Reprinted from

Volume 14 Number 2 September 1980

Artificial Intelligence

AN INTERNATIONAL JOURNAL

Planning with Constraints (MOLGEN: Part 1)

Mark Stefik*

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

Recommended by Daniel G. Bobrow

Planning with Constraints (MOLGEN: Part 1)

Mark Stefik*

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

Recommended by Daniel G. Bobrow

ABSTRACT

Hierarchical planners distinguish between important considerations and details. A hierarchical planner creates descriptions of abstract states and divides its planning task into subproblems for refining the abstract states. The abstract states enable it to focus on important considerations, thereby avoiding the burden of trying to deal with everything at once. In most practical planning problems, however, the subproblems interact. Without the ability to handle these interactions, hierarchical planners can deal effectively only with idealized cases where subproblems are independent and can be solved separately.

This paper presents an approach to hierarchical planning, termed constraint posting, that uses constraints to represent the interactions between subproblems. Constraints are dynamically formulated and propagated during hierarchical planning, and used to coordinate the solutions of nearly independent subproblems. This is illustrated with a computer program, called MOLGEN, that plans gene-cloning experiments in molecular genetics.

1. Introduction

Divide each problem that you examine into as many parts as you can and as you need to solve them more easily. Descartes, *OEuvres*, vol. VI, p. 18; "Discours de la Methods"

This rule of Descartes is of little use as long as the art of dividing . . . remains unexplained. . . . By dividing his problem into unsuitable parts, the inexperienced problem-solver may increase his difficulty. Leibniz, *Philosophische Schriften*, edited by Gerhart, vol. IV, p. 331 from Polya [12]

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

Artificial Intelligence **16** (1981) 111–140

0004-3702/81/0000–0000/$02.50

Subproblems interact. This observation is central to problem solving, particularly planning and design. When interactions can be anticipated, they can guide the division of labor. When they are discovered late, the required changes can be difficult and expensive to incorporate. The difficulty of managing interactions is compounded by problem size and complexity. In large design projects, unforeseen interactions often consume a substantial share of the work of project managers [2].

This paper is concerned with ways to cope with and exploit interactions in design. Section 2 presents the *constraint posting* approach for managing interactions in design. Constraint posting has been implemented in a computer program (named MOLGEN) that has planned a few experiments in molecular genetics. In Section 3, the design of an experiment is used to illustrate the constraint posting ideas. In Section 4, the effectiveness of constraint posting on the sample problem is examined. The remaining sections trace the intellectual connections to other work on problem solving and propose suggestions for further research.

This is the first of two papers about my thesis research on MOLGEN. Both papers are concerned with the use and organization of knowledge to make planning effective. This paper discusses the use of constraints to organize knowledge about subproblems in hierarchical planning. A companion paper [21] discusses the use of levels to organize control knowledge. It also develops a rationale for deciding when a planner should use heuristic reasoning.

The research was carried out as part of the MOLGEN project at Stanford. A long term goal of this project is to build a knowledge-based program to assist geneticists in planning laboratory experiments. Towards that goal, two prototype planning systems have been constructed and used as vehicles for testing ideas about planning [7, 20].

2. The Constraint Posting Approach to Design

The constraint posting approach depends on the view of systems as aggregates of *loosely coupled* subsystems. It models the design of such systems in terms of operations on constraints.

2.1. Nearly independent subproblems

In Sciences of the Artificial [18], Simon discussed the study and design of complex systems. He observed that when we study a complex system, whether it is natural or man-made, we often divide it into subsystems that can be studied separately without constant attention to their interactions. For example, in studying an automobile, we delineate subsystems such as the electrical system, fuel system, engine, and the brake system; in an animal, we delineate the nervous system, circulatory system, and the digestive system.

Similarly, when we design complex systems, we tend to first map out the design in terms of subsystems. Designers have advocated this top-down ap-

proach for the design of such diverse things as computer programs, machines, and buildings. This approach is so familiar and universally practiced that we seldom consider the motivations for it. Some of these motivations are: (1) the apparent complexity of the design problem is often reduced by partitioning it into subproblems, and (2) the partitioning can be done before the specifications of the subsystems are worked out because most of the details are irrelevant to the global design, and (3) the labor and expertise of designing the detailed subsystems can often be divided among several specialists.

A key step in design is to minimize the interactions between separate subsystems. Simon coined the phrase *nearly decomposable system* to characterize the way that a complex system can be built from loosely coupled subsystems. Winograd [26] addressed the same point in representational terms:

> "... we must worry about finding the right decomposition to reduce the apparent complexity, but we must also remember that interactions among subsystems are weak but *not negligible*. In representational terms, this forces us to have representations which facilitate the weak interactions."

The view of a system in terms of *nearly decomposable* subsystems corresponds to the view of the design process in terms of *nearly independent* subproblems. In hierarchical planning a solution is first sketched out in terms of abstract steps, which are refined into specific plan steps during the planning process as shown in Fig. 1. The abstract steps and their subsequent refinements

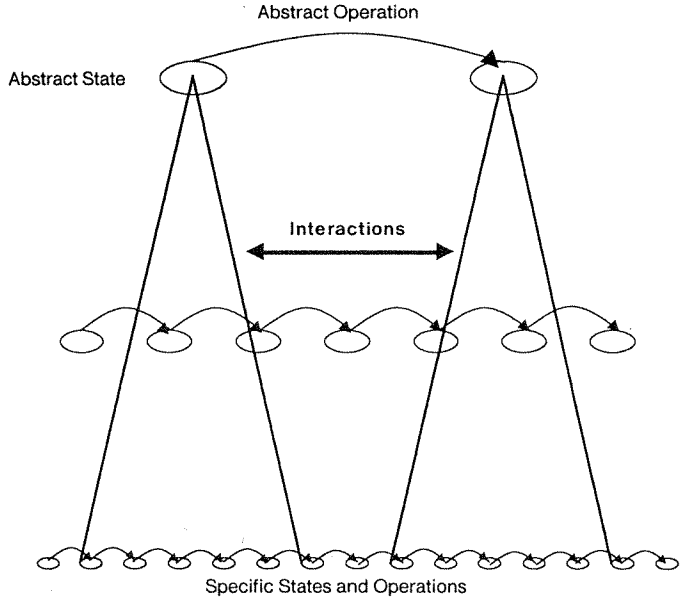

FIG. 1. Nearly independent subproblems. The upper level represents an abstract (or less detailed) plan. The arcs represent abstract operations and the circles represent abstract states. In hierarchical planning, such abstract plans are *refined* into more specific plans, as suggested by the *refinement-cones*. Each cone can be viewed as a refinement-subproblem. Interactions between the refinement subproblems need to be managed during planning.

correspond to *nearly decomposable* subsystems; they are connected together in a plan and the output of each plan step must match the required input of the following plan step. These input/output relationships make a plan work as a whole. The design subproblems, that is, the refinements of the abstract plan steps into specific steps, are only nearly independent. They are not completely independent because their solutions must interface correctly. When the abstractions partition the plan into nearly independent subproblems, interactions do not dominate the planning process. However, efficient planning usually requires that the weak interactions be taken into account during the planning process. The key idea in constraint posting is to use constraints to represent the interactions between subproblems.

2.2. The meanings of constraints

For the purposes of planning, constraints can have several different interpretations. A constraint expresses a relationship among plan variables. Constraints are represented as predicates. For example, consider the following:

(Lambda (Gene DNA-structure) (Contains Gene DNA-Structure)).

If the **Lambda** variables (***Gene*** and ***DNA-Structure***) are bound to the constants *Tc^r-gene* and *DNA-13*, then the predicate **Contains** may be evaluated to determine whether the *Tc^r-gene* is contained in *DNA-13*. The full representation of constraints implemented in MOLGEN allows the plan variables to have different names than the **Lambda** variables and contains other information related to the constraint. Constraints may involve more than two variables; they may also apply over sets of variables.

The first interpretation of constraints is as ***elimination rules*** from the perspective of object selection. A constraint in MOLGEN is associated with a set of plan variables that usually refer to laboratory objects. When the variables are not yet bound, the constraint may be interpreted as a condition to be satisfied. It *constrains* the set of allowable bindings; potential selections are eliminated if they do not satisfy the constraint.

A second interpretation of constraints is as partial descriptions and ***commitments*** from the perspective of plan refinement. During the process of planning, there are many opportunities for deciding which part of a plan to make more specific. A least-commitment approach is to defer decisions as long as possible. A constraint is essentially a partial description of an object; a selection is a full description. By formulating constraints about objects, MOLGEN is able to make commitments about partial descriptions of the objects without making specific selections.

A third interpretation of constraints is as a ***communication medium*** for expressing interactions between subproblems. A constraint represents an in-

tended relationship between (possibly uninstantiated) plan variables. In MOLGEN, these variables are often shared among various steps in the plan; they represent objects having a rich set of relationships. For example, they may represent laboratory objects which are to be constructed out of other laboratory objects. If a problem solver has a calculus of expressions, it can take constraints relating to variables in one part of the plan and infer new constraints relating to variables in another part of the plan, even though the variables themselves are still unbound. This amounts to the propagation of constraints in a plan, and we will see that this enables a problem solver to coordinate the solution of subproblems.

2.3. Operations on constraints

The constraint posting approach is essentially a marriage of ideas from hierarchical planning and constraint satisfaction. It distinguishes three operations on constraints:

(1) constraint formulation,
(2) constraint propagation,
(3) constraint satisfaction.

All of these operations could be broadly characterized as inferences in problem solving. A major point of this paper is that it is useful to consider these operations in terms of the substantially different roles they play in the problem solving process.

Constraint formulation is the adding of new constraints as commitments in the design process. A planner can proceed hierarchically by formulating constraints of increasing detail as planning progresses. Thus, a problem-solver that can introduce new constraints need not work with all of the details at once. This idea is consistent with the common experience of working on problems that are imprecisely formulated, but which become more tightly specified during the solution process. In contrast, the traditional constraint satisfaction approach works with a fixed number of constraints that are all known at the beginning.

Constraint propagation is the creation of new constraints from old constraints in a plan. In MOLGEN, this operation performs communication between refinement subproblems during planning. Refinement subproblems are usually under-constrained when viewed in isolation because there are many choices for refining abstractions in genetics plans. When constraints are propagated, they bring together the requirements from separate parts of the problem. Constraint propagation makes possible a *least-commitment* strategy of deferring decisions for as long as possible. The problem solver works to keep its options open, and reasons by elimination when constraints from other subproblems become known.

Constraint satisfaction is the operation of finding values for variables so that a set of constraints on the variables is satisfied. Constraint satisfaction can take different forms. For example, linear programming is a constraint satisfaction method that assigns numeric values of variables satisfying linear inequalities. Constraints in MOLGEN describe requirements about laboratory objects needed in the plans; constraint satisfaction implements a 'buy or build' decision process. MOLGEN first tries to satisfy the constraints by selecting an available object 'off the shelf'. Computationally, this involves searching MOLGEN's knowledge base for a record of an object that is marked as available and which satisfies the constraints. For example, MOLGEN might search for an organism carrying a particular gene on its chromosome. If the search process fails, MOLGEN marks the constraint as unsatisfied and may later propose building an object to satisfy the constraint. In such a case, the construction of the object becomes a subgoal in the plan. When the constraints and variables come from different subproblems, constraint satisfaction plays a coordinating role by pooling the constraints and intersecting their solutions.

3. An Example of Planning with Constraints

This section illustrates the constraint posting idea with an experiment that was planned by MOLGEN. MOLGEN has been used to plan experiments in a class of synthesis problems known as gene cloning experiments.[1] The goal in gene cloning experiments is to use bacteria as a biological system for synthesizing a desired protein product. The experiments involve splicing a gene coding for the protein into bacteria, so that the bacteria will manufacture it. The laboratory plan illustrated in this example is a solution to a gene cloning problem called the *rat-insulin problem* which was reported by Ullrich et al. in 1977 [24].

Before starting the example, it should be mentioned that the main use of MOLGEN is as a vehicle for testing approaches to reasoning about design. It would be misleading to suggest that MOLGEN is currently a useful computational aid for geneticists. MOLGEN's knowledge base is too narrow and there are serious difficulties in upgrading MOLGEN to a routinely useful system (see Section 6.2). The rat-insulin experiment is one of a few gene cloning experiments that have been planned by MOLGEN. Even in this narrow class of experiments, there are laboratory techniques (e.g., involving protein transcription) that are currently beyond MOLGEN's ken, and experiments which MOLGEN fails to solve satisfactorily. The trace of MOLGEN's reasoning in this experiment took over 30 pages of computer print out (without annotations). The interested reader is referred to Stefik [20] for a complete trace of the planning of this experiment.

[1] A very readable review of these experiments is available in Gilbert and Villa-Komaroff [8].

3.1. First steps

The first part of MOLGEN's trace is similar to the behavior of previous problem-solving programs like GPS (Newell [11]). MOLGEN compares goals, finds *differences*, and chooses operators to reduce the differences. Like several recent planning programs (see Section 5.1), MOLGEN plans hierarchically. It uses a simplified model of genetics to set up an abstract plan, and then refines that to a plan of specific laboratory steps. This section shows how MOLGEN sets up an abstract plan for achieving a synthesis goal. The constraint posting ideas do not appear until Section 3.2.

3.1.1. Abstract objects and operators

MOLGEN views synthesis experiments as compositions of four abstract operators called the '***MARS***' operators. (The word 'MARS' is an acronym formed from the first letters of their names.)

(1) ***Merge***[2] – to put separate parts together to make a whole. *Examples*: connecting DNA structures together (***Ligate***), adding an extrachromosomal vector to an organism (***Transform***).

(2) ***Amplify*** – to increase the amount of something. *Examples*: incubating bacteria in ideal growth conditions (***Incubate***), introducing something from stock (***Get-Off-Shelf***).

(3) ***React*** – altering the properties of something. *Examples*: cleaving DNA with a restriction enzyme (***Cleave***), using alkaline phosphatase to change terminal phosphates to hydroxyl groups in DNA molecules (***Add-Hydroxyl***).

(4) ***Sort*** – to separate a whole into parts according to their properties.[3] *Examples*: separating polynucleotides according to mass and topology (***Electrophoresis***), killing organisms not resistant to a given antibiotic (***Screen***).

MOLGEN's knowledge is represented in a hierarchical knowledge base[4] divided into objects and operators. The knowledge base describes the laboratory entities at various levels of abstraction. The most abstract laboratory operator is called ***Lab-Operator***; the next level contains the four ***MARS*** operators, and the next level contains thirteen specific laboratory operators. The most abstract laboratory object is ***Lab-Object***; the next level contains ***Antibiotic***, ***Culture***, ***DNA-Struc***, ***Enzyme***, ***Organism***, and ***Sample***. This hierarchy is six levels deep and contains descriptions of 74 kinds of objects.

[2] This paper uses the convention that operators are indicated by underlining; objects and steps are indicated by italics. References to units are indicated by capitalizing their first letter; references to slots are indicated by italics with the first letter not capitalized.

[3] This is the usual meaning of *sort* and not the computer science meaning, which requires a linear ordering. Some of the laboratory operators classified as ***Sort*** operators do provide a linear ordering (such as ***Electrophoresis***); others do not (such as ***Screen***). An alternative name for this category of operators would be *separative techniques*.

[4] See Stefik [19] for a discussion of the representation language.

3.1.2. *Finding a difference*

The synthetic goal for the rat-insulin problem is shown in Fig. 2. This goal is a *partial* description of the desired state that leaves some of the details to be filled in by the planner. It describes a culture of an unspecified bacterium, having an unspecified vector that carries gene for rat-insulin. A *vector* is a self-replicating DNA molecule that can be used to transmit genes into bacteria. Bacteriophages and plasmids are typically used as vectors. Determining what bacterium and vector to use is part of the problem. MOLGEN interprets this goal as a request to design a laboratory plan to get the described bacterium.

An important part of creating a laboratory plan is the selection of (possibly abstract) operators. Like several earlier problem solvers, MOLGEN keys its selection of operators by *differences*. MOLGEN's first steps in doing this are shown in Fig. 3. The key item in the figure is the data structure *Difference-1*.

```
Prototpe is SYNTHESIS-PROBLEM

GOAL: [CULTURE-1 with
        ORGANISMS: [BACTERIUM-1 with
             EXOSOMES: [VECTOR-1 with
                  GENES: [RAT-INSULIN]]]]

TO-PLAN:        META-PLAN-INTERPRETER

PLAN-NUMBER:  1

DESCR:          This synthesis problem is to clone the
                 gene for rat-insulin. This problem was
                 discussed by Ullrich et.al. in Science,
                 Vol. 196, pp. 1313-1319.
```

FIG. 2. Goal of the rat-insulin problem. The goal slot contains a symbolic description of the synthetic goal. *Bacterium-1* and *Vector-1* are variables, which will become instantiated during planning.

```
->STRATEGY-STEP-1 (FOCUS)
->PLAN-STEP-1 (FIND-UNUSUAL-FEATURES) Input: ($LAB-GOAL-1)
<-PLAN-STEP-1 DONE SUCCESS Output: (DIFFERENCE-1)

     ----- DIFFERENCE-1 -----

MISMATCH:          EXOSOMES
OBJECT:            BACTERIUM-1
COMPARED-TO:       BACTERIUM
LEVEL:             2
DEFN-ROLE:         PART-OF
TYPE:              MORE-SPECIFIC
EXCEPTIONS:        (VECTOR-1 (*P VECTOR))
REPRESENTATION:    LIST
MAKER:             PLAN-STEP-1
DESCR:             Created by FIND-UNUSUAL-FEATURES
```

FIG. 3. Finding unusual features. The ***Find-Unusual-Features*** design operator compares the objects in *Lab-Goal-1* against their prototypes, and outputs a description of their unique features as *Difference-1*. (It quits after finding the highest level difference.)

Difference-1 summarizes the unusual features of the bacterium described in the goal (*Bacterium-1*) that were found when it was compared to the prototypical bacterium in the genetics knowledge base. The interpretation of *Difference-1* is that *Bacterium-1* was unusual in that it had a specific vector (*Vector-1*) as an exosome.

The rest of Fig. 3 exposes some aspects of MOLGEN's planning machinery that are the topic of the companion paper. For now it is enough to know that in addition to laboratory operators which operate on laboratory objects, MOLGEN has operators which operate on *plans*. These operators are further classified as *planning* (or design) operators, which operate on plans, and *meta-planning* (or strategy) operators, which control the design steps. These operators are described in detail in the companion paper. The design operators represent the constraint posting approach in terms of operators for refining objects and operators, creating and propagating constraints, simulating laboratory steps, and finding differences. The design operator in Fig. 3 is ***Find-Unusual-Features***.

MOLGEN's progress in planning takes place in a series of *steps*, which are executed. In Fig. 3, the start of execution of a step is indicated in the trace by a line beginning with the symbols '→'. The name of the operator follows in parentheses (e.g., ***Focus*** in the first line). The names of the objects input to the operator in the step, if any, follow. The termination of a step is indicated by a line beginning with the symbols '←' and followed by the step name, an indication of the status of the step at termination, an indication of the reason for the status, and the names of the objects output from the step. Finally, a representation of each of the objects output from the step is printed.

3.1.3. Making an abstract plan

Starting with *Difference-1,* MOLGEN goes on to develop an abstract plan. It

```
-> PLAN-STEP-2 (PROPOSE-OPERATORS) Input: (DIFFERENCE-1)
<- PLAN-STEP-2 DONE SUCCESS Output: (LAB-STEP-1)

     ----- LAB-STEP-1 -----

OPERATOR:     MERGE
INPUT:
OUTPUT:
FWD-GOAL:     $LAB-GOAL-1
STATUS:       PROPOSED
REASON:       APPLICABLE
NEXTSTEPS:
PREVSTEPS:
DESCR:        Created to reduce: (DIFFERENCE-1)
MAKER:        PLAN-STEP-2
```

FIG. 4. Proposing the first laboratory step. *Lab-Step-1* is a partially instantiated laboratory step in the abstract plan. It specifies that the operator is ***Merge***. No previous or following steps in the plan are known yet. The *input* and *output* slots will be filled with descriptions of the objects that are input and output to the laboratory step. The *fwd-goal* slot refers to a description of the intended output of the step.

begins by partially instantiating a laboratory step (*Lab-Step-1*) as shown in Fig. 4. This step is created by the design operator, ***Propose-Operators***. It specifies the abstract operator, ***Merge***, but not what objects will be merged or any previous or next steps in the plan. The next few design operations fill in the backwards goals in *Lab-Step-1*, and propose additional steps. They are omitted here for brevity. When they have been executed, MOLGEN has the two-step abstract laboratory plan in Fig. 5. The planning in this part of the trace has been quite straightforward; it is about to get more interesting.

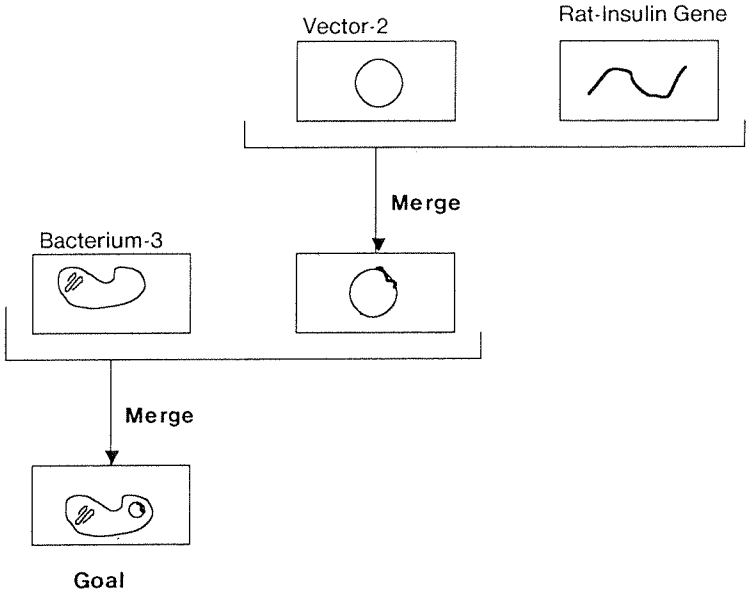

FIG. 5. MOLGEN's abstract plan. This figure characterizes the experiment in terms of two abstract ***Merge*** operations.

3.2. Introducing a constraint

In the next part of the trace, MOLGEN refines its abstract plan. Fig. 6 shows the important developments:

(1) The specific laboratory operator, ***Transform***, has replaced the abstract operator, ***Merge***.

(2) The *input* slot of *Lab-Step-1* has been filled with a description of the objects being combined.

(3) *Constraint-1* has been introduced to the plan.

The formulation of *Constraint-1* illustrates an important aspect of MOLGEN's decision-making. When MOLGEN decided to refine ***Merge*** to the ***Transform*** operator, the bacterium and vector in *Lab-Step-1* were still unspecified. For ***Transform*** to work properly, it is necessary that the bacterium and vector be biologically compatible. One approach would be for MOLGEN to immediately select a bacterium and vector for compatibility, and to bind the

```
->PLAN-STEP-6 (REFINE-OPERATOR) Input: (LAB-STEP-1)
<--PLAN-STEP-6 DONE SUCCESS
 Output: (LAB-STEP-1 REFINEMENT-1 CONSTRAINT-1)

          -----LAB-STEP-1-----

OPERATOR:       TRANSFORM
INPUT:          [SAMPLE-1 with
                 STRUCS: [VECTOR-1 with
                 GENES: [RAT-INSULIN]],
                 CULTURE-1 with
                 ORGANISMS: [BACTERIUM-3]]
OUTPUT:
FWD-GOAL:       $LAB-GOAL-1
STATUS:         REFINED
REASON:         NEW-OPERATOR
NEXTSTEPS:
PREVSTEPS:      [LAB-STEP-2]
DESCR:          Created to reduce: (DIFFERENCE-1)
MAKER:          PLAN-STEP-2

     -----REFINEMENT-1-----

ABSTRACT:       MERGE
SPECIFIC:       TRANSFORM
CONSTRAINTS:    [CONSTRAINT-1]
GOODLIST:
LAB-STEP:       LAB-STEP-1
MAKER:          PLAN-STEP-6
DESCR:

     -----CONSTRAINT-1-----

TYPE:           MANDATORY
ARGS:           [BACTERIUM-3,
                 VECTOR-1]
PREDICATE:      (LAMBDA (BACTERIUM VECTOR)
                 COMPATIBLE BACTERIUM VECTOR))
STATUS:         PROPOSED
REASON:         FORMULATED
MAKER:          PLAN-STEP-6
LAB-STEP:       LAB-STEP-1
DESCR:          From refinement of MERGE to TRANSFORM in LAB-
                 STEP-1
```

FIG. 6. Refinement of *Lab-Step-1* introduces the constraint that the bacterium (*Bacterium-3* from Culture-1) and the vector (*Vector-1* from Sample-1) must be compatible. (The syntax of *Constraint-1* has been simplified slightly by eliminating the expression which tests whether enough information is available to evaluate the constraint.)

plan variables (*Bacterium-3* and *Vector-1*) accordingly. The trouble with this approach is that it would preclude the consideration of other constraints that might be uncovered later in the planning process. Since many combinations of values are possible for these variables, MOLGEN decides to keep its options open. Instead of choosing values for the variables, it formulates a constraint on their values that can be taken into account in a later constraint satisfaction step. *Constraint-1* states that the bacterium and vector input to the ***Transform*** step must be compatible. By posting the constraint, MOLGEN makes the requirement explicit so that it can be combined with other constraints. This

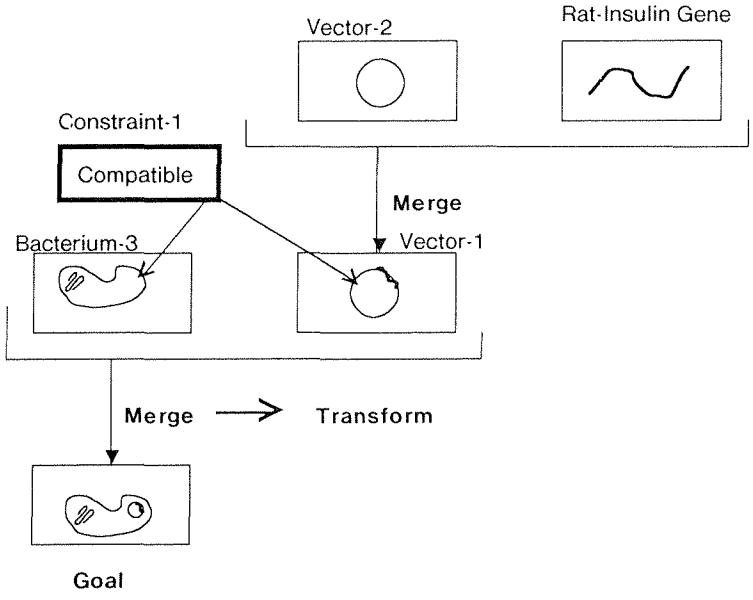

FIG. 7. The plan after introducing the compatibility constraint.

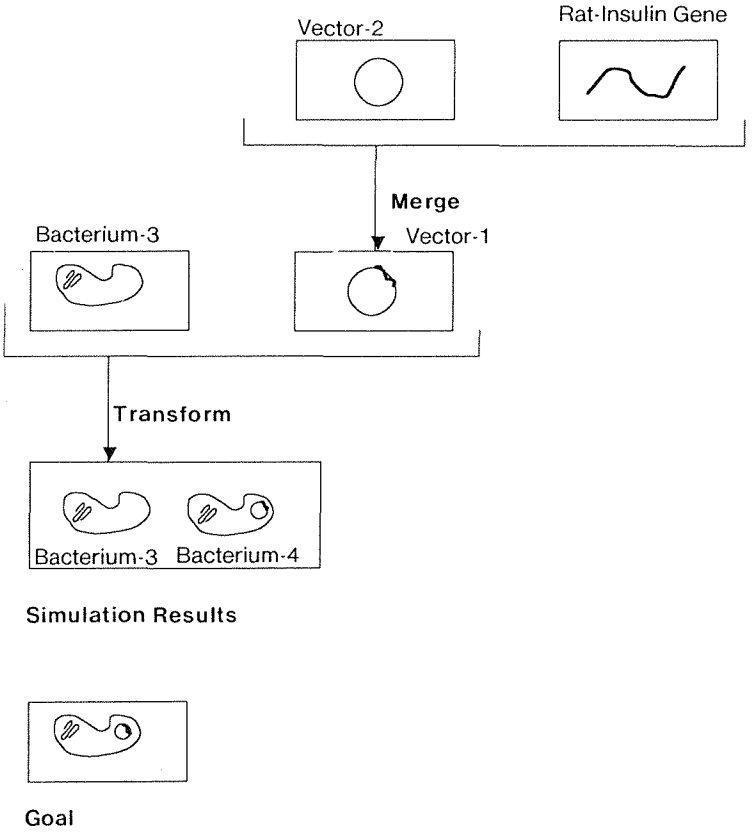

FIG. 8. Simulation of the ***Transform*** step predicts that some of the bacteria will not get the vector.

deferring of decisions until necessary is part of a *least-commitment* approach to problem solving. Fig. 7 illustrates the plan at this stage pictorially.

3.3. Predicting results of a lab step

One of the important ideas for using symbolic representations is symbolic execution. For each of its laboratory operators, MOLGEN has a simulation model which it can use to predict the results of a laboratory step. The simulation of ***Transform*** in *Lab-Step-1* is illustrated in Fig. 8. The simulation takes account of the fact that transformation in the laboratory never works to completion. Transformation is essentially the absorption of vectors across cell membranes. In practice, some of the bacteria inevitably end up without vectors. Thus, the output of *Lab-Step-1* includes *Bacterium-4*, which has the vector, and *Bacterium-3*, which does not.

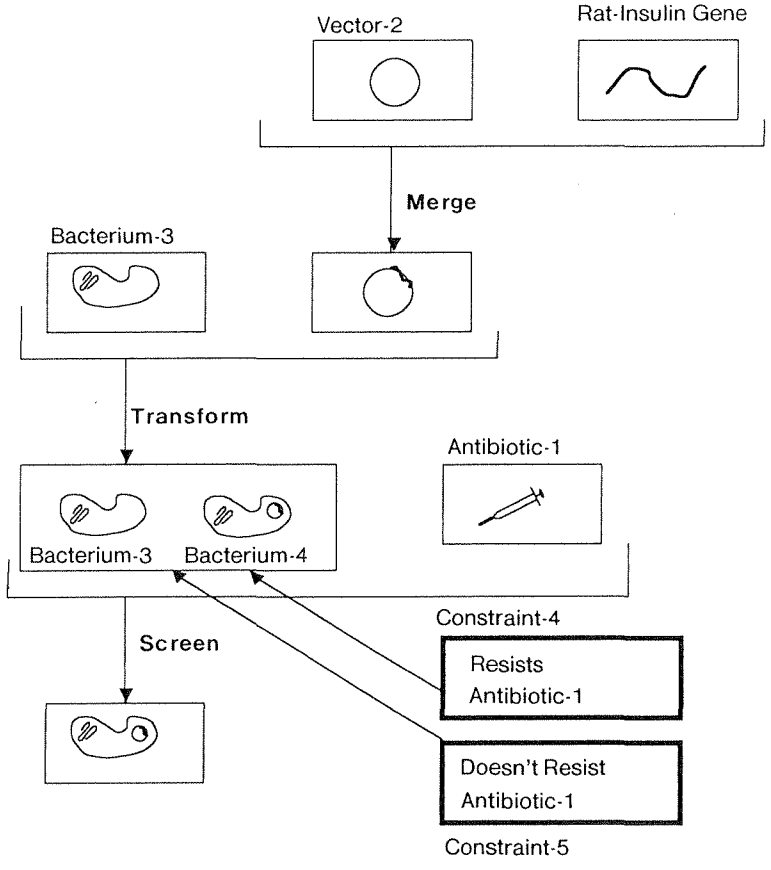

FIG. 9. Introducing an antibiotic. To get rid of the unwanted bacteria, a ***Screen*** step is proposed, which utilizes an antibiotic. At this point in the plan, MOLGEN has not yet determined the type of bacteria. It introduces some new constraints that tie the selection of the bacteria to the selection of the antibiotic.

3.4. Introducing a variable

When MOLGEN compares the simulation output of *Lab-Step-1* with the goals, it discovers the extra bacteria resulting from the incompleteness of the ***Transform*** step. The comparison process yields a difference, which is used to key the selection of an abstract laboratory operator (***Sort***) to remove the bacterium. After several planning steps similar to what we have already seen, MOLGEN refines the ***Sort*** operator to the ***Screen*** operator, which kills bacteria with an antibiotic as shown in Fig. 9. This particular refinement introduces some of the most interesting constraints in the plan. When ***Refine-operator*** looks for a specialized kind of ***Sort*** to remove unwanted bacteria, it finds only the ***Screen*** laboratory operator, which kills the bacteria with an antibiotic. This means that an antibiotic must be introduced into the plan. The antibiotic should kill the extra bacteria, those without the vector (*Bacterium-3*) but not harm the others (*Bacterium-4*). Although MOLGEN could arbitrarily choose an antibiotic at this point, it prudently decides to defer the decision in case other factors are found that bear on the selection. In order to refer to an antibiotic without selecting a particular one from the knowledge base, MOLGEN introduces the variable, *Antibiotic-1*, and posts a pair of constraints to indicate which of the bacteria are supposed to be resistant to it.

3.5. Propagating constraints

Subproblems in plans interact. A simple form of interaction occurs when variables are shared between subproblems. In this case, constraints from the subproblems are combined when a value for the variable is determined. A more complicated way to account for interactions is to propagate symbolic constraints between subproblems. In such cases, new constraint expressions are inferred from other constraint expressions on possibly distinct variables.

The virtue of constraint propagation can be seen by viewing planning as a *generate-and-test* process. Constraints are the rules for pruning in the *test* part of the process. The key to efficiency in finding solutions is to apply constraints as early as possible, so that branches corresponding to possible plans can be eliminated before much computational effort is expended. When constraints can be propagated across partial solutions to subproblems, they enable a planner to anticipate interactions and effectively prune some possible choices without generating them.

An example of constraint propagation is shown in Fig. 10. The figure shows the plan at a much later point than where we left off in the previous section. In this propagation, the constraints on which bacteria resist the antibiotic are converted, through a series of transformations, into a constraint on the selection of the vector at the top of the figure. The propagation process approximates the following genetics argument:

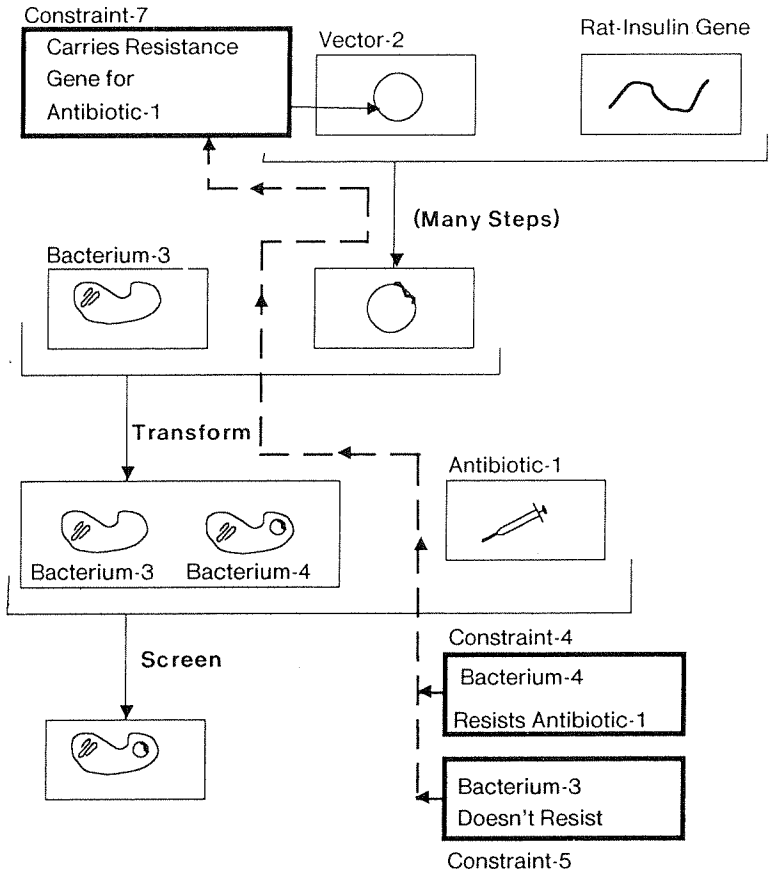

FIG. 10. Propagating Constraints. The constraint propagation process creates new constraints in the plan from existing constraints. This process regresses constraints through the plan in time, one step at a time.

By *Constraint-4*, *Bacterium-4* is resistant to *Antibiotic-1*. By *Constraint-5*, *Bacterium-3* is not resistant. Resistance to an antibiotic is conferred by a resistance gene, which can be either on the bacterial chromosome, or on some extrachromosomal element. The two bacteria are of the same type, that is, they have the same chromosome. This means that the resistance can not be conferred by a resistance gene on the bacterial chromosome. Therefore the resistance gene must be conferred by an extra-chromosomal element. *Vector-1* is the only exosome in *Bacterium-4* that is not in *Bacterium-3*. Therefore, *Vector-1* must carry a resistance gene for *Antibiotic-1*. *Vector-1* was constructed from *Vector-2* and the rat-insulin gene. Since the rat-insulin gene carries no resistance genes for any antibiotic, a resistance gene must be carried by *Vector-2*. (This is the predicate of *Constraint-7*.)

3.6. Satisfying constraints

The third operation on constraints is constraint satisfaction. Constraints in MOLGEN describe restrictions on laboratory object in plans; satisfaction is simply a search of the knowledge base for records of available objects that satisfy the constraints on the plan variables. Fig. 11 illustrates an example of constraint satisfaction. *Constraint-1* is the constraint we saw earlier requiring compatibility for values for the variables *Bacterium-3* and *Vector-1*. **Refine-Object** is the name of the design operator for constraint satisfaction on laboratory objects. It searches the knowledge base for possible bindings and records them in the data structure *Tuple-1*. (These should properly be termed 'tuple-sets', since they represent sets of solutions expressed as n-tuples.) 'Tuple' data structures list the possible solutions to constraints. The interpretation of *Tuple-1* is that *Bacterium-3* can be bound only to *E. coli*, and that *Vector-1* can be bound to any of four plasmids (e.g., *Col. E1*) listed in the figure.

As discussed further in Section 6.3, MOLGEN uses distinct variable names to refer to objects at different times in the plan, that is, in distinct states. Such variables are linked by a *same-type* relationship in the representation language; solutions for one variable imply solutions for others. For example, *Bacterium-3* is known to be the same type of bacterium as *Bacterium-4*. When MOLGEN anchored *Bacterium-3*, it propagated the information to *Bacterium-1* and *Bacterium-4* as well.

As MOLGEN picks constraints to satisfy, it sometimes discovers that the objects are mentioned in several constraints. In such cases, the tuples are combined and the solutions are intersected. This is shown in Fig. 12 where the constraint satisfaction step integrates the results of satisfying *Constraint-7* with

```
-> PLAN-STEP-15 (REFINE-OBJECT) Input: (CONSTRAINT-1)
   (Anchoring BACTERIUM-3 to E.COLI)
   (Updating slots: GRAMSTAIN MORPHOLOGY in BACTERIUM-1)
   (Updating slots: GRAMSTAIN MORPHOLOGY in BACTERIUM-4)
<- PLAN-STEP-15 DONE SUCCESS Output: (TUPLE-1)

                -----TUPLE-1-----

   CONSTRAINTS:      [CONSTRAINT-1]
   VARIABLES:        [BACTERIUM-3,
                     VECTOR-1]
   PRIMARIES:        [BACTERIUM-3,
                     VECTOR-1]
   COMPATIBLES:      (((BACTERIUM-3   E.COLI)
                     VECTOR-1 COL.E1
                     PSC101 PBR322 PMB9)))
   MAKER:            PLAN-STEP-15
   DESCR:
```

FIG. 11. Satisfying a constraint. Constraint satisfaction involves a 'buy or build' decision. Here MOLGEN searches the knowledge base for combinations of bacteria and vectors that satisfy the compatibility constraint.

```
-> PLAN-STEP-33 (REFINE-OBJECT) Input: (CONSTRAINT-6)
<- PLAN-STEP-33 CANCELLED REPLACED Output: (NONE)
-> PLAN-STEP-35 (REFINE-OBJECT) Input: (CONSTRAINT-7)
<- PLAN-STEP-35 DONE SUCCESS Output: (TUPLE-2)

        -----TUPLE-2-----

CONSTRAINTS:    [CONSTRAINT-3, CONSTRAINT-2, CONSTRAINT-1,
                 CONSTRAINT-7]
VARIABLES:      [RESTRICTION-ENZYME-1,
                 VECTOR-2,
                 BACTERIUM-3,
                 VECTOR-1,
                 ANTIBIOTIC-1]
PRIMARIES:      [RESTRICTION-ENZYME-1,
                 VECTOR-2,
                 BACTERIUM-3,
                 ANTIBIOTIC-1]
COMPATIBLES:    (((RESTRICTION-ENZYME-1 RESTRICTION-
                   ENZYME)
                  (BACTERIUM-3 E.COLI)
                  (ANTIBIOTIC-1 COLICIN-E1)
                  (VECTOR-2 COL.E1))
                 ((RESTRICTION-ENZYME-1 RESTRICTION
                   ENZYME)
                  (BACTERIUM-3 E.COLI)
                  (ANTIBIOTIC-1 TETRACYCLINE AMPICILLIN)
                  (VECTOR-2 PBR322))
                 ((RESTRICTION-ENZYME-1 RESTRICTION-
                   ENZYME)
                  (BACTERIUM-3 E.COLI)
                  (ANTIBIOTIC-1 TETRACYCLINE)
                  (VECTOR-2 PMB9 PSC101)))
MAKER:          PLAN-STEP-21
```

FIG. 12. Integrating constraints. MOLGEN uses a *tuple* notation to keep track of possible values for variables. When constraints are considered which tie together variables from different tuples, the requirements are combined.

the other constraints in *Tuple-2*. *Constraint-7* is a constraint requiring that *Vector-2* carry a resistance gene for *Antibiotic-1*.

3.7. Finishing the plan

The rest of the trace of MOLGEN's performance on this experiment uses the same kinds of problem solving techniques that we have seen already. New constraints are introduced about restriction enzymes and resistance genes and more variables are introduced and anchored as the constraints on the plan accumulate. In *Lab-Step-7*, MOLGEN introduced a 'molecular adapter' (*Linker-1*) so that the rat-insulin gene can be readily attached to the vector. At this point, the solution was somewhat predetermined in that MOLGEN's knowledge base only had one available linker (called *Hind3decamer*) that could be used. This narrowed the number of possible solutions to the accumulated constraints more than would have been possible if a full complement of linkers had been available. Even so, MOLGEN had four solutions after satisfying all of the constraints as shown in Table 1. The fourth solution was the one

TABLE 1. Final solutions to the constraints

Solution	Bacterium	Vector	Antibiotic	Enzyme	Linker
1	E. coli	pBR322	Tetracycline	HIND3	HIND3DECAMER
2	E. coli	pBR322	Ampicillin	HIND3	HIND3DECAMER
3	E. coli	pSC101	Tetracycline	HIND3	HIND3DECAMER
4	E. coli	pMB9	Tetracycline	HIND3	HIND3DECAMER

reported by Ullrich et al. [24]. A picture of MOLGEN's plan for the experiment is shown in Fig. 13.

In reporting their experiments, geneticists customarily report only the details of their final experiments. Infrequently they report some of their thoughts in planning an experiment and even less frequently are any of the constraints reported. In review articles (such as Boyer [1]) one can sometimes find a discussion of the constraints or experimental considerations once the technique has worked its way into the methodology of the field. The constraints that MOLGEN formulated in the rat-insulin problem are listed below together with a description of their introduction to the plan:

(1) *The bacterium should be biologically compatible with the vector.*

(Formulated as a commitment when ***Merge*** was refined to ***Transform*** in *Lab-Step-1.*)

(2) *The vector should have sticky ends prior to ligation in Lab-Step-2 for some restriction enzyme (Restriction-Enzyme-1).*

(Formulated as a commitment when ***Merge*** was refined to ***Ligate*** in *Lab-Step-2.*)

(3) *The DNA carrying the Rat-insulin gene should have sticky ends for Restriction-Enzyme-1* prior to ligation in *Lab-Step-2.*

(Formulated as a commitment when ***Merge*** was refined to ***Ligate*** in *Lab-Step-2.*)

(4) *The bacterium carrying the plasmid (Bacterium-4) should be resistant to some antibiotic (Antibiotic-1).*

(Formulated as a commitment when ***Sort*** was refined to ***Screen*** in *Lab-Step-4.*)

(5) *The bacterium without the plasmid (Bacterium-3) should not be resistant to Antibiotic-1.*

(Formulated as a commitment when ***Sort*** was refined to ***Screen*** in *Lab-Step-4.*)

(6) *The vector input to the transformation step (Vector-1) should carry a resistance gene for Antibiotic-1.*

(Result of propagating *Constraint-4* and *Constraint-5* through the ***Transform*** operator in *Lab-Step-1.*)

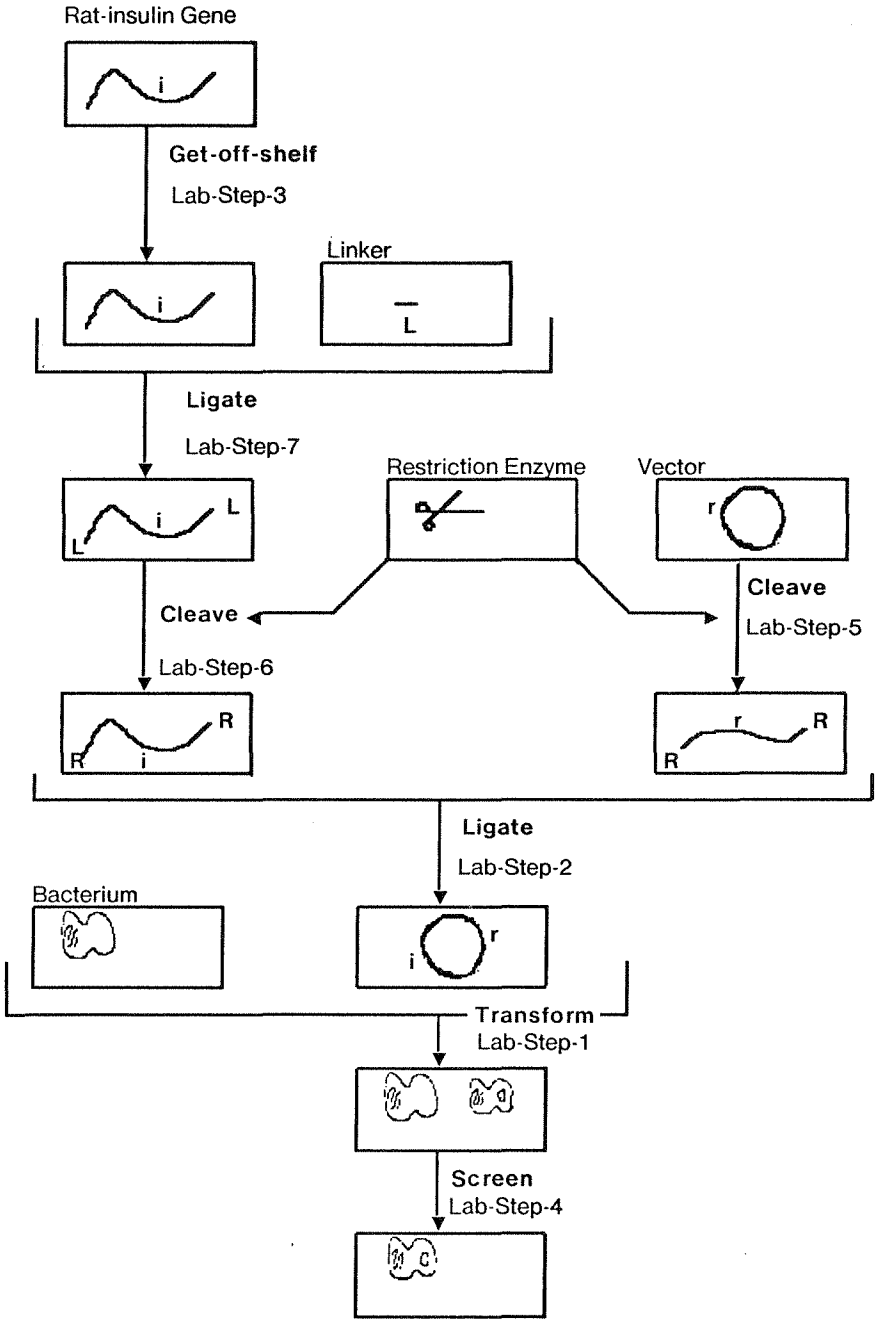

FIG. 13. Final plan for the rat-insulin problem.

(7) *The vector out of which Vector-1 is made should carry a resistance gene for Antibiotic-1.*

(Result of propagating *Constraint-6* through the ***Ligate*** operator in *Lab-Step-2.*)

(8) *Restriction-Enzyme-1 should not cut the resistance gene for Antibiotic-1.*
(Result of propagating *Constraint-7* through the ***Cleave*** operator in *Lab-Step-5.*)
(9) *Restriction-Enzyme-1 should not cut the rat-insulin gene.*
(Result of propagating *Constraint-2* through the ***Cleave*** operator in *Lab-Step-5.*)
(10) *The vector should have a site for Restriction-Enzyme-1.*
(Formulated as a commitment when ***React*** was refined to ***Cleave*** in *Lab-Step-5.*)
(11) *The DNA carrying the rat-insulin gene should have a site for Restriction-Enzyme-1.*
(Result of propagating *Constraint-3* through the ***Cleave*** operator in *Lab-Step-6.*)
(12) *The linker should have a site for Restriction-Enzyme-1.*
(Formulated as a commitment when ***React*** was refined to ***Cleave*** in *Lab-Step-7.*)

If MOLGEN had a more detailed model of genetics (i.e., including the logic of gene promoters) even more constraints would have been formulated.

4. The Effectiveness of Constraint Posting

The power of constraint posting comes largely from two abilities: (1) the ability to plan hierarchically by introducing new constraints and variables, and (2) the ability to anticipate interference between subproblems (using constraint propagation) and to eliminate the interfering solutions. The effectiveness of this during the planning of the rat-insulin problem is illustrated in Table 2.

Each row in the table corresponds to the introduction of a constraint in the plan; the first row shows the situation before any constraints have been introduced. The shaded squares in each row indicate which variables are involved in the constraint. For example, the *compatibility* constraint in the second row involves the variables *Bacterium* and *Vector.* The power of constraint formulation is illustrated by the column labeled 'Total Combinations', which shows how the number of solutions decreases from 3456 to 4 as constraints are added. This column shows the number of acceptable combinations of values for all of the variables. These numbers understate the combinations seen by MOLGEN because they reflect the use of genetics knowledge to reduce the combinatorics. For example, a plan actually contains many variables representing bacteria at different stages of planning. MOLGEN knows that these bacteria represent an *equivalence class* for the purposes of constraint satisfaction, because no laboratory operator will change a bacterium from one type to another. If the variables were counted independently, the columns would be *powers* of the numbers shown. In the first row, no com-

TABLE 2. Elimination of Solutions

Constraint	Combinations Total	Considered	Antibiotic	Bacterium	Enzyme	Linker	Vector
_____	3456	0	9	3	32	1	4
Compatible	1152	4	9	1	32	1	4
Carries Resistance Gene	160	5	3	1	32	1	4
Has Sites	21	21	3	1	6	1	4
Doesn't cut Resistance Gene	10	10	2	1	6	1	4
Has Sites	4	4	2	1	1	1	3

binations are ruled out and 3456 is the product of the number of solutions for each variable. As planning continues, the possible solutions are only a subset of this because some of the combinations are ruled out.

Because MOLGEN does not consider solutions for variables until they are introduced, it works with a substantially reduced number of combinations as shown in the column labeled 'Considered Combinations'. MOLGEN introduces new variables as it plans hierarchically. For example, the variable *Enzyme* is not really considered in the problem until it first appears in a constraint for sites on a vector. Thus, hierarchical planning greatly reduces MOLGEN's bookkeeping requirements during constraint satisfaction. The largest number of combinations that MOLGEN needed to simultaneously record during planning was 21, when the *Enzyme* variable was introduced.

The global control of interactions is illustrated in the declining numbers of solutions for each variable. A decrease in the number of solutions for a variable usually happens only when an additional constraint involving that variable is introduced. For example, the number of solutions for the *Bacterium* variable is reduced from 3 to 1 by the first constraint. In the last row of the

chart, the number of solutions for the *Vector* variable decreases from 4 to 3, even though it is not involved directly in the new constraint. This is because constraint satisfaction implicitly includes all of the previous constraints. All of the solutions involving the eliminated vector (a subset of the 10 solutions satisfying the previous constraints) also involved a particular solution for the *Enzyme* variable. When the last constraint reduced the number of possible enzymes from 6 to 1, it eliminated all of the solutions that permitted the deleted value for the *Vector* variable. This shows how the bookkeeping of constraint satisfaction automatically coordinates requirements from different parts of the problem.

5. Relationships to Other Work

The constraint posting approach builds on previous research in hierarchical planning, subgoal interactions, and constraint satisfaction. Several recent and detailed reviews of this research are available with extensive bibliographies [13, 15, 20]. In the interest of brevity, the following discussion will be limited to the main ideas.

5.1. Hierarchical planning

Many AI programs have had the ability to break a problem into subproblems, that is, to find a solution by a *divide and conquer* strategy. However, a program uses a *hierarchical* approach only if it has the additional capability to defer consideration of the details of a problem. *Non-hierarchical* programs suffer from the tyranny of detail. If in the course of solving a problem there is something they need to know, they must determine it immediately. This fault is expressed in the common wisdom as 'not being able to see the forest for the trees'. Abstraction, as the basis for hierarchical planning, is a way of suppressing detail. It was used in the General Problem Solver (GPS) reported by Newell, Shaw, and Simon, for finding proofs in propositional logic. Hierarchical approaches have been integral to most recent planning programs.

When hierarchical and non-hierarchical approaches have been systematically compared, the former have usually dominated. For example, the ABSTRIPS program (Sacerdoti [17]) was a version of the non-hierarchical STRIPS planning program, retro-fitted with a scheme for abstract reasoning. In this comparative study of the two programs on a sequence of blocks world problems, Sacerdoti reported that ABSTRIPS was substantially more efficient than STRIPS, and that the effect increased dramatically as longer plans were tried. Hierarchical and non-hierarchical methods have also been compared in special purpose applications, such as Paxton's study [14] of approaches to speech recognition. Paxton's measurements indicated that a hierarchical 'island-driving' approach does not necessarily dominate a simpler left-to-right processing approach. When planning islands are formed by an abstraction process, the

abstraction process must be appropriate. In the terminology of this article, the abstraction process must divide the planning decisions into nearly independent (or loosely-coupled) subproblems. As a practical matter, the more loosely the subproblems are coupled, the better the hierarchical approaches have performed because the methods for handling interactions between subproblems have been weak.

MOLGEN differs from these earlier hierarchical planning programs in its ability to add details to a plan by adding constraints. This approach to hierarchical planning avoids the issue of trying to assign global criticality levels to the domain vocabulary (as in ABSTRIPS), and reflects the perspective that commitments in planning can be characterized as new constraints. This facilitates knowledge-based approaches to backtracking that examine the reasons for making commitments in planning.

5.2. Interactions between subproblems

When subproblems in a problem do not interact, they can be solved independently. However, the experience with problem-solving programs in the past few years has shown that this ideal situation is unusual in real world problems. Interactions appear even in highly simplified AI domains such as the blocks world. Recognition of this has led researchers to focus on the nature of interactions to determine how they should be taken into account during planning.

The first of the recent programs to focus on the interactions between steps was the HACKER program reported by Sussman [22]. HACKER solved problems in the blocks world by making some simplifying assumptions to create an initial plan, and then *debugging* the plan. HACKER's main assumption (termed the *linearity assumption*) was that to solve a conjunction of goals, each one may be solved in sequence. In many simple blocks world problems, the effects of satisfying one goal interfere with solving another one. Sussman created procedures called *critics* that could recognize such interference. HACKER was often able to repair the plan by rearranging the steps in the plan.

Other approaches to satisfying conjunctive goals have been explored by Tate and Waldinger. In his INTERPLAN program, Tate's approach was to abstract the original goals and to determine holding periods over which they could be assumed to be true. INTERPLAN analyzed these periods with a view toward moving goals around to ease conflict situations. Waldinger [25] developed an approach called *goal regression* for problems from program synthesis and blocks world. It involved creating a plan to solve one of several goals followed by constructive modifications to achieve the other goals. It differed from HACKER in that it used notation about protection of goals to guide the linear placement of actions in the plan. Thus, rather than building incorrect plans and

then debugging them, it built partial linear plans in non-sequential order. The term *goal regression* is suggestive of the way the program worked, moving goals backwards through the planned actions to where they did not interfere.

A novel approach to planning with interfering conjunctive goals was reported by Sacerdoti [16] for his NOAH program. NOAH avoided HACKER's linearity assumption by considering the plan steps as parallel (that is, partially ordered) as long as possible. NOAH had *constructive* critics which sequenced the steps according to the interactions that were uncovered. If an action for one goal deleted an expression that was a precondition of a conjunctive goal, then the action with the endangered precondition was moved so that it would be performed first. In 1977 Tate [23] extended these techniques somewhat in his planning program, NONLIN, which he applied to blocks world problems and to generator maintenance in power stations.

MOLGEN is like NOAH in its use of a least commitment strategy for handling interactions. NOAH used this idea for resolving the order of operators; MOLGEN used it mostly for object selection. (See Stefik [21] for a discussion of MOLGEN's recourse to heuristic reasoning when least commitment fails.) Constraint propagation in MOLGEN is like Waldinger's goal regression, except that MOLGEN is a hierarchical planner. Section 6 discusses some weaknesses in MOLGEN's representation of time which bear on the use of constraint propagation across planning situations.

5.3. Reasoning with constraints

This section discusses several AI programs that use constraints. It begins with search, a model for problem solving in AI in which solutions are found by traversing a space of possibilities for candidates that satisfy some constraints.

DENDRAL and its descendant CONGEN (Buchanan and Feignenbaum [3]) are examples of programs that use constraint-satisfaction. CONGEN accepts as input a set of constraints about chemical structures – an atomic formula, lists of required and disallowed substructures, and partial specifications of inter-atomic connections. It searches for solutions using a hierarchical *generate-and-test* approach. An exhaustive depth-first generator of chemical structures delivers partial solutions for testing against the constraints. CONGEN's applicability to practical problems depends on (1) the availability and use of powerful problem-specific constraints for limiting the generation of candidates, and on (2) the application of these constraints early in the generation process.

When constraints can be applied early in the solution process on partial solutions, the time to solution can usually be reduced. This leads to the idea of processing the constraints in ways that facilitate their early application. In 1970, Fikes [5] reported a problem-specification language and problem solver, REF-ARF, that was able to represent and solve a number of discrete numeric and symbolic constraint satisfaction problems. REF-ARF combined backtracking

with constraint manipulation routines. Given a partial instantiation of the variables, these routines attempted to simplify the remaining constraints by reducing choices for the other variables or by deriving a contradiction. For example, an unbound variable could be expressed as a function of bound variables to yield an immediate solution. By alternating constraint manipulation and variable instantiation, REF-ARF demonstrated an impressive performance that was much superior to backtracking methods, which require complete variable instantiation before acceptance tests can be applied. Mackworth [9] and Freuder [6] have recently reviewed some sources of redundancy in backtracking and have suggested ways to improve efficiency.

For several years, several researchers as MIT have been working on programs for electronic circuit analysis and design. In 1977, McDermott [10] reported an ambitiously conceived program (NASL) for designing electrical circuits. NASL designed circuits hierarchically by combining and instantiating schemata representing functional subcircuits; it was capable of propagating and manipulating various kinds of algebraic constraints about circuits. Although NASL was never fully implemented and relied on human intervention for the more difficult aspects of constraint manipulation, this research established some of the ideas for later design programs. In 1978, Sussman and de Kleer [4] reported the SYN program for the *synthesis* phase of circuit design, that is, for determining the parameters of a circuit given desiderata for its behavior. Solution of the parameters by algebraic means (i.e., solving equations) is infeasible. SYN introduces constraints by making engineering assumptions about the operation of various components (e.g., by assuming that a transistor is in its linear operating region) and then propagates them through the circuit using electrical laws. The constraints are composed of algebraic expressions with variables. In some cases, SYN introduces variables for unknowns. It combines and reduces the resulting algebraic expressions using an adaptation of a rational simplifier from MACSYMA.

MOLGEN differs from constraint satisfaction programs like DENDRAL and REF-ARF in that it is not limited to the initial set of constraints. MOLGEN formulates constraints dynamically as it runs. NASL and SYN both augmented the constraint satisfaction idea with the use of constraint propagation between subproblems. Many of the ideas that were important for MOLGEN were anticipated in NASL, although they were not implemented.

6. Limitations and Further Research

Research often raises more questions than it answers. While this paper offers some suggestions for understanding the process of design in terms of constraint posting, it leaves open several fundamental questions about the constraints themselves. The following sections raise some issues about constraints and the

representation of time, the generality of MOLGEN's ability to use constraints, and some practical limitations of MOLGEN.

6.1. Constraints and meta-constraints

The generality of MOLGEN's ability to reason with constraints stems from the simple requirements of constraint satisfaction. Constraint satisfaction requires only the ability to *evaluate* constraints. As long as MOLGEN can *generate* potential solutions, it can easily test whether arbitrary constraints are satisfied. This use of constraints for testing ignores the more powerful idea of using them to guide generation, by applying them early to partial solutions.

MOLGEN's ability to apply constraints early depends on its implementation of constraint propagation, which has some serious weaknesses. MOLGEN's constraint propagation operators are based strictly on *syntactic matches* of the constraints. Unfortunately, MOLGEN has no capability for recognizing the equivalence of logical predicates in constraints. Although MOLGEN is able to propagate constraints that it was generated, it has no ability to propagate logically equivalent variations of these constraints or arbitrary constraints outside of its limited vocabulary. This results in a practical limitation on MOLGEN's ability to use constraints; while it may eventually generate a plan that satisfies a new constraint, it may practically take too long for MOLGEN to propose a satisfactory plan if it can only apply the constraint late in the planning process.

A second limitation is that MOLGEN's does not use constraints to describe processes; all of the examples in this paper deal only with object specification. The simplest example of this would be to constrain the selection of laboratory operators. The difficulty is that MOLGEN lacks powerful ways to describe processes. No constraints on *partial process descriptions* have been developed within the representational framework used in MOLGEN.

A third limitation is that MOLGEN's does not use *meta*-constraints. First-order constraints are about the objects in the plans; *meta*-constraints would be about the plan or the planning process. For example, there could be a constraint that the plan have no more than twelve steps, or constraints on its overall yield or time of execution. In the companion paper, we will see that MOLGEN's interpreter is organized in layers. Within this layered structure, the knowledge about manipulating constraints simply appears at too low a level to support constraint reasoning about the design process.

6.2. The knowledge acquisition bottleneck

Although ideas like *meta-constraints* have some exotic appeal, it is difficult to assess their impact on making a practical system. To keep things in perspective, it is worth remarking on a serious practical limitation to the use of computers in problem solving: the difficulty of getting the relevant knowledge into the

computer. This difficulty is compounded in a rapidly expanding field like molecular genetics because the knowledge can quickly become out of date. Most knowledge-based systems (including MOLGEN) fail to use what they know to make the transfer of expertise less painful. They don't take an active part in trying to understand what they are told and don't improve their ability to acquire new knowledge.

Constraint posting is a knowledge intensive style of problem solving; it requires substantial knowledge about when to formulate constraints and how to propagate them. Missing knowledge about constraint formulation has a more serious effect than missing knowledge about constraint propagation. When MOLGEN fails to formulate some necessary constraint in planning, it fails to model the genetics accurately and may propose experiments that will not work in the laboratory. When MOLGEN fails to propagate constraints, interference between planning decisions will not be discovered until much extra work is done. This results in only a *soft failure* in planning; MOLGEN may still plan successfully, but only after much extra backtracking. In practical terms, the amount of extra computation can sometimes mean that MOLGEN will never finish. The difficulty of incorporating such knowledge easily into a knowledge base illustrates the need for more research in knowledge acquisition.

6.3. Representing time

MOLGEN uses an inadequate representation of time. To deal with the changes in objects over time, MOLGEN changes the names of the objects. At different points in a plan, a bacterium may be known as *Bacterium-1*, or *Bacterium-3*, or some other name. These different names refer to the same bacterium in different 'states'. The determination of the times during which various constraints are satisfied is indicated indirectly by the names of the objects that are referenced. While this approach is good enough to indicate when constraints are satisfied (in terms of states), it does not provide a satisfactory representation of time for further planning work. For example, it does not facilitate (1) reasoning explicitly about the periods of satisfaction of constraints or (2) maintaining records of distinct possible worlds.

Reasoning about possible futures is tricky because what will happen depends on what we do and on things that we do not know about. For example, we want to reason as far into the future as knowledge and commitments permit. I know of no planning programs which can realistically reason about the future or construct useful scenarios. To do this, they would need to understand the limits of their knowledge and the sources of uncertainty about the future.

7. Summary

This paper presents an approach to hierarchical planning which focusses on the

use and interpretation of constraints. Constraints are viewed (1) as elimination rules for ruling out solutions, (2) as commitments made by the planner to partially describe solutions, and (3) as a communication medium for expressing interactions between subproblems. Constraint posting is an approach to hierarchical planning which exploits the different interpretations of constraints to plan effectively. It formulates constraints during hierarchical planning to add new commitments and propagates them so that they can be utilized early in the design process to eliminate interfering solutions.

A computer program has been implemented with a genetics knowledge base to test the idea of constraint posting. It models the experiment design process in terms of operations on constraints: formulation, propagation, and satisfaction. Constraint formulation adds details to parts of the plan. Constraint propagation spreads information between the nearly independent subproblems. Constraint satisfaction finds values for the variables subject to constraints from the subproblems.

Constraint posting is a knowledge intensive approach to problem solving. An impediment to the routine application of such approaches is the lack of effective means for transferring such information into a computer. This work does not address the knowledge acquisition issue but has identified several kinds of inferential knowledge for handling constraints.

ACKNOWLEDGMENT

The research reported here was drawn from my thesis [20]. Special thanks to my advisor, Bruce Buchanan, and the other members of my reading committee: Edward Feigenbaum, Joshua Lederberg, Earl Sacerdoti, and Randall Davis. Thanks also to the members of the MOLGEN project – Douglas Brutlag, Jerry Feitelson, Peter Friedland, and Lawrence Kedes for their help. Thanks to Daniel Bobrow, Lewis Creary, and Austin Henderson 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. Boyer, H.W., Betlach, M., Bolivar, F., Rodriguez, R.L., Shine, J. and Goodman, H.M., The construction of molecular cloning vehicles, in: Beers, R.F. and Bassett, E.G., Eds., *Recombinant Molecules: Impact on Science and Society* (Raven Press, New York, 1977).
2. Brooks, F.P., *The Mythical Man-month: Essays on Software Engineering* (Addison-Wesley Publishing Company, Reading, MA, 1975).
3. Buchanan, B.G. and Feigenbaum, E.A., DENDRAL and Meta-DENDRAL: Their applications dimension, *Artificial Intelligence* **11** (1978) 5–24.
4. de Kleer, J. and Sussman, G.J., Propagation of constraints applied to circuit synthesis, MIT AI Memo 485 (September 1978).
5. Fikes, R.E., REF-ARF: A system for solving problems stated as procedures, *Artificial Intelligence* **1** (1970) 27–120.
6. Freuder, E.C., Synthesizing constraint expressions, *Communications of the ACM* **21**(11) (1978) 958–966.
7. Friedland, P., Knowledge-based hierarchical planning in molecular genetics, Doctoral Dis-

sertation, Computer Science Department, Stanford University. (Also Computer Science Department Report STAN-CS-79-771.)
8. Gilbert, W. and Villa-Komaroff, L., Useful proteins from recombinant bacteria, *Sci. Am.* (April 1980) 74–94.
9. Mackworth, A.K., Consistency in networks of relations, *Artificial Intelligence* **8** (1977) 99–118.
10. McDermott, D.V., Flexibility and efficiency in a computer program for designing circuits, Doctoral Dissertation, Massachusetts Institute of Technology, Report AI-TR-402 (June 1977).
11. Newell, A. and Simon, H.A., GPS, A program that simulates human thought, in: Feigenbaum, E.A. and Feldman, J., Eds., *Computers and Thought* (McGraw-Hill, New York, 1963).
12. Polya, G., *Mathematical Discovery. 2* (John Wiley and Sons, New York, 1965).
13. Nilsson, N.J., *Principles of Artificial Intelligence* (Tioga Publishing Co., Palo Alto, 1980).
14. Paxton, W.H., A framework for speech understanding, Doctoral Dissertation, Computer Science Department, Stanford University (1977). (Also SRI Artificial Intelligence Center Technical Note 142).
15. Sacerdoti, E.D., Problem solving tactics, *Proceedings of the Sixth International Joint Conference on Artificial Intelligence* (August 1979) 1077–1085.
16. Sacerdoti, E.D., *A Structure for Plans and Behavior.* (American Elsevier Publishing Company, New York, 1977 (originally published, 1975)).
17. Sacerdoti, E.D., Planning in a hierarchy of abstraction spaces, *Artificial Intelligence* **5**(2) (1974) 115–135.
18. Simon, H.A., The science of design and the architecture of complexity, in: *Sciences of the Artificial* (MIT press, Cambridge, 1969).
19. Stefik, M.J., An examination of a frame-structured representation system, *Proceedings of the Sixth International Joint Conference on Artificial Intelligence* (1979) 845–852.
20. 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.)
21. Stefik, M.J., Planning and meta-planning, *Artificial Intelligence* **16**(2) (1981) 141–170 [this issue].
22. Sussman, G.J., *A Computer Model of Skill Acquisition* (American Elsevier, New York, 1975).
23. Tate, A., Generating project networks, *Proceedings of the Fifth International Joint Conference on Artificial Intelligence* (1977) 888–900.
24. Ullrich, A., Shine, J., Chirgwin, J., Pictet, R., Tischer, E., Rutter, W.J. and Goodman, H.M., Rat insulin genes: construction of plasmids containing the coding sequences, *Science* **196** (June 1977) 1313–1319.
25. Waldinger, R., *Achieving several goals simultaneously*, SRI Artificial Intelligence Center Technical Note 107 (July 1975).
26. Winograd, T., Frame representations and the procedural/declarative controversy, in: Bobrow, D.G. and Collins, A., Eds., *Representation and Understanding: Studies in Cognitive Science* (Academic Press, New York, 1975).

Received June 1980; revised version received September 1980