Reprinted from

Volume 14 Number 2 September 1980

Artificial Intelligence

AN INTERNATIONAL JOURNAL

Planning and Meta-Planning (MOLGEN: Part 2)

Mark Stefik*

Computer Science Department, Stanford University, Stanford, CA 94305, U.S.A.

Recommended by Daniel G. Bobrow

1931 1981

North-Holland Publishing Company
Amsterdam

Planning and Meta-Planning (MOLGEN: Part 2)

Mark Stefik*
Computer Science Department, Stanford University, Stanford, CA 94305, U.S.A.

Recommended by Daniel G. Bobrow

ABSTRACT

The selection of what to do next is often the hardest part of resource-limited problem solving. In planning problems, there are typically many goals to be achieved in some order. The goals interact with each other in ways which depend both on the order in which they are achieved and on the particular operators which are used to achieve them. A planning program needs to keep its options open because decisions about one part of a plan are likely to have consequences for another part.

This paper describes an approach to planning which integrates and extends two strategies termed the least-commitment and the heuristic strategies. By integrating these, the approach makes sense of the need for guessing; it resorts to plausible reasoning to compensate for the limitations of its knowledge base. The decision-making knowledge is organized in a layered control structure which separates decisions about the planning problem from decisions about the planning process. The approach, termed meta-planning, exposes and organizes a variety of decisions, which are usually made implicitly and sub-optimally in planning programs with rigid control structures. This is part of a course of research which seeks to enhance the power of a problem solvers by enabling them to reason about their own reasoning processes.

Meta-planning has been implemented and exercised in a knowledge-based program (named MOLGEN) that plans gene cloning experiments in molecular genetics.

1. Introduction

Method consists entirely in properly ordering and arranging the things to which we should pay attention. Descartes, *OEuvres*, vol. X, p. 379; "Rules for the Direction of the Mind," from Polya [17].

Verily, as much knowledge is needed to effectively use a fact as there is in the fact, de Kleer et al. [5].

Problem solvers repeatedly decide what do do. A problem solver has goals and a repetoire of possible actions. It decides when the actions can be applied and

* Current address: Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304, U.S.A.

how they should be combined. In computational systems, such decisions about actions are called *control* and a framework for organizing these decisions is called a *control structure.*

A sophisticated control structure should provide flexibility for decision-making—so that a problem solver can take advantage of new information, make guesses, and correct mistakes. It should be able to recognize when an approach is succeeding (even by serendipity), and also recognize when it is failing. In substantial planning problems, there are too many possibilities to try everything, so a planner must exercise control by deciding what to try. To plan effectively, a planner must know when to make commitments and when to wait. These capabilities place a premium on flexibility and raise challenges for finding ways to use information effectively.

This paper considers the control of decision making in planning. A computer program, named MOLGEN, has been implemented and used as a vehicle for studying planning. This is the second of two papers about MOLGEN. The first paper [23] considers experiment design as a hierarchical process and characterizes planning decisions in terms of operations on constraints. This paper focusses on the control of those planning decisions.

Experimentation with flexible control structures is of increasing significance in *knowledge-based* problem solvers for which we have an apparent wealth of information in knowledge bases and increased ambitions for intelligent behavior. Almost 20 years ago, Newell [15] surveyed several organizational alternatives for problem solvers. Only a few substantial experiments have been done in the intervening years. The elaboration of the principles for creating effective control structures is hindered by the substantial effort involved in building systems that use them. Most experiments consider only one control structure and a small set of control issues.

This paper describes a control structure, termed *meta*-planning, which enables a planner to reason (to some degree) about its own reasoning process. *Meta*-planning provides a framework for partitioning control knowledge into layers so that flexibility is achieved without the complexity of a large monolithic system. The rationale for this is discussed in Section 2. The implementation of MOLGEN's layered control structure is presented in Section 3. The final sections consider the conceptual ties of this work to other research on layered control and show how additional capabilities not implemented in MOLGEN increase the need for flexible control.

2. The Rationale for Layers

This section presents the rationale for organizing problem solving knowledge as a control hierarchy. It begins with a discussion of the shortcomings of monolithic agenda systems.

2.1. The trouble with agendas

The idea of organizing a problem solver around an *agenda*, that is, around a queue of competing processes, is currently popular as a flexible control structure [2, 8, 13]. The agenda control structure is a generalization of the fetch-execute cycle that is used in the hardware of most digital computers (see Fig. 1). In the fetch-execute cycle, instructions are retrieved by a processor and executed. Execution of the instructions causes changes in memory and (presumably) brings the system closer to the completion of a problem. In agenda systems (see Fig. 2), the tasks are similar to instructions except that they are usually more complicated than machine instructions and the retrieval and selection criteria are richer. Still, the basic organization is the same and the potential for programming the system by altering the tasks and selection criteria is appealing.

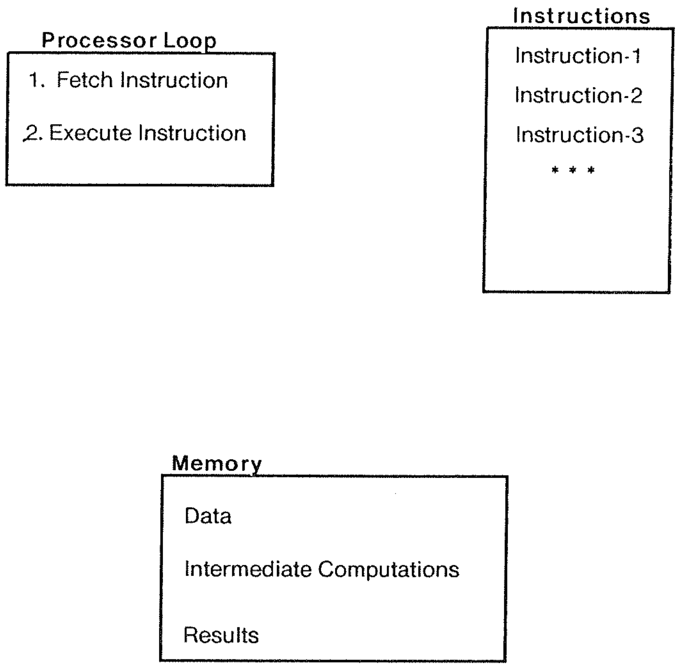

FIG. 1. The Fetch-Execute Cycle. Instructions are retrieved by a processor and executed. Execution of the instructions causes changes in memory and (presumably) brings the system closer to the completion of a problem.

In the basic agenda organization the knowledge for selecting tasks is contained in the interpreter. Hayes [11] has argued that this approach is a return to the 'uniform black-box problem solver':

> We have now come full circle, to a classical problem-solving situation. How can the interpreter decide what order to run the processes in? It doesn't know anything about any parti-

cular domain, so it can't decide. So we have to be able to tell it. . . . This is exactly the situation which . . . the proceduralists attacked. In removing the decision to actually *run* from the code and placing it in the interpreter, advocates of [agenda systems] . . . have re-created the uniform black-box problem-solver.

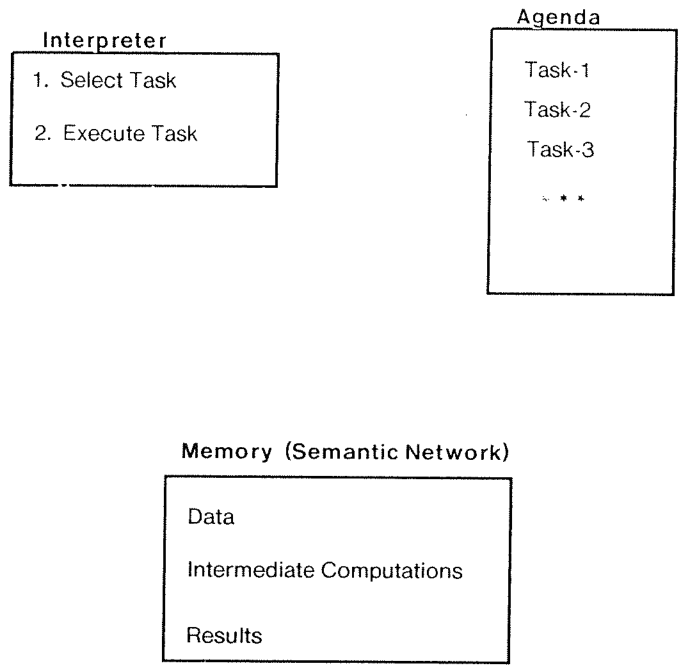

FIG. 2. The agenda control structure. This control structure has essentially the same architecture as a digital computer tasks except that tasks are usually more complicated than machine instructions, and the retrieval and selection criteria are richer. The memory in agenda systems is often structured as a semantic network.

Several modifications in this scheme have been proposed to simplify the interpreter by removing the task selection criteria. One approach is to provide an initial set of tasks and arrange that new tasks are created by earlier tasks as they are run. Tasks are run in a standard order, such as the order in which they were created. This approach limits the amount of scheduling information in the interpreter by the drastic expedient of eliminating it altogether. The priority queue approach (see Fig. 3), which recognizes that some tasks are more important than others, is to assign priorities to tasks and select those with the highest priorities.

Several embellishments are possible on the priority queue approach. One embellishment is to raise or lower priorities to reflect changing conditions. For example, the reasons for running a task may lose validity if other tasks have been executed between the time that a task is created and the time that it

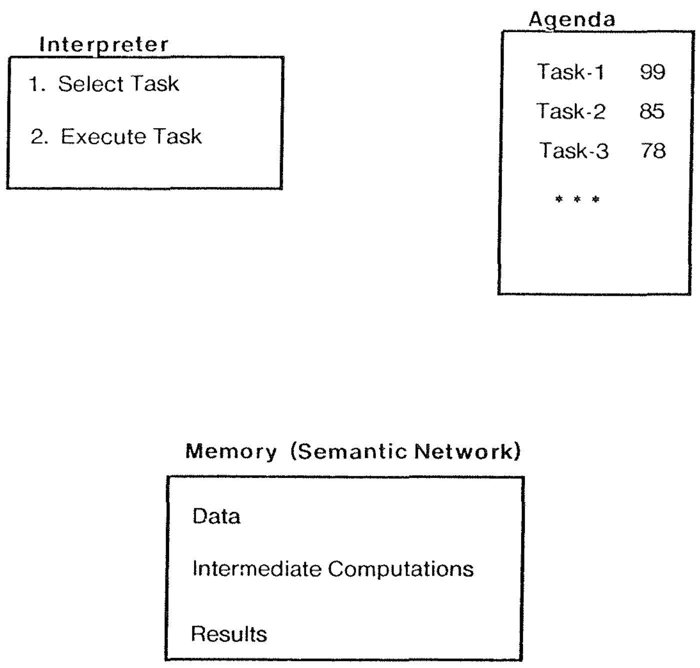

FIG. 3. Priority Queue. The task description in agenda control structures can be augmented to include priority information to represent the idea that some tasks are more important than others. This begs the question of where to put the knowledge for setting the priorities.

reaches the top of the agenda. Some programs distinguish between activation-conditions and pre-conditions to handle this case. In programs like Lenat's AM a task may increase in importance as reasons for running it accumulate.

Task-centered scheduling is an augmentation of the priority queue idea that associates priority-estimating functions with each of the tasks, instead of numeric priorities (see Fig. 4). However, the treatment of complexity is not necessarily much improved. If the problem solver has multiple goals, each priority-estimating function must potentially know about all of them. In the worst case, the priority-estimating functions for each task need to take into account all of the other possible tasks. Unfortunately, the number of possible interactions grows rapidly with the number of tasks. Even if we consider only pairwise interactions, their number is proportional to the square of the number of tasks; if we count interactions between groups of tasks, the number of possible interactions grows exponentially with the number of tasks.

While the worst case does not usually hold in practice, this shows how the control knowledge can become unmanageably complicated in a monolithic organization. The fix for the complexity problem is not simply a choice between a centralized or decentralized organization. In the absense of some other kind of simplifying organization, we have only a choice between (1) maintaining an arbitrarily complex central function, or (2) maintaining a set of interacting task-centered functions.

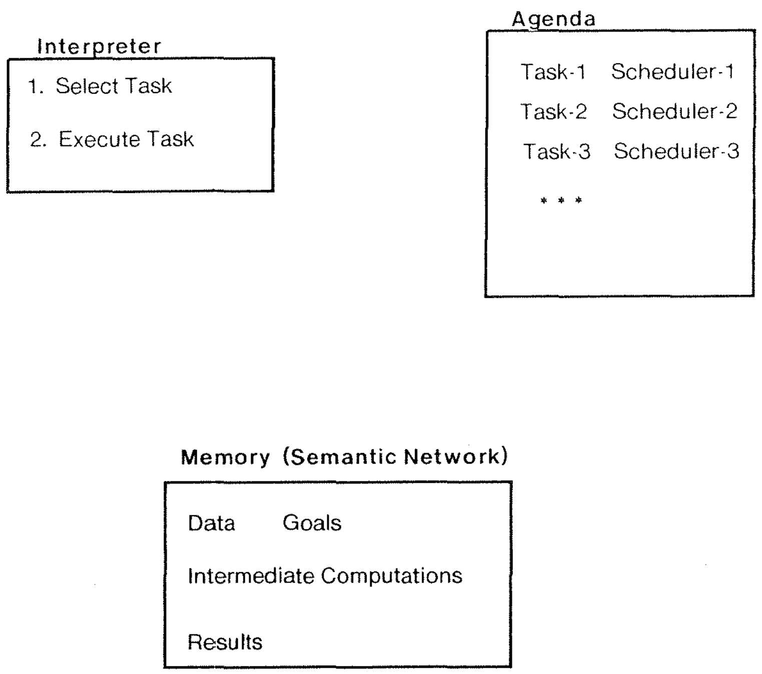

FIG. 4. Task-centered computation of priorities. In some tasks, priorities (or even applicabilities) change as conditions change. To account for this, some systems associate functions with tasks to compute the current priority on demand. Unfortunately, if there are many possible goals, each function must be able to take all of them into account.

2.2. Recognizing the *meta*-problem

Continuing with Hayes's argument:

> A somewhat more sophisticated idea is to allow descriptors for subqueues and allow processes to access these descriptors. . . . But none of these ideas seem very convincing. And we have now moved down another level, to the interpreter of the interpreter-writing language of the representation language.
>
> The only way out of this descending spiral is upwards. 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 argument is supported by the observation that many of the important actions, goals, and constraints can be characterized as being on a *meta*-level. For example, in the classical Missionaries and Cannibals puzzle, a first-level action would be a trip across the river specifying various occupants of the boat. The first-level goal is to get the people across the river safely and the first-level constraints relate to the eating habits of the people. Introspection while trying to solve the puzzle suggests that much of the thought process is actually on a *meta*-level, that is, it is about the process of solving the puzzle. For example, higher level actions would include (1) generating plausible sequences of first-level actions to find a solution, or (2) describing possible intermediate states in the boating plan, or (3) changing the representation of the first-level problem. The meta-level goal is to find a solution to the puzzle; limitations on the availability of computational resources are examples of meta-level constraints. In general, any choices or evaluation criteria which relate to the process of problem solving can be characterized as *meta*-level considerations.

That many planning decisions are about the *meta*-problem explains the source of the combinatorially explosive number interactions in monolithic organizations. If the tasks in the agenda refer only to first-level actions, then the scheduling functions must take into account not only the applicability

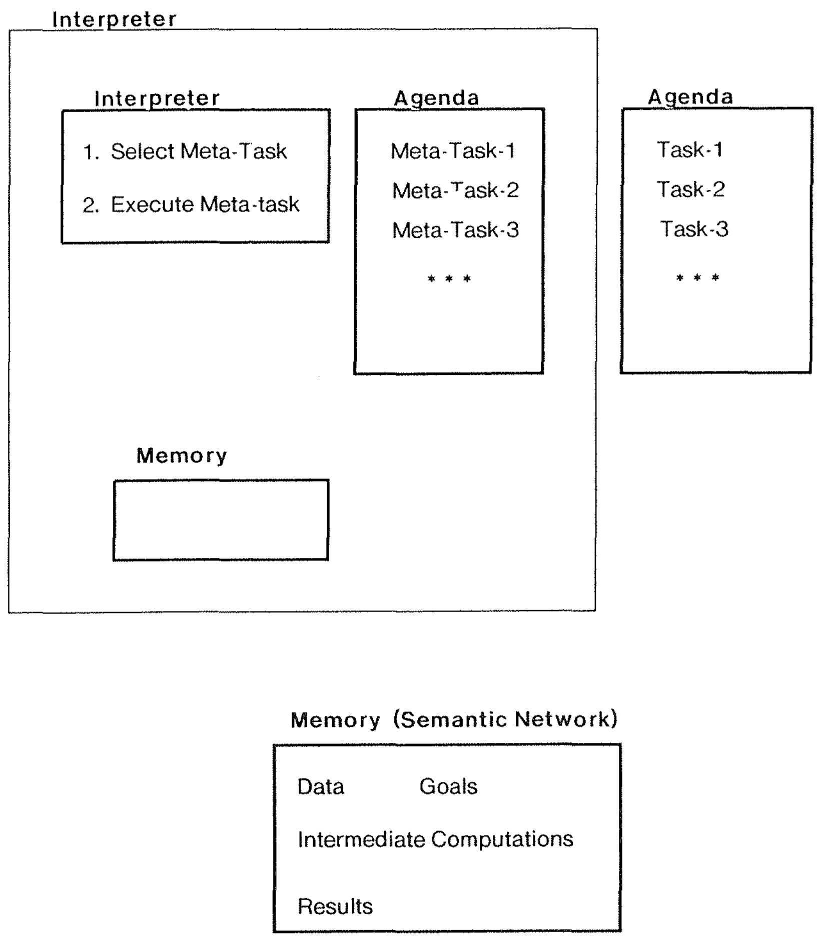

FIG. 5. Layered Agenda Structures. The original interpreter can be replaced by another agenda-based problem solver dedicated to the scheduling problem. The higher problem solver should represent the control concepts necessary for solving the original problem. Tasks in the higher problem solver select and execute tasks in the original problem.

considerations of the first-order problem, but also the problem-solving considerations of the meta-problem. If the meta-level tasks are not represented explicitly and are not hidden in a 'black box interpreter', then the higher-level considerations will surface in a confusing way as task interactions on the first level. The basic difficulty with all of the monolithic agenda approaches is that they provide no hierarchical framework for complex control. They provide no meta-level concepts or global perspectives to bear on scheduling and arbitration.

How then might this knowledge be organized? One approach is to extend the agenda idea to a multiple-layered structure with a separate problem solver for the meta-problem. We can replace the complex interpreter in the original agenda structure with a second agenda-based problem solver dedicated to the meta-problem (see Fig. 5). In this *multiple-layered system*, the interpreter of each agenda is essentially another agenda-based system. Tasks in the second layer act collectively as the interpreter of the lower agenda system by creating, ordering, and running the lower tasks.

The layering idea is not limited to two layers; it can be applied recursively. To reduce the apparent complexity of a system, layers can be created until the knowledge remaining in the uppermost interpreter is trivial.

The use of layers has been essential to the creation of computer systems for many years. Most computer programs are built on a succession of layers (or virtual machines)—through hardware, firmware, operating system calls, programming languages, and application software. This practice reduces the amount of expertise that is needed to program a system by providing layers of concepts appropriate for the application. This paper argues for the use of such layers for organizing the control knowledge in a problem solver.

2.3. Advice and control

Many books about problem solving contain advice. For example, the following advice was offered by Polya [18]:

(1) Think on the end before you begin. · · · Let us inquire from what antecedent the desired result could be derived.

(2) Examine your guess. · · · Don't let your suspicion, or guess, or conjecture grow without examination till it becomes ineradicable.

(3) A wise man changes his mind, a fool never does.

(4) Look around when you have got your first mushroom or made your first discovery; they grow in clusters.

It is generally conceded by researchers in AI (artificial intelligence) that there is a considerable gap between advice such as this and its realization in problem solving programs. As Mostow and Hayes-Roth [14] have observed, considerable knowledge is sometimes required in order to interpret such advice. Another part of the difficulty is that there is often no apparent place to put advice in a problem solver. Heuristics like these deal essentially with control concepts, so the absence of an explicit vocabulary of control concepts necessarily impedes the representation of such advice. This research takes some first steps towards defining a vocabulary of control concepts and suggests that layers of control can provide a useful framework for representing them.

3. A Model for Planning

This section presents the layers of control (termed *planning spaces*) that are used to model hierarchical planning in MOLGEN. The main features of the implementation are

(1) a trivial finite-state machine as the top-level interpreter,

(2) the factoring of the knowledge for using plausible and logical reasoning from the planning operations, and

(3) the development of a vocabulary of operators and concepts for hierarchical planning with constraints.

MOLGEN uses three layers and an interpreter as shown in Fig. 6. The three spaces have parallel structure: each space has operators and objects and steps. Each layer controls the creation and scheduling of steps in the layer below it. The spaces are described here starting with the bottom or *domain* space:

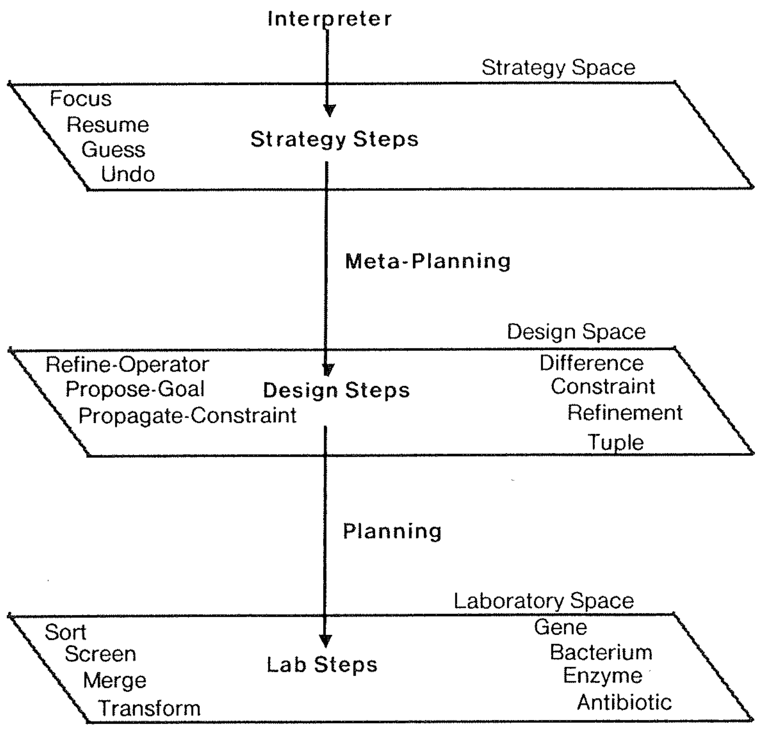

FIG. 6. MOLGEN's planning spaces. The design space *plans* by selecting and executing laboratory steps; the strategy space *meta-plans* by selecting and executing design steps.

(1) ***Laboratory space*** (*or domain* space)—knowledge about the objects and operations of a genetics laboratory. The operators in this space represent actions that can be performed by a laboratory technician; the objects are the things that can be manipulated in the genetics laboratory. (Laboratory space also contains abstractions of these objects and operators.) Steps (i.e., tasks) in laboratory space are executed in order to simulate a real genetics experiment. This bottom space is not a *control* level at all; it represents knowledge about genetics. Laboratory space describes what can be done in the laboratory, but not when to do it in an experiment.

(2) ***Design space***—knowledge about designing plans. This space defines a set of operators for sketching plans abstractly and for propagating constraints around in a laboratory plan as it is refined. These operators model the actions of an *experiment designer.* Steps are executed in design space in order to create and refine the laboratory plan.

(3) ***Strategy space***—knowledge about strategy. This space has two problem-solving approaches: heuristic and least-commitment. Steps are executed in strategy space in order to create and execute the steps in the design space.

(4) The ***Interpreter***—this program is MOLGEN's outermost control loop. It creates and executes steps in the strategy space.

The design operators *plan* by creating and scheduling laboratory steps; the strategy operators '*meta-plan*' by creating and scheduling design steps.

3.1. Control messages

The stratification of control knowledge introduces some organizational requirements:

(1) The operators in a meta-level need to be able to create and schedule first-level tasks.

(2) Meta-level operators should be able to reference and describe first-level entities.

(3) For convenience in a changing knowledge base, an interface between the spaces should isolate the meta-level operators from trivial name changes in the first-level space.

In MOLGEN, the translation of domain-level information into design-level concepts was implemented using an object-centered approach (see Bobrow and Winograd [2]). MOLGEN's operators were represented as 'objects' (called units) that communicated by passing standardized messages. This enabled operators in a *meta*-space to look up information in a lower space and to communicate uniformly with the operators in the lower space. A message passing protocol was implemented using facilities provided by the Units Package representation language [25]. No claim is made that a message-passing protocol is essential for implementing a layered control structure. Indeed, more sophisticated methods for insulating problem solving layers and translating between vocabularies are possible, but were not implemented in MOLGEN.

The following sections discuss the vocabulary and rationale for each of the planning spaces. For concreteness, the operators in each planning space will be described in terms of the *message-passing* protocols that were used. The specific messages will be introduced as needed.

3.2. Laboratory space

Laboratory space is MOLGEN's model of the objects and actions relevant to gene cloning experiments. It was described in the companion paper and will be summarized here briefly. Laboratory space defines the set of possible laboratory experiments by describing the allowable laboratory objects and operators.

The objects in laboratory space represent physical objects that can be manipulated in the genetics laboratory. They include such things as antibiotics, DNA structures, genes, plasmids, enzymes, and organisms. Seventy-four different generic objects are represented in total. The knowledge base includes annotations which indicate which of these objects are available 'off the shelf'.

The operators in laboratory space represent physical processes that can be carried out in the genetics laboratory. They are organized into four groups depending on whether they

(1) combine objects together (***Merge***),
(2) increase the amount of something (***Amplify***),
(3) change the properties of something (***React***), or
(4) separate something into its components (***Sort***).

Collectively, these abstract (or generic) operators are called the **'MARS'** operators. Thirteen specific operators are represented as specializations of these. For example, ***Cleave*** is a ***React*** operator which cuts a DNA molecule with a restriction enzyme; ***Screen*** is a ***Sort*** operator which removes unwanted bacteria from a culture by killing them with an antibiotic.

Steps in laboratory space describe the application of (possibly abstract) genetics operators to genetics objects. When MOLGEN *runs* (i.e., executes) a step in a higher level space, the step is said to have been done and corresponding changes in the plan structure are made. MOLGEN can not actually run the laboratory steps in the sense of doing them in the laboratory; executing the code is interpreted as *simulating* the laboratory step.

Laboratory space does not contain the knowledge about how to effectively plan experiments, that is, how to arrange laboratory steps to achieve experimental goals. This knowledge is organized in the design and strategy spaces.

3.3. Design space

Design space is MOLGEN's first control space. It contains operators for planning, that is, for creating and arranging steps in laboratory space. This section discusses the concepts and operations of meta-planning in enough detail to give a sense of how MOLGEN worked. No claim is advanced that the particular operators described here are universally applicable in problem solving, or that the partitioning of functionality is ideally chosen. Rather, this description is offered as an example of the kinds of operations that can be treated explicitly in problem solving, and it is hoped that the example will provoke the kind of careful thinking that will lead to defining and organizing control information in other systems.

The main idea in organizing MOLGEN's design space is that planning can be viewed as operations on constraints. Three operations on constraints are important: formulation, propagation, and satisfaction. *Constraint formulation* is the dynamic creation of constraints that set limits on the acceptable solutions. Constraints correspond to commitments in planning. By formulating constraints about abstract objects (variables), MOLGEN creates partial descriptions of the objects and postpones complete instantiation until later. *Constraint propagation* performs communication by passing information between nearly independent subproblems. *Constraint satisfaction* refines abstract entities into specific ones. It pools the constraints from the nearly independent problems to work out solutions. The operations on constraints are an important subset of MOLGEN's design operators. These operators provide a repertoire of possible actions that MOLGEN can use to plan hierarchically. Fig. 7 gives an outline of the objects and operators in MOLGEN's design space.

- Design-Object
  - Constraint
  - Difference
  - Refinement
  - Tuple
- Design-Operator
  - **Comparison**
    - Find-Unusual-Features
    - Check-Prediction
  - Temporal-Extension
    - Propose-Operator
    - Propose-Goal
    - Predict-Results
  - Specialization
    - Refine-Operator
    - Propagate-Constraint
    - Refine-Object

FIG. 7. Outline of the objects and operators in design space.

3.3.1. *Design operators*

MOLGEN has three categories of design operators:

(1) ***Comparison operators*** that compare goals and compute differences,

(2) ***Temporal-extension operators*** that extend a plan forwards or backwards in time, and

(3) ***Specialization operators*** that make an abstract plan more specific.

3.3.1.1. *Comparison operators*

Comparison is a fundamental operation in planning. The results of comparison are represented as *differences.* Differences are represented as objects in MOLGEN's design space. Other design objects include constraints, refinements, and tuples. Examples of these objects are given in the planning trace in the previous paper. Thus, unlike the objects in laboratory space which represent *physical* objects, the objects in design space represent *conceptual* objects.

Typically, MOLGEN chooses laboratory operators that can reduce specific differences. This basic formulation goes back to the Logic Theorist program and has appeared in many planning programs. MOLGEN has two comparison operators: ***Find-Unusual-Features*** and ***Check-Prediction***.

Find-Unusual-Features is a design operator that examines laboratory goals. Sometimes a good way to select abstract operators to synthesize objects in cloning experiments is to find features in which the objects are highly specialized or atypical, and then find operators that act on those features. ***Find-Unusual-Features*** does this by comparing objects (e.g., *Bacterium*-1) with their prototypes (e.g., *Bacterium*). ***Find-Unusual-Features*** searches recursively through units representing the parts of an object and stops when it has found differences at any depth of processing.

Check-Prediction is a design operator that compares the predictions from simulation of a laboratory step with the forward goal for the step. This operator is useful for detecting cases where a plan needs to be adjusted because the predicted results of a laboratory step do not quite match the goals. MOLGEN discovers this mismatch after simulating the laboratory step when the knowledge in its simulation model is more complete than the knowledge that was used for selecting the laboratory operator.

3.3.1.2. *Temporal-extension operators*

A design operator that extends a plan forwards or backwards in time is called a *temporal-extension* operator. MOLGEN has three such operators: ***Propose-Operator***, ***Propose-Goal***, and ***Predict-results***.

Propose-Operator proposes abstract laboratory operators to reduce differences. It is activated when new differences appear in the plan and it creates partially instantiated units to represent laboratory steps. It is responsible for linking the new laboratory steps to the neighboring laboratory steps and goals. ***Propose-Operator*** must determine which of the abstract laboratory operators (i.e., the ***MARS*** operators) are applicable. ***Propose-Operator*** takes advantage of the hierarchical organization of the laboratory operators by considering only the most abstract operators. It sends an *apply?* message to each of the abstract laboratory operators. Each laboratory operator has a procedure for answering the message that determines whether the operator is applicable (given a list of differences and constraints). If more than one laboratory operator is applicable, ***Propose-Operator*** puts the list of candidates in a refinement unit and suspends its operation pending messages from strategy space.

The ***Propose-Goal*** design operator creates goals for laboratory steps. It uses messages to access specialized information for laboratory operators. For example, when it sends a *make-goals* message to the ***Merge*** operator, a local procedure creates goals for each of the parts being put together.

Predict-Results is the design operator for simulating the results of a proposed laboratory step. It activates a simulation model associated with each laboratory operator. In the case that the information in the laboratory step is too incomplete for simulation at this stage of planning, ***Predict-Results*** suspends its execution pending messages from strategy space.

3.3.1.3. *Specialization operators*

A hierarchical planner first makes plans at an abstract level and then adds details to its plans. MOLGEN's specialization operators all add details to partially specified plans. The design operators for this are ***Refine-Operator***, ***Propagate-Constraint***, and ***Refine-Object***.

Refine-Operator is the design operator that replaces abstract domain operators (i.e., the MARS operators) in laboratory steps with specific ones. ***Refine-Operator*** is invoked when there are laboratory steps that have their goals and inputs specified but have abstract specifications of the laboratory operator (i.e., ***Merge***). The inputs to laboratory steps are usually incompletely specified when the operator is chosen. For example, the input may be a 'culture of bacteria' without being precise about the type of bacteria. Because laboratory operators often have specific requirements, the process of refinement is accompanied by the introduction of specific constraints on the input. These constraints make the requirements of the laboratory operator specific, without requiring a full specification of the input at the same time. Like other operators in the design space, ***Refine-Operator*** uses several messages in the design space/laboratory space interface to retrieve information about specific laboratory operators.

Propagate-Constraint creates new constraints from existing constraints in the plan. It is organized around the observation that even long-distance propagations can be decomposed into a series of *one-step* propagations through individual laboratory steps. ***Propagate-Constraint*** is invoked when a new constraint appears in the plan. While constraints can, in principle, be propagated in either a forward or backward direction in a plan, only the backward direction (in time) is implemented in MOLGEN. ***Propagate-Constraint*** is activated whenever new constraints appear in the plan. After trying to propagate a constraint, the design task is suspended for possible reactivation if some new laboratory steps appear in the plan. These tasks are cancelled if a constraint is marked as replaced in the plan.

Refine-Object is MOLGEN's constraint satisfaction operator. It is activated when new constraints appear in the plan. Constraint satisfaction involves a 'buy or build' decision in MOLGEN.[1] MOLGEN first tries to find an available object that satisfies the constraints. If this fails, the constraint is marked as failed and the refinement task is suspended. If the constraint is never replaced by a different constraint and MOLGEN runs out of things to do, it may guess that it should make a subgoal out of building the object—thus making the build decision.

Refine-Object evaluates constraints (**Lambda** expressions) using objects from the knowledge base as arguments. The solutions are pooled with those of other constraints on the same objects in design objects called 'tuples' that keep track of the sets of solutions. Sometimes a new constraint will include objects that are included in disjoint tuples; in such cases ***Refine-Object*** combines the subproblems by integrating the tuples into a new tuple with intersected solutions. When enough constraints have been found to make the solution for any abstract variable unique, that variable is anchored to the solution.

3.3.2. *Interface to laboratory space*

Fig. 8 summarizes the messages that were used in MOLGEN. Each laboratory operator includes a procedure to respond to each kind of message. This

Message	Meaning
APPLY?	Asks a lab operator whether it is applicable to reducing a list of differences given a set of constraints.
REFINE	Instructs a lab operator that it has been chosen as a refinement in the plan. Returns a list of new constraints.
MAKEGOALS?	Asks a lab operator whether it needs to modify the goals of a step.
MAKEGOALS	Instructs a lab operator to modify the goals of a laboratory step as needed.
SIMULATE?	Asks a lab operator whether the input to a lab step is specified precisely enough to do a detailed simulation of the lab step.
SIMULATE	Instructs a lab operator to provide a detailed simulation of laboratory step.
BKWD-PROPAGATE	Instructs a laboratory operator to propagate a constraint backwards (in time) from its forward goals to its input.
FWD-PROPAGATE	Instructs a laboratory operator to propagate a constraint forwards (in time) from its input to its forward goals. (This message was not implemented in MOLGEN.)

FIG. 8. Message-protocol interface to laboratory space. These messages are sent by design space operators to control and retrieve information from laboratory space.

[1] Post-thesis examination of MOLGEN's logic has revealed some gaps and confusions in the implementation of this 'buy or build' decision. Some aspects of this are discussed in Section 5.

approach redundantly represents the knowledge about the laboratory operators, since the queries can be about redundant information and there is a separate attached procedure provided for each kind of query from the design space. A direction for future research is to develop an approach for stating the information declaratively once, and then possibly *compiling* it into the procedures like these.

3.4. Strategy space and its interpreter

The distinction between *least-commitment* and *heuristic* approaches to problem solving is the key to the organization of the knowledge in MOLGEN's strategy space. A least commitment approach requires the ability to defer decisions when they are under-constrained. It relies on a synergistic relationship between subproblems, so that constraints from different parts of a problem can be combined before decisions are made. A heuristic approach utilizes plausible reasoning, to make tentative decisions in situations where information is incomplete.

MOLGEN's strategy space is organized as four strategy operators: ***Focus***, ***Resume***, ***Guess***, and ***Undo*** as described in the next section. Fig. 9 shows how

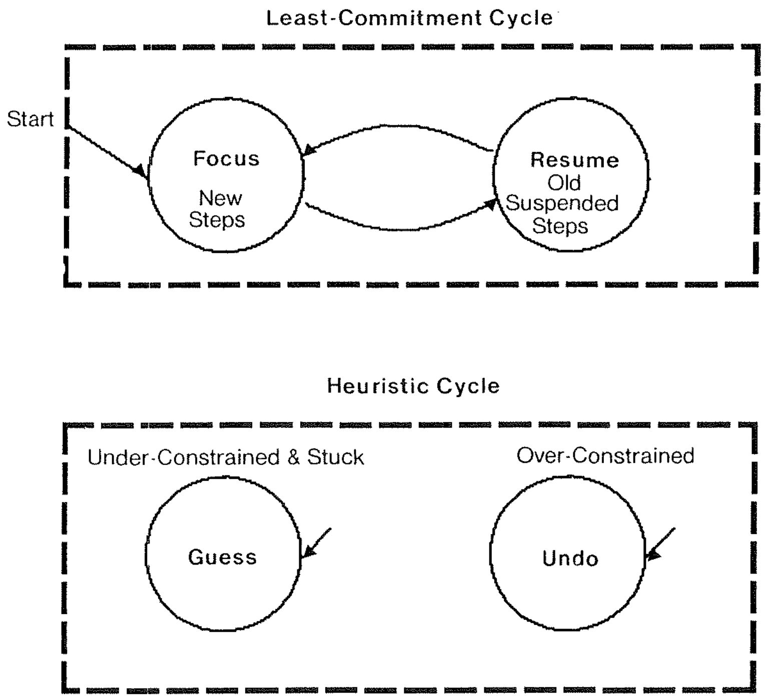

FIG. 9. Least-commitment and heuristic cycles. This diagram shows how the strategy operators are controlled by MOLGEN's interpreter. The least-commitment cycle makes conservative changes in the plan, depending on synergy between subproblems (and constraint propagation) to keep going. When MOLGEN runs out of least-commitment steps, it resorts to guessing using the heuristic cycle.

the strategy operators are controlled by a simple finite state machine, the interpreter, which is composed of two main parts. Section 3.4.2 describes the message-passing protocol that interfaces these operators with the design space. The significance of these ideas is discussed in Section 3.4.3.

3.4.1. *Strategy operators*

The four strategy operators partition the knowledge about logical and plausible reasoning out of the design operators that create the experimental plans. This section describes MOLGEN's strategy operators and how their control of design space is implemented.

3.4.1.1. *Focus*

The ***Focus*** strategy operator is used to create and execute new design tasks. ***Focus*** sends a *find-tasks* message to every design operator. This causes the procedures associated with the design operators to search for work in the current plan and to report back where they can be applied. These procedures mark the places where they have looked in the plan, so that they can avoid redundant checking. The application points are recorded in design steps and the design steps are put into an agenda. In the simplest case, only one design step is ready at any given time. When several design steps are ready simultaneously (usually from different parts of the plan), ***Focus*** has to choose one of them to go first. In MOLGEN, priorities were assigned to the design operators as described in the following paragraphs. These priorities were used by the ***Focus*** and ***Resume*** operators to schedule design steps when several were simultaneously ready.

Focus executes a design task by sending it a *simulate* message. A task may terminate in any of four possible states: done, failed, suspended, or cancelled. ***Focus*** iterates through its agenda. After each execution, it sends out the *find-tasks* message again. As long as steps are successful or are suspended due to being under-constrained, ***Focus*** continues through the agenda. However, if a design-step is over-constrained, ***Focus*** stops processing and returns to the interpreter with the status over-constrained. (This causes the ***Undo*** operator to be invoked.)

The priorities for scheduling competing design tasks are shown in Fig. 10. They reflect a bias towards performing comparison before temporal-extension, and temporal-extension before specialization. This was intended to encourage MOLGEN to look first for differences, then to use them to sketch out an abstract plan, and finally to refine to specific objects and operators. Given such a control scheme, it is interesting to ask whether it was effective or necessary. While no comprehensive set of measurements was done, an experiment was performed accidentally. The original versions of the ***Focus*** and ***Resume*** operators had a bug which caused them to use the priorities precisely backwards, so that the design operators with the lowest priority were tried first. Interestingly,

Operator Class	Design Operator	Priority
Comparison		
	Check-Prediction	9
	Find-Unusual-Features	9
Temporal-Extension		
	Propose-Goal	7
	Propose-Operator	6
	Predict-Results	5
Specialization		
	Refine-Operator	4
	Propagate-Constraints	3
	Refine-Object	2

FIG. 10. Priorities of the design operators. When the strategy operators have more than one design task that seems applicable in the least-commitment cycle, these priorities are used to order the tasks. They reflect a bias towards extending a plan in time before extending it in depth.

MOLGEN was still able to plan correctly, except that it did a lot of unnecessary work. Design tasks were scheduled, and immediately suspended because of insufficient information. Although this buggy version of MOLGEN never completed a plan, it seemed to make the correct decisions, but only after a great deal of fuss; starting tasks, suspending them, and picking them up later.

3.4.1.2. *Resume*

Resume is the strategy operator that restarts suspended design steps. A design step may be suspended because it is under-constrained, or because there is potentially additional work to be found later. ***Resume*** works very much like ***Focus*** in that it creates an agenda and uses priorities to schedule design tasks when more than one is ready. It differs from ***Focus*** in that it does not look for new work to do, only old work to start up again. It sends a *resume?* message to every suspended design-step telling it to indicate if it is ready to run again. ***Resume*** reactivates the design steps that are ready to run by sending them a *resume* message.

Fig. 11 shows an example from a cloning experiment where the ***Resume*** operator was used to activate a design step. The design step in this case is the ***Propagate-Constraint*** step shown in the right center portion of the figure. This step was activated when the constraint (dark box) was added to the plan. The constraint required that the enzyme corresponding to the sticky-ends of the vector should not cut the desired gene (rat-insulin) that it carries. At the time of formulation, there was no place to propagate the constraint, because the ***Cleave*** step in the plan had not yet been added to the plan. Later, other design steps proposed and instantiated the ***Cleave*** step, and the suspended ***Propagate-Constraint*** design step could be resumed.

3.4.1.3. *Guess*

Occasionally during planning, information is not adequate for making any

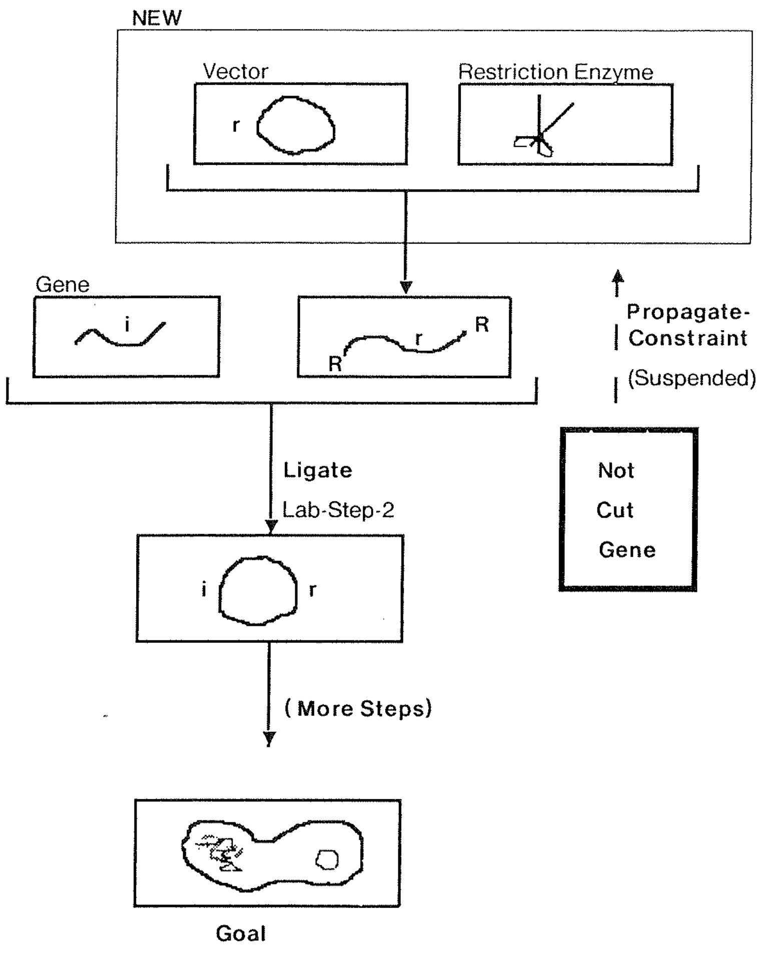

FIG. 11. Example of resuming. The ***Propagate-constraints*** design step was created when the constraint was created, but was suspended until enough of the plan had evolved to create a place for propagating the constraint.

irrevocable decision. When MOLGEN has run out of *least-commitment* changes to a plan, it looks for a plausible commitment that will allow it to continue with the design process. This is recognized by MOLGEN when the ***Focus*** and ***Resume*** strategy operators have nothing to do.

The ***Guess*** operator sends a *guess?* message to the operator of every suspended design step. These design steps represent under-constrained decision points in planning. The *guess?* message causes each suspended step to examine its options and to compute a numerical estimate of the utility of commiting to one of its choices. ***Guess*** then activates the task with the highest rating by sending it a *guess* message. After making a single guess, the ***Guess*** operator returns to the interpreter and another ***Focus*** step is started.

3.4.1.4. *Undo*

Undo is the strategy operator for backtracking when a plan has become over-constrained. It is the least developed of the strategy operators in this research. Other researchers have developed a more comprehensive approach to dependency-directed backtracking than has been done for MOLGEN; most of the effort in this research has gone towards avoiding backtracking. In Section 3.4.3 it is argued that it is not always feasible to avoid revoking decisions made in planning.

For the record, MOLGEN's (primitive) ***Undo*** operator works as follows. First it picks a candidate design step to undo. It begins by making a list of design steps that were guessed and searches this list for a step whose *output* was the *input* of the over-constrained design step. If ***Undo*** finds such a step, it sends it an *undo* message. This tells the design step to remove the effects of its execution from the plan. The design step is then marked as undone. If ***Undo*** finds no candidate, it prints out an apology and quits. ***Undo***, as implemented in MOLGEN, is not capable of tracking down all of the consequences of a decision to be undone and does not check that the undoing actually alleviates the over-constrained situation.

3.4.2. *The interface to design space*

The preceding account of the strategy operators has discussed a number of messages that are sent from the strategy space to the design space. These messages provide an interface to design space that is analogous to the message interface between design space and laboratory space. The interface provides a way for the strategy operators to communicate with the design operators uniformly. They enable strategy operators to invoke design procedures without knowing the names of the procedures. The interface messages in the current implementation are shown in Fig. 12.

Message	Meaning
FIND-TASKS	Instructs the design task to search for new work to do. Returns a list of tasks to do.
SIMULATE	Causes a design task to be executed.
RESUME?	Asks a suspended design task whether it is ready to be re-started.
RESUME	Instructs a suspended design task to resume execution.
GUESS?	Asks a suspended design task whether it can make a plausible guess. Returns a numeric rating of the guess.
GUESS	Instructs a suspended design task to make its best guess.
UNDO	Instructs a finished (guessed) design task to undo the effects of its execution.

FIG. 12. Message-protocol interface from strategy space to design space.

3.4.3. *Significance of the strategy space*

The heuristic and least commitment cycles are reminiscent of two earlier AI programs, HACKER (Sussman [26]) and NOAH (Sacerdoti [20]), respectively. These programs epitomize two points on a spectrum of time of commitment. HACKER epitomizes heuristic early commitment and NOAH epitomizes late (or least) commitment. HACKER guesses its way to a solution using debugging to fix things when the assumptions are bad; NOAH defers decisions and invites the possibility that information will become available later that narrows the possibilities. After NOAH successfully and optimally solved some of the problems that were troublesome for HACKER, Sacerdoti [20] observed that

> HACKER does a lot of wasted work. While the problem solver will eventually produce a correct plan, it does so in many cases by iterating through a cycle of building a wrong plan, then applying all known critics to suggest revisions of the plan, then building a new (still potentially wrong) plan.

The bugs arose, in Sacerdoti's view, from premature and inappropriate decisions by the problem-solver. By delaying judgment, a problem-solver can achieve a considerable savings in computational effort.[2] Sussman and later Goldstein disagreed on the power of the least-commitment principle. Bugs in a design are to be expected; they result from *heuristically justifiable* but incorrect inferences in the design process. Goldstein [10] observed that

> Many bugs are just manifestations of creative thinking—the creation and removal of bugs are ***necessary*** steps in the normal process of solving a complex problem.

The formulation of strategy knowledge in MOLGEN integrates and extends the two earlier approaches to planning. By integrating the least commitment cycle with the heuristic cycle in strategy space, MOLGEN makes sense of the need for guessing: we can guess, but only when we have to. Bugs are inevitable, but only when we guess. The amount of guessing is a measure of missing knowledge; the more we know (and are able to use what we know), the less we need to guess. Guessing is used to compensate for the limited knowledge of a problem solver. With increased expertise we expect reduced guessing and backtracking. By increasing MOLGEN's knowledge about constraint formulation and propagation, we decrease its need to revoke decisions. The least commitment approach is conservative reasoning; the heuristic approach is plausible reasoning.

The appeal of the least commitment cycle is that it uses a monotonic approach towards a solution; as long as MOLGEN stays in this cycle, it is

[2] Barstow [1] illustrated this in an example of program refinement when abstraction trees are skinny at the top, and bushy at the bottom. He cited a case where delaying a choice reduced the number of rule applications in half.

guaranteed to make no wrong moves. The ***Focus*** operator calls on new design operators to make infallible (i.e., irrevocable) changes in the developing plan; the ***Resume*** operator re-starts any suspended design operators which have received additional information. The power of this cycle derives from the ability of the various design tasks to reinforce each other in their decisions. The operators can be suspended when they have insufficient information and restarted when it becomes available. Reinforcement is a consequence of constraint propagation, which passes partial results between subproblems. As long as there are new things to do in the plan or suspended things to restart, MOLGEN stays in the least-commitment cycle. If MOLGEN runs out of things to do (and the plan is incomplete), the plan is said to be *under-constrained* and it calls upon the ***Guess*** operator to make some tentative decision that will enable planning to continue. The ***Guess*** operator calls again upon the design operators to make moves that are plausible, even if they cannot be guaranteed. If MOLGEN discovers at any point that the plan is over-constrained, it calls on the ***Undo*** operator to revoke some of the design decisions, typically undoing a choice that was guessed.

4. Relationships to Other Work

This section considers other problem-solving programs with layered control structures. While the idea of layered control was reported as early as 1963, programs which substantially utilize this idea have appeared only recently. Several researchers (e.g., de Kleer et al. [5] and Georgeff [9]) have proposed approaches for controlling inference; this section will consider only approaches that are layered.

4.1. GPS

The idea of layers of control with problem-solving operators was anticipated in 1963 by Simon [22] in his experiments with the heuristic compiler in the GPS framework:

> It should . . . be feasible, by modifying the top-level programs, to bring the Heuristic Compiler into a form which would allow its problem-solving processes to be governed by GPS. . . . . That is, GPS would first be applied to the task environment of the General Compiler; applying an operator in this environment would consist in applying GPS to the task environment.

This suggestion anticipates the use of problem-solving operators that are distinct from the domain operators. The idea for the Heuristic Compiler was recursive in that it used an instantiated version of GPS as an operator in the *difference tables* of a higher version of GPS.

4.2. TEIRESIAS

TEIRESIAS (Davis [3, 4]) with its *meta-rules* also used a layered control structure. TEIRESIAS was developed in the context of MYCIN (Shortliffe [21]), a medical-consultation system. MYCIN performs an exhaustive goal-directed search through a diagnostic AND/OR tree. At each stage of the diagnosis, MYCIN retrieves the set of production rules which conclude about a premise of interest. In TEIRESIAS, the system was modified so that object-level production rules could be reordered and pruned according to explicit criteria in *meta-rules*. These criteria were used by TEIRESIAS to shorten or re-order the list of potentially applicable rules considered. The idea of higher order meta-rules (e.g., meta-meta-rules) that would act on other meta-rules was also considered, but the medical domain offered no examples.

4.3. HEARSAY-like systems

Several recent AI programs with layered control structures have been based on ideas from the *unlayered* HEARSAYII program [6] for speech understanding. The architecture of HEARSAY-II incorporated three main ideas which have influenced the design of the later programs:

(1) ***Hierarchical hypothesis structure.*** Each level was more abstract than the level below it. The hypotheses were kept on a global data structure termed the *blackboard*.

(2) ***Knowledge Sources.*** Operators termed *KS*s (for Knowledge Sources) made hypotheses at the different abstraction levels.

(3) ***Focus of attention.*** A centralized control mechanism was used to focus attention on parts of the hypothesis space and to coordinate the KSs.

4.3.1. *SU-X and SU-P*

In 1977, Nii and Feigenbaum [16] described two computer programs, SU-X and SU-P, that did signal interpretation tasks. SU-X interpreted instrument signals in a military context and SU-P (also known as CRYSALIS) interpreted X-ray crystallography data to determine protein structure. These programs extended the HEARSAY-II architecture as follows:

(1) HEARSAY-II's single-layered control structure (the hypothesize and test paradigm) was extended to multiple layers and

(2) HEARSAY-II's blackboard was partitioned into distinct areas.

The control layers in both programs were called *hypothesis-formation*, *hypothesis-activation*, and *strategy*. KSs on the first layer formed hypotheses from the incoming signal. The two operators on the second layer, the *hypothesis-activation layer*, were called the *event-driver* and the *expectation-driver*. They corresponded to data-driven and goal-driven policies for activating KSs on the first layer. The KSs on the third or strategy layer decided (1) how close the system was to a solution and (2) how well the KSs on the second level were performing and (3) when and where to redirect the focus of attention in the data space.

The control layers in MOLGEN are an adaptation of the control layers used in the SU-X and SU-P programs; the differences reflect MOLGEN's more elaborate concern about coordination of subproblems. MOLGEN's explicit management of the communication between nearly independent subproblems led to many more operators on the second level. Thus, the strategy level in SU-X had only to mediate between two analysis operators: goal-driven and event-driven analysis; MOLGEN's strategy operators mediate between eight design operators. Another source of complexity is MOLGEN's ability to save partial results of computations. Operators in SU-X merely succeed or fail without saving partial results; operators in MOLGEN can be suspended with partial results on under-constrained problems and can be restarted with instructions to try again, guess, or undo these steps.

4.3.2. *The Hayes-Roth planning model*

A cognitive model for an errand planning task has been developed by Barbara and Frederick Hayes-Roth [12] that is intended to model the mixture of goal-driven and data-driven behavior observed in human planners. The Hayes-Roths' model proposes pattern-directed invocation and resource allocation as the basic control concepts. Planning knowledge is factored into KSs that suggest decisions about how to approach a problem, what knowledge to use, and what actions to try.

MOLGEN research has paralleled the Hayes-Roths' work and there has been a considerable sharing of ideas. The Hayes-Roths' model evolved from the analysis of human problem-solving behavior in protocols taken from an errand-running task. It characterizes planning as follows:

> Our first assumption is that people plan ***opportunistically***..... [This] implies that the decisions they make can occur at non-adjacent points in the planning space..... A decision at a given level of abstraction specifying action to be taken at a given point in time may precede and influence decisions at either higher or lower levels of abstraction ... [or] at either earlier or later points in time.

This characterization is consistent with the behavior of MOLGEN using constraint posting.

Like SU-X and SU-P, the Hayes-Roths' model extends the HEARSAYII model by partitioning the blackboard into separate *planes*. In their model, an *executive* plane corresponds roughly to MOLGEN's *strategy* plane; a *meta-plan* plane corresponds to the design plane, and the three remaining planes correspond to the domain plane factored into intermediate states of planning in the errand running task. Resource allocation is governed by procedures in the executive plane. The separation of domain and control knowledge in the Hayes-Roths' model, however, is not rigorously enforced. For example, both domain-level facts and meta-level operations for setting goals appear on the *Knowledge-Base* plane.

Although the authors describe planning behavior as the result of the 'uncoordinated actions' of KSs acting opportunistically, the KSs in their computational model are far from uncoordinated. Specific KSs, such as *middle-management* and *referee*, perform critical control functions by determining focus, setting priorities, and establishing policies. While there is no explicit grouping of productions to make layered interpreters, some productions serve mainly as control functions. Unlike MOLGEN, the operators (production rules) in the model are organized as a monolithic set invoked by pattern invocation. Control is achieved by pattern-directed invocation from records placed in the blackboard planes. Some records represent control information, such as priorities and scheduling policies. The shift in attention from the problem to the meta-problem is controlled by the specification of flags in the planes; these flags are mentioned by the preconditions of the productions and tested by the interpreter. This practice invites the mixing of meta-level and first-level considerations in the rules.

The Hayes-Roths describe two planning paradigms: *hierarchical* and *opportunistic*. The *hierarchical* model is characterized as a systematic top-down exploration of possible plans. This differs from our use of the term *hierarchical planning*. For our purposes, the important feature of hierarchical planning is the use of planning islands, that is, a simplified planning model. While the direction of hierarchical planning is generally top-down, it need not be explored breath-first. Any planning model which makes use of abstractions would be termed hierarchical.

Opportunistic planning in the Hayes-Roth model is described as bi-directional (i.e., top-down and bottom-up) and heterarchical. This allows subplans to be developed independently, possibly at different levels of abstraction, for eventual incorporation into a final plan. The opportunistic idea is manifested in the constraint posting behavior of MOLGEN. Both approaches move the focus of planning activity between fruitful subproblems; both approaches work with constraints and nearly-independent subproblems. For example, the protocols in the Hayes-Roth model reveal time constraints: groceries perish, people get hungry at lunch time, the auto mechanic finishes with the car late in the day. In both cases, success depends on viewing plans as *structured objects* rather than action sequences.

The Hayes-Roths' cite examples of how the bottom-up level observations and decisions can trigger changes in higher-level activity in planning. There is an important distinction to be made here – that has been muddled somewhat in their discussion: the distinction between (1) bottom-up processing and (2) feedback of information to the meta-level. The behavior of a planning program without goals would seem very erratic; similarly a planning program with no event-driven component can have no feedback and can make no advantage of observation. In both cases, it is the *behavior of the planner* that is under scrutiny. This *problem-solving behavior* is controlled by the meta-level, so information relevant to changing problem-solving behavior must be utilized here.

In the Hayes-Roths' model, there seem to be no explicit planning operators for dealing with constraints. Constraints are simply mixed together with other records in the blackboard and somehow it all works. There are also no formal *hierarchical* planning operators and no differentiation of guessing or undoing, as in MOLGEN's heuristic cycle. These differences reflect the different interests of the researchers: the Hayes-Roths want to model human problem-solving as observed in their protocol studies, and are less interested in studying organizations of problem-solving knowledge.

5. Limitations and Further Research

This paper has argued for the use of a multi-layered organization as an antedote for the complexities of a monolithic control structure. Of course, this research has barely scratched the surface in considering the capabilities and organizations of planning systems. Several ideas and issues that go beyond the present work are listed below. This section argues that they expose an even greater need for factoring the knowledge used in a control structure.

(1) ***Guessing and Solution Density.*** When the solution space contains many solutions, almost any plan would probably work. In such situations, it would be reasonable to guess early, before performing all of the bookkeeping entailed in least-commitment approaches. This is related to the allocation of effort to *thinking* versus *thinking about thinking* in that cost/benefit estimates could be associated with the cost of computation and the risk of guessing incorrectly.

MOLGEN's conservative approach is based on a view of genetics experiment planning as a sparse solution space. Random experiments are unlikely to work. In general, the density of solutions varies with the particulars of the problem. It is possible to create additional layers of control to account for this. For example, a second layer of strategy could allow more sophisticated switching between the least-commitment and heuristic cycles. To speed up the planning process, it could recommend, for example, (1) that MOLGEN should play a sufficiently strong hunch instead of waiting until it knows that the problem is under-constrained or (2) that MOLGEN should debug (partially undo) some existing plan if its goals are sufficiently similar to those in the current problem.

(2) ***Incorporating new information.*** Experiments involving real-world feedback push the planning technology in several ways. For example, to do execution-monitoring of experiments, MOLGEN would need to inquire about

the success of laboratory steps. It would need to make judgments about what to observe, and what to do when steps violate expectations. Potentially, it would need to recognize when an unexpected event is a research opportunity and to decide (at a *meta*-level) whether to pursue it (see Feitelson and Stefik [7]). This would provide a setting to study the balance between planning *before* execution and planning *during* execution, that is, between goal-driven and event-driven planning. The ability to defer some of the planning until execution would reduce the burden of planning for all possible contingencies.

(3) ***Reasoning about theories.*** MOLGEN has no sense of the scientific method, which guides the creation of experiments to test hypotheses. A full-fledged experiment planner should be able to plan experiments in order to disambiguate and extend a theory. This enterprise would require a system to balance its efforts between proposing, modifying, and testing theories.

(4) ***Reasoning about scenarios.*** There is currently a research opportunity to combine the ideas of 'truth maintenance' and hierarchical reasoning about scenarios. Such a program might reason about a future that depends in part on its own commitments and activities. It would need to consider events caused by its own actions as well as those caused by other actors. The consideration of other actors considerably increases the complexities of planning.

(5) ***Reasoning about failures.*** A geneticist observing MOLGEN would distinguish between the following reasons for not finding a plan to an experiment: (1) conflicting constraints in the problem statement, (2) incompatible constraints introduced during problem solving, (3) incomplete knowledge of the objects and materials available in the laboratory, (4) incomplete knowledge about how to plan an experiment. MOLGEN does not currently distinguish between these possible causes for failure. Knowledge about sources of failure and about the completeness of its knowledge base could be used by MOLGEN to discriminate between these types of failure.

The need for partitioning control knowledge into layers is even more acute in resource-limited problem solvers which must balance these additional issues during computation.

6. Summary

Many of the actions, goals, and constraints that are important in planning can be best understood as belonging on *meta*-levels. That is, some of the decisions and goals refer to the process of problem solving, and not to the particulars of the problem at hand. This paper argues that the organization of a problem solver can be simplified by partitioning problem solving knowledge into layers. Monolithic organizations provide no distinction for meta-level considerations. By factoring out the meta-level considerations, we can reduce the apparent complexity of the interactions between first-level tasks.

MOLGEN is organized into laboratory, design, and strategy spaces. The laboratory space represents MOLGEN's knowledge about objects in the laboratory and operators that can be used to manipulate them to achieve laboratory goals. The design space provides an explicit repertoire of operators for hierarchical planning. The organizational idea behind this space is that hierarchical planning can be understood as operations on constraints. Tasks in the design space are created and executed by the strategy space. The organizational idea behind the strategy space is the distinction between *least-commitment* and *heuristic* modes of reasoning. MOLGEN's strategy space relies on the synergy between subproblems (via constraint propagation) to stay in the least commitment cycle as long as it can, and to resort to guessing only when it has to. The design operators *plan* by creating and scheduling laboratory steps; the strategy operators *meta-plan* by creating and scheduling design steps.

ACKNOWLEDGMENT

The research reported here was drawn from my thesis [24]. Special thanks to my advisor, Bruce Buchanan, and the other members of my reading committee: Edward Feigenbaum, Joshua Lederberg, Earl Sacerdoti, and Randall Davis. The idea of *meta*-planning was originally inspired by Davis' research on *meta*-rules in the TEIRESIAS program. Thanks also to the members of the MOLGEN project—Douglas Brutlag, Jerry Feitelson, Peter Friedland, and Lawrence Kedes for their help. At several key points in this research I benefitted from expansive discussions with Frederick and Barbara Hayes-Roth and Stanley Rosenschein. Thanks to Daniel Bobrow, Lewis Creary, and Michael Genesereth for helpful comments on earlier drafts of this paper. Research on MOLGEN was funded by the National Science Foundation grant NSF MCS 78-02777. General support for the planning research was provided by DARPA Contract MDA 903-77-C-0322. Computing support was provided by the SUMEX facility under Biotechnology Resource Grant RR-00785.

REFERENCES

1. Barstow, D., A knowledge-based system for automatic program construction, *Proceedings of the Fifth International Joint Conference on Artificial Intelligence* (1977) 382–388.
2. Bobrow, D.G. and Winograd, T., An overview of KRL, A knowledge representation language, *Cognitive Sci.* **1** (1) (1977) 3–46.
3. Davis, R., Applications of meta level knowledge to the construction, maintenance and use of large knowledge bases, Doctoral Dissertation, Computer Science Department, Stanford University (1976) (Also Stanford Computer Science Department Report No. STAN-CS-76-552.
4. Davis, R., Generalized procedure calling and content-directed invocation, *SIGPLAN Notices* **12** (8) (1977) 45–54.
5. de Kleer, J., Doyle, J., Steele, G.L. and Sussman, G.J., Explict control of reasoning, MIT AI Memo 427 (June 1977).
6. Erman, L.D. and Lesser, V.R., A multi-level organization for problem solving using many, diverse, cooperating sources of knowledge, *Proceedings of the Fourth International Joint Conference on Artificial Intelligence* (1975) 483–490.
7. Feitelson, J. and Stefik, M., A case study of the reasoning in a genetics experiment, Heuristic Programming Project Report 77-18 (working paper), Computer Science Department, Stanford University (April 1977).
8. Fikes, R.E. and Hendrix, G., A network-based knowledge representation and natural deduc-

tion system, *Proceedings of the Fifth International Joint Conference on Artificial Intelligence* (1977) 235–246.
9. Georgeff, M., A framework for control of production systems, *Proceedings of the Sixth International Joint Conference on Artificial Intelligence* (1979) 328–334.
10. Goldstein, I.P. and Miller, M.L., Structured planning and debugging, a linguistic theory of design, MIT AI Memo 387 (December 1976).
11. Hayes, P.J., In defence of logic, *Proceedings of the Fifth International Joint Conference on Artificial Intelligence* (1977) 559–565.
12. Hayes-Roth, B. and Hayes-Roth, F., A cognitive model of planning, *Cognitive Sci.* **3** (1979) 275–310.
13. Lenat, D.B., *AM*: An artificial intelligence approach to discovery in mathematics as heuristic search, Doctoral dissertation, Stanford University, Computer Science Department, 1976. (Also Stanford Computer Science Report AIM-286.)
14. Mostow, J. and Hayes-Roth, F., Operationalizing heuristics: Some AI methods for assisting AI programming, *Proceedings of the Sixth International Joint Conference on Artificial Intelligence* (1979) 601–609.
15. Newell, A., Some problems of basic organization in problem-solving programs, in: Yovits, M.C., Jacobi, G.T. and Goldstein, G.D., Eds., *Proceedings of the Second Conference on Self-Organizing Systems* (Spartan Books, Chicago, Il., 1962).
16. Nii, H.P. and Feigenbaum, E.A., Rule-based understanding of signals, in: Waterman, D.A. and Hayes-Roth, F., Eds., *Pattern-Directed Inference Systems* (Academic Press, New York, 1978).
17. Polya, G., *Mathematical Discovery. 2* (John Wiley and Sons, New York, 1965).
18. Polya, G., *How to solve it* (Doubleday Anchor Books, New York, originally published 1945).
19. Sacerdoti, E.D., Planning in a hierarchy of abstraction spaces, *Artificial Intelligence* **5** (2) (1974) 115–135.
20. Sacerdoti, E.D., *A structure for plans and behavior* (American Elsevier Publishing Company, New York, 1977 (originally published, 1975)).
21. Shortliffe, E.H., *MYCIN*: *Computer-based medical consultations* (American Elsevier, New York, 1976).
22. Simon, H.A., Experiment with a heuristic compiler, *J. Assoc. Comput. Mach.*, **10** (4) (1963) 493–506.
23. Stefik, M.J., Planning with constraints, *Artificial Intelligence* **16**(2) (1981) 111–140 [this issue].
24. Stefik, M.J., *Planning with constraints*, Doctoral Dissertation, Computer Science Department, Stanford University (January 1980). (Also Stanford Computer Science Department Report No. STAN-CS-80-784.)
25. Stefik, M.J., An examination of a frame-structured representation system, *Proceedings of the Sixth International Joint Conference on Artificial Intelligence* (1979) 845–852.
26. Sussman, G.J., *A computer model of skill acquisition* (American Elsevier Publishing Company, New York, 1975 (originally published 1973)).

Received June 1980; revised version received September 1980