SUBGOAL PROTECTION AND UNRAVELLING DURING PLAN SYNTHLSIS

Chuck Rieger and Phil London
Computer Science Department
University of Maryland
College Park, Maryland 20742

Abstract: Subgoal annihilation occurs during plan synthesis when the solution of a later subgoal. undoes or interferes with a completed solution to a preceding subgoal. Subgoal annihilation during plan synthesis is discussed, and a uniform solution to the general problem is presented. The two phases of the theory are detection and correction of annihilation errors. Detection is carried out via a theory of spontaneous computation in which so-called guardian clusters" watch over synthesized goals. Correction is accomplished by a give and take technique called "unravelling," wherein conflicting parts of the plan back off temporarily to allow other parts to complete. The theoretical framework of the plan synthesizer is described, a detailed computer example of detection and unravelling is given, and other applications motivated by this theory are suggested.

Keywords: problem solving, subgoal protection, subgoal annihilation, spontaneous computation, unravelling, inference.

1. Introduction

One of the subtler issues of automatic problem solving and program synthesis is subgoal annihilation or infringement. Subgoal annihilation occurs when. during the course of solving one aspect of a larger problem, some other aspect of ttie problem whose solution has already been obtained is interiered with, or undone completely. Subgoal infringement is a milder form of subgoal annihilation in which the prior solution of one subgoal causes the solution of anottier later subgoal to become more difficult or tedious.

in this paper, we describe a uniform procedure for detecting and resolving subgoal annihilations. The scheme is applicable to both small-domain and general problem solvers, and relies on an associative style of computing which we call "spontaneous computation" for the detection phase, and a technique we call "unravelling" for the correction phase.

We first provide a brief motivation of the subject and a review of existing techniques for subgoal protection. Then we describe the problem solving framework from which we have approached the problem, and present the paradigm for spontaneous computation. Next, we describe the protection-unravelling theory and illustrate it on a small blocks-world example run on our system. Finally, we discuss the implications and extent of our scheme, pointing out some possible future developments and related applications of the theory.

2. Motivation and Background

In solving a complex problem, it is nearly always necessary to break the initial problem down irvto smaller problems such that, if each subproblem were to be solved, the initial problem would also be solved. This approach is commonly called problem reduction or means-ends analysis (see e.g. [ENI] or [HI]) and has been the paradigm for most problem solving systems in AI.

Problem reduction is an intuitively adequate approach to problem solving, especially if the point is to propose models of human problem solving. However, problem reduction suffers from an inherent difficulty. This is that the act of delegating responsibility for the solution of the initial problem to smaller subproblems must obviously give some degree of autonomy to the processes which solve the subproblems. (Otherwise, there would be no point in reducing the initial problem.) At one extreme, there is total autonomy, where each subgoal solver has complete freedom to generate whatever type of solution seems best for the subproblem, disregarding the fact the tthe subproblem participates in a larger plan which must eventually be consistent. At the other extreme, every subgoal solver takes utmost care in interacting and cooperating with all other subgoal solvers so that harmony of the overall plan will be ensured.

The first extreme suffers from too much independence; the second extreme suffers from too little independence. However, the first approach is conceptually and computationally much cleaner than the second, because it makes possible a more modular representation of problem solving strategies. If there were some uniform method of noticing and resolving conflicts among relatively autonomous subgoal solvers, then we could have the best of both worlds: modularity in problem solving knowledge, yet harmony in the synthesis of large plans.

Subgoal annihilation problems are extraordinarily pervasive, both in expert domains, and in everyday human problem solving. Consider a simple example from the blocks world:

GOAL: Achieve the state (AND (ON A B) (ON B C)) starting from the state shown below under the constraint that only one block at a time can be lifted.

If the problem solver attempts to solve the subgoals (ON A B) and (JN B C) in the order given in the statement of the goal, then it will back itself into a corner in which it must undo part of the plan for the first goal in order to solve the second goal, as shown above.

To illustrate an analogous annihilation problem in everyday problem solving, consider the midnight snack scenario: an AI researcher has just made a delicious (but greasy) cold-cut sandwich, and wishes to clean up before consuming it. "Clean up" might mean: (a) wash his hands, and (b) throw the greasy cuttings away. Obviously, if he solves these subgoals in this order, he will not only have to wash his hands twice, he might also lose his job at the Al labi

The central problem in both these examples, and indeed in many instances of subgoal annihilation, is ordering. In the absense of any prior knowledge, it is difficult to see how to have the problem solver prefer any particular order for the solution of subgoals. Hence, it must undertake them in some arbitrary order, simply to begin. The object of any system of subgoal protection must therefore be (a") to detect when following the simple syntactic ordering leads to subgoal annihilations or infringements, and (b) to recover from such violations with the minimum degree of resynthesis.

Not all subgoal annihilations stem from ordering problems. The other major source of annihilations is inappropriate strategy selection. This occurs when, having several alternative strategies for some subgoal, the solver selects one (perhaps even the best, based on the local evidence surrounding the subgoal) which turns out to be inimical to some other subgoal in the larger plan. For example, if our AI researcher wishes to insert a stake in his garden for the purpose of nailing a cookie tin to its top as a bird feeder, selecting and applying the "pound it in with a sledgehammer strategy might severely splinter the top of the stake, making it impossible or difficult to nail anything to the stake. In this case, selecting some other strategy would be called for. We will comment on this type of subgoal violation later, but concentrate principally on annihilations stemming from bad orderings.

Past Work

Much attention has been paid recently to the subgoal annihilation problem. This attention seemingly stems from Sussman's HACRER system [S2], In fact, the predisposition to the blocks world by researchers in this area may be somewhat Sussman's doing; he was able to cite some conceptually simple but theoretically troublesome blocks world problems. The very nature of HACKER as a system which produces plans under the assumption of absolute subgoai independence, and then proceeds to debug them for the errors they inevitably have be-

cause of subgoal interaction, points out the need for subgoal interaction schemes.

Several interesting subgoal interaction schemes have been devised. In Sacerdoti's NO All system [SI], subgoals are solved independently and in parallel. At each step in the planning process, the plan is incrementally made more detailed. After each such step, procedural experts examine the plan to remove any potential interaction errors or redundant operations. A powerful idea here is the use of a two-dimensional (non-linear) representation for plans, also a central feature of our Commonsense Algorithm system.

Waldinger [WI] and Tate [TI] both advocate the modification of partial plans to alleviate detected protection violations. Their schemes involve detecting that the current goal violates a previously protected goal and that the violation can be relieved by specifying that the solution to the current goal occur before the violated goal.

Because of space constraints, we must omit a detailed discussion of the relative merits of the various problem solving strategies dealing with subgoal annihilation problems. Such discussions, formulated from various points of view can be found, for example, in [S1] or [WI], What should be pointed out, however, is that there is general agreement that some form of dynamic subgoal interaction scheme is required for solving even barely complex problems. We expect that the scheme we are proposing will provide us the latitude to explore the subtler issues of violation detection in complex domains and the relation of strategy selection errors to subgoal violation.

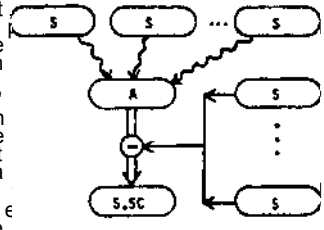
3. Theoretical Framework

There are two aspects of our subgoal protection theory that relate to ongoing work within our Commonsense Algorithms (CSA) Project at Maryland. These are (1) the plan synthesizer and its representation for plans and cause-effect knowledge, and (2) our paradigm of spontaneous computation. We describe each of these briefly in this section. For more complete discussions, the reader is referred to [R1J, 1R2J, and [R3J.

3.1. CSA Problem Solver Representation

The basis of the problem solver's representation both for the strategies it will use in constructing plans, and for the final plans themselves, is a set of 16 links. Each link connects two or more events and represents one of the primitive[causality-related event-event relations in the CSA theory. We will not discuss all the links in this paper, but instead concentrate on one of the constructions of most interest to our problem solver.

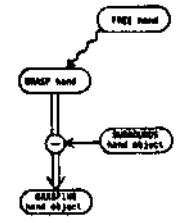
This construction is the gated causal link with explicit enablements. It is represented graphically as shown to the right. The central vertical link with the bulb is the c relation. Its "syntax" mands that the event on its tail be an action-like event and that the event on its head be either a state or st a event. The events f into the bulb from the right are called the gating states, in that they represent a collection of states which must be true in order for the action to effect the state or statechange on the causal link s head. Semantically, these are the conditions that govern not the action itself, but the influence the action will have on its surroundings. As such, the interpretation is that the gates must all be true simultaneously, and for the duration that "causality must flow" from the action to the caused state or statechange. In this sense, gates will constitute subgoals to be solved before performing the action.



The wavy links are enablement links. These indicate explicit enabling conditions that must be in effect to initiate and sustain the action itself. The syntax of the enablement link demands that its tail be a state and its head be an action-like event. Enablements and gates are distinct in theory, because the enablements are prerequisite to the action itself, whereas the gates are prerequisite to the causal influence of the action on its environment. Enablements will also contribute subgoals to be solved before performing the action.

To illustrate, to express one simple strategy for picking up a small physical object by grasping it, we write a pattern such as:

```
($STRATEGY (
  (NAME *GRASP-IT)
  (VARIABLES hand object)
  (EVENTS (1 S (FREE hand))
    (2 A (GRASP hand))
    (3 S (SURROUNDS hand object))
    (4 S (GRASPING hand object)))
  (LINKS  (C-CAUSE (2 4) (3))
    (OS-ENABLE (1 2)))
  (ACCOMPLISHES 4)
  (APPROPRIATE-WHEN
    ($FEATURE (SIZE object *) HANDFUL)
    ($FEATURE (CLASS object *) PHYS-OBJ)) ))
```



We express this in the LIbP rorm shown to the lert of the diagram, and enter it into the database of problem solver strategies each of which we call an abstract algorithm". Each such strategy is a schema that can be as simple as this one, or as complex as we desire. More complex patterns typically include additional levels of specification on the gates and enaolenients, providing more than one level of solution within the strategy itself. Roughly speaking, the larger the strategy pattern is, the more stereotyped a solution it represents, in that it provides a more thorough specification of the next several levels of the plan at once. Smaller patterns contribute only the central causal idea and the gates and enablements, leaving the strategies for solving all subgoals up to the synthesizer.

3.2. Operation of the CSA Synthesizer

In the CSA system, the act of inserting a new strategy pattern in the synthesizer s database of strategies inherently also suggests to the system where to store the pattern and when to retrieve it as a relevant strategy for some goal. This organization occurs via the APPROPRIATE-WHEN feature in the strategy's input description.

The APPROPRIATE-WHEN conditions specify the circumstances under which it might be most appropriate to prefer this strategy over the possibly numerous other strategies available to the system for the sarae general goal. For example, the APPROPRIATE-WHEN conditions for the "send it by mail" strategy are different from the appropriateness conditions for the hand carry it" stragegy, even though both strategies accomplish the same goal of causing a change in location of a small object.

The system uses the APPROPRIATE-WHEN conditions to decide where in one of the system's discrimination networks to situate the strategy. There is one discrimination network for each goal predicate (e.g. LOC) in the system. It is the purpose of a network, whenever a goal involving the net's predicate appears during plan synthesis, to select the most appropriate strategy in that particular instance. For this reason, we call these networks "causal selection networks" (CSN's). CSN's are described in more detail in [RI].

The selection process is sensitive to the features of all known strategies which best differentiate those strategies according to their contextual relevance. The nodes of a CSN therefore are intended to probe as much as necessary of the context of the goal whose solution is sought and its environment to make a "most appropriate" selection in that environment. In using a central CSN structure for this selection process, rather than adopting a more distributed scheme in which strategies compete (e.g. as do PLANNER theorems [SW1] or production rules [DK1]), we are reflecting our very strong theoretical bias that intelligent, coordinated selection from among alternate strategies is one of the central issues of AI.

Our system therefore consists of a potentially large number of CSN's, each of which contains (at

its terminal nodes) a set of strategies. Each strategy is expressed via cause-effect patterns similar to the one illustrated above.

In this environment, the plan synthesizer's overall behavior is as follows. The initial goal (there may be several concurrent goals, but we will consider the simplest case where there is just one) causes the relevant CSN to be called up and applied. The CSN poses queries to the database-deductive component in order to select the most appropriate strategy, within the limits of its available strategies and available information about the context of the problem. Once selected, the synthesizer creates as its subgoals any gating and enabling states (as specified by the gates and enablements of the selected strategy) which cannot be expected to be true at plan execution time. In this manner, the synthesizer recursively attacks the initial goal until no details remain or until a specified level of detail (depth bound) is achieved. The paradigm is that of problem reduction, where the reduction operators are (possibly quite complex) CSA .strategy patterns, and where alternate strategy selection occurs "intelligently" via the CSN's.

The most common source of subgoal violation in this environment seems to be (contextually) bad orderings on the gating and enablement states in one or more subgoal strategies. Although it seems to be feasible to learn and record good orderings, initially, the synthesizer cannot be assumed to know anything about potential ordering problems. By definition, therefore, in the absence of explicit ordering information, the synttiesizer always attacks the gating and enablement conditions in the order they appear in the internal representation. Similarly, for compound initial goals (represented by anotner of our links), the components are attacked in the order specified.

We will return to the overall synthesizer strategy later. We now turn to a description of the spontaneous computation component of the synthesizer, and describe how it relates to the synthesizer strategy just outlined.

3-3. Spontaneous Computation

The spontaneous computation (SC) component of the CSA system provides us with a demon-like, associative access mechanism. It is of use and interest in other phases of our project (especially as a model of certain categories of inference in language comprehension). However, we focus here only on the aspects which relate to the synthesizer.

The CSA SC component is a generalized implementation of the notion of pattern-directed invocation in that it provides for more complex invocation patterns and for a more sophisticated hierarchical organization of invocation patterns. A CSA SC invocation (trigger) pattern is a collection of nested n-tuples composed via AND, OR, and ANY relations to virtually any complexity. Each nested n-tuple is identified as (1) an associative component of the trigger pattern, (z) a non-associative component of the trigger pattern, or (J) a "computable", an S-expression which must evaluate non-NIL. Associative components are those that have the potential for triggering the execution of the spontaneous computation. Non-associative components are patterns that must be true (i.e. in the database or iieducible) in order for an SC to run. but which themselves have no potential for initially triggering the computation. We denote associative components by the symbol "+", non-associative components by the symbol "-", and computables by any other LISP predicate.

In a system with this type of invocation pattern, an important issue is now to organize the associative parts of triggers so that, given some stimulus, all "nibblers , i.e. trigger patterns which contain a component that matches the stimulus, can be accessed all at once. For this purpose, we employ a construction called a "trigger tree", and use the metaphor of "planting" a trigger pattern in a trigger tree. The Internal structure of trigger trees are irrelevant to this discussion, and are described in [R3]. We need only the term here, and that is because each trigger tree will correspond to a population of SC's, namely the collection of SC's whose trigger patterns' associative components have been planted in the tree.

To illustrate, suppose we wish to create an SC which reacts associallveiy whenever BLOCKI comes to be located at location A while BLOCK2 is located at location B, ££ whenever any block greater than 10 cm in height comes to be located at location C. Then we would define this SC to our system as follows:

```
(^PLANT  ' (OK (AND        (LOG BLOCK1 A))
                           (LOG ULOCK2 B)))
               (AND        (LOC -X C))
                       1   (HEIGHT -X -Y))
                   (GREAT ERP -Y 10)))
        <some body> <some trigger tree>)
```

This illustrates all three types of trigger components (associative, non-associative. and computable). The body is any EVAL-able LISP S-expression, and the trigger tree is the name of the trigger tree (e.g. "TTi) in which we want the associative components of this pattern planted. The integers (all l's in this case) in the pattern are used by the database-deductive component, and represent the energy budget (measured in database fetches) which can be expended in attempting to demonstrate that the associated component of the pattern is true after an initial triggering by some other component. Atoms prefixed with hyphen signs denote variables. Variables are global to the entire trigger pattern, in that two variables with the same name must be consistently bound to the same constant in a stimulus pattern (as in PLANNER and CONNIVER [MS 1]).

Associative access to all SC's which "nibble" at a stimulus pattern is caused by a call having either of the following forms:

```
(SACTIVATE <trigeer-tree> <stimulus>)
or  (<trigger tree> <stimulus>)
```

(i.e. trigger trees can be used semantically as functions also). In either case, <stimulus> is a fully constant nested n-tuple.

Application of a trigger tree in this manner results in a queue of SC bodies which are ready to run. In the process of activation and the determination of which nibblers are actually ready to run, there is considerable Interaction with the database-deductive component in determining which patterns are fully satisfied (beyond the superficial associative triggering). Some other theoretical applications of this interaction are discussed in [R3].

Trigger trees can then be regarded as populations of contextually related SC s to participate in a larger construct we call a "channel '. It is the channel construct that gives us a final pattern directed invocation facility akin to PLANNER and C0NNIVER, except that channels provide for more generality in the hierarchical control over populations of bC's. Since we use channels only trivially in the plan synthesizer at present, we do not discuss them further here. Interested readers are referred to [3].

We now return to the main topic of the paper, first describing the subgoal annihilation detection strategy.

4 Subgoal Violation Detection

As the synthesizer attacks and solves each subgoal in the manner described earlier, it protects that subgoal by planting one or more violation patterns in a central synthesizer-related trigger tree. A violation pattern, conceptually, is any pattern which would be directly inimical to the pattern representing the subgoal which is being protected. For this reason, we call protection patterns "guardian clusters." It is the responsibility of a guardian cluster, for the duration of its existence, to ensure the continuation of the state it protects, spontaneously firing when inimical patterns materialize.

To illustrate, to protect the state (LOC BLOCKI A), we would plant a relatively simple pattern such as:

```
(AND (+ 1 (LOC BLOCKI -X) )
     (NOT-EUAL -X 'A))
```

i.e. a guardian that would react if the location of

BLOCKl is ever predicated to be anywhere but location A, the protected state.

The planting of a guardian cluster occurs immediately upon the successful solution of the state the cluster is designed to protect. (i.e. all gating and enabling states). Gating and enabling states which are found to exist already, and which therefore require no synthesis, are also protected as though they had been solved by the synthesizer.

The point of guarding each subgoal as it is solved Is to ensure that at the end of all the gate and enablement solutions, all subgoals will still be in effect so that (a) the action of the CSA schema can be performed (actually, added to the output action stream) in the presence of all its enablements, and (b) performance of the action will achieve the intended result in the presence of all its gates. Therefore, guardian SC s are destroyed (actually, masked) after the synthesizer finally generates the action. Semantically, since the action has been performed, the protected states are *no* longer needed for the time being.

There are some interesting theoretical questions abouTwhat, exactly, constitutes a guardian cluster of SC's for any given state. In its simplest form, we conceptualize the guard to trigger on "the negation" of the guarded state. However, things are. seldom so simple that planting just the syntactic negation of the pattern will be adequate.

We do not yet have what we would call a general theory of guardian cluster generation. However, we expect the semantics of the predicate in the protected state can provide the synthesizer with direction in constructing the guardian cluster. For example, for those predicates which are uni-valued, but which have a continuous range (such as LUC), we plant a pattern of the form shown earlier, namely:

Protect (LOG X Y)————> (AND (+ 1 (LOG X Z))
(NOT-CqUAL Z Y))

for predicates which are either true or false, we plant a pattern involving the predicate which is the opposite of the protected state's predicate, e.g. to protect the state (FREE <hand>), we plant:

Protect (FREE -il)————> (+ 1 (GRASPING -il))

Guardian Clusters vs. Inference

It is difficult at this time to assess the range of problems involved in planting good guardian clusters. However, one requirement is assured: we must rely heavily on a good inference system which generates a moderately rich set of inferences from each event arising during synthesis, i.e. meaning paraphrases, implications, and so forth. Our bias is to keep the guardian cluster fairly concise, relying on Inference to hit this violation target in *a* shotgun fashion.**

** We feel that studying the processes of subgoal annihilation detection will shed some light on the interesting question of how problem solving and inference interact. As an example of problem solver - inference interaction, reconsider the midnight snack exampLe. There, the two goals are (1) have clean hands, and (2) get the greasy leftovers (GL) into the trash can. Now, the essence of the model concept that represents the greasy leftovers is SC-like itself; namely, it represents an object, one of whose features is: whenever this object is touched (i.e. whenever an assertion matching (GRASPING -H GL) is made), the grasping object will become greasy. This is an inference which "plays the role of an "active" component in the object's definition (similar to an "imp" in Winograd's terminology [W2J).

ii our synthesizer were to synthesize and protect the plan for clean hands first, one state that would be directly asserted at some point during the "throwing away" subgoal (the second one) would be (GRASPING liANI) GL), where GL is the model concept ior the leftovers. This grasping state, being a component of the trigger pattern for this active feature of GL, would cause the inference (GREASY HAND) to be generated. That in turn would set off the guardian cluster for the clean hands state, assuming of course that this cluster were sufficiently rich to include (GREASY HAND) as a component. This could be arranged, for instance, by including a relatively exhaustive definition of the state "clean hands", namely, an enumeration of all closely-related inimical states (DIRTY, GREASY, etc.) Alternatively, some of these problems might be avoided if we were to break all these concepts down into further primitives, and arrange to trigger on the more primitive symbols rather than the higher level ones. In any event, the point is that without this inference that connects a grasping of something greasy to the resulting greasiness of the grasper, the grasping state asserted during the solution of one of the subgoals would not trigger the guardian cluster. However, with such an inference, the guardian cluster is hit. Thus, details aside, we get a glimpse of how inference can interact with plan synthesis.

To summarize, there are two issues relating to spontaneous subgoal guardians: (1) how to express the guardian cluster, and (2) how much responsibility to delegate to the guardian vs. how much to place on a more general inference facility.

5. Unravelling

The protection paradigm described above suggested itself relatively soon after the development of the SC component of the CSA system. We then puzzled for a while about what should actually happen at subgoal violation time. Our criteria were twofold: (1) that violations be detected at the earliest possible phase of synthesis, and (2) that as much as possible of the plan up to the violation point be salvaged.

The first criterion seems to be solved by the SC detection paradigm. The general motivation for early detection is that, during the earlier phases, there is more fundamental knowledge about the how s and why's of the plan's construction. This makes possible both more accurate diagnosis of violations and more accurate analysis of the implications of plan rearrangement. The second criterion of maximum plan salvage has more practical motivations, but also coincides with our intuition that human problem solving involves considerable salvaging and stitching together of plans.

Since the source of conflicts we are primarily considering is bad orderings, the repair of a conflict will amount to reordering parts of the plan in a way that avoids the conflict. The simplest and most obvious strategy would be simply to start over tfith a new order, possibly resynthesizing many gate and enablement suboals. This is not a good approach, because it is inherently combinatorial, and because it duplicates elfort* It would be more desirable to have a give-and-take arrangement wherein conflicts can be resolved by having one subgoal back off for a moment to allow the other subgoal to complete, then having the retreated subgoal simply rejoin without having to resynthesIze.

Uur theory oi correction follows these ideals. We call the technique that implements this give-and-take strategy "unravelling". The unravelling scenario goes as follows (see figure below). Suppose gating state A has been achieved as a subgoal, and"has been protected. Suppose that during the synthesis of another brother subgoal B (e.g. a brother gate), A's guardian cluster is triggered, indicating that some state lust achieved in the partially completed synthesis oi B has violated state A. The plan which leads up to A then unravels one level by reverting to the context which existed immediately prior to the attainment of A, i.e., the unravelling mechanism undoes the action which causes A. It then removes A, alleviating the immediate violation, and notes that it will have to reconstruct this last step in the plan leading up to A.

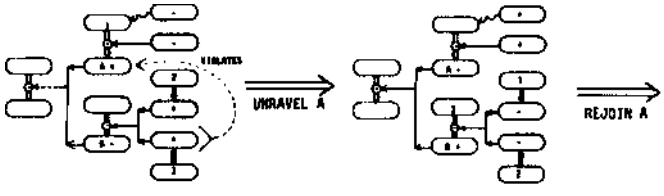
As A is unravelled, the gates on A's causal link and the enablements of A's causing action, once protected themselves during the earlier part of the synthesis, are re-protected by reawakening their guardian clusters. These gates and their clusters are then melded into the current context (i.e. that in which B is being synthesized). Then B is allowed to proceed. When B completes, A rejoins (n.b. A does not resynthesIze, but rather simply closes in "around" the plan for B, which now has conceptually been spliced in) by reissuing the action

which achieves state A. A's gates and enablements are then once again unprotected, and A itself is again protected. At that point, both A and B will coexist, permitting the syntliesizer to get on with the next gate or enablement, or, in case all have been completed, issue the action at the level awaiting such a completion.

The semantics of this unravelling-rejoining are this: Ordinarily,, brother subgoals do not have to know of each other s existence. However, when one does something which is inimical to the other, they must at that point become aware of each other. By unravelline the first one a single level, we alleviate the immediate problem, but at that point also force pieces of the two goals (actually, the plans leading up to them) to coexist in the same context. This is because, the process of unravelling A casts A s gates and enablements into the population of currently guarded states.

Unravelling is tree to occur wherever and whenever it is needed to permit the current subgoal to proceed. This means that one unravelling can set off other unravellings, e.g., that subgoal B might cause more than one unravelling inside of A's plan after being allowed to proceed past the first conflict, that the melding; of A's prior context (i.e. its gating and enabling states) into the existing context could in turn trigger an unravelling of part of B's plan, and so forth. All problems arise because of now having to force two brothers (or, in the limit, distant cousins) which previously knew nothing of each other to live in the same environment.

This give and take paradigm is, we feel, the best possible conflict resolution paradigm. At one end of the spectrum, unravelling amounts to one simple backoff, while at the other end of the spectrum, it can amount to multiple backoffs which in the limit amount to a complete reversal of all subgoals (i.e. each later one comes to be spliced in before former ones). The important point (and the feature that we feel gives this approach elegance) is that only the unravellings that have to be done are actually done. There is no wholesale reordering of subgoals, and (conceptually) little overhead for conflict-free plans.



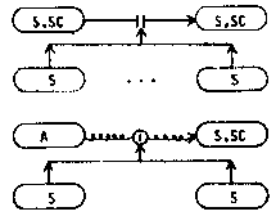
'-• DENOTES: UNPROTECTED
•♦• DENOTES: PROTECTED
NUMBERS IN NODES DENOTE SEQUENCE OF ACTIONS

b.l. Context Requirements

When an unravelling occurs, previously protected states come to be unmasked and reprotected. The demand on the context mechanism therefore is that it be able to revert to the context immediately prior to the first masking of those subgoals. As the reprotection occurs and the prior context is exhumed, items from that context are melded into the existing context. This process can itself trigger other subgoal violations pending in the current context. Although any given example can grow to be quite complex in its unravelling behavior, unravel-

6. Example

We now illustrate the theory via a simple blocks world example which conveys some of the subtleties of the unravelling process. Starting with an initial configuration as shown in Figure 1(a), we pick up the. action as it stands in Figure 1(b), solving the goal (uN A B). The synthesizer has been able to construct a standard synthesis without any protection violations up to this point and is about to attack the. subgoal (FREE-HAND) — state SII.



First let us remark that there are two CSA links (shown to the right) in this graph that have not been previously mentioned in this paper. They are the "state-coupling" link, and the "gated one-shot byproduct" link. (See [R1], [H2] and [RG1] for a complete coverage of the CSA links.) The state-coupling link connects two states and implies that the state at its head can be considered to be true because of the truth of the state at its tail. The gated byproduct link is identical in syntax and semantics to the gated causality link. However, the intent of the causality link distinguishes it from the Dyproduct link, in that the state caused by the action on a byproduct link is not the state intended by the performance of the action, but merely a byproduct. Thus, the fate of a Dyproduct state is irrelevant to plan synthesis and it need not be protected. But a byproduct state can cause a violation as decisively as any other state. Therefore, byproduct states must be treated as other achieved states with respect to violation detection.

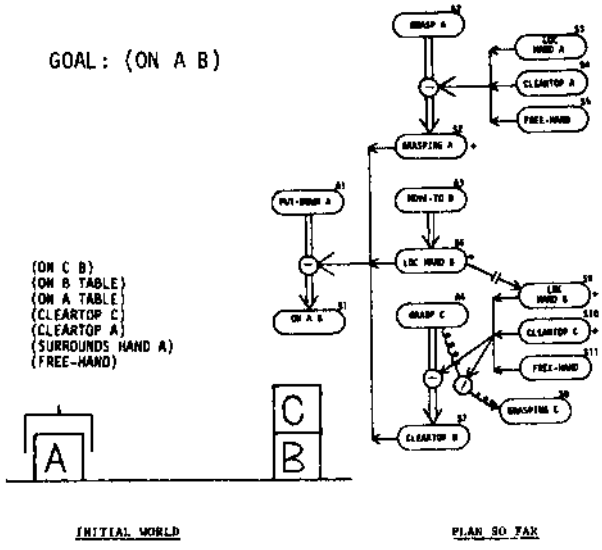
In attempting to solve the goal (ON A B), the synthesizer has set up subgoals (GRASPING A), (LOC HAND B), and (CLEARTOP B). (GRASPING A) has been solved (its subgoals S3, S4, and S5 were already true In the initial configuration) by performing the action (GRASP A). Similarly, the second subgoal, S6, has been solved by performmg action A3. (S6 had no subcoals). To this point, the synthesizer has proposed an action sequence (A2,A3).

The last subgoal to be solved is (CLEARTOP B), S7. The action AA can only achieve this goal after the subgoals S9, S10, and SII have been solved. S9 is true because S6 is true and this fact is explicitly denoted by a state coupling link (built by the synthesizer). S10 is true in the initial configuration and nothing lias negated it. Now the synthesizer attempts to solve SII, (FREE-HAND). The currently protected states S2, Sb, S9, and SiO are marked with '+' in Figure 1(b). Since SII is a violation of the protected state S2, let us describe the steps taken by the synthesizer to complete this plan.

— The action (A2) which causes the violated state (S2) is unravelled. This leaves a new action sequence (A3). The context in which S2 s subgoals were protected is exhumed and melded in with the current context, yielding a set of protected states S3,SA,S5,S6,S9,S10. An item (S2.A2) is pushed onto an "unravel stack".

— A test for secondary violations is made (i.e. a test to determine whether any of the currently active states (S6,S9,S10) violate any of the newly reprotected states (S3,S4,S5)). This is done in the order of the most recently solved state first (i.e. SIO, then S9, then S6). The violation of S9 violating S3 is detected, S3 is unravelled, and its entry is pushed onto the unravel stack. The unravel stack now looks like: (S3.NIL),(S2.A2). The action entry for S3 is NIL because there is no action to be unravelled for S3. This NIL entry will signify that the synthesizer must construct a plan tor S3 at the time S3 is rejoined. (The reason for this new synthesis is that S3 is no longer true — that is why it was violated!)

— No more secondary violations are detected and SII can now be protected. Since all the subgoals



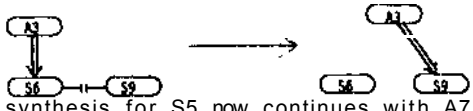
(a) Figure 1. (b)

for S7 are now solved, these subgoals' guardian clusters can be hidden (by popping contexts), leaving only states S4,S5,Sb protected. The action A4 is appended to the action sequence, yielding (A3,A4). A test must now be raade to see if S7 or tiie byproduct state S8 cause a violation of any of the currently protected states.

— It is detected that S8 violates S3. S5 is unravelled, yielding an unravel stack: (S5.N1L), (S3.NIL),(S2.A2). No secondary violations need be considered since secondary violations are caused only by the reactivation of hidden guardian clusters. S5 has no subgoals and, thus, no guardian clusters are reactivated. The action sequence remains (A3.A4). The currently protected states are S4,So,S/.

— Since subgoals of S7 caused violations, The top entry on the unravel stack, (S3.NIL), is popped. If the action slot it is time to start rejoining, on the unravel stack entry is not NIL, the action is threaded onto the action sequence and the violated state is reprotected. In this case, the action slot is NIL, so that there is no action to be threaded. Thus, the synthesizer is called recursively to generate a solution to the goal S5. Of course, there is the possibility of a new protection violation, which is exactly what happens in this case. The plan generated by the synthesizer as a solution to (FREE-HAND), S5, Involves state S12 and actions A 6 and A7 as seen in Figure 2. A6 is threaded onto the action sequence (A3,A4,A6), but when a test is made to determine if S1 2 causes any violations, the contradiction between (LUC HAND FREE-SPACE) and (LOC HAND B) is detected. Since S6 was protected, a violation occurs.

— As in this case, when a violated goal is state-coupled .to another, special care must be taken in unravelling. It should be recalled that the reason for the coupling is to denote the fact that, for example, S9 is true only because Sb is true. Thus, if we were to unravel S6 by splicing A3 out of the action sequence, then S9 would not have been true when we thought it was. The solution is to break the causal link between the unravelled action (A3) and the unravelled state (S6), and construct a causal link between the action and the coupled state (SV). This transformation is depicted below. State Sb is left without a causing action as its entry is pushed onto the unravel stack: (S6.N1L).(S3.NIL),(S2.A2). The protected states are



— The synthesis for S5 now continues with A7 being threaded onto the action sequence: (A3,A4,A6, A7). The guardian for (LOC HAND FREE-SPACE) is hidden and Sb can be rejoined.

— The entry for S6 is popped off the unravel stack and the synthesizer is again called recursively to generate a new solution for (LOC HAND B). This process does not cause any further violations. Figure 2 displays the new plan; A8 is threaded onto the action sequence, yielding (A3,A4,A6,A7,A8). (LOC HAND B) is again protected and the synthesizer returns to the level that was solving S5.

— S5 is reprotected, leaving the set of states S4,S5,Sb,S7 protected. The synthesizer exits from this recursive call, returning to the level that was working on S7 (remember?). It was at this level that S8 violated S3.

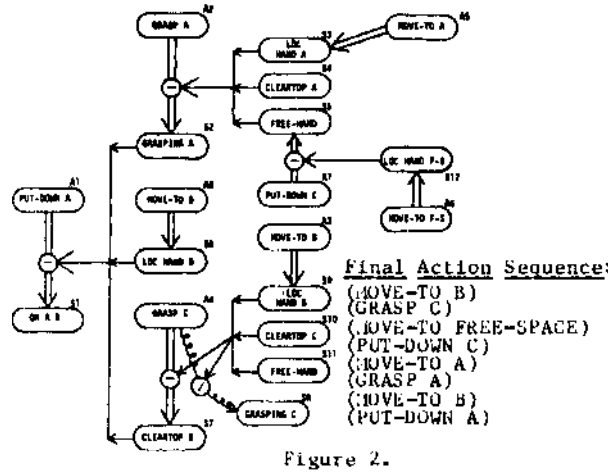


Figure 2.

-- At this level, there are still two entries on the unravel stack; (S3.NIL),(S2.A2). Again, the first entry is popped off, the synthesizer is called recursively to take care of its NIL action slot, and then tests for violations, detecting S3 violates S6. The new action, A5 (Figure 2), is threaded, and S3 is reprotected. If the synthesizer were to return at this point and attempt to unravel S6, we would run into a major problem; S6 and S3 would alternately violate one another indefinitely. This is because unravelling S6 at this point violates the Golden Rule of Unravelling, namely, that an unravelled state cannot be rejoined until the state which violated it has been used as a subgoal, i.e. until it has been unprotected. (The complexities of the algorithm described here merely serve the purpose of preserving the integrity of the Golden Rule.) Thus, if all the subgoals of an unravelled state (e.g. S2) have been solved, then that state must be rejoined. Therefore, the unravel stack entry for S2 is removed, A2 is added to the action sequence, S3, S4, and S5 are unprotected, S2 is protected, and the synthesizer returns. The action sequence is now (A3,A4,A6,A7,A5,A2). The protected goals are S2,S7.

-- The last unravel entry (S6.A8) is popped. A8 is threaded onto the action sequence, S6 is reprotected, and the synthesizer returns. The action sequence is (A3,A4,A6,A7,A5,A2,A8). The protected goals are S2,S6,S7.

-- The synthesizer has returned to the level of S1, (ON A B). All of this goal's subgoals are solved, so its action, Al can be threaded onto the action sequence, and the truth of (ON A B) can be asserted.

The final result of this synthesis is the complete plan representation of Figure 2 and an action sequence (MOVE-TO B), (GRASP C), (MOVE-TO FREE-SPACE), (PUT-DOWN C), (MOVE-TO A), (GRASP A), (MOVE-TO B), and (PUT-DOWN A).

7. Advanced Subgoal Infringement Problems

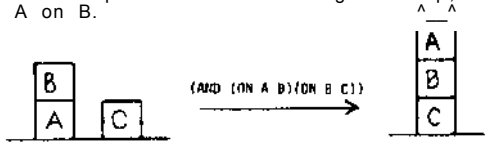
We are beginning to consider some of the more sophisticated (but more common) problems of subgoal violation. One of the most tenacious of these problems, that of of "fuzzy violation" or "infringement", requires the introduction of costs into what is otherwise a discrete, symbolic pattern matching annihilation detection facility. To il-

lustrate subgoal Infringement in our everyday domain, consider the Battle of the Trousers scenario: ;iost of us know that it is easier to put long pants on before shoes! Strictly speaking, nutting the shoes on first does not violate the gating condition for pulling the pants on: "path is unobstructed . However, it certainly increases the cost of the subsequent operation. Somehow, the. synthesizer must have access to nominal efforts required for strategies, be able to halt synthesis whenever the nominal effort for some strategy seems to have been seriously exceeded, then identify and unravel the culprit. We consider subgoal infringement to be an open topic.

b. Other Applications

The associative detection aspect of our design is applicable to certain forms of plan optimization as well as subgoal violation detection. One interesting form concerns a process we call "object, adoption . This is the process wherein the synthesizer commits itself to particular objects, times, locations, etc. in the instantiation of a plan in the concrete terms required for actual execution.

One instance of object adoption that provides an interesting example of SC-based plan optimization is that of free area selection. Consider the blocks problem illustrated below. A computer-wise mundane solution is to place B on the table, place A on B, attempt to place B on C, by clearing B's top, generating a subgoal violation, and proceeding as described above. The plan eventually gets built, Dut contains a superfluous intermediate location for B, namely the table. Clearly, the simpler solution would have been to place B directly on C in the first operation of clearing A's top, then placing A on B.



The problem here could have been avoided had the synthesizer been willing to leave B's location "floating" after clearing A's top in the first step (i.e., to defer the obiect adoption). In our system, this can be accomplished by describing some features of B's location, together with a default recommendation, via a CSA descriptor:

```
(*D* X ((CLASS X AREA)
        (XYSIZe X <the xy-size of the block>))
       (REC (PART-OF X TABLE)))
```

and by delegating responsibility to this descriptor ror eventually ensuring that it binds itself to some concrete object (a location in this case) before the end of the synthesis. This responsibility is phrased by associating with the descriptor an SC wnich will react to patterns (ON B -X), since the descriptor is willing to commit itself to any -X.

Now, each time the synthesizer is about to undertake the synthesis or a subgoal, it first shouts at this population of "concretization" SC's, in essence asking whether there are any floating references which would make short shrift of the subgoal, i.e. obviate a standard synthesis. In this example, the synthesizer eventually shouts (ON B C) before undertaking it as a subgoal. This shout excites our floating concretization SC, it responds affirmatively, committing itself at that point to C (since the tops of blocks are legitimate areas also), and, voiia!, the synthesizer is spared the synthesis !

There are some other interesting forms of subgoal optimization in this fashion. A future report will treat them in more detail.

9. Discussion and Conclusions

We have presented a general theory of detection and correction for subgoal annihilation problems which stem from bad orderings. The theory is, we feel, efficient, uniform, and general. It is efficient in the sense that it does only the minimum amount of work necessary to ameliorate annihilations, and does not incur resvnthesis. it is uniform in the sense that it is a theory of control that will function with any suitably formatted problem solving representation (in our case, CSA cause-effect schemata). It is general in the sense that it does not rely on any sort of domain-specific knowledge, with the possible exception of having to know the semantics of the system s predicates when planting appropriate guardian clusters.

There are many complex issues of subgoal annihilation that remain to be explored. One problem with any performance system such as ours is that it often has a myopic view of what it is doing. There seems to be a need for "meta" watchers (SC s) whose responsibility it is to watch the overall detection-unravalling process for recurring patterns of activity. Since our scheme is a form of relaxation, such an overseer would presumably intervene when it perceived the unravelling to be looping (i.e. trying to solve an insoluble ordering problem), and so forth.

other interesting problems have to do with bad strategy selection, as opposed to bad ordering. We have steered clear of this topic in this phase of our research, primarily because switching strategies in mid-stream seems to call for more radical reconsideration of the. total plan, possibly leading to partial or complete resynthesis. We do not propose to leave this problem at rest, only to sidestep it temporarily. In short, we will be working on our synthesizer for quite some time to come.

Acknowledgments

We wish to thank the other members of the .laryland CSA group (Milt Grinberg, Mache Creeger, John Boose, and Georgy Fekete) ior their participation in numerous discussions and for their suggestions. We wish also to thank NASA for the support of this research under Grant NSG-7253.

[DK1] Davis, K. and J. King, An Overview of Production Systems, Stanford AIM 271, 197b.

[EN1] Ernst, G. . and Newell, A., GPS: A case Study in Generality and Problem Solving, Academic Press, 1969

[MS1] McDermott, Drew V. and G.J. Sussman, The CONN1VER Reference Manual. MIT AI Memo 259a, Jan. 1974.

[N1] Nilsson, Nils J., Problem Solving Methods in Artificial Intelligence* McGraw-Hill, 197TT

[R1] Kieger, C, An Organization of Knowledge for Problem Solving and Language Comprehension, Artificial intelligence, vol. 7, no. 2, 1976.

[R2] Rieger, C, The Representation and Selection of Commonsense Knowledge for Natural Language Comprehension, Proc. Georgetown University Linguistics Roundtable, 1976.

[R3] Rieger, C., Spontaneous Computation in Cognitive Models, to appear in Cognitive Science.

[RG1] Rieger, C, and Grinberg, M., The Declarative Representation and Procedural Simulation of Causality in Physical Mechanisms, Proc. UCAI-77, MIT, August 1977.

[S1] Sacerdoti, Earl, The Nonlinear Nature of Plans, Proc. 4IJCAI, Tbilisi, USSR, Sept. 1975.

[S2] Sussman, Gerald J., A Computer Model of Skill Acquisition. MIT AI-TR 297, Aug. 1973.

[SW1] Sussman, Gerald Jay and Terry Winograd., Micro-Planner Reference Manual. MIT AI Memo 203, July, 1970.

[T1] Tate, A., Interacting Goals and their Use, Proc. 41JCAI. Tbilisi, USSR, Sept. 1975.

[W1] Waldinger, Richard, Achieving Several Goals Simultaneously, SRI Tech. Note 107, July, 1975.

[W2] Winograd, Terry, Frame Representations and the Procedural-Declarative Controversy, in Representation and Understanding, D.G. Bobrow and A. CoTTTns, ed. Academic Press, 1975.