Control of Refitting during Plan Reuse'

Subbarao Kambhampati
James A. Hendler

Center for Automation Research, Department of Computer Science
University of Maryland, College Park, MD 20742. *email: rao@alv.wnd.edu*

Abstract

In plan reuse, refitting is the process of modifying an existing plan to make it applicable to a new problem situation. An efficient refitting strategy needs to be *conservative,* i.e., it should minimally modify the existing plan to fit it to the new problem situation. In this paper we present techniques for conservative refitting control by utilizing the annotated dependency structures of an existing plan. The dependency structures arc used to select the refitting choices that minimize disturbance to the applicable parts of the existing plan. This localizes the refitting process and minimizes the cost of refitting. We describe how these techniques are incorporated into PRIAR, a framework for flexible plan reuse.

1. Introduction

The value of enabling a planning system to remember the plans it generates for later use was acknowledged early in planning research [4]. An important part of reusing an existing plan in a new situation is modifying its inapplicable parts so as to to make the overall plan applicable to the new problem situation. This modification, called *refitting,* is essential for the flexible reuse of existing plans. For the efficiency of reuse, it is imperative that this process be controlled to produce a plan for the new problem with minimal planning effort and minimal disturbance to the parts that are already applicable. However, very little work in plan reuse has concentrated on the issues involved in the control of refitting.

We have proposed an annotation-based framework for the flexible reuse of nonlinear hierarchical plans in the presence of a generative planner, and implemented it in a system called PRIAR [6,71. A major theme of the PRIAR reuse framework is that the annotated internal dependency structure of a plan can be utilized in focusing and controlling various phases of its reuse in a new problem situation. During reuse, the PRIAR system interprets an old plan in a new problem situation, localizes and characterizes the applicability failures of the interpreted plan, suggests appropriate *refit-tasks* for those failures, and finally reduces the refit-tasks with the help of a generative planner. In this paper, we present techniques to control this final refitting step in the PRIAR reuse framework. These techniques exploit the constraints imposed by the applicable parts of the old plan to control the generative planner during refitting. In particular, we introduce the concept of *task kernels* and describe how they are used to control the refitting process by ordering the refitting choices. We will show that choices based on this ordering reduce the planning effort by minimizing the disturbance to the applicable parts of the old plan. This in turn leads to minimization of refitting cost by localizing the refitting process.

After briefly describing the relevant parts of the PRIAR plan reuse cycle in section 2, we discuss the issues involved in refitting in this framework in section 3. In section 4, we discuss the ordering of choices during refitting. Section 5 gives examples and section 6 discusses related work. In the final section, we discuss the merits of this approach and describe some planned extensions. The techniques described in this paper are implemented in the PRIAR plan reuse system and are currently being tested.

The terminology used in this paper is generally consistent with the previous descriptions of hierarchical planning [3,13] and plan reuse [1,5]. We distinguish between two types of schema applicability conditions: preconditions and filter conditions (see [3]). In blocks world, *Clear (A)* is an example of a precondition as the planner can achieve it, while *Is-Block (A)* is an example of a filter condition since this condition can not be achieved by a blocks world planner.

If *n* is a node in the hierarchical task network representing a plan, we use the notation *R(n)* for the sub-reduction rooted at *n*, *A (n*) for the annotations on *n*, and *K (n*) for the the *task kernel* (to be defined later) of *n.* External preconditions (or *e-preconds)* of *n* are the preconditions of some node in *R (n*) that are validated by a node outside of *R (n*). For example, in Figure 1, *e-p* and *e-p-$* are among the *e-preconds* for *n*. We define the persistence conditions *(p -conds*) of a node *n* as the conditions that have to be protected over all or part of the range of *R* (n), to leave the rest of the plan undisturbed. In Figure 1, *p —c* 2 is a *p —cond* of *n*. The *e —conds* of a node are the effects of any node in *R (n*) that are used elsewhere in the plan. In Figure 1, e_r and e_u are the *e-conds* of *n.*

2. Overview of PRIAR Reuse Framework

The PRIAR system implements a plan reuse capability for a domain independent, hierarchical, nonlinear planner [7]. The planner in the system, based on NONUN [13], has the capability of reducing a task network into an executable plan, and resolving interactions between steps. Unlike normal nonlinear planners,

The support of the Defense Advanced Resean h Projects Agency and the U.S. Army Engineer Topographic Laboratories under contract DACA76-88-C-0008 is gratefully acknowledged.

however, it keeps the hierarchical task network showing the development of the plan, and annotates that with a description of the internal causal and decision dependency structure of the plan. These annotated plans are then stored in the plan library for later reuse.

PRIAR keeps two types of annotation structures on the plans it produces: node annotations and annotation-states. The annotations on a node n in a hierarchical task network reflect the dependencies between the sub-reduction *R(n)* rooted at *n* and the rest of the plan. The following information is included in the annotations on the node *n:* The *required-effects (goals)* of *R (n*), the filter conditions of the schema instance that reduced node n, the external preconditions of all the nodes belonging to *R(n*\ the *effects* of all the nodes belonging to *R(n)* which are used outside of *R (n*), and the conditions of the plan that have to persist over any part of *R(n).* Information regarding the nodes that are validated or affected by each of these conditions is also included in the annotations.

Annotation-states are maintained between successive steps of the developed plan. The annotation-state following a plan step consists of the useful outcomes of that plan step, and also those facts of the previous annotation-state that have to persist over this plan step for the validation of the rest of the plan. The Annotation-states are used for locating and characterizing applicability failures, while the node annotations are used to guide refitting. Most of the information in the PRIAR annotation structures is available to the planner during the planning, and is either incrementally annotated or derived from the datastructures used by the planner (in our case, generalized versions of Table of Multiple Effects and Goal Structure Table [13]).

Given a new planning problem consisting of an input situation and a goal specification, the reuse procedure progresses in the following stages:

1. Retrieval: A plan that solved a problem similar to the new problem is retrieved from the planner's library. As the focus of our research is on reuse methodology, at present the system uses a rather simple retrieval procedure based on partial unification with the goals of the new problem. We have however developed a technique that utilizes the annotation structures of a plan to judge the utility of reusing that plan in a new problem situation [8].

2. Interpretation: The old plan along with its annotations is mapped into the new problem situation marking the important differences between the old and the new situations.

3. Annotation Verification: The annotations of the interpreted plan are verified and upon finding various forms of verification failure, various types of refit-tasks are suggested to take care of the failures. The procedure distinguishes between precondition validation failure and filter condition validation failure. A refit-task to *reachieve precondition* is suggested in the former case, while a refit-task to *replace reduction* is suggested in the latter case. In addition, *dephantomize* refit-tasks are suggested to take care of failing phantom[1] node validations and *achieve extra goal* refit-tasks are suggested to take care of goals of the new problem that are not present in the interpreted plan. The annotation verification procedure also removes any parts of the plan whose sole purpose is to supply validations to the reductions that are being replaced or to goals that are unnecessary in the new problem situation.

4. Reduction of refit-tasks: The planner is invoked to reduce the annotation verified task network, which contains the applicable parts of the interpreted plan and the suggested refit-tasks, to produce a plan for the new problem.

The annotation verification procedure preserves the applicable portions of the old plan, and leaves the inter-step order of the old plan undisturbed to the extent possible. Because of this, many goals need not be reachieved and many interactions do not have to be re-analyzed during refitting. (For more details on the basic reuse cycle in PRIAR framework, see [7])

[1] Phantom goals refer to the goals of a planning problem that are achieved without step addition [11,13]. Such goals get established either through helpful interactions from other steps in the plan, or by persistence from the initial state.

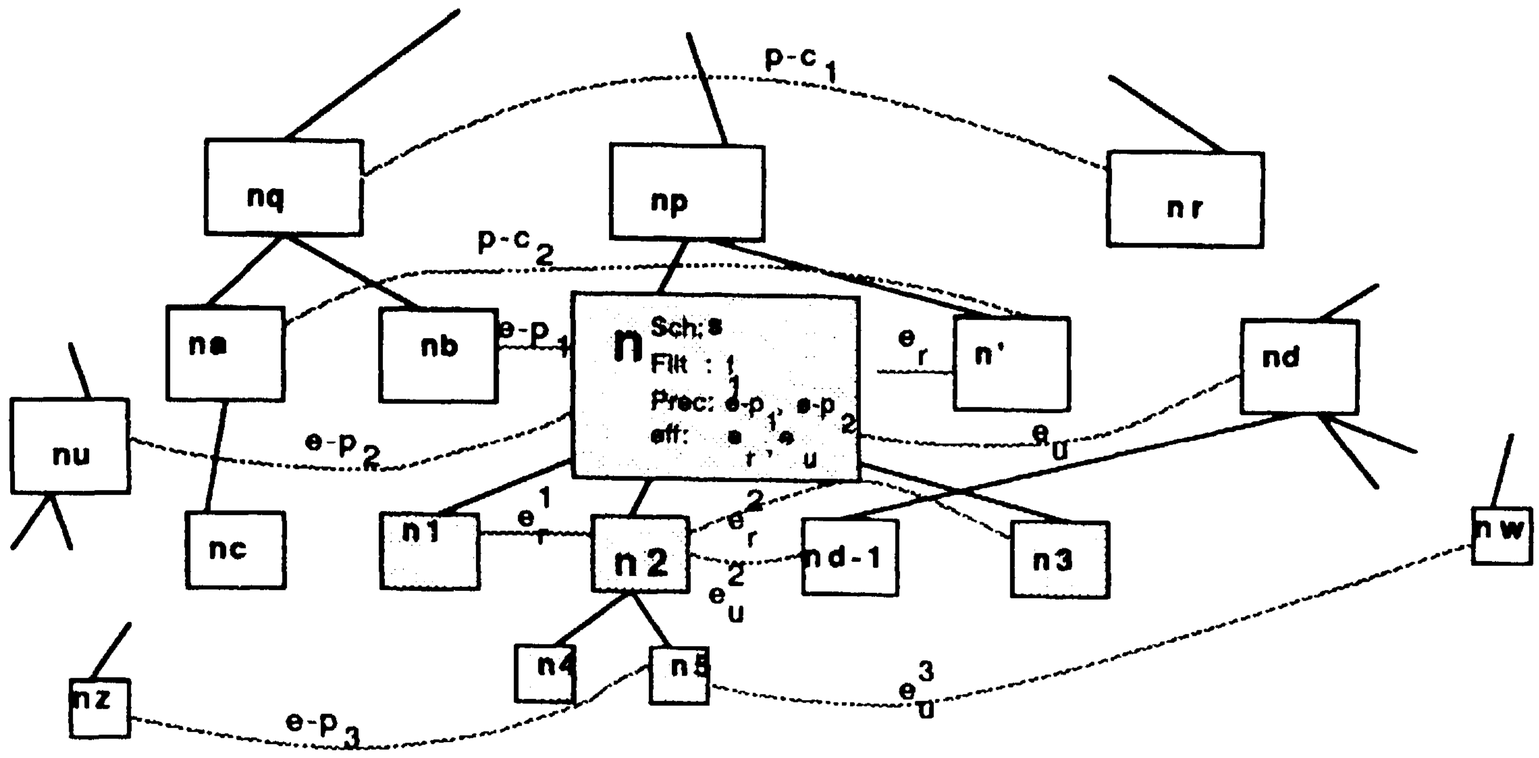
3. Refitting in PRIAR

Although the basic PRIAR reuse framework can suggest appropriate refit-tasks using the dependency information, it can not control the planner during the reduction of these refit-tasks. As we discussed in section 1, the ability to localize refitting such that it will leave the applicable parts of the old plan unaffected, is essential to a conservative reuse strategy. In this section we describe a technique to control refitting process by influencing the selection of the schema instance that is used to reduce a refit-task.

In general, the planner reduces refit-tasks just as it reduces other tasks during from-scratch planning. The default selection strategy is to fetch schema instances that are indexed under the goals of the refit-task, discard the instances whose filter conditions are not satisfied in the new problem situation, and select one of the remaining schema instances (arbitrarily) to reduce the refit-task. (This is representative of the normal schema selection procedure in most hierarchical nonlinear planners.) In many domains, the number of schema instance choices remaining at this stage is still quite large. In the case of the reduction of refit-tasks, this choice can be effectively controlled by exploiting the constraints afforded by the failing validations (that necessitated the refit-task) and by minimizing the disturbance to the rest of the applicable plan. To do this, we use the following choice selection procedure when the planner reduces a refit-task:

Step 1. Getting Plausible Choices: This step basically duplicates the default choice selection mechanism of the planner. In PRIAR's case, this consists of fetching all the schema instances that are indexed under the *?todo* pattern of the refit-task, and filtering out those instances whose filter conditions do not hold in the new problem situation. In the case of a *replace reduction* refit-task, this fetch can be made efficient by exploiting the fact that any instance that shares the failing filter conditions of the reduction being replaced would also be inapplicable.

Step 2. Ordering the choices to minimize interactions: This step is the heart of the refitting procedure. It uses a technique called *task kernel based ordering* to rank the remaining choices by the amount of disturbance they would cause to the applicable parts of the plan. Reducing the refit-task by the choice that is ranked best by this ordering would localize refitting, as it causes the least amount of disturbance to the rest of the plan. For the purpose of this ordering, we define a structure called task kernel for each refit-task. It encapsulates the conditions that should be preserved by any choice of task reduction for that refit-task, so as to leave the rest of the plan undisturbed.



Step 3. Selecting and Installing the Best choice: The schema instance ranked best by the task kernel based ordering is selected to reduce the refit-task. Next it has to be *installed* in the task network properly. This requires a comparison between the chosen schema instance, *S*, and the annotations $A(n)$ on the node n being reduced, to take care of the validations that *S* does not promise to preserve. It will involve (i) pruning parts of the task network whose sole purpose is to achieve *e—preconds* of $A(n)$ that are not required by S, (ii) adding refit-tasks to take care of any effects of *A (n)* that are used elsewhere in the plan, but are not guaranteed by S, and *(iii)* adding refit-tasks to take care of the *p—conds* of $A(n)$ that do not persist over *S*.

Once a schema instance is selected by the above procedure and installed properly, the control is passed back to the planner. The planner then reduces the refit task at n by the chosen schema instance in the normal way, detecting and resolving any interactions caused in the process. In the rest of the paper, we concentrate on step 2 of this refitting procedure.

4. Refitting with Task Kernels

4.1. Representation of Task Kernels

The task kernel is calculated for every refit-task node of an annotation-verified plan, prior to refitting. It estimates the potential interactions that a reduction choice at this node will have with the rest of the annotation-verified plan. It contains three different types of conditions— the effects of this node that are used elsewhere in the plan (called *e—conds* of the task kernel), the conditions that have to persist over this node *(p-conds),* and the external preconditions that are required by this node *(e-preconds).* Depending upon its type, a refit-task may not have all three types of task kernel conditions. (In fact, only *replace reduction* refit-tasks have all three.) The task kernel $K(n)$ of a node n is computed as follows:

(1) The *e—conds* of *K(n)* consist of (i) effects that arc inherited by *n* (these would be the *required effects* of the reduction of n), and *(ii)* any other effects of n that are used elsewhere in the plan. Thus, for example, in the hierarchical task network of Figure 1, the *e—conds* of the task kernel of node n will be e_r and e_u.

(2) The *p-conds* of $K(n)$ consist of (i) *p-conds* that are inherited by n from upper levels (these persist over some sub-reduction of which n is a part) and *{ii) p—conds* that have to persist over n at this level. Thus, in Figure 1, the *p-conds* of the task kernel of node n will consist of $p\text{-}C_1$ (which is inherited), and $p—c_2$ (which is at the level of n).

(3) The *e-preconds* of $K(n)$ consist of the external (unsupervised) preconditions of the schema instance that reduced n. Thus, in Figure 1, *e—preconds* of the task kernel of node n will *be e-pi* and $e\sim p_2$.

Suppose that the sub-reduction $R(n)$ of node n in Figure 1 has to be replaced (i.e., there is a *replace reduction* refit-task at n). Then the task kernel for this refit-task, $K(n)$, would be:

$$K(n) = [\ \textbf{e—cond:}\ e_r, e_u \quad \textbf{p—cond:}\ p-c_1, p-c_2 \quad \textbf{e—precond:}\ e-p_1, e-p_2\ ]$$

Notice that by this definition, the task kernel of a node *n* is a subset of the annotations $A(n)$ on that node. The reason kernel contents are not the same as the contents of the node annotations is the following. Since the purpose of the task kernel is to control the choices for the reduction of a refit-task, the task kernel conditions should be at the same level of detail as the conditions in the schema instances that comprise the reduction choices. Thus, *e-preconds, e-conds* and *p-conds* of $A(n)$, which come from parts of the task network that are below the level of n, are not made part of $K(n)$. For example, in Figure 1, the task kernel of node *n* will not include the *e—cond* e_u^3 and the *e-precond e-p3.* It is not possible to tell if a given choice of schema instance to reduce the refit-task at n will need, achieve or preserve such conditions, before it is completely reduced to the primitive level.

4.2. Ordering refitting choices using Task Kernels

The essential idea of using task kernels to order refitting choices is to choose the schema instance that preserves as many of the task kernel conditions as possible. This strategy has the following desirable effects:

(1) By preserving the *e-conds* of the task kernel, we avoid the need to add additional steps to reachieve the useful effects of this node.
(2) By preserving the *p-conds* of the task kernel, we minimize the interactions that refitting at this node will have with the rest of the plan, thus preserving as much of the old applicable plan as possible and reducing the cost of refitting.
(3) By preserving the *e-preconds* of the task kernel, we utilize the already existing sub-reductions that give rise to those preconditions and thereby reduce the cost of reduction of the current node (by avoiding the need to reachieve a new set of external preconditions).

We use the following procedure for ordering the schema instance choices to reduce a refit-task.

Step 1. Order the refitting choices (schema instances) according to the number of task kernel e-conds they preserve. For each *e-cond* e_c of the task kernel, if a schema instance *S* has an effect *e* that unifies with e_c, we consider that 5 preserves e_c.

Step 2. Pick the set of schema instances that are ranked best by step 1. Order them according to the number of task kernel p—conds they preserve. A task kernel *p—cond p—c* is considered preserved by a schema instance *S,* if the effects of *S do not negate p —c.*

Step 3. Pick the set of schema instances that are ranked best by step 2. Order them according to the number of task kernel e-preconds they preserve. An *e—precond e-p* is considered preserved by a schema instance *S*, if *S* has an external precondition that unifies with *e —p*

The best ranked schema instances at the end of this three-layered process are returned as the schemas that are preferred by the task kernel based ordering. One of them will then be selected by the planner to reduce the refit-task.

Notice that this three-layered procedure effectively imposes levels of importance on the three types of task kernel conditions, preferring the preservation of *e -conds*, *p —conds* and *e-preconds* in that order. These implicit levels of importance reflect the effect of violation of those conditions upon the overall refitting cost. We can also differentiate among the conditions of the same type, based on the relative effect the violation of those conditions will have on the cost of refitting. We suggest some methods for doing this in section 7.

Let us consider again the *replace-reduction* refit-task at node *n* in Figure 1. We discussed the task kernel of this refit-task in section 4.1. Suppose there are four schema instances capable of reducing this refit-task, specified as follows:

Si: ?todo: e, prec: $e\text{-}p_6, e\text{-}p_5$
eff: e,^,-.p-c_2
S_2: ?todo: e, prec: e-p!,$e\text{-}p_6, e\text{-}p_7$
eff: e_r,-ip-c_2-'P-C7
S_3: ?todo: e, prec: $e\text{-}p_{l.}e\text{-}p_7$
eff: e_r, e_u,-ip-c_2.-'P-C7
S_4: ?todo: e_r prec: $e\text{-}p_6, e\text{-}_5$
eff: ey.e^,-ip-c2,-ip-ci

The ordering with respect to the task kernel *e-conds* would prefer the schema instances $S_i S_3$ and S_4, since S_2 does not preserve e_u. Next, from these three, the ordering with respect to the task kernel *p-conds* would prefer S_x and S_3 since they preserve one p-cond *(p-c_1)* while S_4 preserves none. Finally, from these two choices, the ordering with respect to the task kernel *e-preconds* picks S_3 since it can utilize one previously achieved precondition *(e-pb.* Thus, S_3 would be suggested as the best refitting choice.

5. Example

Consider the example of reusing the plan for the three block stacking problem (3bs) shown in Figure 2(a), to make a plan for the four block stacking problem (4bs) shown in Figure 2(b). PRIAR interprets the 3bs into 4bs using the object mapping *[A →LB,→K,C→*J]. Figure 2(c) shows the task network after the annotation verification. The annotation verification procedure reveals two validation failures in the interpreted plan. The validation to the phantom node *A [Clear* (L)] is failing, and *On (J J*) is an extra goal. PRIAR then places a refit-task *A [Clear (L*)] at the site of the failing validation for re achieving the phantom goal condition (node *n* 11 in Figure 2(c)), and a refit-task *A [On (JJ*)] parallel to the interpreted plan to take care of the extra goal (node *n* 10 in Figure 2(c)). (For details of this process, see [7].)

Consider the refit-task *A [Clear* (L)] at node *n* 11 in figure 2(c). From the methods described in the previous section, we can calculate nil's *task kernel* as:

AT (nil) = [e-cond: *Clear* (L)
p--ond: *Clear(J\Clear(K\On(JJ)*]

Since this refit-task does not replace any parts of the previous plan, its task kernel will not have any *e-preconds. On (J J*) occurs in the task kernel because *A [On (J,J*)] is not yet ordered with respect to the plan and thus it is preferred that *R* (n 11) preserve *On (J,J).*

The default selection strategy finds that this refit-task can be reduced by the following three schema instances:

A : MakeClear-Table(LJ)
eff: Clear (L),On (J.Table)
B : MakeClear -Block (L,J*K)*
eff: Clear (L),On (J,K -Clear (K)
C: *MakeClear-Block (LJJ*)
eff: Clear (L),On (JJ),-,Clear (I)

Where, the schema *MakeClear-Block(?X,?Y,?Z)* clears *?X* by putting *?Y* (which is on top of *?X)* on top of ?Z, and the schema *MakeClear -Table (?X,?Y)* clears *?X* by putting *?Y* on the *Table* (which is always clear).

When the above schema instances are ordered using *K(n* we find that the task kernel *e-cond Clear*(L) is preserved by all the three choices. So, they all survive to the next layer ordering with respect to the task kernel *p-conds*. At this stage, however, the *p-cond Clear(K)* is violated by choice B and the *p-cond On (J,J)* is violated by choices A and B. So the task kernel based ordering prefers choice C, *Make-Clear {L,J,J).* This will be sent to the planner as the schema instance with which node n ll should be reduced. Notice that this choice would in fact minimize the refitting cost. It also has the serendipitous side effect of achieving the extra goal, *On (J,l).*

In this example, while the choice between *B* and C could have been made through a delayed binding of objects (e.g., the *merge objects* critic in NOAH [11]), such a strategy will not be able to deal with *A*, which is an instance of a different schema. Thus, controlling the choice among alternative schema instances involves more than delayed binding of objects. In particular, there may be different schemas that can reduce the same refitting task, with significant differences among their effects and preconditions. Choices made with the help of task kernel based ordering will be able to effectively control refitting in such cases also.

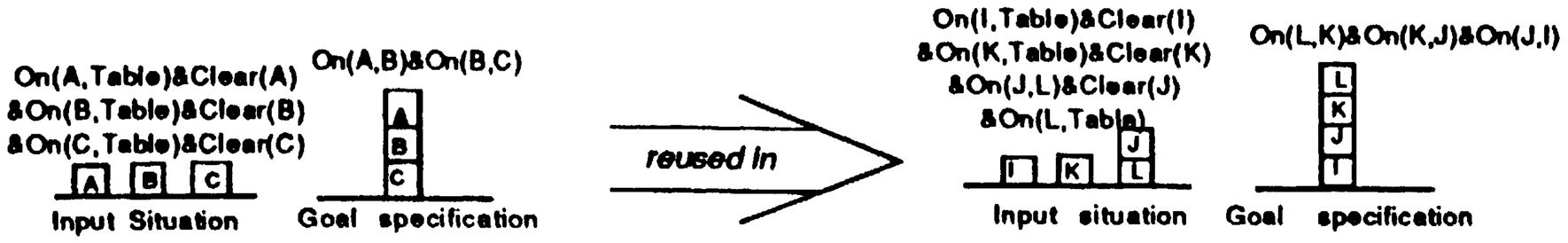


Figure 2(a). Three blocks problem

Figure 2(b). Four blocks problem

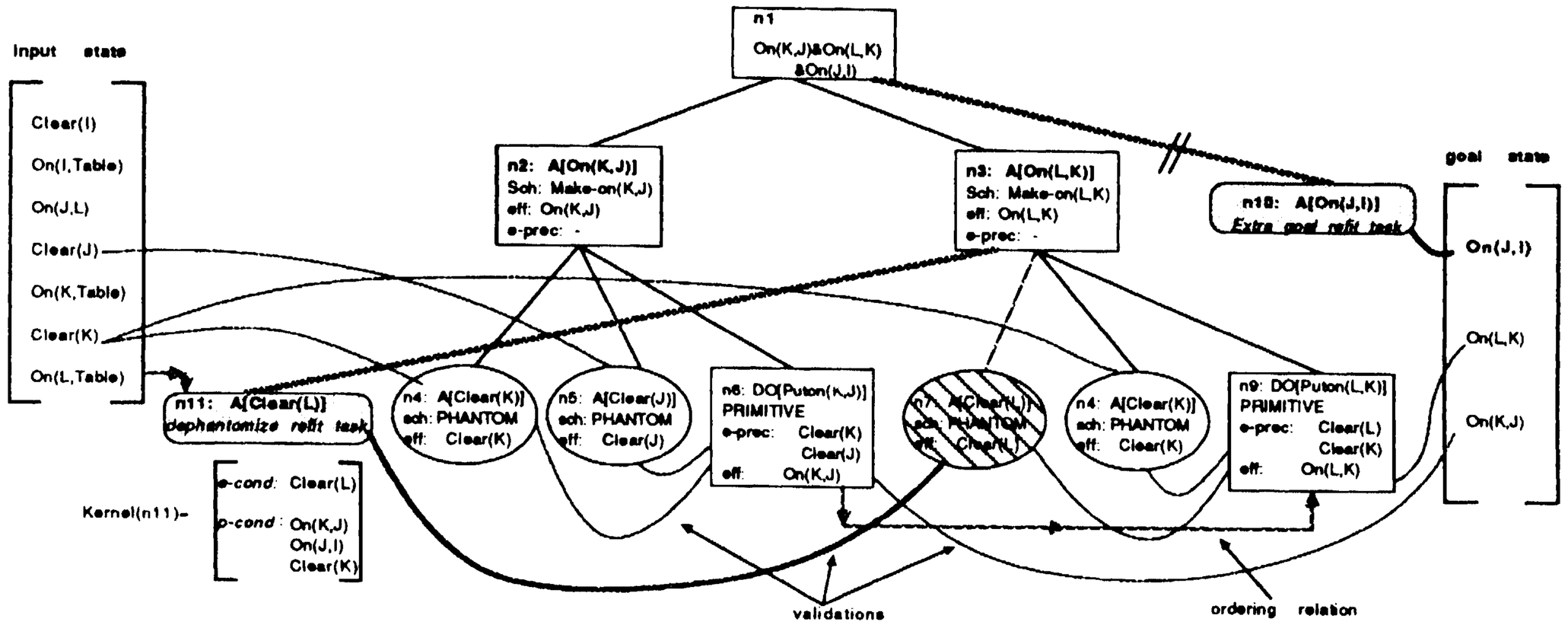


Figure 2(c). The task network showing the annotation-verified plan prior to refitting

As another example, consider the following plan to attend a birthday party at San Fransisco:

$$GO\text{-}BY\text{-}CAR(Airport) \rightarrow BUY(Ticket) \rightarrow \rightarrow GOTO\text{-}SHOP(Giftshop, Airport) \rightarrow \rightarrow BUY\text{-}FROM\text{-}SHOP(?Gift) \rightarrow TAKE\text{-}FLIGHT(Sf)$$

Suppose this plan has to be reused in a new situation where the airport does not have a gift shop, i.e., *—chouse (Gift shop .Airport).* This breaks the *filler condition* for the sub-plan *GOTO(Giftshop .Airport),* necessitating a *replace reduction* refit-task at this place. The task kernel of this refit-task will be:

$$[\ \textbf{e–cond}: at(?Shop), sells(?Shop, ?Gift)$$
$$\textbf{p–cond}: at(Airport), poss(Ticket), poss(Money)$$
$$\textbf{e–precond}: at(Airport)]$$

Thus, the task kernel based ordering prefers choices that do not take the agent away from the airport, do not involve spending too much money (because money will be needed at *BUY-FROM-SHOP (? Gift)),* and do not cause the agent to lose possession of the ticket. Consider two alternatives for refitting this plan:

A: $GOTO\text{-}SHOP(Candyshop, Airport)$
B: $GOTO\text{-}SHOP(Giftshop, Plaza\text{–}near\text{–}home)$.

The task kernel based ordering prefers choice A to choice B. This is because A preserves the *p-cond at (Airport)* and also utilizes the old *e-precond at (Airport),* while B does not. Notice that to localize the refitting and reduce refitting cost, this is the best choice. However, this may not necessarily be the choice that leads to a plan with minimal execution cost.

6. Related Work

Interal dependency structures of plans have been used previously in replanning and plan revision to locate failures and to undo the wrong decisions (e.g., [15]). The novelty of oui approach is that we use the dependency structures also to control the refitting of the plan. Minimization of disturbance to the overall plan has been used as a basis for modification and repair of plans in other systems. Hammond's case-based planner, CHEF [5], uses the explanation of an execution time failure to suggest various minimally interactive ways of repairing that failure, and then uses domain dependent heuristics to select among the repair strategies. The debugger in Simmons' GTD system [12] selects among possible repairs by doing a causal simulation of the plan with the suggested repairs, followed by an assessment of the global effect of the suggested repair on the final outcome of the plan. Aiterman's PLEX system [1] depends on the helpful cues from the new problem situation to trigger the retrieval of appropriate refitting choices. In comparison to these systems, PR1AR utilizes the already existing dependency structures of a plan to anticipate the interactions that would be caused by potential refits, and chooses the ones which cause least amount of disturbance to the rest of the plan. Domain dependent solutions to plan refitting are embodied in PRIDE [9] and CAS [14] which store specific hand-coded strategies for repairing individual failing preconditions, and use them to guide refitting. We realize that information such as these domain dependent repair rules, and justifications for planning decisions [2], may be able to exert a stronger control over refitting. The goal of our work, however, has been to demonstrate that even in the absence of

such domain specific information, refitting can be effectively guided by the dependency structures of the plan.

7. Discussion

The utility of the refitting control strategy presented in this paper ultimately depends on the trade-off between the effort invested in ordering the refitting choices with the use of task kernels, and the effort that would be required to do the refitting without such an ordering. Since poor refitting choices cause interactions and give rise to increased planning cost (by increasing the plan length and possibly causing costly backtracking), we believe that this trade-off falls in favor of the control strategy. The strategy is especially useful in cases where the number of refit-tasks is small compared to the length of the plan. In such cases, choices based on the task kernel based ordering lead to substantial savings in the refitting cost by keeping the refitting localized. *Conservatism* is thus an emergent property of this refitting control strategy.

There are two types of costs associated with planning—the cost of planning, and the cost of execution of the produced plan. By localizing changes to the plan being reused, the refitting control strategy reduces the cost of planning for the new problem. It does not however guarantee the optimality of the cost of execution of the plan. This may not be a serious limitation since many domains where reuse has utility are also domains where it is more important to arrive at a plausible plan quickly than to guarantee the optimality of the produced plan.

As the refitting control strategy relies on the dependency structures that are provided by the planner, an important issue to be addressed is the effect of incorrect and incomplete domain models on refitting. When the planner's domain models are not correct and complete, it is possible that both the *plans* that are generated from scratch and the plans that are produced by PRIAR through reuse may fail during execution, thereby necessitating plan repair and debugging (e.g., CHEF [5], GTD [12]). One of the goals of PRIAR's refitting strategy is to ensure that the plans it produces are at the same level of correctness as the the ones that are produced by the planner from scratch (the difference would be in the efficiency of producing the plan). If a refitted plan fails, it would be because of the inadequacy of the planner's domain models rather than due to incorrectness of modification. This is important since debugging is a costly operation. PRIAR's philosophy is that robust refitting of plans is not possible in the absence of internal dependency structures of the plan, and that dependency structures of the plan, even if they are incomplete, can help in locating applicability failures and localizing the modification.

The task kernel based ordering presented in section 4 can be refined further by gauging the importance of the conditions in each layer of the kernel more precisely. For example, the hierarchical level of a validation could be used to measure the importance of that validation. In this sense, in the task kernel of node *n* in Figure 1, the *p-cond p-c1* which is inherited from an upper level, should be considered more important than $p\text{-}c_2$, since it may take more effort to refit the plan if $p\text{-}c_x$ is violated than *if p-c 2* is violated. Another way of measuring the importance of a validation is to measure the number of nodes in the reduction which give rise to or use that validation. Thus in Figure 1, the external precondition $e\text{-}p_x$ is given by the node n_b, and so a measure of its importance is the number of nodes in $R(n_b)$. We are currently exploring such alternative ordering strategies to make the kernel-based ordering reflect the estimated cost of refitting more faithfully.

Finally, although we presented the task kernel based ordering in the context of plan reuse, we believe that it can also be profitably used in guiding replarming (repairing a plan in response to an execution failure due to an unexpected event). Another area where we expect potential utility for this type of control strategy is design reuse [10]. Notice that all these problems share the requirement of minimal modification to an existing structure to make it satisfy the constraints of a new problem.

Acknowledgements

We would like to thank Lindley Darden, Jack Mostow and one of the IJCAI referees for providing helpful comments on an earlier draft of this paper. Thanks are also due to Chaitali Chakrabarti for her help in preparing this document.

References

1 R. Alterman, "An Adaptive Planner'*, *Proceedings of 5th AAAI,* 1986, 65-69.

J. Carbonell and M. Vclose, "Integrating Derivational Analogy into a General Problem Solving Architecture", *Proceedings of Case-Based Reasoning Workshop,* 1988.

E. Charaniak and D. McDermott, "Chapter 9: Managing Plans of Actions", in *Introduction to Artificial Intelligence,* Addison-Wesley Publishing Company, 1984, 485-554.

R. Fikes and N. Nilsson, "STRIPS: a New Approach to the Application of Theorem Proving to Problem Solving", *Artificial Intelligence 2* (1971), 189-208.

K. J. Hammond, "CHEF: A Model of Case-Based Planning'*, *Proceedings of 5th AAAI,* 1986, 267-271.

S. Kambhampati and J. A. Hendler, "Adaptation of Plans via Annotation and Verification", *1st Intl. Conf on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems,* 1988, 164-170.

S. Kambhampati and J. A. Hendler, "Flexible Reuse of Plans via Annotation and Verification", *Proceedings of 5th IEEE Conf. on Applications of Artificial Intelligence,* 1989, 37-44.

8. S. Kambhampati, An Annotation-Based Framework for Flexible Plan Reuse, (Ph.D. Thesis in preparation).

9. S. Mittal and A. Araya, "A Knowledge-Based Framework for Design*', *Proceedings of 5th AAAI,* 1986, 856-865.

10 J. Mostow, "Design by Derivational Analogy: Issues in the Automated Replay of Design Plans", Rutgers University ML-Tech. Rep.-22, March 1987. (To appear in Artificial Intelligence Journal).

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

12. R. Simmons, "A Theory of Debugging Plans and Interpretations", *Proceedings of 7th AAAI,* 1988, 94-99.

13. A. Tate, "Generating Project Networks", *Proceedings of 5th IJCAI,* 1977, 888-893.

14. R. M. Turner, "Issues in the Design of Advisory Systems: The Consumer-Advisor System", GIT-ICS-87/19, School of Information and Computer Science, Georgia Institute of Technology, April 1987.

15. D. E. Wilkins, "Recovering from execution errors in SIPE", *Computational Intelligence 1* (1985), 33.