Achieving Several Goals Simultaneously*

Richard Waldinger

Artificial Intelligence Center
Stanford Research Institute

In the synthesis of a plan or computer program, the problem of achieving several goals simultaneously presents special difficulties, since a plan to achieve one goal may interfere with attaining the others. This paper develops the following strategy: to achieve two goals simultaneously, develop a plan to achieve one of them and then modify that plan to achieve the second as well. A systematic program modification technique is presented to support this strategy. The technique requires the introduction of a special "skeleton model" to represent a changing world that can accommodate modifications in the plan. This skeleton model also provides a novel approach to the "frame problem."

The strategy is illustrated by its application to three examples. Two examples involve synthesizing the following programs: interchanging the values of two variables and sorting three variables. The third entails formulating tricky blocks-world plans. The strategy has been implemented in a simple QLISP program.

It is argued that skeleton modelling is valuable as a planning technique apart from its use in plan modification, particularly because it facilitates the representation of "influential actions" whose effects may be far reaching.

The second part of the paper is a critical survey of contemporary planning literature, which compares our approach with other techniques for facing the same problems. The following is the outline of contents.

CONTENTS

0. INTRODUCTION

1. SIMULTANEOUS GOALS, PROGRAM MODIFICATION, AND THE REPRESENTATION OF ACTIONS
   - 1.1 A description of our approach
     - 1.1.1 Achieving primitive goals
     - 1.1.2 Goal regression
     - 1.1.3 Plan modification
     - 1.1.4 Protection
     - 1.1.5 A very simple example
     - 1.1.6 Skeleton models
   - 1.2 Interchanging the values of two variables
     - 1.2.1 Relations that refer to variables
     - 1.2.2 The Solution to the two-variable problem
   - 1.3 Sorting three variables
     - 1.3.1 Sorting two variables
     - 1.3.2 Achieving an implication
     - 1.3.3 The solution to the three-sort problem
   - 1.4 The Sussman "anomaly"
   - 1.5 Limitations and next steps

2. THE REPRESENTATION OF ACTIONS AND SITUATIONS IN CONTEMPORARY PROBLEM SOLVING
   - 2.1 The classical problem solvers
   - 2.2 Regression and STRIPS operators
   - 2.3 The use of contexts to represent a changing world
   - 2.4 Influential actions
   - 2.5 Escaping from the STRIPS assumption
   - 2.6 The use of contexts to implement skeleton models
   - 2.7 Hypothetical worlds
   - 2.8 Complexity
   - 2.9 Recapitulation

ACKNOWLEDGMENTS

REFERENCES

*The research reported herein was sponsored by the National Science Foundation under Grant GJ-36146.

INTRODUCTION

My feet want to dance in the sun
My head wants to rest in the shade
The Lord says "Go out and have fun!"
But the landlord says "Your rent ain't paid!"

E.Y. Harburg, *Finian's Rainbow*

It is often easier to achieve either of two goals than to achieve both at the same time. In the course of achieving the second goal we may undo the effects of achieving the first. Terry Winograd points out in a *Psychology Today* article (Winograd, 1974) that his blocks program

> cannot carry out the command, "Build a stack without touching any pyramids," because it has no way to work on one goal (building a stack) while keeping track of another one (avoiding contact with pyramids).

The reasoning subprograms of his natural language processor "have a sort of one-track mind unsuited to complicated tasks."

In program synthesis, such "simultaneous goal" problems are rampant. A

typical example: the goal of a sort program is to rearrange an array in ascending order while ensuring at the same time that the resulting array is a permutation of the original. Simultaneous goal problems occur in mathematical equation solving, in robot tasks, and in real life as well.

An earlier paper (Manna and Waldinger, 1974) proposes a method for dealing with simultaneous goal problems in program synthesis. The present paper elaborates on the description of the method, reports on its implementation, discusses its application to general planning and robot problem solving, and points out some of its shortcomings and some projected improvements.

The general strategy proposed in (Manna and Waldinger, 1974) is: in order to construct a plan to achieve P and Q, construct a plan to achieve P, and then modify that plan to achieve Q as well. In the course of the modification, the relation P is "protected": no modifications that might make P false are permitted. If no satisfactory modification is found, the same strategy is attempted with the roles of P and Q reversed.

The earlier paper considers the construction of programs with branches and recursive loops; here, the discussion is strictly limited to the construction of straight-line programs. The simultaneous goal strategy can be integrated with the branch and loop formation techniques discussed in our earlier paper; however, this integration has not yet been implemented. Furthermore, the straight-line case is rich enough to be interesting in its own right.

The paper is divided into two main parts. Part 1 describes the simultaneous goal strategy in full detail, the program modification technique, and the modelling structure that the strategy requires. The strategy is illustrated by several examples, including the development of programs to interchange the values of two variables and to sort three variables, and the solution of the "anomaly" blocks-world problem from Sussman's (Sussman, 1973) thesis. These examples are not chosen to be impressive; they have been refined to present no difficulties other than the simultaneous goal problem itself.

Part 2 tries to relate this work to some other problem-solving efforts, and provides a critical survey of the way these systems represent a changing world in terms of the framework developed in Part 1. A summary of Part 2 appears in Section 2.9.

PART 1

SIMULTANEOUS GOALS, PROGRAM MODIFICATION, AND THE REPRESENTATION OF ACTIONS

1.1 A description of our approach

1.1.1 Achieving primitive goals

Below, the boarhound and the boar
Pursue their pattern as before
But reconciled among the stars.

T.S. Eliot, *Four Quartets*

Before we are ready to face multiple simultaneous goals, it may be helpful to say a few words about our approach to simple goals. Our system has a number of built-in techniques and knowledge of the kinds of goal to which each technique applies. When faced with a new goal, the system tries to determine if that goal is already true in its model of the world–if the goal is true it is already achieved. Otherwise, the system retrieves those techniques that seem applicable. Each of these techniques is attempted in turn until one of them is successful.

An important clue to the choice of technique is the form of the given goal. For instance, suppose we are working on blocks-world problems, and we are faced with the goal that block A be directly on top of block B. Assume that we have an arm that can move only one block at a time. Then we may build in the following strategy applicable to all goals of form, "Achieve: x is on y": clear the top of x and the top of y, and then put x on y. That x be clear is a new goal, which may already be true in the model, or which may need to be achieved itself (by moving some other block from the top of x to the table, say). "Put x on y" is a step in the plan we are developing. If we can successfully apply this technique, we have developed a plan to put A directly on top of B. However, for a variety of reasons this technique may fail, and then we will have to try another technique.

An example from the program synthesis domain: suppose our goal is to achieve that a variable X have some value b. One approach to goals of this form is to achieve that some other variable v has value b, and then execute the assignment statement $X \leftarrow v$.* Here again, the relation "v has value b" is a subgoal, which may already be true or which may need to be achieved by inserting some other instructions into the plan. The assignment statement $X \leftarrow v$ is an operation that this technique itself inserts into the plan. (Note that if we are not careful, this technique will be applicable to its own subgoal, perhaps resulting in an infinite computation.) Of course, there may be other techniques to achieve goals of form "X has value b"; if the original technique fails, the others are applied.

The practice of retrieving techniques according to the form of the goal and then trying them each in turn until one is successful is called "pattern-directed function invocation," after Hewitt (Hewitt, 1972). A problem solver organized around these principles can be aware of only one goal at a time: hence the single-mindedness that Winograd complains of. When given multiple simultaneous goals, we would like to be able to apply the techniques applicable to each goal and somehow combine the results into a single coherent plan that achieves all of them at once.

1.1.2 Goal regression

Change lays not her hand upon truth.

A.C. Swinburne, *Poems: Dedication*

*We use a lower case "v" but an upper case "X" because here, X is the name of a specific variable while v is a symbol that can be instantiated to represent any variable.

Our approach to simultaneous goals depends heavily on having an effective program modification technique. Our program modification technique in turn depends on knowing how our program instructions interact with the relations we use to specify the program's goals.

Suppose P is a relation and F is an action of program instruction; if P is true, and we execute F, then of course we have no guarantee that P will still be true. However, given P, it is always possible to find a relation P′ such that achieving P′ and then executing F guarantees that P will be true afterwards. For example, in a simple blocks world if P is "block C is clear" (meaning C has no blocks on top of it) and F is "Put block A on block B," then P′ is the relation "C is clear or A is on C": for if C is clear before putting A on B, then C will still be clear afterwards, while if A is on C before being put on B, then the action itself will clear the top of C.*

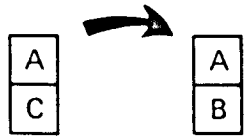

FIGURE 1

We will demand that P′ be the weakest relation that ensures the subsequent truth of P; in other words, if P′ is not true before executing F, P may not be true afterwards. Otherwise, we could always take P′ to be the relation that is always false. We will call P′ the result of passing P back over F, and we will call the operation of passing P back "regression."

Another example: suppose F is an assignment statement "X ← t" where X is a variable and t an expression, and let P be any relation between the values of the variables of our program, written P(X). Then P′ is P(t), the relation obtained by replacing X by t in P(X). For if P(t) is true before executing X ← t, then P(X) will certainly be true afterwards. For instance, if P(X) is "X=A*B," and F is "X ← U*V," then P′=P(U*V) is "U*V=A*B," for if U*V=A*B before executing X ← U*V, then X=A*B afterwards. Furthermore, if U*V=A*B is false before the assignment, then X=A*B will be false afterwards.

Note that if X does not occur in P(X), then P(t) is the same as P(X); the instruction has no effect on the truth of the relation.

Regression will play an important part in our program modification technique and also in the way we construct our models. The use of a static relational description to describe a dynamic program has been variously attributed to (Floyd, 1967), (Naur, 1966), (Turing, 1950), and (Goldstine and von Neumann, 1947), but the observation that it is technically simpler to look at the "weakest preconditions" of a relation (passing it back), as we do, instead of the "strongest postconditions" (passing it forward), appears to be due to (Manna, 1968), (Hoare, 1969), and (King, 1969). The term "weakest precondition" is Dijkstra's (1975); we will not use it because the word "precondition" has a different meaning in the artificial intelligence literature. All these authors apply the idea to proving the correctness of programs; (Manna, 1974) contains a survey of this application. We now go on to show how the idea applies to program modification as well.

*We assume that the blocks are all the same size, so that only one block can fit immediately on top of another.

1.1.3 Plan modification

It is a bad plan that admits of no modification.

Publilius Syrus, *Sententiae*

In order to achieve a goal of form P and Q, we construct a plan F that achieves P, and then modify F so that it achieves Q while still achieving P. The simplest way to modify F is to add new instructions to the end so as to achieve Q. This method is called a "linear theory plan" by Sussman (Sussman, 1973). However, this linear strategy may be flatly inadequate; for instance, executing the plan F may destroy objects or information necessary to achieve Q. Furthermore, even if Q can be achieved by some composite plan ⟨F:G⟩ (execute F, then execute G), how can we be sure that plan G will not cause P to be made false?

However, we may also modify F by adding new instructions to the beginning or middle, or by changing instructions that are already there. Let us assume that F is a linear sequence of instructions $\langle F_1,...,F_n\rangle$. As we have seen, in order to achieve Q after executing F, it suffices to achieve Q′ immediately before executing F_n, where Q′ is the result of passing Q back over F_n. Similarly, it suffices to achieve Q″ immediately before executing F_{n-1}, where Q″ is the result of passing Q′ back over F_{n-1}.

How can we benefit by passing a goal back over steps in the plan? A goal that is difficult or impossible to achieve after F has been executed may be easier to achieve at some earlier point in the plan. Furthermore, if achieving Q after executing F destroys the truth of P, it is possible that planning to achieve Q′ or Q″ earlier will not disturb P at all; a planner should be free to achieve Q in any of these ways.

How is the planner supposed to know how to pass a relation back over a given plan step? First of all, the information can be given explicitly, as one of a set of rules. These "regression rules," which can themselves be expressed as programs, are regarded as part of the definition of the plan step. Alternatively, if a relation is defined in terms of other relations, it may be possible to pass back those defining relations. Furthermore, if the plan step is defined in terms of simpler component plan steps, then knowing how to pass relations back over the components allows one to pass the relation back over the original plan step. Finally, if no information at all exists as to how to pass a relation back over a plan step, it is assumed that the plan step has absolutely no effect on the relation. This assumption makes it unnecessary to state a large number of rules, each saying that a certain action has no effect at all on a certain relation. Thus we avoid the

so-called "frame problem" (cf. [McCarthy and Hayes, 1969]).

In modifying a program it is necessary to ensure that it still achieves the purpose for which it was originally intended. This task is performed by the protection mechanism we will now describe.

1.1.4 Protection

Protection is not a principle, but an expedient.

Disraeli, Speech

Our strategy for achieving two goals P and Q simultaneously requires that after developing a plan F that achieves P we modify F so that it achieves Q while still achieving P. This strategy requires that in the course of modifying F the system should remember that F was originally intended to achieve P and check that it still does. It does this by means of a device called the protection point: we attach P to the end of F as a comment. This comment has imperative force: no modifications are permitted in F that do not preserve the truth of P at the end of the modified plan. We will say that we are protecting P at the end of F. Any action that destroys the truth of P will be said to violate P. Relations may be protected at any point in a plan; if a relation is protected at a certain point, that relation must be true when control passes through that point.*

Protection has purposes other than ensuring that simultaneous goals do not interfere with each other: for instance, if an action requires that a certain condition be true before it can be applied, we must protect that condition at the point before the action is taken to see that no modification in the plan can violate it.

In order to ensure that a modification cannot violate any of the protected relations, we check each of these relations to see that it is still true after the proposed modification has been made: otherwise, the modification must be retracted.

In the next section we will examine a very simple example involving two simultaneous goals in order to demonstrate the techniques we have described.

1.1.5 A very simple example

Suppose we have three blocks, A, B, and C, sitting on a table.

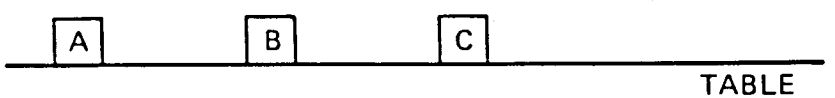

FIGURE 2

*(Sussman, 1973) was the first to use protection in program synthesis, and to apply it to the simultaneous goal problem.

Our goal is to make a tower of the three blocks, with A on top and C on the bottom.

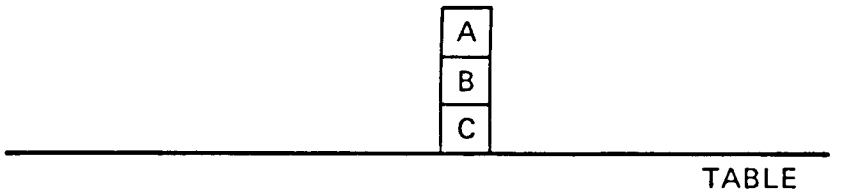

FIGURE 3

We express this goal as a conjunction of two goals. "A is on B" and "B is on C." (We'll forget about saying that C is on the table.) Of course, if we approach these goals in the reverse order we have no problem: we simply put B on top of C and then put A on top of B; no destructive interactions arise. However, if we approach them in the given order we run into a blind alley.

We first attempt to achieve that A is on top of B. In order to do this, we see if A and B are clear (they are), and then we plan to put A on top of B. We have thus planned to achieve our first goal. Because we will now work on another goal to be achieved simultaneously we protect the relation that A is on top of B. We will adopt a notation for representing plans under development in which the left-most column will represent the steps of the plan, the second column will represent the anticipated model or state of the world between the respective plan steps, and the third column will represent any goals that we have yet to achieve, and relations that have already been achieved but must be protected at that point. In this notation our plan so far is as follows:

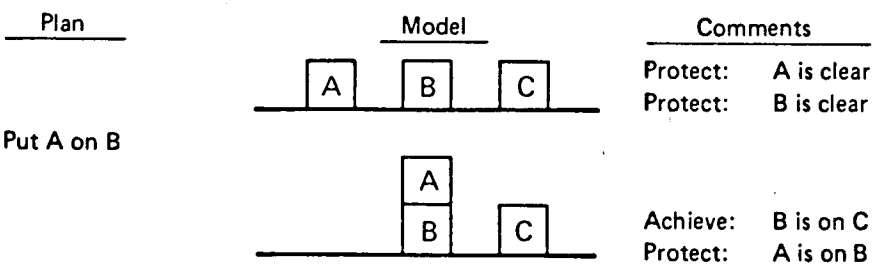

FIGURE 4

In order to put A on top of B we must be sure that A and B are both clear: therefore we have protected these two relations at the point before the action is applied. (Of course, the action itself violates one of the conditions afterwards: we merely want to ensure that the conditions will be true immediately before the action is applied, regardless of what modifications are made to the plan.) We put the goal "Achieve: B is on C" after the plan step and not before because we

are initially attempting to achieve the goal by adding steps to the end of the plan and not the beginning.

Now, since our arm can lift only one block at a time, we will be forced to put A back on the table again in order to get B on top of C. This will violate our protected relation (A is on B) so we cannot hope to achieve our second goal by adding instructions to the end of the plan. But we can still try to pass the goal back over the plan. The goal "B is on C" passed back over the plan "Put A on B" is simply "B is on C" itself, because putting A on B will not alter whether or not B is on C. The plan state so far is as follows:

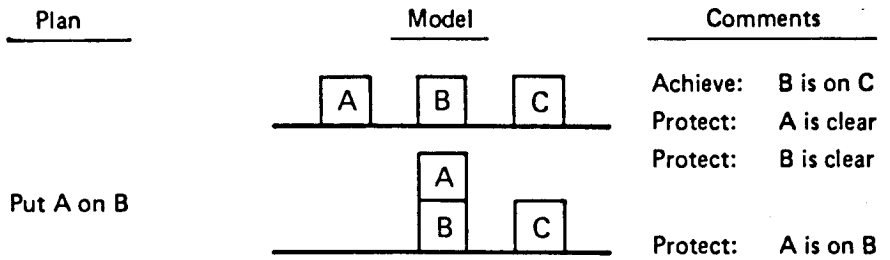

FIGURE 5

The goal "Achieve: B is on C" now occurs before the plan step.

Our goal "B in on C" can now be achieved by simply putting B on C; the appropriate plan step will be added to the beginning of the plan instead of to the end. The resulting plan state is illustrated in Figure 6.

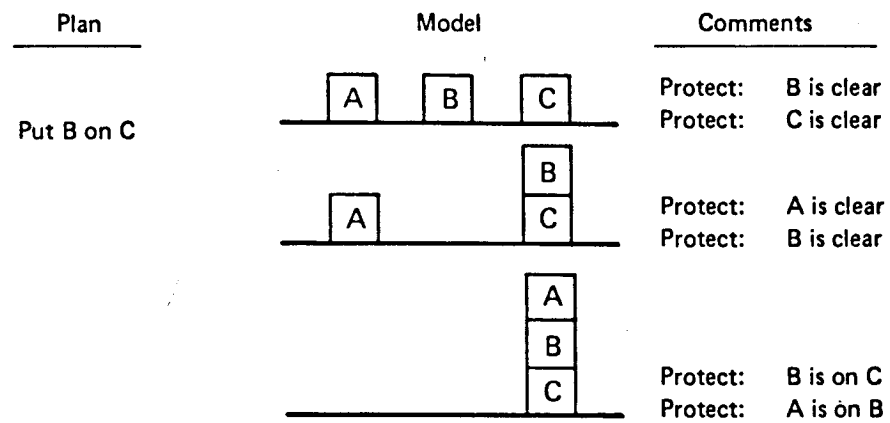

FIGURE 6

Note that the new plan step did not interfere with any of the protected relations: otherwise we would have had to retract the step and find some other solution. As it is, the two-step plan "Put B on C; Put A on B" achieves the desired goal. The method of passing goals back over plan steps has enabled us to avoid backing up and reversing the order in which the goals are approached. This technique will not always prevent such goal reordering; however, we will see that it will allow us to solve some problems that cannot readily be solved, regardless of the order in which the goals are attempted.

The reader may note that the model following the plan step "Put A on B" changes between Figure 5 and Figure 6, because of the insertion of the earlier plan step "Put B on C." It we maintained a model corresponding to each plan step, we would be faced with the task of updating the entire sequence of models following every insertion to reflect the action of the new plan step. This can be an arduous chore if the model is at all large. Instead we maintain only a scanty "skeleton" model that is not affected by an alteration, and generate or "flesh out" other portions of the model as needed, using the same regression method that we introduced earlier as a program modification technique.

1.1.6 Skeleton models

Following each step in the developing plan we have a model, which for our purposes may be regarded as a list of relations that are certain to be true following the execution of that plan step. For instance, following the step "Put A on B" we include in the model the relation "A is on B" and perhaps the relation "A is clear," meaning that no block is on top of A. However, we do not usually include any information about the location of B, for example, because, unless protected, the location of B can be changed by inserting new steps earlier in the plan.

Similarly, after an assignment statement $X \leftarrow t$ we do no generally include the fact that X has value 2 even if we believe that t has value 2 before the statement is executed, because subsequent modifications to the beginning of the program could change the value of t, unless that value is protected. In fact, the model following an assignment statement may be absolutely empty.

In addition to the models that follow each statement in the plan, we have an initial model that describes that situation at the beginning (as given by the problem statement), and we have a global model of the "eternal verities," relations such as x=x, that are unchanged by any action or the passage of time. Information in the global model is implicitly present in all the other models. .

The models that follow each action in the plan are incomplete: much knowledge about the situation is not included explicitly. How are we to compensate for this deficiency?

Suppose that we are given a plan $F_1,...,F_n$, and we need to know whether some relation Q is true after execution of step F_i. We first see if Q is explicitly in the model following F_i; in other words, we see if Q is an immediate consequence of the execution of F_1. If not, we simply pass Q back over the plan step F_i, yielding a perhaps altered relation Q'. We then check if Q' is in the preceding model. The justification for this measure is clear: Q' has been defined as the relation that must be true before the execution of F_i in order that Q will be true afterwards.

If we fail to determine if Q' is true, we pass Q' back over F_{i-1} and repeat the

process until we have passed Q all the way back to the intitial model. If we are still unable to determine whether Q is true we must give up. Even if we determine that Q is true, we must generally resist the temptation to add it to the model that follows F_i: unless Q is protected, later plan alterations could make Q false, and then the model would be inaccurate.

An example: suppose we are given a model in which block A is on C, but blocks A and B both have a clear top.

FIGURE 7

We somehow develop the plan step "Put A on B," and we are led to inquire if C is clear. We cannot determine this from the model that follows "Put A on B," because that model only contains the relations "A is on B" and "A is clear." However, we can pass that relation back over the plan step using a regression rule (as described in Section 1.1.2), leading us to ask if "C is clear or A is on C." Since we know "A is on C" initially, we can conclude "C is clear" in the model following the plan step.

The skeleton model is a technique in which the partial plan that has been constructed is regarded as a central part of the model. Important relationships and the plan itself are in the model explicitly; other relationships may be inferred using the regression rules.

It is traditional in problem solving to distinguish between rules that work backwards from the goal and rules that work forwards from the present state of the world. In Hewitt's (Hewitt, 1972) terminology, these rules are called "consequent theorems" and "antecedent theorems" respectively. Regression rules are a special kind of consequent theorem that can refer explicitly to steps in the plan as well as relations in the model. (Kowalski, 1974) and (Warren, 1974) also discuss the application of regression rules as a modelling technique.

The use of skeleton models means that if a relation P is protected at the end of a plan, no modification can be made at any point in the plan that will not leave P true at the end, because, in checking the truth of P after the modification has been made, we will percolate P back up through the plan, and the unfortunate interaction between P and the new plan step will be discovered.

For instance, suppose a plan step $X \leftarrow Y$ achieves a protected relation P(X), and a new instruction $Y \leftarrow Z$ is inserted at the beginning of the plan, where P(Z) is false. We will try to check that the protected relation P(X) is still true at the end of the modified program. Using regression, we will therefore check if P(Y) is true in the middle of the program, and thus that P(Z) is true at the beginning. Since P(Z) is false, we will detect a protection violation and reject the proposed modification.

This mechanism means that it is necessary to protect a relation only at the point at which we need it to be true. In the previous example, we must protect P(X) after the assignment statement $X \leftarrow Y$, but we need not protect P(Y) before the statement; the latter protection is implicit in the former.

A description of how skeleton models can be implemented using the "context" mechanism of the new artificial intelligence programming languages occurs in Section 2.7.

We have concluded the general description of our approach to simultaneous goals. The balance of Part 1 concerns how this technique has been applied to specific subject domains in order to solve the sample problems.

1.2 Interchanging the values of two variables

1.2.1 Relations that refer to variables

> So first, your memory I'll jog,
> And say: A CAT IS NOT A DOG.
>
> T.S. Eliot, *Old Possum's Book of Practical Cats*

In the next section we will show the synthesis of a more complex program whose specification is represented as a set of simultaneous goals. The subject domain of this program will be variables and their values. However, we must first examine a certain kind of relation more closely: the relation that refers directly to the variable itself, as opposed to its value. For instance, the relation "variable X has value a," written "X:a," refers both to the variable X and its value a. The relations "variable X is identical to variable Y," written "$X \approx Y$," and its negation "variable X is distinct from variable Y," written "$X \not\approx Y$" refer to variables X and Y, but do not refer at all to their values. $X \not\approx Y$ means "X and Y are not identical," and is true regardless of whether X and Y have the same value. Relations such as $\approx$, which do not refer to values at all, are not affected by assignment statements or any program instructions we are going to consider. Relations such as ":" are more complicated. For instance, the relation X:a passed back over the assignment statement $X \leftarrow Y$ yields Y:a, where X and Y are both variables. (A more general rule covers the case in which an arbitrary term plays the role of the variable Y, but we will have no need to consider this case in the following examples.) A more complex situation arises if the variable in the relation is existentially quantified. Such a situation arises if the relation is a goal to find a variable with a certain value. For instance, how do we pass back a goal such as "Find a variable v such that v:a" over the instruction $X \leftarrow Y$? If there is a variable v such that v:a before the assignment statement is executed, and if that variable is distinct from X, then certainly v:a after the execution of $X \leftarrow Y$. Furthermore, if Y:a before the execution, then v can be identical to X as well. Therefore, passing the goal "Find a variable v such that v:a" back over the assignment statement $X \leftarrow Y$ yields

"Find a variable v such that

v≠X and v:a

or v≈X and Y:a."

We will assume the system knows verities such as x≈x, X≠Y, or X≠Z. In the example of the next section we will use one additional fact about the relation ≠: the fact that we can always invent a new variable. In particular, we will assume we can find a variable v such that v≠X by taking v to be the value of a program GENSYM that invents a new symbol every time it is called.

There is, of course, much more to be said about these peculiar relations that refer to variables themselves. They do not follow the usual Floyd-Naur-Manna-King-Hoare rule for the assignment statement. However, the discussion in this section will be enough to carry us through our next example.

1.2.2 The solution to the two variable problem*

> But above and beyond there's still one name left over,
> And that is the name that you never will guess
> The name that no human research can discover–
> But THE CAT HIMSELF KNOWS, and will never confess.
>
> T.S. Eliot, *Old Possum's Book of Practical Cats*

The problem of exchanging the values of two variables is a common beginner's programming example. It is difficult because it requires the use of a "temporary" variable for storage. Part of the interest of this synthesis involves the system itself originating the idea of using a generated variable for temporary storage.

We are given two variables X and Y, whose initial values are a and b; in other words, X:a and Y:b. Our goal is to produce a program that achieves X:b and Y:a simultaneously.

Recall that our strategy when faced with a goal P and Q is to try to form a plan to achieve P, and then to modify that plan to achieve Q as well. Thus our first step is to form a plan to achieve X:b.

For a goal of form X:b we have a technique (Section 1.1.1) that tells us to find a variable v such that v:b and then execute the assignment statement X ← v. We have such a v, namely Y. Therefore, we develop a plan, X ← Y, that achieves X:b. We must now modify this plan to achieve Y:a while protecting the relation X:b that the plan was developed to achieve. In our tabular notation:

Plan	Model	Comments
	X:a Y:b	
X ← Y		
		Achieve: Y:a
	X:b Y:b	Protect: X:b

FIGURE 8

(In our table we record the full model at each stage even though the implementation does not store this model explicitly.

In trying to achieve Y:a we attempt to find a variable v such that v:a. Once we have executed X ← Y, no such variable exists. However, we pass the goal "Find v such that v:a" back over the plan step X ← Y, yielding

Find v such that

v≠X and v:a

or v≈X and Y:a,

as explained in the preceding section. We now attempt to achieve this goal at the beginning of the plan. In tabular form

Plan	Model	Comments
	X:a Y:b	Achieve: Find v such that v≠X and v:a
		or v≈X and Y:a
X ← Y		
	X:b Y:b	Protect: X:b

FIGURE 9

Once the outstanding goal is achieved, we will add an assignment statement Y ← v to the end of the program, where v is the variable that achieves the goal.

If we work on the goals in the given order, we try to find a v such that v≠X. Here we know that GENSYM will give us a new variable name, say G_1, guaranteed to be distinct from X. Our problem is now to achieve the first conjunct, namely G_1:a. But this can easily be achieved by inserting the assignment statement G_1 ← X at the beginning of the plan, since X:a initially. Inserting this instruction does not disturb our protected relation.

We have been trying to find a v satisfying the disjunction

v≠X and v:a

or v≈X and Y:a

We have satisfied the first disjunct, and therefore we can ignore the second. (We

*Another way of approaching this problem is discussed in (Green *et al.*, 1974). Green's system has the concept of temporary variable built in. He uses a convention of inserting a comment whenever information is destroyed, so that a patch can be inserted later in case the destroyed information turns out to be important.

will consider later what happens if we reverse the order in which we approach some of the subgoals.)

We have thus managed to find a v such that v:a at the end of the program, namely $v \approx G_1$. Since our ultimate purpose in finding such a v was to achieve Y:a, we append to our program the assignment statement $Y \leftarrow G_1$. This addition violates no protected relations, and achieves the last of the extant goals. The final program is thus

Plan	Model	Comments
	X:a Y:b	
$G_1 \leftarrow X$		
	X:a Y:b G_1:a	
$X \leftarrow Y$		
	X:b Y:b G_1:a	
$Y \leftarrow G_1$		
		Protect: Y:a
	X:b Y:a G_1:a	Protect: X:b

FIGURE 10

The program has "invented" the concept of "temporary variable" by combining two pieces of already existing knowledge: the fact that GENSYM produces a variable distinct from any given variable, and the rule for passing a goal "Find a v such that v:a" back over an assignment statement. Of course, we could have built in the temporary variable concept itself, and then the solution would have been found more easily. But in this case the invention process is of more interest than the task itself.

Notice that at no point in the construction did we violate a protected relation. This is because of the fortunate order in which we have approached our subgoals. For example, if we had chosen to work on the disjunct

$$v \approx X \text{ and } Y{:}a$$

instead of

$$v \not\approx X \text{ and } v{:}a,$$

we would have inserted the assignment statement $Y \leftarrow X$ at the beginning of the program in order to achieve Y:a, and we would have proposed the program

$$Y \leftarrow X$$
$$X \leftarrow Y$$
$$Y \leftarrow X$$

which violates the protected relation X:b. Other alternative choices in this synthesis are either successful or terminated with equal dispatch.

1.3 Sorting three variables*

1.3.1 Sorting two variables

In our next example we will see how to construct a program to sort the values of three variables. This program will use as a primitive the instruction sort2, which sorts the values of two variables. Before we can proceed with the example, therefore, we must consider how to pass a relation back over the instruction sort2.

Executing sort2(X Y) will leave X and Y unchanged if X is less than or equal to Y ($X \leqslant Y$), but will interchange the values of X and Y otherwise. Let P(X Y) be any relation between the values of X and Y. We must construct a relation P'(X Y) such that if P'(X Y) is true before sorting X and Y, P(X Y) will be true afterwards. Clearly, if $X \leqslant Y$, it suffices to know that P(X Y) itself is true before sorting, because the sorting operation will not change the values. On the other hand, if $Y < X$ it suffices to know P(Y X), the expression derived from P(X Y) by exchanging X and Y, because the values of X and Y will be interchanged by the sorting. Therefore, the relation P'(X Y) is the conjunction

$$\text{if } X \leqslant Y \text{ then } P(X\ Y)$$
$$\text{and if } Y < X \text{ then } P(Y\ X)$$

A similar argument shows that the above P' is as weak as possible. The same line of reasoning applies even if X or Y does not actually occur in P. For instance, if X does not occur, P(Y X) is simply P(X Y) with Y replaced by X.

Given the appropriate definition of sort2, it is straightforward to derive the above relation mechanically (e.g., see [Manna, 1974]). However, that would require the system to know about conditional expressions, and we do not wish to discuss those statements here. For our purposes, it suffices to assume that the system knows explicitly how to pass a relation back over a sort2 instruction.

1.3.2 Achieving an implication

We have excluded the use of conditionals in the programs we construct. However, we cannot afford to exclude the goals of form "if P then Q" from the specifications for the program being constructed. For instance, such specifications can be introduced by passing any relation back over a sort2 instruction.

*This problem is also discussed in (Green, *et al.*, 1974). Green allows the use of program branches and the program he derives has the form of a nested conditional statement. Green's use of the case analysis avoids any protection violations in his solution: the interaction between the subgoals plays a much lesser role in Green's formulation of the problem. Some other work in the synthesis of sort programs (see [Green and Barstow, 1975], [Darlington, 1975]) does not consider "in-place" sorts at all; goal interactions are still important, but protection issues of the type we are considering do not arise. However, Darlington's concept of "pushing in" a function is the analogue of regression for programs in which nested functional terms play the role of sequential program instructions.

The form of these specifications suggests that the forbidden conditional expression be used in achieving them. Therefore, for purposes of this example we will introduce a particularly simple-minded strategy: to achieve a goal of form "if P then Q," first test if P is known to be false: if so, the goal is already achieved. Otherwise, assume P is true and attempt to achieve Q.

The strategy is simple-minded because it does not allow the program being constructed to itself test whether P is true; a more sophisticated strategy would produce a conditional expression, and the resulting program would be more efficient. However, the simple strategy will carry us through our next example.

1.3.3 The solution to the three-sort problem

Given three variables, X, Y, and Z, we want to rearrange their values so that X≤Y and Y≤Z. Either of these goals can be achieved independently, by executing sort2(X Y) or sort2(Y Z) respectively. However, the simple linear strategy of concatenating these two instructions does not work; the program

sort2(X Y)
sort2(Y Z)

will not sort X, Y, and Z if Z is initially the smallest of the three. On the other hand, the simultaneous goal strategy we have introduced does work in a straightforward way.

In order to apply our strategy, we first achieve one of our goals, say X≤Y, using the primitive instruction sort2(X Y). We then try to modify our program to achieve Y≤Z as well. In modifying the program we protect the relation X≤Y. In tabular form, the situation is as follows:

Plan	Model	Comments
sort2(X Y)		
		Achieve: Y≤Z
	X≤Y	Protect: X≤Y

FIGURE 11

As we have pointed out, simply appending a plan step sort2(Y Z) will violate the protected relation X≤Y. Therefore we pass the goal back to see if we can achieve it at an earlier stage. The regressed relation, as explained in the previous section, is

if X≤Y then Y≤Z
and if Y<X then X≤Z.

(This relation effectively states that Z is the largest of the three numbers.) Our situation therefore is as follows:

Plan	Model	Comments
		Achieve: if X≤Y then Y≤Z and if Y<X then X≤Z
sort2(X Y)		
	X≤Y	Protect: X≤Y

FIGURE 12

We must now try to achieve the remaining goal. This goal is itself a conjunction and is handled by the simultaneous goal strategy. The first conjunct, "if X≤Y then Y≤Z," is an implication. Therefore we first test to see if X≤Y might be known to be false, in which case the implication would be true. However, nothing is known about whether X≤Y, so we assume it to be true and resign ourselves to achieving the consequent Y≤Z: this can easily be done using the primitive instruction sort2(Y Z). Inserting this instruction at the beginning of the plan does not interfere with the protected relation X≤Y: the protection point is immediately preceded by the instruction sort2(X Y). Our situation is therefore as follows:

Plan	Model	Comments
sort2(Y Z)		
		Achieve: if Y<X then X≤Z
	Y≤Z	Protect: if X≤Y then Y≤Z
sort2(X Y)		
	X≤Y	Protect: X≤Y

FIGURE 13

(Notice that we do not reproduce the complete model for this example, but only include the skeleton model.)

We have achieved the goal "if X≤Y then Y≤Z," which is one of two simultaneous goals. We therefore protect the relation we have just achieved and attempt to modify the program to achieve the remaining goal, "if Y<X then X≤Z." Again, we cannot disprove Y<X and therefore we attempt to achieve the consequent, X≤Z. This goal can be achieved immediately by executing sort2(X Z), but we must check that none of the protected relations is disturbed. Our situation is

Plan	Model	Comments
sort2(Y Z)		
	Y≤Z	
sort2(X Z)		
	X≤Z	Protect: if X≤Y then Y≤Z
sort2(X Y)		
	X≤Y	Protect: X≤Y

FIGURE 14

The second protected relation X≤Y is still preserved: the first presents us with a bit more difficulty, but is in fact true: a human might notice that Z is the largest of the three numbers at this point. Perhaps it is worth explaining how the system verifies this protected relation, thereby illustrating the use of the skeleton model.

After executing the second instruction sort2(X Z), the only information in skeleton model is that X≤Z. This is not enough to establish that the protected relation is undisturbed. The system therefore passes the relation back to an earlier model and tries to prove it there. The regressed relation is

if X≤Z then (if X≤Y then Y≤Z)
and if Z<X then (if Z≤Y then Y≤X).

The earlier model tells us that Y≤Z [because we have just executed sort2(Y Z)]. The first conjunct is thus easy to prove: the conclusion Y≤Z is known explicitly by the model. The second conjunct follows from transitivity: since we know Y≤Z from the model and Z<X from the hypothesis we can conclude that Y≤X. (This sort of reasoning is performed by a mechansim described in [Waldinger and Levitt, 1974]). The program in Figure 14 is therefore correct as it stands (although additional relationships should be protected if the plan is to undergo further modification).

It is pleasing that this last bit of deduction was not noticed by Manna and Waldinger in preparing the 1974 paper, but was an original discovery of the program, which was implemented afterwards. Manna and Waldinger assumed the protected relation would be violated and went through a somewhat longer process to arrive at an equivalent program. This is one of those not-so-rare cases in which a program debugs its programmer.

In order to show how these ideas apply to robot-type problems we discuss one further example, Sussman's "anomaly," in the next section.

1.4 The Sussman "anomaly"

We include this problem because it has received a good deal of attention in the robot planning literature (e.g., [Sussman, 1973; Warren, 1974; Tate, 1974; Hewitt, 1975; Sacerdoti, 1975]). However, for reasons that we will explore in Part 2, the solution does not exercise the capabilities of the system as fully as the previous two examples. We are given three blocks in the following configuration:

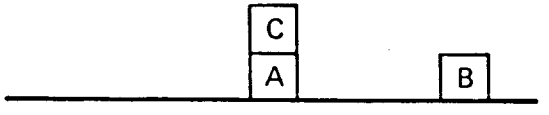

FIGURE 15

*We are asked to rearrange them into this configuration:**

FIGURE 16

The goal is thus a simple conjunction "A is on B and B is on C." (We will forget about the table.)

The anomaly is one of the simplest blocks-world problems for which the linear strategy does not work regardless of the order in which we approach the subgoals: if we clear A and put A on B we cannot put B on C without removing A:

FIGURE 17

(Remember the arm can lift only one block at a time.)

On the other hand, if we put B on C first, we have buried A and cannot put it on top of B without disturbing the other blocks:

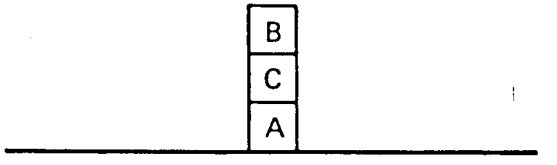

FIGURE 18

Our technique can solve this problem regardless of the order in which it attacks the goals. We will consider just one of these orderings: Assume we attempt to achieve "A is on B." The system will generate subgoals to clear A and B. B is already clear, and A will be cleared by putting C on the table. Then A will be put on B. This much can be done by the elementary strategy for achieving the "on" relationship (Section 1.1.1). Our situation is as follows:

*This problem was proposed by Allan Brown. Perhaps many children thought of it earlier but did not recognize that it was hard.

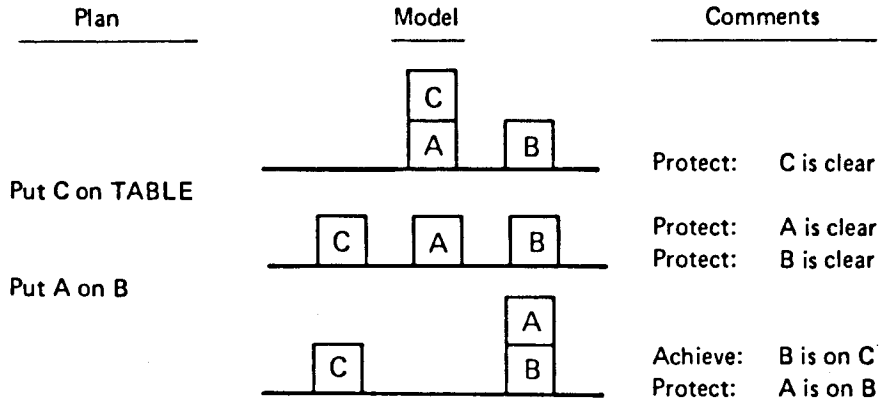

FIGURE 19

We protect "A is on B" because we want to modify the plan to achieve "B is on C" while still achieving "A is on B." We protect "A is clear" and "B is clear" earlier in order to make sure that the operation "Put A on B" will still be legal after the modifications are made.

Now, we have seen that we cannot achieve "B is on C" by adding new steps to the end of the plan without disturbing the protected relation "A is on B." Therefore we again pass the goal back to an earlier stage in the plan, hoping to achieve it before the protected relationship is established.

Passing "B is on C" back over the plan step "Put A on B" yields "B is on C" itself: whether B is on C or not is unaffected by putting A on B. The situation is thus:

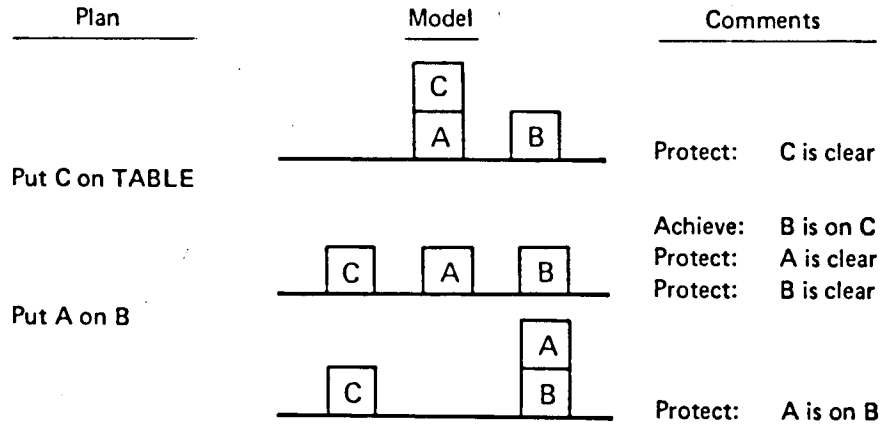

FIGURE 20

The goal "B is on C" can be easily achieved at the earlier stage: B and C are both clear, so we can simply put B on C. Furthermore this operation does not violate any of the protected relations. Since all goals have been achieved, our final plan is as follows:

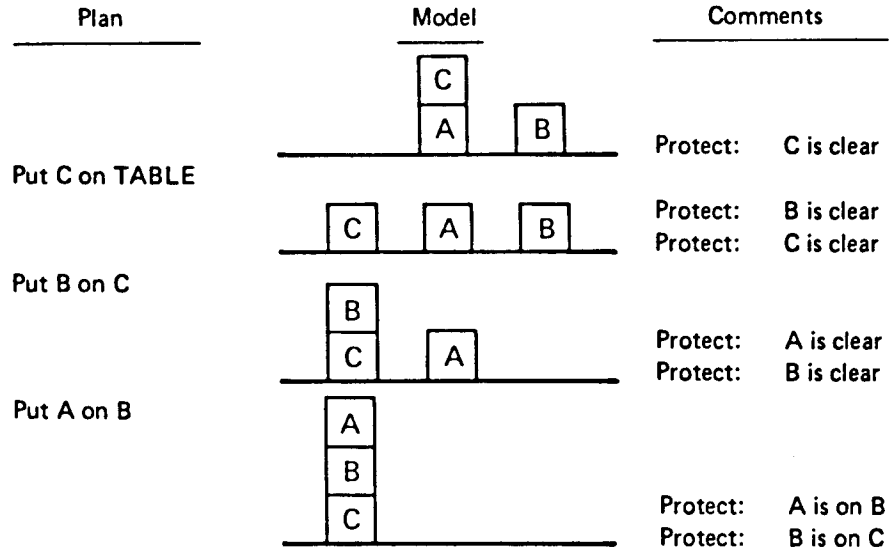

FIGURE 21

The solution is similar if the order in which the goals are attempted is reversed.

This completes the last of our examples. In the next section we discuss some of the limitations of this approach, and consider how they might be transcended.

1.5 Limitations and next steps

> Odin. . .of all powers mightiest far art thou
> Lord over men of Earth, and Gods in heaven,
> Yet even from thee thyself hath been withheld.
> One thing: to undo what thou thyself hast ruled.
>
> Matthew Arnold, *Balder Dead*

The policy maintained by our implementation is to allow no protection violations at all: if a proposed modification causes a violation, that modification is rejected. This policy is a bit rigid and can sometimes inhibit the search for a solution.

For instance, consider the blocks problem in which initially the blocks are as follows:

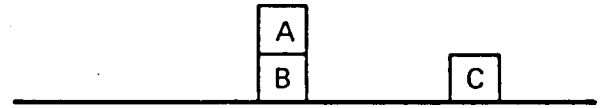

FIGURE 22

and in which the goal is to construct the following stack:

FIGURE 23

The goal may be considered to be the conjunction of two goals, "A is on B" and "B is on C." If these goals are approached in the reverse order, the system has no problem: it clears B by putting A on the table, puts B on C and then puts A on B. However, if the system approaches the goals in the given order, it will attempt to achieve "A is on B" first. This relation is already true, so the system protects it while trying to achieve the goal "B is on C." Here the system is baffled: it cannot put B on C without clearing B, thereby violating the protected relation. Passing the goal backwards into the plan is of no use: there are no plan steps to back it over. Clearly we would like to relax the restriction against protection violation until B is safely on C, and then reachieve the relation "A is on B," but our policy does not permit such a maneuver. The system is forced to reorder the goals in order to find a solution.

The restriction against violating protected relations also lengthens the search in generating the program to sort three variables. If these violations were permitted, a correct program

```
sort2(X Y)
sort2(Y Z)
sort2(X Y)
```

could be constructed without the use of regression at all. Why not permit violations, under the condition that a "contract" is maintained to reachieve protected relations that have been violated?

Indeed, such a strategy is quite natural, but we have two objections to it. First, suppose in the course of reachieving one protected relation we violate another. Are we to reachieve that relation later as well, and so on, perhaps ad infinitum? For example, in searching for a plan to reverse the contents of two variables it is possible to generate the infinite sequence of plans

```
X ← Y,
Y ← X
X ← Y,
X ← Y
Y ← X
X ← Y,
Y ← X
X ← Y
Y ← X
X ← Y, . . .
```

Each plan corrects a protection violation perpetrated by the previous plan–but commits an equally heinous violation itself. (This objection is a bit naive: one could invent safeguards against such aberrations, as has been done by Sussman (Sussman, 1973) and Green et al. (Green et al., 1974).

The second objection: allowing temporary protection violations can result in inefficient plans. For example, we could generate the following plan for solving the Sussman anomaly:

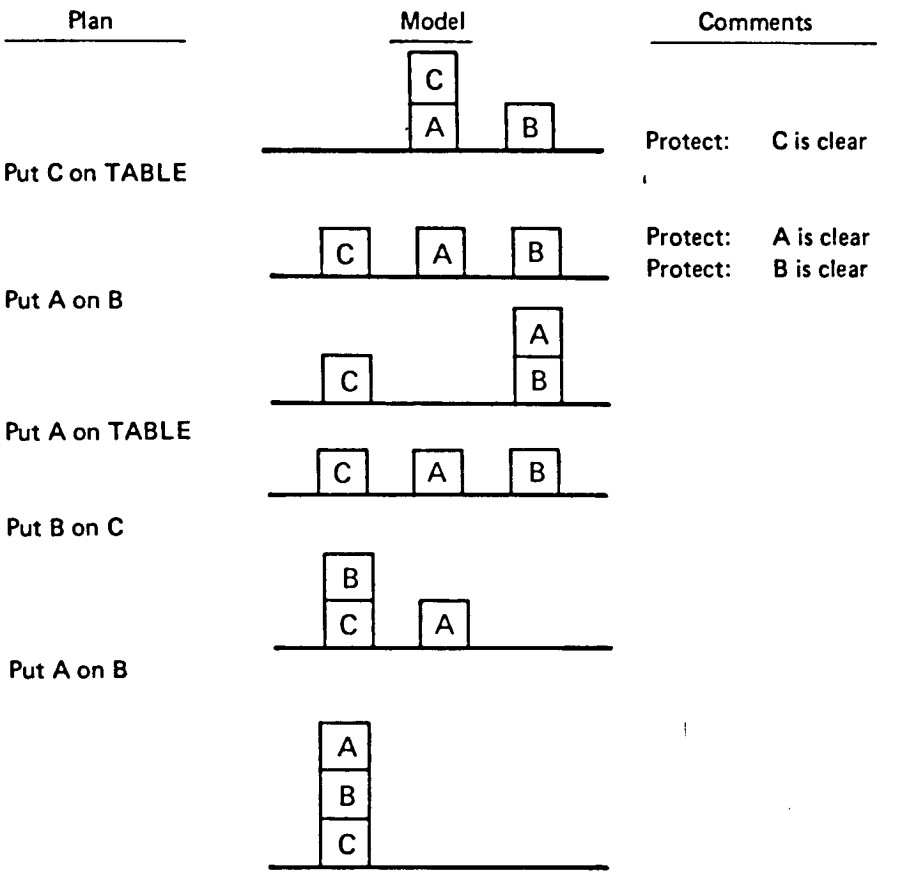

FIGURE 24

This plan is correct but inefficient: We have put A on B only to put A back on the table again because a protection violation was temporarily admitted. In a similar way, Sussman's HACKER produces an equally inefficient plan, approaching the goals in the opposite order. Of course, the plans could later be optimized, but allowing protection violations seems to encourage inefficiency in the plan produced.

Nevertheless, we feel that permitting temporary protection violations in a controlled way is a natural strategy that may be admitted in future versions of the program.

A more serious limitation of our implementation is that the only way it can modify plans is by adding new instructions, never by changing instructions that are already there. For example, suppose we have the initial configuration

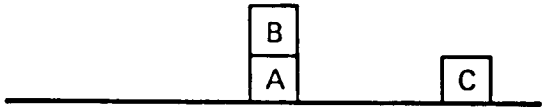

FIGURE 25

and our goal is to construct the stack

FIGURE 26

Assuming we approach the goal "A is on B" first, we are quite likely to put B on the table and then put A on B. In modifying the plan to achieve "B is on C," it would be clever to plan to put B on C instead of the table, but this sort of modification is beyong the system's capabilities. The "formal object" approach of Sussman (Sussman, 1973) would handle this properly: there, the decision about where to put B (in clearing A) would be deferred until we attempted the second goal "B is on C." However, other sorts of modifications require achieving the same goals in entirely different ways in order to accomodate the demands of the additional specification. Certain protected relations might never be achieved at all in the modified program if the higher level goal that constituted the "purpose" of the protected subgoal could be achieved in some other way. To effect such modifications will require that in the course of modifying a program we retain some of the goal-subgoal hierarchy that caused the original program to be constructed. Such modifications are in the spirit of our approach, but beyond the capabilities of our simple implementation.

The plans we have constructed in our paper are "straight-line" programs; they involve no loops or branches. The system as it exists contains a subsystem for constructing programs with branches and recursive loops (cf. [Manna and Waldinger, 1974]); however, these programs are free of side effects. Since the mechanisms for loop branch construction have not been integrated with the system that constructs structure-altering programs of the sort we have discussed in this paper. Nevertheless, these mechanisms are entirely consistent, and we intend to unite them. Our hand simulations indicate that the system will then be able to construct a variety of array-sorting routines.

The use of goal regression for these more complex programs has been studied by many as a way of proving programs correct. Passing relations back into branches is straightforward (Floyd, 1967, Hoare 1969); passing a relation back into a loop, on the other hand, may require ingenuity to generalize the relation. This problem is discussed by (Katz and Manna, 1973; Wegbreit, 1974; Boyer and Moore, 1973; Moore, 1975) and others, but it is by no means "solved."

All the loops constructed by our synthesizer will initially be recursive: we intend to introduce iteration only during a subsequent optimization phase, following (Darlington and Burstall, 1973).

The way we have implemented skeleton modelling may be remarkably inefficient, particularly if the plan being constructed is to have many steps. It may take a long time to pass a relation back so far, and the transformed relation may grow alarmingly. There are many ways one might consider to make skeleton modelling more efficient. We prefer not to speculate on which of these ways will actually help until we have tried to implement some of them.

We regard program modification as a valuable synthesis technique apart from its role in achieving goals simultaneously. Often we can construct a program by modifying another program that achieves a goal that is somehow similar or analogous. For instance, in (Manna and Waldinger, 1974) we show how a unification algorithm could be constructed by modifying a pattern matcher. Another sort of program modification is optimization: here we try to modify the program to achieve the same goal more efficiently. It is our hope that systems with the ability to modify their own programs will be able to adapt to new situations without needing to be "general." Before that can happen, however, program modification techniques must be developed beyond what has been done here.

This concludes our discussion of the simultaneous goal strategy. In the next part of this paper we discuss how some other problem solvers have approached some of the same problems.

PART 2

THE REPRESENTATION OF ACTIONS AND SITUATIONS IN CONTEMPORARY PROBLEM SOLVING

Time present and time past
Are both perhaps present in time future,
And time future contained in time past.
If all time is eternally present
All time is unredeemable.

T.S. Eliot, *Four Quartets*

In the rest of this paper we will examine a number of problem-solving systems, asking the same question of each system: how are actions and their effects on the world represented? Thus we will not emphasize simultaneous goals

in this section, and in discussing a system we will often ignore the very facets that make it unusual. Many of these systems approach problems of far greater complexity than those we have addressed in Part 2, problems involved in manipulating many more objects, and more complex structures. When we compare our approach to theirs, please bear in mind that our implementation has not been extended to handle the problems that our hand simulation dispatches with such ease.

2.1 The classical problem solvers

In the General Problem Solver (GPS) (see [Newell, Shaw, and Simon, 1960]), the various states of the world were completely independent. For each state, GPS had to construct a new model: no information from one state was assumed to carry through to the next automatically, and it was the responsibility of each "operator" (the description of an action) to tell how to construct a new model. The form of the states themselves was not dictated by GPS and varied from one domain to another.

The resolution-based problem solvers (e.g., [Green, 1969; Waldinger and Lee, 1969]) maintained the GPS convention that every action was assumed capable of destroying any relation: in other words it was necessary to state explicitly such observations as that turning on a light switch does not alter the location of any of the objects in a room. To supply a large number of these facts (often called "frame axioms") was tedious, and they tended to distract the problem solver as well. Since most actions leave most of the world unchanged, we want our representation of the world to be biased to expect actions not to affect most existing relations. For a number of reasons we demand that these "obvious" facts be submerged in the representation, so that we (and our system) can focus our attention on the important things, the things that change.

The STRIPS problem solver (Fikes and Nilsson, 1971) was introduced to overcome these obstacles. In order to eliminate the frame axioms, STRIPS adopted the assumption that a given relation is left unchanged by an action unless it is explicitly mentioned in the "addlist" or the "deletelist" of the action: relations in the addlist are always true after the action is performed, while relations in the deletelist are not assumed to be true afterwards even if they were true before. Thus the frame axioms are assumed implictly for every action and relation unless the relation is included in the addlist or deletelist of the action. For instance, a (robot) action "go from A to B" might have "the robot is at B" in its addlist and "the robot is at A" in its deletelist. A relation such as "box C is in room 1" would be assumed to be unaffected by the action because it is not mentioned in either the addlist or the deletelist of the operator.

Henceforth, we shall refer to the belief that an action leaves all the relations in the model unchanged, unless specified otherwise, as the "STRIPS assumption."

A STRIPS model of a world situation, like a STRIPS operator, consists of an addlist and a deletelist: the addlist contains those relations that are true in the *corresponding situation but that may not have been true in the initial situation, and the deletelist contains those relations that may not be true in the corresponding situation even though they were true initially.* Thus one can determine which relations are true, given the current model and the intial list of relations. Also, given a model and an operator, it is easy to apply the operator to the model and derive a new model. The STRIPS scheme keeps a complete record of all the past states of the system, while allowing the various models to share quite a bit of structure.

STRIPS operators are appealingly simple. In the next section we will examine how the sorts of techniques we have discussed apply if the actions are all STRIPS operators.

2.2 Regression and STRIPS operators

Suppose an action is represented as a STRIPS operator, and that the members of the addlist and the deletelist are all atomic—that is, they contain no logical connectives or quantifiers. It is singularly simple to pass a relation back over such an operator, because the interaction between the operator and the relation are completely specified by the addlist and the deletelist. In order for a relation to be true after the application of such an operator, it must (1) belong to the addlist of the operator, or else (2) be true before application of the operator and not belong to the deletelist of the operator. Thus the rule for passing any relation back over such a STRIPS operator is implicit in the operator description itself.

For instance, an operator such as "move A from B to C" might have addlist "A is on C" and "B is clear" and deletelist "A is on B" and "C is clear." Thus, when passed back over this operator, the relation "A is on C" becomes true, "A is on B" becomes false, and "C is on D" remains the same. The simplicity of regression in this case indicates that we should express our actions in this form whenever possible.

The problem-solver WARPLAN (Warren, 1974) uses precisely the same sort of skeleton model as we do, and uses an identical strategy for handling simultaneous goals, but restricts itself to an atomic add-deletelist representation for operators, thus achieving a marvelous simplicity. Although we imagine that WARPLAN would require extension before it could handle the sort problem or the interchanging of variable values, the principles involved in the WARPLAN design are a special case of those given here.

Thus the clarity of actions expressed in this form makes reasoning about them exceedingly easy. However, many have found the add-deletelist format for representing actions too restrictive. With the advent of the "artificial intelligence programming languages," it became more fashionable to represent actions "procedurally" so that the system designer could describe the effects of the action using the full power of a programming language. We shall examine the impact of the STRIPS assumption on some of these systems in the next section.

2.3 The use of contexts to represent a changing world

What is past, even the fool knows.

Homer, *Iliad*

The new AI languages include PLANNER (Hewitt, 1972), QA4 (Rulifson, *et al.*, 1972). CONNIVER (McDermott and Sussman, 1972) and QLISP (Wilber, 1976), a variant of QA4. A comparative survey of these languages is provided in (Bobrow and Raphael, 1974). Implementers of problem solvers in these languages are fond of saying their systems represent actions "procedurally," as computer programs, rather than "declaratively," as axioms or add-delete lists. Yet in each of these systems the STRIPS assumption is firmly embedded, and the procedures attempt to maintain an updated model by deleting some relations and adding others; which relations an action adds or deletes depends on a computation instead of being explicitly listed beforehand. The STRIPS assumption is expressed not procedurally or declaratively but structurally: it is built into the choice of representation. The more primitive systems (e.g., [Winograd, 1971; Buchanan and Luckham, 1974]), implemented in an early version of PLANNER, maintained a single model which they updated by adding and deleting relations.* This scheme made it impossible for the system to recall any but the most recent world situation without "backtracking," passing control back to an earlier state and effectively undoing any intermediate side effects. The more recent trend† has been to incorporate the assumption by a particular use of the "context" mechanism of the newer implementation languages. We must now describe the context mechanism and its use in building what we will call an "archeological model."

The context mechanism in QA4, CONNIVER, QLISP, AP/1, and HBASE operates roughly as follows: Each of these systems has a data base; assertions can be made and subsequently retrieved. Assertions and queries in these systems are always made with respect to an implicit or explicit context. If T_1 is a context, and we assert that B is on C with respect to T_1, the system will store that fact and answer accordingly to queries made with respect to T_1. There is an operation known as "pushing" a context that produces a new context, an immediate "descendant" of the original "parent" context. We may push T_1 any number of times, each time getting a new immediate descendant of T_1. If T_2 is a descendant of T_1, any assertion made with respect to T_1 will be available to queries made with respect to T_2.

*We do not mean to imply that all these systems were copying STRIPS; Winograd's work was done at the same time.

†See, for example, (Derksen, *et al.*, 1972; Sussman, 1973; Fahlman, 1974; McDermott, 1974; Fikes, 1975). (Balzer, *et al.*, 1974 and Tate, 1974) use the context mechanism of the AP/1 programming system and the HBASE data base system (Barrow, 1974), respectively, in exactly the same way.

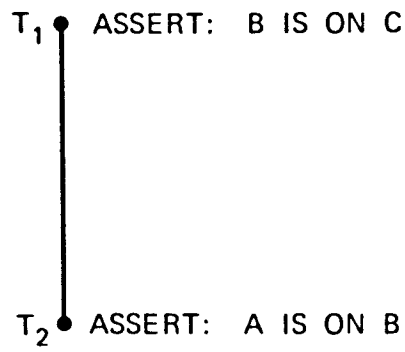

FIGURE 27

Thus if we ask whether B is on C with respect to T_2, we will be told "Yes" (in some fashion). However, assertions made with respect to that descendant are "invisible" to queries made with respect to its parent or any other context aside from its own descendants. For instance, if A is asserted to be on B with respect to T_2, that information will not be available to queries made with respect to T_1 (see Figure 27).

It is also possible to "delete" a relation with respect to a given context. If I delete the fact that B is on C with respect to T_2, the system will be unable to determine whether B is on C with respect to T_2 (on any of its descendants), but it will still know that B is on C with respect to T_1:

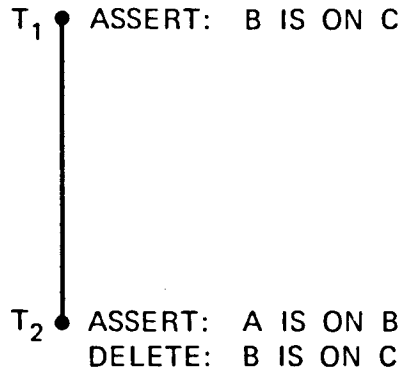

FIGURE 28

The convention taken in planning systems implemented in languages with such a "context-structured data base" has been to equate each situation with a context. Furthermore, if some action occurs in a given situation T_1, resulting in a new situation, the usual practice has been to equate the new situation with an immediate descendant T_2 of the given context T_1. Any relations that are produced by the action are asserted with respect to T_2; any relations that may be

disturbed by the action are deleted with respect to T_2. Other relations are still accessible in the new context. Thus if we are in situation T_1 and move block A onto block B from on top of block C, we construct a descendant T_2, asserting that A is on B and deleting that A is on C with respect to T_2. If B was known to be on block D in situation T_1, that information will still be available in situation T_2.

If T_2 is succeeded by another situation T_3, T_3 will be represented by a descendant of T_2, and so on. The structure of the sequence of contexts is represented as

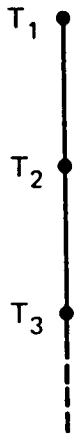

FIGURE 29

Each context is a descendant of the preceding context.

We will call this representation of the world an "archeological" model because it allows us to dig into successive layers of context in order to uncover the past.

In the balance of this paper we will propose that the archeological model is not always ideal. Because any assertion true in a context is automatically true in its descendants (unless specifically deleted), the use of archaeological models implicitly incorporates the STRIPS assumption, and accepts the STRIPS solution to the frame problem. Therefore, most of the planning systems implemented in the new AI languages use representations like that of STRIPS. We have been paying so much attention to the STRIPS assumption for the following reason: we are about to argue that in the future we may not want this assumption so firmly implanted in the structure of our problem solvers; indeed, some researchers have already begun to feel its constriction.

2.4 Influential actions

For want of a nail the shoe was lost,
For want of a shoe the horse was lost,
For want of a horse the rider was lost,
For want of a rider the battle was lost,
For want of a battle the kingdom was lost,
And all for the want of a horseshoe nail. Nursery Rhyme

*The STRIPS assumption, embedded in the archeological model, has been so universally adopted because it banishes the frame axiom nightmare: it is no longer necessary to mention when an action leaves a relation unaffected because every action is assumed to leave every relation unaffected unless explictly stated otherwise. The assumption reflects our intuition about the world, and the archeological model represents the assumption in an efficient way. Having found a mechanism that rids us of the headaches of previous generations of artificial intelligence researchers, shouldn't we swear to honor and cherish it forever?

Indeed, so much can be done within the STRIPS-archeological model framework, and so great are the advantages of staying within its boundaries, that we only abandon it with the greatest reluctance. If we were only modelling robot acts we might still be content to update our models by deleting some relations and adding others. The death blow to this approach is dealt by programming language instructions such as the assignment statement.

Suppose we attempt to express an assignment statement $X \leftarrow Y$ by updating an archeological model. We must delete any relation of form P(X); furthermore, for every relation of form P(Y) in the model we must add a relation of form P(X). In addition, we may need to delete a relation of form "there is a z such that z has value b" even though it does not mention X explicitly. We may need to examine each relation in the model in order to determine whether it depends on X maintaining its old value. The consequences of this instruction on a model are so drastic and far reaching that we cannot afford to delete all the relations that the statement has made false.

How are we to represent the effects of an instruction such as sort2(X Y) on a model? If P(X Y) is the conjunction of everything that is known about X or Y, we might delete P(X Y) and assert $X \leqslant Y$ and (P(X Y) or P(Y X)). This is a massive and unworkable formula if P(X Y) is at all complex; furthermore, it does not express our intuition about the sort, that whether P(X Y) or P(Y X) holds depends on whether or not X was less than or equal to Y before the sort took place. Knowledge of the previous relation between X and Y has been lost.*

Even in the robot domain, for which the STRIPS formalism was orginated, the archeological representation becomes awkward when considering actions with indirect side effects. For example, if a robot is permitted to push more than once box at a time, an operation such as "move box A to point x" can influence the locations of boxes B, C, and D.

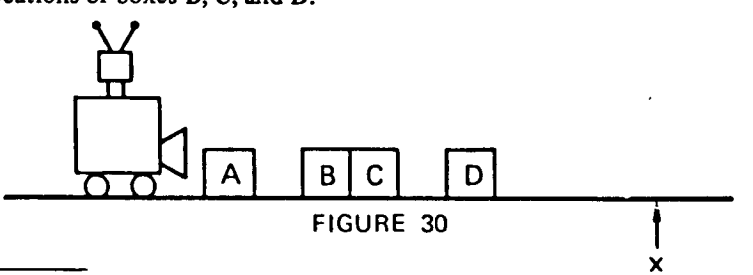

FIGURE 30

*A reply to some of this criticism appears in (Warren, 1976).

This situation becomes worse as the number of elements in the world increases: in moving a complex subassembly of a piece of equipment, we must change the location of every component of the subassembly. If we turn a subassembly upside down, we must replace every relation of form "x is on y" by the relation "y is on x," if x and y are components of the inverted assembly.

These actions are clumsy to model archeologically because so many relations need to be added and deleted from the model, and these relations may involve objects that are not explicitly mentioned by the operator. Furthermore, the operators are insensitive to whether or not these relations are relevant to the problem being solved.

Many of the more recent planning and modelling systems have been attempting to represent these "influential" actions, and we will soon examine how they have overcome the above obstacles. First let us point out that regression provides one technique for modelling these actions; for instance, we need not determine the location of any component indirectly affected by an action until a query concerning that component arises: thus, though many components may be moved, the system need only be concerned with a few of them. When a query about the location does arise, the regression technique will allow the new location to be determined from the original location and from the sequence of actions that has been performed on the subassembly. In particular, if the robot in the previous example (Figure 30) has moved the stack 10 feet to the right in moving box A to point x, the new location of box C will also be several feet to the right of the old location: of course, there is no need to compute the new location of C unless that information is requested.

We have seen that archeological models embed the STRIPS assumption; however, many of the more recent planning systems, while retaining the archeological structure, have been attempting to model actions that must be classified as influential. We will see in the next section how they have resolved the discrepancy.

2.5 Escaping from the STRIPS assumption

Once the archeological model was adopted, the designers of problem solvers devised mechanisms to loosen the STRIPS assumption embedded in their choice of representation.

Fahlman (Fahlman, 1974), using CONNIVER, wanted to simulate a robot that could lift and transport an entire stack or assembly of blocks in one step by carefully raising and moving the bottom block. We characterize this action as "influential" because many blocks will have their location changed when the bottom block is moved. Aware of the difficulty of maintaining a completely updated model, Fahlman distinguishes between "primary" and "secondary" relations. Primary relationships, such as the locations of the blocks, are fundamental to the description of the scene: an updated model is kept of all primary relationships. Secondary relationships, such as whether or not two blocks are touching, are defined in terms of the primary relationships and therefore can be deduced from the model, and added to it, only as needed. The system has thereby avoided deducing large quantities of irrelevant, redundant secondary relationships.

Notice, however, that keeping an updated model of just the primary relationships may still be a sizable chore: for instance, at any moment the system must know the location of every block in the model, even though these locations are often themselves redundant; when a large subassembly is moved, the locations of each of the blocks in the subassembly can be computed from the location of the subassembly itself.

Furthermore, in Fahlman's system if a primary relationship is changed, all the secondary relationships that have been derived from that primary relationship and added to the model must be deleted at once to avoid potential inconsistency.

The modelling system of the SRI Computer Based Consultant (Fikes, 1975), implemented in QLISP, distinguishes between derived and explicitly asserted relations for the same reason that Fahlman distinguishes between primary and secondary data. However, in the SRI system the same relation might be derived in one instance and explictly asserted in another. Thus the location of a component could very well be derived from the location of a subassembly.

Like the Fahlman system, the SRI system deletes all the information derived from an assertion when it deletes the assertion itself.

Note that the SRI system does not behave at all well if the user tries to assert a complex relationship explicitly, say in a problem description. For instance, suppose the user says that block B is between blocks A and C. If the system then moves block A, it will still report that B is between A and C, because that relationship was explicitly asserted and not derived: the system has no way of knowing that it depends on the location of A.

The Fahlman system avoids this difficulty only by forbidding the user to assert any secondary relationships.

Both the Fikes and the Fahlman systems have the following scheme: define actions in terms of the important relationships that they modify, and then define the lesser relationships in terms of the important relationships. This simplifies the description of actions, makes model updating more efficient, and allows the system designer to introduce new relationships without needing to modify the actions' descriptions.

However, it may be impossible to define some lesser relationships in terms of the important ones; we may need to know directly how the lesser relationships are affected by actions. The moving of subassemblies provides a convenient example of this phenomenon.

Consider a row of blocks on a table.

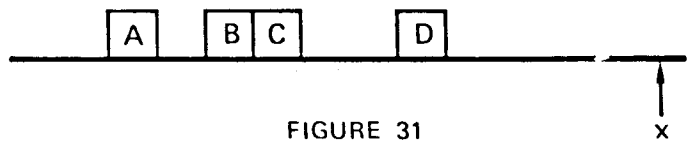

FIGURE 31

We want to move A several feet to the right, to point x. We can either slide A or lift it. If we lift it, blocks B, C, and D will stay where they are, whereas if we slide it, we will inadvertently carry the others along. It is expensive to expect the slide operator to update the model to include the new locations of all the blocks it affects: there may be many of these intermediate blocks and they may not be important to the problem being solved. On the other hand, we cannot expect an archeological system to deduce the new location of B from the new model in case that information turns out to be needed: in order to compute the location of B, the system needs to know whether A has been lifted or slid, and that information is not part of a conventional model. Thus, in an archeological model, locations of intermediate blocks must always be computed at the time the slide is added to the plan.

If skeleton models are adopted, on the other hand, the actions in the plan form an integral part of the model. If A is slid to x, only the new location of A would be explicitly included in the new model. If subsequently we need to determine the location of B, a regression rule sees that A has been slid and asks whether B is in the path of the slide; if not, the location of B after the slide is the same as before; otherwise, the new location of B is somewhere to the right of A.

In both the archeological and the skeletal representations, knowledge about the side effects of sliding must be explicitly expressed. In the skeleton model, the new locations of the intermediate blocks need not be computed until they are needed.

In archeological modelling, the description of an action must be expressed completely in a single operator. For an action with many side effects, the operator is likely to be a rather large and opaque program. Skeleton modelling does not eliminate the need to describe the effects of an action explicitly; however, it does allow the description to be spread over many smaller, and usually clearer programs. Furthermore, one can alter a system to handle new relations merely by adding new regression rules, without changing any previously defined operators. In short, skeleton modelling can sometimes make a system more transparent and modular, as well as more efficient.

Skeleton models do not discard the STRIPS assumption. If this assumption were abandoned, the frame problem would be back upon us at once: for every relation and action it would be necessary to state or deduce a regression rule whether or not the action had any effect at all on the relation. Instead, skeleton models contain a default rule stating that if no other regression rule applies, a given relation is assumed to be left unchanged by a given action. This rule states the STRIPS assumption precisely but does not freeze it into a structure. We have lost in efficiency if actions really do have few side effects, because the archeological model does embed the STRIPS assumption in a structural way and requires no computation if it applies, whereas a skeleton model can only apply the assumption after all the regression rules have failed. The extent to which this modelling technique will be economic depends entirely on the "influence" of actions of the plan–the degree to which they affect the relations in the model.

If skeleton models are adopted, the context mechanism need not be dropped altogether as a way of representing distinct world situations; however, descendent contexts cannot be used to represent successive world states. Our implementation of skeleton models uses contexts in a different way, which we will outline in the next section.

2.6 The use of contexts to implement skeleton models

Recall that we can "push" a given context any number of times, creating a new immediate descendant with every push. These new contexts are independent from each other–none of them is descended from any of the others, and an assertion made with respect to one of them will be invisible to the rest.

In our implementation of skeleton models we represent each situation by a context, but successive situations are all immediate descendents of a single global context T. Thus if situation T_2 results from situation T_1 by performing some act, T_1 and T_2 will both be immediate descendants of T, created by pushing T; T_2 will not be a descendant of T_1. We can represent the skeleton model context structure as follows:

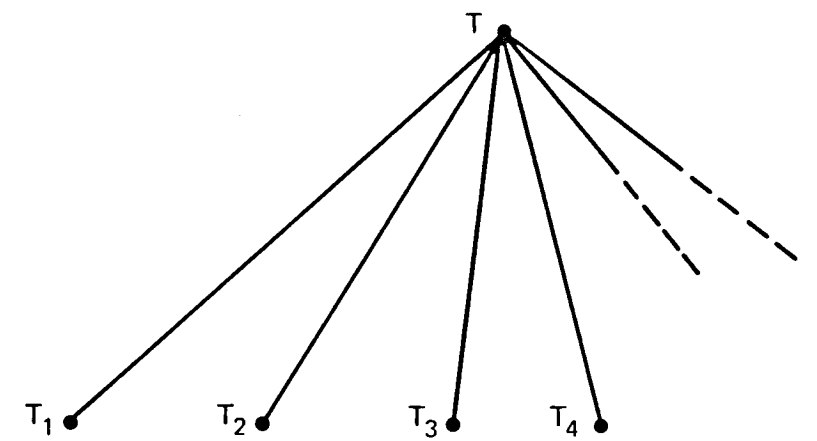

FIGURE 32

Asserting a relation with respect to T_1 does not automatically make it true with respect to T_2, and so on. The only relations asserted in the global context T are the eternal verities.

Since the structure of the skeleton model does not imply any relationship at all between successive states, we represent such knowledge procedurally, by the regression rules for passing a relation back from one state to the preceding one. We suffer a possible loss of efficiency in abandoning the archeological model, but we gain in flexibility and in our ability to represent influential operators efficiently. We do not need to struggle against the assumption incorporated into our representation.

Of course, it is possible to implement skeleton models without using a context mechanism. Problem solvers of the sort advocated by Kowalski (Kowalski, 1974) or implemented by Warren (Warren, 1974) embed a skeleton model representation in a predicate logic formalism in which states of the world are represented by explicit state variables, just as in the early theorem-proving approach. These systems are especially elegant in that the regression rules are indistinguishable from the operator descriptions. They both accept the STRIPS add-deletelist operator representation, but we can envision their incorporating the sort of regression we have employed without requiring any fundamental changes in structure. Hewitt (Hewitt, 1975) has indicated that a version of what we have called skeleton modelling has also been developed independently in the actor formalism, and Sacerdoti (Sacerdoti, 1975) uses another version in conjunction with the procedural net approach.

2.7 Hypothetical worlds

> What might have been is an abstraction
> Remaining a perpetual possibility
> Only in a world of speculation.
> What might have been and what has been
> Point to one end, which is always present.
> Footfalls echo in the memory
> Down the passage which we did not take
> Towards the door we never opened
> Into the rose-garden.
>
> T.S. Eliot, *Four Quartets*

Although so far we have avoided discussing the formation of conditional plans in this paper, it may now be useful to note that using descendent contexts to split into alternate hypothetical worlds (cf. [Rulifson, *et al.*, 1972; McDermott, 1974; Manna and Waldinger, 1974]) is entirely consistent with using independent contexts in skeleton models, but presents something of a problem to archeological models.

In both archeological and skeletal models it is common to represent hypothetical worlds by descendent contexts. For instance, to prepare alternate plans depending on whether or not it is raining in a situation represented by context T_1, two new contexts T_1' and T_1'' are formed, corresponding to the cases in which it is raining and it is not raining, respectively. T_1' and T_1'' are both descendants of T_1, so that any relations known in T_1 will automatically be assumed about T_1' and T_1'' also, as one would have hoped. Furthermore, in T_1' it is asserted to be raining, while in T_1'' it is asserted not to be raining.

The plan for the rainy case would be represented as a sequence of contexts that follows T_1'. In an archeological model these would be successive descendants of T_1' (Figure 33), while in a skeleton model these would be independent contexts linked by regression rules. A similar sequence of contexts beginning with T_1'' would correspond to the plan for the case in which it is not rainy.

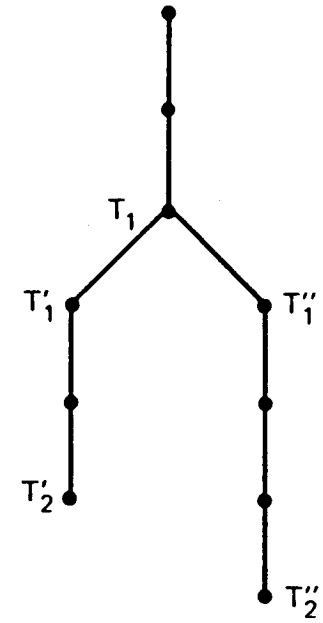

FIGURE 33

Eventually we may reach a situation T_2' and T_2'' in each plan, respectively, after which it becomes irrelevant whether or not it was raining in T_1. In other words our ultimate goal may now be achieved by a single plan that will work in either T_2' or T_2''. Therefore we would like to join our two plans back together into a single plan; we want to form a new context T_2 such that P is true in T_2 if and only if it is true in both T_2' and T_2''. This can be done in a skeleton model by creating an independent context T_2 linked to the previous contexts by the following regression rule: to establish R in T_2, establish R in both T_2' and T_2''.

The situation becomes more difficult if one attempts to maintain an updated archeological model. One could take the following approach: if P and Q are the conjunction of all that is known in T_2' and T_2'', respectively, then assert (P or Q) with respect to T_2. However, (P or Q) is likely to be an unwieldy formula, and we may have lost the information that P corresponds to the rainy situation and Q to the nonrainy one.

We regret that our treatment of hypothetical situations is so terse. A discussion of our own approach, with examples, is given in (Manna and Waldinger, 1974).

2.8 Complexity

> Home is where one starts from. As we grow older
> The world becomes stranger, the pattern more complicated
> Of dead and living.
>
> T.S. Eliot, *Four Quartets*

Perhaps we should say a few words contrasting the work reported here with recent work of Sussman (Sussman, 1973) and Sacerdoti (Sacerdoti, 1975). Although both of these works deal in some of their aspects with simultaneous goals, the principal thrust of their interests is different from ours, and so comparisons are likely to be shallow.

Sussman's main interest is the acquisition of knowledge. Thus he wants his system to learn how to handle simultaneous goals, and is more concerned with learning than with simultaneous goals themselves. We, on the other hand, want our system to know how to handle simultaneous goals from the start, and are not (at present) concerned with learning at all.

The sort of program modification we do is distinct from debugging: the program we are modifying correctly achieves one goal, and we want it to achieve another. We also refrain from actually executing our programs, and ultimately produce programs that are guaranteed correct, whereas Sussman produces programs that may have undiscovered bugs. It is plausible that in tackling more complex problems we will want to introduce bugs and later correct them. We imagine this happening in problems involving serveral levels of detail: a program may work correctly in a crude way, but still contain many minor errors. The problems we have been considering are simple enough so that we have not been forced into using these techniques.

Similarly we view Sacerdoti's procedural nets, like his earlier abstraction hierarchies (Sacerdoti, 1974) as a way of dealing with complexity by submerging detail until a grossly correct plan has been developed. Then the plan is examined in greater depth, and difficulties are ironed out as they emerge. The Sacerdoti formalism can easily represent actions with many subsidiary side effects: these effects are considered only after the initial (approximate) plan has been formulated.

In approaching several simultaneous goals, Sacerdoti develops plans to achieve each of the goals separately; as interactions between the plans are observed, the system will impose orderings on the steps ("Step F_i from plan F must be executed before step G_j from plan G") and even alter the plans themselves to make them impervious to the effects of the other plans. Actions are represented essentially by addlists and deletelists, and the "critics" (cf. [Sussman, 1973]) that recognize the interactions between plans rely strongly on this representation, although the critic principle is more general.

Sacerdoti's approach to simultaneous goals is partially dictated by his application: a consultant system advising a human amateur in a repair task. The user may choose to order the plan steps in any of a number of valid ways; the system cannot force an order except where that order is necessary to avoid harmful interactions; therefore it maintains a highly parallel plan whenever possible until the user himself has selected the order. In a sense, Sacerdoti's system must anticipate all possible plans to achieve a task.

Sacerdoti's idea, deciding what order in which to approach goals only after having done some planning for each of them, is intriguing and avoids a certain *amount of goal reordering. However, we believe we will not make best use of hierarchical planning until we are ready to wade into deeper waters of complexity.*

2.9 Recapitulation

> You say I am repeating
> Something I have said before. I shall say it again.
>
> T.S. Eliot, *Four Quartets*

In this section we will briefly repeat the main points of the argument in Part 2.

The earliest problem solvers maintained entirely separate models corresponding to each state of the world. In GPS, each operator had the responsibility of constructing a completely new model, whereas in the resolution-based systems the description of the new model created by an action was distributed between several axioms, some describing how relationships were changed by the action, and others (the frame axioms) telling which relationships remained the same.

In an effort to do away with troublesome and obvious frame axioms, later problem solvers adopted what we have called the "STRIPS assumption," that any action will not change most relations, and therefore they described an action by telling which relations it adds and which relations it deletes from the model. The "addlists" and "deletelists" were either given explicitly or computed. Any relation not explicitly added or deleted by an action was assumed to be unaffected.

Systems implemented in artificial intelligence programming languages having a "context" feature tended to incorporate the STRIPS assumption by equating states of the world with contexts, and representing states that occur after a given state by successive descendants of the given context; since any relation asserted with respect to the given context is considered to be true with respect to any of its descendents unless explicitly deleted, the STRIPS assumption is expressed structurally in this "archeological" representation.

Meanwhile, the designers of problem-solving systems entered domains in which the STRIPS assumption began to break down: areas in which the world was modelled in such detail, or in which objects were so highly interrelated, that actions might have many consequences, most of which were irrelevant to the problem at hand. The STRIPS assumption and the archeological structure that expresses it become an obstacle here: it would be cumbersome and inefficient for the description of the action to have to make all these changes in the model. Recent problem solvers have attempted to escape from the STRIPS assumption by distinguishing between the important relations, which are always updated in the model, and the lesser relations, which are defined in terms of the important relations and which are only updated as necessary. These measures are inadequate largely because the designer of the system is prevented from stating

explicitly how the lesser relationships are affected by the various actions.

The regression technique advocated here and elsewhere provides a method whereby the actions in the plan become an important part of the model, from which a relational description of the world can be "fleshed out" as necessary. The context mechanism can be used to represent this type of "skeleton model," but successive states are represented as parallel contexts instead of descendants. This latter representation has the additional advantage of being consistent with the use of descendent contexts to represent hypothetical worlds, and with the program modification technique introduced in Part 1.

In my end is my beginning.
T.S. Eliot, *Four Quartets*

ACKNOWLEDGMENTS

This work has been developed through discussions with Zohar Manna and Earl Sacerdoti. The manuscript has benefited from the comments of Rich Fikes, Earl Sacerdoti, and Bert Raphael. The ideas presented here have also been influenced by conversations with Mike Wilber, Bob Boyer, Nachum Dershowitz, Rod Burstall, John Darlington, Gordon Plotkin, Bob Kowalski, Alan Bundy, Bernie Elspas, Nils Nilsson, Peter Hart, Ben Wegbreit, Carl Hewitt, Harry Barrow, Cordell Green, Avra Cohn, Dave Barstow, Doug Lenat, and Lou Steinberg. Mike Wilber has been of special assistance in the use of the QLISP system, which is based on INTERLISP. Our efforts have been encouraged by the environments provided by the Artificial Intelligence Center at SRI, the Department of Artificial Intelligence at the University of Edinburgh, and the Applied Mathematics Department at Weizmann Institute. Linda Katuna and Lorraine Staight prepared many versions of the manuscript.

The National Science Foundation Office of Computing Activities supported this work through Grant GJ-36146.

REFERENCES

Balzer, R.M., N.R. Greenfeld, M.J. Kay, W.C. Mann, W.R. Ryder, D. Wilczynski, and A.L. Zobrist (1974) Domain-independent automatic programming. *Information Processing 74: Proc IFIP 74*, 2, 326-330.

Barrow, H.G. (1974) HBASE. *POP-2 Library Documentation*, Department of Artificial Intelligence, University of Edinburgh, Edinburgh.

Bobrow, D.G. and B. Raphael (1974) New programming languages for artificial intelligence research. *ACM Computer Surveys*, 6, 3, 155-174.

Boyer, R.S. and JS Moore (1973) Proving theorems about LISP functions. *Proc IJCAI3*, 486-493, Stanford, CA, also in *JACM*, 22, 1, 129-144.

Buchanan, J.R. and D.C. Luckham (1974) On automating the construction of programs. *Informal Memo*, Artificial Intelligence Laboratory, Stanford University, Stanford, CA.

Darlington, J. (1975) Application of Program Transformation to Program Synthesis. *Proc Colloques IRIA:* Proving and improving programs, 133-144, Arc et Senans, France.

Darlington, J. and R.M. Burstall (1973) A system which automatically improves programs. *Proc IJCAI3*, 479-485, Stanford, CA.

Derksen, J., J.F. Rulifson and R.J. Waldinger (1972) The QA4 language applied to robot planning. *AFIPS* 41, Part II, 1181-1187.

Dijkstra, E.W. (1975) Guarded commands, non-determinacy and a calculus for the derivation of programs. *Proceedings, International Conference on Reliable Software* 2-2.13, Los Angeles, CA.

Fahlman, S. (1974) A planning system for robot construction tasks. *Artificial Intelligence*, 5, 1, 1-49.

Fikes, R.E. (1975) Deductive retrieval mechanisms for state description models. *Technical Note 106*, Artificial Intelligence Center, Stanford Research Institute, Menlo Park, CA.

Fikes, R.E. and N.J. Nilsson (1971) STRIPS: A new approach to the application of theorem proving in problem solving, *Artificial Intelligence*, 2, 3/4, 189-208.

Floyd, R.W. (1967) Assigning meanings to programs. *Mathematical Aspects of Computer Science*, Proceedings of a Symposium on Applied Mathematics Vol. 19, American Mathematical Society, 19-32.

Goldstine, H.H. and J. von Neumann (1947) Planning and Coding Problems for an Electronic Computer Instrument. *Collected Works of John von Neumann* 5, 80-235 (Pergamon Press, New York, 1963).

Green, C.C. (1969) Application of theorem proving to problem solving. *Proc IJCAI* 219-239, Washington, DC.

Green, C.C., R.J. Waldinger, D.R. Barstow, R. Elschlager, D.B. Lenat, B.P. McCune, D.E. Shaw and L.I. Steinberg (1974) Progress report on program-understanding systems. *Memo AIM-240*, Stanford Artificial Intelligence Laboratory, Stanford University, Stanford, CA.

Green, C.C. and D. Barstow (1976) A hypothetical dialogue exhibiting a knowledge base for a program-understanding system. *Machine Intelligence* 8, (eds. Elcock, E.W. and Michie, D.), Ellis Horwood Ltd. and John Wiley.

Hewitt, C. (1972) Description and Theoretical Analysis (using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot. *Ph.D. Thesis* Massachusetts Institute of Technology, Cambridge, Mass.

Hewitt, C. (1975) How to use what you know. *Proc IJCAI4*, 189-198, Tbilisi, Georgia, USSR.

Hoare, C.A.R. (1969) An axiomatic basis for computer programming. *CACM*, 12, 10, 576-580, 583.

Katz, S.M. and Z. Manna (1976) Logical analysis of programs, *CACM* 19, 4, 188-206.

King, J.C. (1969) A Program Verifier. *Ph.D. Thesis*, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA.

Kowalski, R. (1974) Logic for Problem Solving. *Memo No. 75*, Department of Computational Logic, University of Edinburgh, Edinburgh.

Manna, Z. (1968) Termination of Algorithms. *Ph.D. Thesis* Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA.

Manna, Z. (1974) *Mathematical Theory of Computation* McGraw Hill, New York.

Manna, Z. and R.J. Waldinger (1974) Knowledge and reasoning in program synthesis. *Artificial Intelligence* 6 2, 175-208.
CA.

McCarthy, J. and P. Hayes (1969) Some philosophical problems from the standpoint of artificial intelligence, *Machine Intelligence* 4, (eds. Meltzer, B. and Michie, D.), American Elsevier, New York.

McDermott, D.V. (1974) Assimilation of new information by a natural language-understanding system, *AI Memo 291*, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA.

McDermott, D.V. and G.J. Sussman (1972) The Conniver Reference Manual. *AI Memo 259* (revised 1973), Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA.

Moore, JS (1975) Introducing iteration into the pure LISP theorem prover. *IEEE Transactions on Software Engineering*, SE-1, 3, 328-338.

Naur, P. (1966) Proof of algorithms by general snapshots. *BIT* 6, 4, 310-316.

Newell, A., J.C. Shaw, and H.A. Simon (1960) Report of a General Problem-Solving Program for a Computer. Information Processing. *Proceedings of an International Con-*

ference on Information Processing, 256-264. UNESCO, Paris, France.

Rulifson, J.F., J.A.C. Derksen, and R.J. Waldinger (1972) QA4: A procedural calculus for intuitive reasoning. *Technical Note 73*, Artificial Intelligence Center, Stanford Research Institute, Menlo Park, CA.

Sacerdoti, E.D. (1974) Planning in a Hierarchy of Abstraction Spaces *Artificial Intelligence*, 5, 2, 115-135.

Sacerdoti, E.D. (1975) The non-linear nature of plans. *Proc IJCAI4*, 206-214, Tbilisi, Georgia, USSR.

Sussman, G.J. (1973) A Computational Model of Skill Acquisition. *Ph.D. Thesis*. Massachusetts Institute of Technology, Cambridge, MA.

Tate, A. (1974) INTERPLAN: A plan generation system that can deal with interactions between goals. *Memorandum MIP-R-109*, Machine Intelligence Research Unit, University of Edinburgh, Edinburgh.

Turing, A.M. (1950) Checking a large routine. *Report of a Conference on High Speed Automatic Calculating-Machines*, 66-69. University of Toronto, Toronto.

Waldinger, R.J. and R.C.T. Lee (1969) PROW: A step toward automatic program writing. *Proc IJCAI*, 241-252, Washington, D.C.

Waldinger, R.J. and K.N. Levitt (1974) Reasoning about programs. *Artificial Intelligence*, 5. 3. 235-316.

Warren, D.H.D. (1976) Generating Conditional Plans and Programs. *Proc AISB Summer Conference*, 344-354, Edinburgh.

Wegbreit, B. (1974) The synthesis of loop predicates, *CACM* 17, 2, 102-112.

Wilber, M. A QLISP Reference Manual. *Technical Note 118*, Artificial Intelligence Center, Stanford Research Institute, Menlo Park, CA.

Winograd, T. (1971) Procedures as a Representation for Data in a Computer Program for Understanding Natural Language. *Ph.D. Thesis*, Massachusetts Institute of Technology, Cambridge, MA. also appears as *Understanding Natural Language* Academic Press, New York, NY.

Winograd, T., (1974) Artificial Intelligence–when will computers understand people? *Psychology Today*, 7, 12, 73-79.