June 1990 Report No. STAN-CS-90-1312

A Validation Structure Based Theory of Plan Modification and Reuse

by

Subbarao Kambhampati and James A. Hendler

Department of Computer Science

Stanford University

Stanford, California 94305

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED

4. TITLE AND SUBTITLE

A Validation Structure Based Theory of Plan Modification

5. FUNDING NUMBERS

..

6. AUTHOR(S)

Subbarao Kambhampati and James A. Hendler

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Computer Science Department
Stanford University
Stanford, CA 94305

8. PERFORMING ORGANIZATION REPORT NUMBER

STAN-CS-90-1312

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

DARPA

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

N00014-88-K-0560

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

A framework for the flexible and conservative modification of plans enables a planner to modify its plans in response to incremental changes in their specifications, to reuse its existing plans in new problem situations, and to efficiently replan in response to execution time failures. We present a theory of plan modification applicable to hierarchical nonlinear planning. Our theory utilizes the validation structure of stored plans to yield a flexible and conservative plan modification framework.

14. SUBJECT TERMS

15. NUMBER OF PAGES
53

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	16. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
			unlimited

A Validation Structure Based Theory of Plan Modification and Reuse*

Subbarao Kambhampati

Center for Design Research and Department of Computer Science
Stanford University
Bldg. 530, Duena Street
Stanford CA 943054026
email: *rao@sunrise.stanford.edu*

James A. Hendler

Computer Science Department
University of Maryland
College Park, MD 20742

Abstract

A framework for the flexible and conservative modification of plans enables a planner to modify its plans in response to incremental changes in their specifications, to reuse its existing plans in new problem situations, and to efficiently replan in response to execution time failures. We present a theory of plan modification applicable to hierarchical nonlinear planning. Our theory utilizes the validation structure of stored plans to yield a flexible and conservative plan modification framework. The validation structure, which constitutes a hierarchical explanation of correctness of the plan with respect to the planner's own knowledge of the domain, is annotated on the plan as a by-product of initial planning. Plan modification is formalized as a process of removing inconsistencies in the validation structure of a plan when it is being reused in a new (changed) planning situation. The repair of these inconsistencies involves removing unnecessary parts of the plan and adding new non-primitive tasks to the plan to establish missing or failing validations. The resultant partially reduced plan (with a consistent validation structure) is sent to the planner for complete reduction. We discuss the development of this **theory in the PRIAR** system, present an empirical evaluation of this theory, and characterize its completeness, coverage, efficiency and limitations.

* The support of the Defense Advanced Research Projects Agency and the U.S. Army Engineer Topographic Laboratories under contract DACA76-88-C-0008 (to the University of Maryland Center for Automation Research), and that of Office of Naval Research under contract N00014-88-K-0620 (to Stanford University Center for Design Research), and the Washington D.C. Chapter of A.C.M. through the ''1988 Samuel N. Alexander A.C.M. Doctoral Fellowship Grant''. Partial support for this research also came from ONR grant N00014-88-K-0560 and NSF grant IRI-8907890, the Systems Research Center and UM Institute for Advanced Studies.

1. Introduction

The ability to incrementally modify existing plans to make them conform to the constraints of a new or changed planning situation is very useful in plan reuse (reusing existing plans to solve new planning problems), replanning (modifying a current plan in response to executing time failures), and incremental planning (updating a plan in response to evolving specifications during interactive planning). Two important desiderata for the plan modification capability are *flexibility* and *conservatism.* Flexibility is the ability to modify a plan to handle a wide variety of changes in the specification. Conservatism is the ability to minimally change the existing plan to make it fit to the new problem situation. The former is required for effective coverage of modification, while the latter is needed to ensure efficiency.

While the value of plan modification has been acknowledged early in planning research [6, 9], the strategies developed were inflexible, in that they could reuse or modify a given plan in only a limited number of situations, and could deal with only a limited variety of applicability failures. There was no general framework for conservatively modifying an existing plan to fit it to the constraints of a new problem situation. A major shortcoming with these approaches was that the stored plans did not represent enough information about the internal dependencies of the plan to permit flexible modification. For example, reuse based on macro-operators [6] built from sequences of primitive plan steps was unable to modify intermediate steps of the macro-operators as the macro-operators did not represent the intermediate decisions and dependencies corresponding to their internal steps. Even in cases where the need for the dependency information was recognized (e.g. [5, 34]), a systematic representation and utilization of such structures in plan reuse and modification was not attempted.

We present a theory of plan modification that allows flexible and conservative modification of plans generated by a hierarchical nonlinear planner. Hierarchical planning is a prominent method of abstraction and least-commitment in domain-independent planning [4]. Our theory of plan modification proposes *validation structure* as a way of representing the internal dependencies of a hierarchical plan and provides algorithms for annotating the validation structure on the plans during plan generation. It systematically explores the utility of the annotated validation structure in guiding and controlling all the processes involved in flexible plan reuse and modification. **The PRIAR reuse** system [15, 16, 17, 14, 13, 12] is our implementation of this theory. This paper presents the plan modification framework used in **PRIAR** and evaluates its performance, completeness, coverage, efficiency and limitations.

1.1. Overview of the PRIAR Plan Modification Theory

The plan modification problem that is addressed in **PRIAR** is the following: *Given (i* **) a** planning problem P^n (specified by a partial description of the initial state I^n and goal state G"), *(ii)* an existing plan R^o (generated by a hierarchical nonlinear planner), and the corresponding planning problem P^o, *Produce* a plan for P^n by minimally modifying R^o. Figure 1 shows the schematic overview of the **PRIAR** plan modification framework.

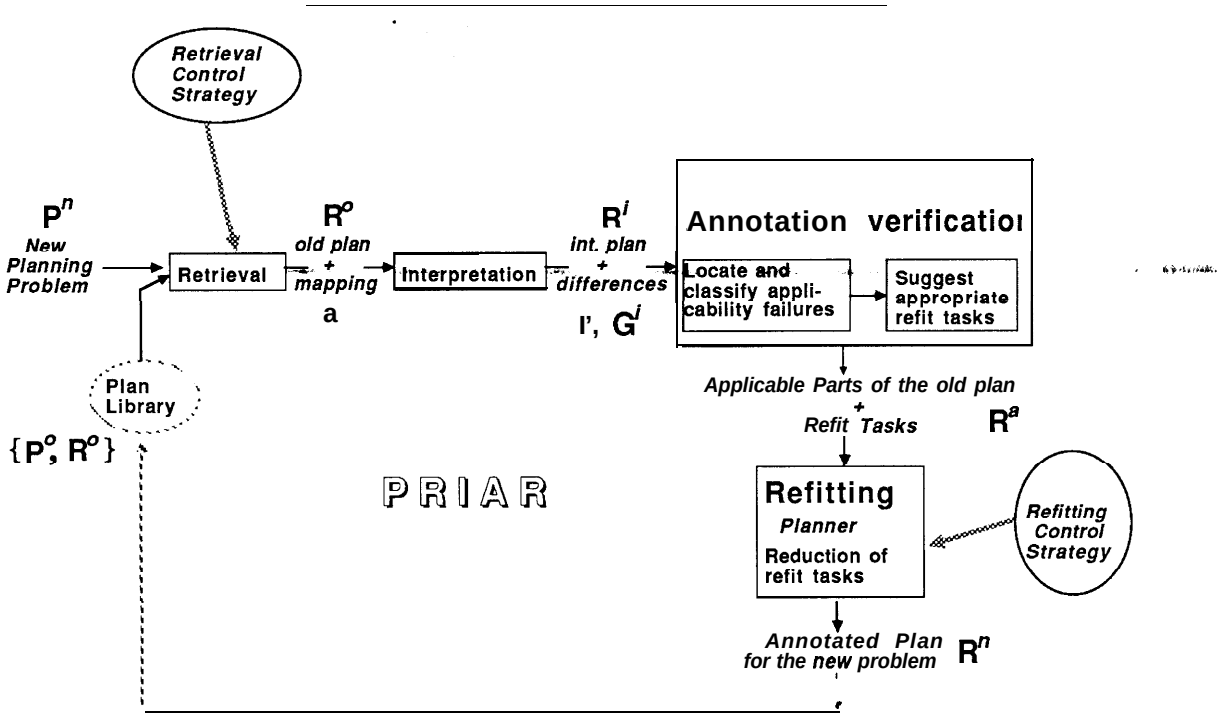

Figure 1. Schematic overview of PRIAR

In the PRIAR modification framework, the internal causal dependencies of a hierarchical plan which are relevant to guide its reuse and modification are formalized as the *validation* structure of the plan. The validation structure can be seen as a form of hierarchical explanation of correctness for the plan with respect to the planner. Individual tasks of the hierarchical plan are annotated with information about their role in the plan validation structure. **PRIAR** provides efficient algorithms for acquiring these annotations as a by-product of planning.

When an existing plan is modified to solve a new planning problem, the applicability failures, the redundancies, and the shortcomings that may arise in the process are formally characterized as *inconsistencies* in the plan's validation structure. Modification in the **PRIAR** framework is formalized as a process of repairing the inconsistencies in the validation structure of a given plan when it is mapped into the new problem situation. Given the new problem P^n, and *an annotated* plan R^o, **PRIAR'S** modification process proceeds in the following steps:

(1) Mapping and Interpretation: An appropriate mapping a between the objects of $[P^o, R^o]$ and P^n is chosen with the help of the validation structure of R^o, and R^o is mapped into P^n with it. Next, the differences between the initial and goal state specifications of P^o and P^n are marked. The resulting interpreted plan, R^i, is typically a plan with an inconsistent validation structure.

(2) Annotation Verification: The inconsistencies in the validation structure of R^i are located, and appropriate repairs are suggested. The repairs include removing parts of R^i that are unnecessary and adding non-primitive tasks (called *refit tasks)* to establish any required new validations. The resulting annotation-verified plan R^a will have a consistent validation structure but is typically only partially reduced. It consists of all the applicable parts of R^i and any newly introduced refit tasks.

(3) Refitting: The refit tasks specified during the annotation verification phase constitute sub-planning problems for the hierarchical planner. The refitting process involves reducing them with the help of the planner. Conservatism is ensured during this process through the use of a heuristic control strategy which minimizes the disturbance to the applicable parts of R^a by estimating the disturbance caused to its validation structure.

Computational savings stem from the fact that the complexity of solving the sub-planning problems during refitting is much less than the complexity of solving the entire planning problem from scratch. This is supported by the results of the empirical studies in blocks world, which showed that plan modification provides 20-98% savings (corresponding to speedup factors of 1.5 to 50) over pure generative planning, with the highest gains shown for the most complex problems tested (the details of these studies are provided in section 4.1).

1.2. Comparison to Previous Work

Here we will briefly summarize some broad distinctions between our theory, and the previous approaches to plan modification; a more detailed discussion of related work appears in section 6 and in [13].

Representations of plan internal dependency structure have been used by several planners previously to guide plan modification (e.g., the triangle tables and the macro operators of [6] and [11]; the decision graphs of [9] and [5]; the plan rationale representation of [34]). However, our work is the first to systematically characterize the nature of such dependency structures and their role in plan modification. It subsumes and formalizes the previous approaches, provides a better coverage of applicability failures, and allows the reuse of a plan in a larger variety of new planning situations. Unlike the previous approaches, it also explicitly focuses on the flexibility and conservatism of the plan modification. The modification is fully integrated with the generative planning, and aims to reduce the average case cost of producing correct plans. In this sense, **PRIAR's** strategies are complementary to the plan debugging strategies proposed in **GORDIUS** [27] and **CHEF** [8], which use an explanation of correctness of the plan with respect to an external (deeper) domain model-generated through a causal simulation of the plan to guide the debugging of the plan-to compensate for the inadequacies of the planner's own domain model. Similarly, PRIAR's validation structure based approach to plan modification stands in contrast to approaches which rely on domain dependent heuristic modification of the plan (e.g. [S, 1,221). Our approach of grounding plan modification on validation structure guarantees the correctness of the modification with respect to planner's domain

model and reduces the need for a costly modify-test-debug type approach.

1.3. Organization of the Paper

The rest of this section introduces some preliminary notation and terminology used throughout the paper. Section 2 presents the notion of plan validation structure, explains the motivation behind remembering it along with each generated plan, and develops a scheme for annotating the plans. Section 3 develops the basic modification processes, and explains how they utilize the plan validation structure. In particular, it provides the details of the mapping, annotation verification and refitting processes and presents an example of plan modification in this framework. It also includes a brief discussion of the control strategies for guiding refitting and retrieval. Section 4 contains an empirical and theoretical analysis of the **PRIAR** plan modification theory. It summarizes the results of the empirical evaluation experiments conducted on the implemented system, and discusses the completeness, coverage, flexibility and efficiency of the modification framework. Section 5 contains a detailed discussion of related work and section 6 summarizes the research. Appendix A contains an annotated trace of the **PRIAR** system solving a problem and Appendix B contains the specification of the domain used in the empirical evaluation.

1.4. Preliminaries, Notation and Terminology

This paper develops a theory of plan modification in the context of hierarchical nonlinear planning. Hierarchical nonlinear planning (known also as hierarchical planning) is a prominent method of abstraction and least commitment in domain independent planning. A good introduction to this methodology can be found in [4]. Some well known hierarchical planners include **NOAH** [25], **NONLIN** [30] and **SIPE** [33]. For a review of these and other previous approaches to planning, see [10].

In hierarchical planning, a partial plan is represented as a task network consisting of high level tasks to be carried out. A task network is a set of tasks with partial chronological ordering relations among the tasks. Planning involves reducing these high level tasks with the help of predefined ''task reduction schemas,'' to successively more concrete subtasks. The task reduction schemas are given to the planner *a priori* as part of the domain specification. The collection of task networks, at increasing levels of detail showing the development of the plan, is called the "hierarchical task network" or ''HTN'' of the plan. Planning is considered complete when the all the leaf nodes of the **HTN** are either primitive tasks (tasks that cannot be decomposed any further) or phantom goals (tasks that are achieved as side-effects of some other tasks). The entire tree structure in Figure 2 shows the hierarchical plan for a simple blocks world planning problem. In the following, we provide formal definitions of some of these notions, to facilitate the development in the rest of the paper.

$1.1. Partial Plans and Task networks: A partial plan P is represented as a task network and can be formalized [37] as a 3-tuple $\langle T, O, \Pi \rangle$, where T is a set of tasks, 0 defines a partial

ordering among elements of T, and Π is a set of conditions along with specifications about where those conditions must hold. Each task T has a set of applicability conditions, denoted by *conditions(T),* and a set of expected effects, denoted by effects(T), where each set consists of literals in first order predicate calculus. Elements of Π *are* called *protection intervals* [4], and are represented by 3-tuples $\langle E, t_1, t_2 \rangle$, where $t_1, t_2 \in T$, $E \in effects(t_1)$ and E has to necessarily persist up to t_2.

91.2. Schemas and Task Reduction: A task reduction schema S can itself be formalized as a mini task network template that can be used to replace some task $t \in T$ of the plan P, when certain applicability conditions of the schema are satisfied. Satisfying the applicability conditions this way involves adding new protection intervals to the resultant plan. Thus when the set of applicability conditions $\{C_f\}$ of an instance S_i of a task reduction schema S can be satisfied at a task t in a partial plan P, then t can be reduced with S_i. The reduction, denoted by $S_i(t)$, is another task network $\langle T_s, O_s, \Pi_s \rangle$. The task t will be linked by a *parent* relation to each task of T_s[1]. The plan P' resulting from this task reduction is constructed by incorporating $S_i(t)$ into P. During this incorporation step, some harmful interactions may develop due to the violation of established protection intervals of P. The planner handles these harmful interactions either by posting additional partial ordering relations, or by backtracking over previous planning decisions. When the planner is successful in incorporating $S_i(t)$ into P and resolving all the harmful interactions, the resultant plan, P' can be represented by the task network $P': \langle T \cup T_s - \{t\}, O' \cup O_s \cup O_I, \Pi' \rangle$, where:

(1) O' is computed by appropriately redirecting the ordering relations involving the reduced task t to its children

(2) O_I are the ordering relations introduced during the interaction resolution false

(3) Finally, the protection intervals II' is computed by *(i)* combining II and II,, *(ii)* adding any protection intervals that were newly established to support the applicability conditions of the schema instance S_i and *(iii)* appropriately redirecting the protection intervals involving the reduced task t to its children.

During the redirection in the last step, the planner converts any protection interval $\langle E, t_1, t_2 \rangle \in \Pi$ where $t_1 = t$ to $\langle C, t_{sb}, t_2 \rangle$, and converts any protection intervals where $t_2 = t$ to $\langle C, t_1, t_{se} \rangle$ (where t_{sb}, t_{se} are appropriate tasks belonging to T_s). The various implemented planners follow different conventions about how the appropriate t_{sb} and t_{se} are computed. For example, irrespective of the protected condition E, NONLIN [29] makes t_{sb} to be t_{beg}, and t_{se} to be t_{end}, where t_{beg} and t_{end} are the beginning and ending tasks of T_s (i.e., no task of T_s precedes t_{beg} or follows t_{end}) respectively. Other conventions might look at the effects and

[1] When the task t is of the form *achieve* (C), and C can be achieved directly by using the effects of some other task $t_c \in T$, *then,* t becomes a *phantom* task and its reduction becomes $\langle \{phantom(C)\}, \emptyset, \emptyset \rangle$. A new protection interval $\langle C, t_c, t \rangle$ will be added to the resultant plan.

conditions of tasks belonging to T_s to decide t_{sa} and t_{sb}. For the purposes of this paper, either of these conventions is admissible.

§ 1.3. Completed Plan: A task network is said to represent a *completed plan* when none of its tasks have to be reduced further. The planner cannot reduce certain distinguished tasks of the domain called *primitive tasks.* (It is assumed that the planner already knows how to execute such tasks.) Further, if all the required effects of a task are already true in a given partial plan, then that task does not have to be reduced any further (such tasks are called *phantom goals* [4]).

§ 1.4. Hierarchical Task Network (HTN): The hierarchical development of a plan $P:\langle T, O, \Pi\rangle$ is captured by its hierarchical task network (abbreviated as **HTN).** HTN(P) is a 3-tuple $\langle P:\langle T, O, \Pi\rangle, T^*, D\rangle$, where T^* is the union of set of tasks in T and all their ancestors, and D represents the parent-child relations between the elements of T^*. *The* set Π is the set of protection intervals associated with HTN(P). (For convenience, we shall abbreviate HTN(P) to **HTN,** where the reference to P is unambiguous, and also refer to the members of T^* as the nodes of HTN.) The **HTN** of a plan thus captures the development of that plan in terms of the corresponding task reductions. We shall refer to the number of leaf nodes in the HTN, $|T|$ as the length of the corresponding plan, and denote it by N_P.

For the sake of uniformity, we shall assume that there are two special primitive nodes n_I and n_G in the HTN corresponding to the input state and the goal state of the planning problem, such that ***effects*** (n_I) comprise the facts true in the initial specification of the problem, and *conditions* (n_G) contain the goals of the problem. The notation **"$n_1 < n_2$''** (where n_1 and n_2 are nodes of **HTN)** is used to indicate that n_1 is ordered to *precede* n_2 in the partially orderered plan represented by the **HTN** (i.e., $n_1 \in predecessor^*(n_2)$, where the *predecessor* relations enforce the partial ordering among the nodes of the HTN). Similarly, ''$n_1 > n_2$'' denotes that n_1 is ordered to ***follow*** n_2, and ''$n_1 \,/\!/\, n_2$'' denotes that there is no ordering relation between the two nodes (n_1 is parallel to n_2). The set consisting of a node n and all its descendents in *the* **HTN** is defined as the *sub-reduction* of n, and is denoted by $R(n)$. Following [4, 30], we also distinguish two types of plan applicability conditions: the preconditions (such as *Clear (A)* in the blocks world) which the planner can achieve, and the filter conditions (such as *Block*(A)) which the planner cannot achieve. We shall use the notation ''$F \vdash f$'' to indicate that f deductively follows from the set of facts in F. Finally, the modal operators ''$\Box$'' and ''$\Diamond$'' denote necessary and possible truth of an assertion.

2. Validation Structure and Annotations

Here we formally develop the notion of the validation structure of a plan as an explicit representation of the internal dependencies of a plan, and provide motivation for remembering such structures along with the stored plan. We will begin the discussion by defining our notion of validation, present a scheme for representing the validation structure locally as

annotations on individual nodes of a HTN, and finally discuss algorithms for efficient computation of these node validations.

2.1. Validation Structure

92.1. Validation: A *validation v* is a 4-tuple $\langle E, n_s, C, n_d \rangle$, where n_s and n_d are leaf nodes belonging to the HTN, and the effect E of node n_s (called the *source)* is used to satisfy the applicability condition C of node n_d (called the *destination). C* and *E* are referred to as the *supported condition* and *the supporting effect* respectively of the validation. As a necessary condition for the existence of the validation v, the partial ordering among the tasks in HTN must satisfy the relation $n_s < n_d$. The *type* of a validation is defined as the type of the applicability condition that the validation supports (one of *filter condition, precondition, phantom goal). Notice* that every validation $v : \langle E, n_s, C, n_d \rangle$ corresponds to a protection interval $\langle E, n_s, n_d \rangle \in \Pi$ of the HTN (that is, the effect E of node n_s is protected from node n_s to node n_d). This correspondence implies that there will be a finite set of validations corresponding to a given HTN representing the development of a plan; we shall call this set V. (If ξ is the maximum number of applicability conditions for any action in the domain, then IV I $\leq \xi N_P$, where N_P is the length of the plan as defined above [13].)

Figure 2 shows the validation structure of the plan for solving a block stacking problem ,3BS (also shown in the figure). Validations are represented graphically as links between the effect of the source node and the condition of the destination node. (For the sake of exposition, validations supporting conditions of the type $Block(?x)$ have not been shown in the figure.) As an example, $\langle On(B,C), n_{15}, On(B,C), n_G \rangle$ is a validation belonging to this plan since *On (B ,*C) is required at the goal state n_G, and is provided by the effect *On (B ,C)* of node n_{15}.

The level of a validation is defined as the reduction level at which it was first introduced into the HTN (see [13] for the formalization of this notion). For example, in Figure 2, the validation $\langle Block(A), n_1, Block(A), n_{16} \rangle$ is considered to be of a higher level than the validation $\langle On(A, Table), n_1, On(A, Table), n_{16} \rangle$, since the former is introduced into the HTN to facilitate the reduction of task n_3 while the latter is introduced during the reduction of task n_9. A useful characteristic of hierarchical planning is that its domain schemas are written in such a way that the more important validations are established at higher levels, while the establishment of less important validations is delegated to lower levels. Thus, the level at which a validation is first introduced into an HTN can be taken to be predictive of the importance of that validation, and the effort required to (re)establish it.[2] The validation levels can be pre-computed efficiently at the time of annotation.

[2] We assume that domain schemas having this type of abstraction property are supplied/encoded by the user in the first place. What we are doing here is to exploit the notion of importance implicit in that abstraction.

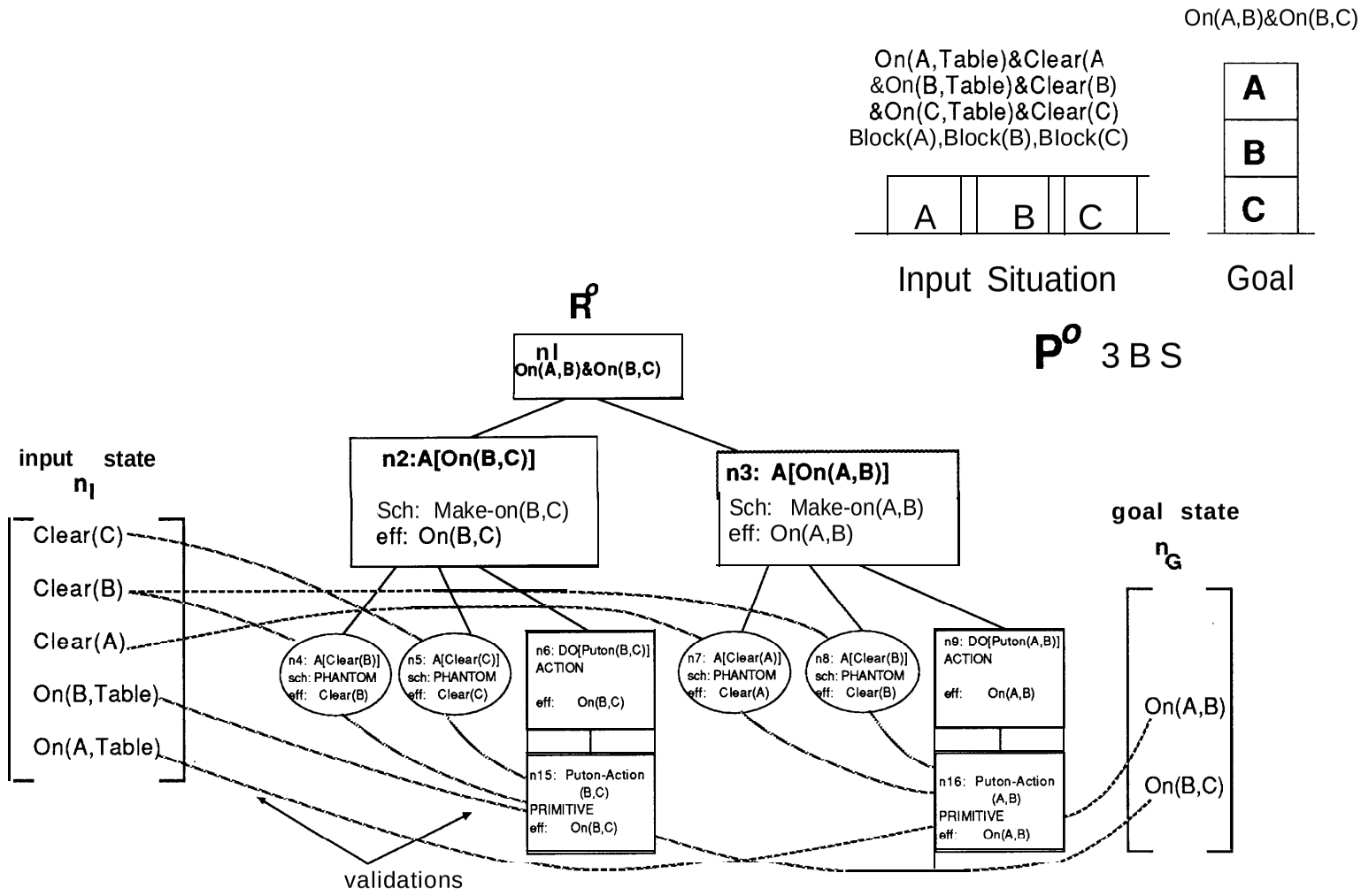

Figure 2. Validation Structure of 3BS Plan

As the specification of the plan changes or as the planner makes new planning decisions, the dependencies of the plan as represented in its validation structure get affected. The notions of consistency and inconsistencies, developed below, capture the effects of such changes on the plan validation structure.

92.2. Inconsistencies and Consistency of Validation Structure: A HTN is said to have a *consistent validation structure* if it does not have *any unnecessary, missing* or *failing* validations. The unnecessary, missing or failing validations in a HTN will be referred to as *inconsistencies* in its validation structure.

- A validation v : $\langle E, n_s, C, n_d \rangle$ is considered a *failing validation* when the corresponding protection interval is undone. More formally, v is a failing validation iff:

$$[\ effects(n_s) \vdash E\ ] \vee [\ \exists n'\ s.t.\ \Diamond\ (n_s < n < n_d) \wedge effects(n') \vdash \neg E\ ]$$

Thus, for every non-failing validation v : $\langle E, n_s, C, n_d \rangle$ of a HTN, the effects of n_s should entail E, and E should *necessarily* persist from n_s to n_d.

- A validation v : $\langle E, n_s, C, n_d \rangle$ is considered *an unnecessary validation* if it is not required to support the condition C at node n_d. This could happen either because n_d is no longer a part of the HTN or because it no longer requires the condition C.

- There is a *missing validation* corresponding to a condition, node pair $\langle C', n' \rangle$ of the HTN iff $\nexists v : \langle E, n_s, c, n_d \rangle$ *s.t.* $C = C' \wedge n_d = n'$ (i.e., the condition C' is not supported by any validation).

Let us consider the example of the 3BS plan shown in Figure 2. If the specification of this plan is changed such that *On (A ,B)* is no longer a goal, then $\langle On(A,B), n_{16}, On(A,B), n_G \rangle$ will be an unnecessary validation. Further, if the new specification contains a goal *On (A ,D)*, then since there is no validation supporting the condition node pair $\langle On(A,D), n_G \rangle$, there is a missing validation corresponding to this pair. Finally, if we suppose that the new specification contains *On (D ,*A) in its initial state, then the validation $\langle Clear(A), n_I, Clear(A), n_7 \rangle$ will be failing, as *effect (n,)* $\vdash$ *Clear (A)*.

From these definitions, it should be clear that in a HTN with a consistent validation structure, each applicability condition of a node (including each goal of n_G) will have a non-failing validation supporting it. (A completely reduced HTN with a consistent validation structure constitutes a valid executable plan.)

2.2. Annotating Validation Structures

Having developed the notion of validation in a plan, our next concern is representing the validation structure of the plan locally as annotations on individual nodes of a HTN. The intent is to let these annotations encapsulate the role played by the sub-reduction below that node in the validation structure of the overall plan, so that they can help in efficiently gauging the effect of

any modification at that node on the overall validation structure of the plan. We achieve this as follows: For each node $n \in$ HTN we define the notions of *(i)* e-conditions(n), which are the externally useful validations supplied by the nodes belonging to $R(n)$ (the sub-reduction below n) *(ii) e* -preconditions(n), which are the externally established validations that are consumed by nodes of *R(n),* and *(iii) p-conditions(n),* which are the external validations of the plan that are required to persist over the nodes of $R(n)$.

§2.3. E-Conditions (External Effect Conditions): The e -conditions of a node *n* correspond to the validations supported by the effects of any node of $R(n)$ which are used to satisfy applicability conditions of the nodes that lie outside the sub-reduction. Thus,

$$e\text{-}conditions\ (n) = \{ v_i : \langle E, n_s, C, n_d \rangle \mid v_i \in \mathbf{V};\ n_s \in R(n);\ n_d \notin R(n) \}$$

For example, the e-conditions of the node n_3 in the HTN of Figure 2 contain just the validation $\langle On(A,B), n_{16}, On(A,B), n_G \rangle$ since that is the only effect of $R(n_3)$ which is used outside of $R(n_3)$. The e-conditions provide a way of stating the externally useful effects of a sub-reduction. They can be used to decide when a sub-reduction is no longer necessary, or how a change in its effects will affect the validation structure of the parts of the plan outside the sub-reduction.

From the definition, the following relations between the e-conditions of a node and the e-conditions of its children follow:

(1) If *n* is a leaf node, then $R(n) = \{n\}$ and the e -conditions of *n* will simply be all the validations of HTN whose source is *n.*

(2) If *n* is not a leaf node, and $n_c \in children(n)$, and $v_c : \langle E, n_s, C, n_d \rangle$ is an e-condition of n_c, then v_c will also be an e-condition of *n* as long as $n_d \notin R(n)$ (since $R(n_c) \subseteq R(n)$, $[n_s \in R(n_c)] \Rightarrow [n_s \in R(n)]$).

(3) If $v : \langle E, n_s, C, n_d \rangle$ is an e-condition of *n,* then $\exists\, n_c \in children(n)$ such that v is an *e*-condition of *n,*. This follows from *the* fact that if $n_d \notin R(n)$ then $\forall n_c \in children(n)$, $n_d \notin R(n,)$, and that if $n_s \in R(n)$, then $\exists\, n_c \in children(n)$ such that $n_s \in R(n,)$.

These three relations allow PRIAR to first compute the e-conditions of all the leaf nodes of the HTN, and then compute the e-conditions of the non-leaf nodes from the e-conditions of their children.

§2.4. *E-Preconditions* **(External Preconditions):** The e-preconditions of a node *n* correspond to the validations supporting the applicability conditions of any node of $R(n)$ that are satisfied by the effects of the nodes that lie outside of $R(n)$. Thus,

$$e\text{-}preconditions(n) = \{ v_i : \langle E, n_s, C, n_d \rangle \mid v_i \in \mathbf{V};\ n_d \in R(n);\ n_s \notin R(n) \}$$

For example, the e-preconditions of the node n_3 in the HTN of Figure 2 will include the validations $\langle Clear(A), n_I, Clear(A), n_7 \rangle$ and $\langle Clear(B), n_I, Clear(B), n_8 \rangle$. The e-preconditions of

a node can be used to locate the parts of rest of the plan that will become unnecessary or redundant, if the sub-reduction below that node is changed.

From the definition, the following relations between the e-preconditions of a node and the e-preconditions of its children follow:

(1) If n is a leaf node, then $R(n) = \{n\}$ and the e -preconditions of n will simply be all the validations of HTN whose destination is n .

(2) If n is not a leaf node, and $n_c \in children(n)$, and $v_c : \langle E, n_s, C, n_d \rangle$ is an e-precondition of n_c , then v_c will also be an e-precondition of n as long as $n_s \notin R(n)$ (since $R(n_c) \subseteq R(n)$, $n_d \in R(n)$).

(3) If $v : \langle E, n_s, C, n_d \rangle$ is an e-precondition of n , then $\exists\, n_c \in children(n)$ such that v is an *e*-precondition of *n,*. This follows from the fact that if $n_s \notin R(n)$ then $\forall n_c \in children(n)$ $n_s \notin R(n,)$ and that if $n_d \in R(n)$, then $\exists\, n_c \in children(n)$ such that $n_d \in R(n,)$.

These three relations allow PRIAR to first compute the e-preconditions of all leaf nodes of the **HTN,** and then compute the e-preconditions of the non-leaf nodes from the e-preconditions of their children.

From the definitions of e-conditions and e-preconditions, it should be clear that they form the forward and backward validation dependency links in the HTN. For the sake of uniformity, the set of validations of type $\langle E, n_i, G, n_G \rangle$, (where G is a goal of the plan) are considered *e*-preconditions of the goal node n_G. Similarly, the set of validations $\langle I, n_I, C, n_i \rangle$, (where I is a fact that is true in the input state of the plan) are considered e-conditions of the input node n_I.

$2.5. P-Conditions (Persistence Conditions): P-conditions of a node n correspond to the protection intervals of the HTN that are external to $R(n)$, and have to persist over some part of $R(n)$ for the rest of the plan to have a consistent validation structure. We define them in the following way:

A validation $v_i : \langle E, n_s, C, n_d \rangle \in V$ is said to *intersect* the sub-reduction $R(n)$ below a node n (denoted by "$v_i \otimes R(n)$") if there exists a leaf node $n \in R(n)$ such that n falls between n_s and n_d for some total ordering of the tasks in the **HTN.** In other words,

$$v_i : \langle E, n_s, C, n_d \rangle \otimes R(n) \;\; iff \;\; \Diamond (n_s < n < n_d)$$

Using the definition of the validation, we can re-express this as

$$v_i : \langle E, n_s, C, n_d \rangle \otimes R(n) \;\; iff \;\; \begin{bmatrix} \exists\, n' \in R(n) \; s.t. \; children(n') = \emptyset \wedge \\ (n_s < n' < n_d \vee n_s \parallel n' \vee n_d \parallel n') \end{bmatrix}$$

(Note: Given that $n_s < n_d$, the only cases in which $\Diamond (n, < n' < n_d)$ are *(i)* n' is already totally ordered between n_s and n_d, i.e., $\Box (n_s < n' < n_d)$ or *(ii)* $n' < n_d \wedge n' \parallel n_s$ or *(iii)* $n_s < n' \wedge n' \parallel n_d$ or *(iv)* $n' \parallel n_s \wedge n' \parallel n_d$. Using the transitivity of "<" relation, We can simplify this disjunction to $n_s < n' < n_d \vee n_s \parallel n' \vee n_d \parallel n'$.)

A validation $v_i : (E , n_s , C , n_d \rangle \in \mathbf{V}$ is considered a p-condition of a node n iff v_i *intersects R (n)* and neither the source nor the destination of the validation belong to *R(n).* *Thus,*

$$P\text{-}conditions(n) = \{ v_i : \langle E , n_s , C , n_d \rangle \mid v_i \in \mathbf{V}; n_s , n_d \notin R(n); v_i \otimes R(n) \}$$

From this definition, it follows that if the effects of any node of $R(n)$ violate the validations corresponding to the p-conditions of n, then there will be a potential for harmful interactions. As an example, the p-conditions of the node n_3 in the HTN of Figure 2 will contain the validation $\langle On(B, C), n_{15}, On(B, C), n_G \rangle$ since the condition $On(B, C)$, which is achieved at n_{15} would have to persist over $R(n_3)$ to support the condition (goal) $On(B, C)$ at n_G. The *p-conditions* are useful to gauge the effect of changes made at the sub-reduction below a node on the validations external to that sub-reduction. They are of particular importance in localizing the changes to the plan during refitting [15].

From the definition of p -conditions, the following relations follow:

(I) $p\text{-}conditions(n_I) = p\text{-}conditions(n_G) = 0$.

(2) When n is a leaf node, (i.e., $children(n) = \emptyset$), $R(n)$ will be $\{n\}$, and the definition of *p-conditions(n)* can be simplified as follows. From the definition of $\otimes$,

$$v_i : \langle E , n_s , C , n_d \rangle \otimes \{n\} \equiv n_s \not\parallel n \vee n_d \not\parallel n \vee (n_s < n < n_d) \equiv \neg(n < n_s \vee n > n_d)$$

and, thus when n is a leaf node

$$P\text{-}conditions(n) = \{ v_i : \langle E , n_s , C , n_d \rangle \mid v_i \in \mathbf{V}; n_s \neq n; n_d \neq n; \neg(n < n_s \vee n > n_d) \}$$

(3) If $n_c \in children(n)$, and $v_c : (E , n_s , C , n_d \rangle \in p\text{-}conditions\&)$, then $v_c \in p\text{-}conditions(n)$ iff $n_s , n_d \notin R(n)$. This follows from the fact that if $v_c \otimes R(n,)$ then $\exists\, n' \in R(n,)$ which satisfies the ordering restriction of "$\otimes$". Since $R(n_c) \subseteq R(n)$, *we* also have $n' \in R(n)$ and thus $v_c \otimes R(n)$. *So,* as long as $n_s , n_d \notin R(n)$, v_c will also be a p-condition of n.

(4) If n is not a leaf node and $v \in p\text{-}conditions(n)$ then $\exists\, n_c \in children(n)$ *s.t.* $v \in p\text{-}conditions(n_c)$. This follows from the fact that for v to be a p-condition of *n,* there should exist a leaf node *n'* belonging to $R(n)$ such that the ordering restriction of the "$\otimes$" relation is satisfied. But, from the definition of sub-reduction, any leaf node of $R(n)$ should also have to be a leaf node of the sub-reduction of one of its children. So, $\exists\, n_c \in children(n)$ *s.t.* $n' \in R(n,)$. Moreover, as the source and destination nodes of v do not belong to $R(n)$, they will also not belong to *R (n,).*

These relations provide a way of computing the p-conditions of a non-leaf node from the *p-conditions* of its children, which will be exploited in computing the annotations.

$2.6. Validation States: If n is a *primitive task* belonging to the **HTN,** then we define structures called *preceding validation state,* $A^p(n)$, and *succeeding validation state,* $A^s(n)$, as follows:

$$A^p(n) = \text{e-preconditions}(n) \cup \text{p-conditions}(n)$$

$$A^s(n) = \text{e-conditions}(n) \cup \text{p-conditions}(n)$$

Thus, the validation states $A^p(n)$ and $A^s(n)$ are collections of validations that should be preserved by the state of the world preceding and following the execution of task *n*, for the rest of the plan to have a consistent validation structure. For example, the plan can be successfully executed from any state *W* of the world such that

$$\forall v : \langle E, n_s, C, n_d \rangle \in A^s(n_I),\ W \vdash E$$

Thus the validation states can be used to gauge how a change in the expected state of the world will affect the validation structure of the plan. This is useful both in reuse, where an existing plan is used in a new problem situation, and in replanning, where the current plan needs to be modified in response to execution time expectation failures. The validation states can be seen as a generalization of **STRIPS** ' triangle tables [6], for partially ordered plans.

The validation states also provide a clean framework for execution monitoring for partially ordered plans. If **EXEC** denotes the set of actions of the plan *P* that have been executed by the agent until now, and *W* denotes the current world state, then the set of actions of the plan that may be executed next, *E (P ,W* **,EXEC),** is computed as (see [20]):

$$E(P, W, \mathbf{EXEC}) = \{ n_e \mid primitive(n_e) \wedge \forall v : \langle E, n_s, C, n_d \rangle \in A^p(n_e)\ s.t.\ n_d \notin \mathbf{EXEC},\ W \vdash E \}$$

As long as the agent executes any of the actions in *E (P ,W* **,EXEC)** next, it is assured of following the plan, while taking into account any unexpected changes in the world state. When *E (P ,W* ,EXEC)=∅, replanning (or modification of the current plan *P)* will be necessitated (see [20]).

2.3. Computing Annotations

In the **PRIAR** framework, at the end of a planning session, the HTN showing the development of the plan at various levels of abstraction is retained, and each node of the HTN is annotated with the following information: (1) *Schema(n),* the schema instance that reduced node *n (2) Orderings (n),* the ordering relations that were imposed during the expansion of *n (see* §1.2)[3] (3) e -preconditions(n) (4) e -conditions(n), and *(5) p-conditions(n).*

Schema(n) and *Orderings (n*) are remembered in a straight forward way during the planning itself. The rest of the node annotations are computed in two phases: First, the annotations for the leaf nodes of the **HTN** are computed with the help of the set of validations[4], V, and the partial ordering relations of the HTN. Next, using relations between the annotations of a

[3] This information is useful for undoing task reductions during plan modifications; see section 3.2.1.

[4] As mentioned previously, the set of validations can be computed directly from the set of protection intervals associated with the plan. Most hierarchical planners keep an explicit record of the protection intervals underlying the plan. NONLIN [30], for example, maintains this information in its GOST data structure.

node and its children, the annotations are propagated to non-leaf nodes in a bottom up breadth-first fashion. The exact algorithms are given in [13], and are fairly straightforward to understand given the development of the previous sections. If N_P is the length of the plan (as measured by the number of leaf nodes of the HTN), the time complexity of annotation computation can be shown to be $\mathbf{0}$ (N_P^2) [13]. Note that the ease of annotation computation is to a large extent an advantage of integrating planning and plan modification, as all the relevant information is available in the plan-time datastructures. With respect to storage, the important point to be noted is that PRIAR essentially remembers only the HTN representing the development of the plan and not the whole explored search space. If the individual validations are stored in one place, and the node annotations are implemented as pointers to these, the increase in storage requirements (as compared to the storage of the un-annotated HTN) is insignificant. This small increase in the storage requirements can be justified in light of the multiple uses of the stored information.

While the procedures discussed above compute the annotations of a HTN in one-shot, often during plan modification, **PRIAR** needs to add and remove validations from the **HTN** one at a time. To handle this, **PRIAR** also provides algorithms (called *Add-Validation* and *Remove-Validation*) to update node annotations consistently when incrementally adding or deleting validations from the HTN [13]. PRIAR uses these procedures to re-annotate the HTN and to maintain a consistent validation structure after small changes are made to the **HTN** during the modification process. They can also be called by the planner any time it establishes or removes a new validation (or protection interval) during the development of the plan, to dynamically maintain a consistent validation structure. The time complexity of these algorithms is $\mathbf{0}$ (N_P). Whenever these procedures add or remove a validation, they also update the protection intervals (II) of the HTN appropriately.

3. Modification by Annotation Verification

We will now turn to the plan modification process, and demonstrate the utility of the annotated validation structure in guiding plan modification. Throughout the ensuing discussion, we will be following the blocks world example case of modifying the plan for the three block stacking problem 3BS *(i.e.,* R^o = 3BS) shown on the left side in Figure 3 to produce a plan for a five block stacking problem S5BS1[5] *(i.e.,* P^n = S5BS1), shown on the right side. We shall refer to this as the 3BS→S5BS1 example.

3.1. Mapping and Interpretation

In **PRIAR,** the set of possible mappings between [P^o ,R ^{o}] and P^n are found through a partial unification of the goals of the two problems. There are typically several semantically consistent mappings between the two planning situations. While the PRIAR modification framework

[5] It may be interesting to note that S5BS1 contains an instance of what is known as the *Sussman Anomaly* [3]

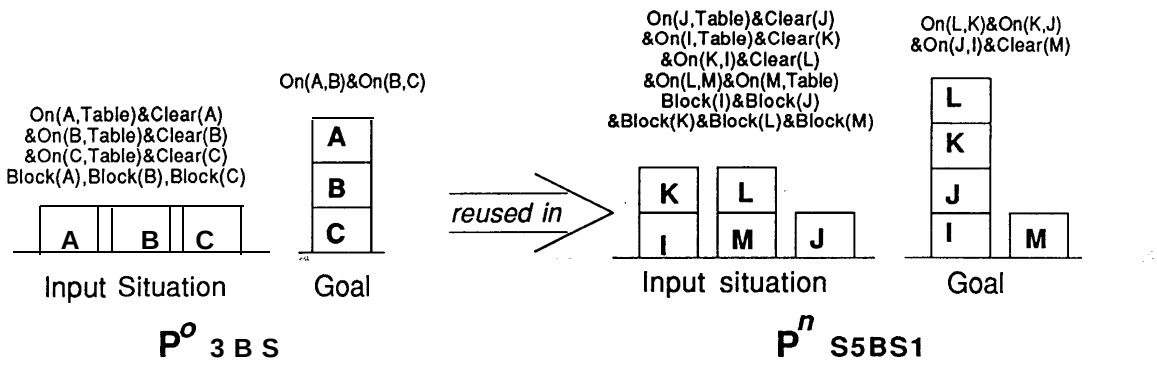

Figure 3. 3BS→S5BS1 Modification problem

would be able to succeed with any of those mappings, selecting the right mapping could considerably reduce the cost of modification. The mapping and retrieval methodology used by **PRIAR** [13,171 achieves this by selecting mappings based on the number and type of inconsistencies that would be caused in the validation structure of R^o. While the details of this strategy are beyond the scope of this paper, a brief discussion appears in section 3.4.2. For the present, we shall simply assume that such a mapping is provided to us. (It should be noted that the mapping stage is not important when PRIAR is used to modify a plan in response to incremental changes in its specification, as is the case during incremental planning or replanning for example [20])

The purpose of the interpretation procedure is to map the given plan, R^o along with its annotations into the new planning situation P^n, marking the differences between the old and new planning situations. These differences serve to focus the annotation verification procedure (see section 3.2.1.) on the inconsistencies in the validation structure of the interpreted plan. Let I^o and G ^{o} be the partial descriptions corresponding to the required initial state, and the set of goals to be achieved by R^o respectively. Similarly, let I^n and G ^{n} be the corresponding descriptions for the new problem P^n. The interpreted plan R^i is constructed by mapping the given plan R^o along with its annotations into the new problem situation, with the help of the mapping a. Next, the interpreted initial state I', and the interpreted goal state, G^i are computed as $I^i = I^n \cup I^o \cdot \alpha$ and $G^i = G^n \cup G^o \cdot \alpha$ (where "·" refers to the operation of object substitution). Finally, some facts of I' and G^i are marked to point out the following four types of differences between the old and new planning situations:

(1) A description (fact) $f \in I^i$ is marked an *out* ***fact*** iff $(f \in I^o \cdot \alpha) \wedge (I'' \nvdash f)$.

(2) A description (fact) $f \in I'$ is marked a *new* ***fact*** iff $(f \in I^n) \wedge (I^o \cdot \alpha \, P\text{-} f)$.

(3) **A** description (goal) g $\in$ G ^{i} is marked an *extra goal* iff $(g \notin G^o \cdot \alpha) \wedge (g \in G^n)$.

(4) A description (goal) (g $\in$ G ^{i}) is marked an *unnecessary goal* iff **(g** $\in G^o \cdot \alpha) \wedge$ (g $\notin$ G"). At the end of this processing, R^i, I^i and **G** ^{i} are sent to the

annotation verification procedure.

3.1.1. Example

Let us assume that the mapping strategy selects a = $[A \rightarrow K, B \rightarrow J, C \rightarrow I]$ as the mapping from 3BS to S5BS1. Figure 4 shows the result of interpreting the 3BS plan for the S5BS1 problem. With this mapping, the facts *Clear* (L) and *On* $(K, Table)$, which are true in the interpreted 3BS problem, are not true in the input specification of S5BS1. So they are marked *out* in I^i. The facts *Clear (L), On (M ,Table), On (I ,Table), On (L ,M)* and *On* $(K$,I) are true in S5BS1 but not in the interpreted 3BS. These are marked as *new* facts in I'. Similarly, the goals *On (L ,K)* and *Clear (M)* of S5BS 1 are not goals of the interpreted 3BS plan. So, they are marked *extra goals* in G^i. There are no *unnecessary goals.*

3.2. Annotation Verification and Refit Task Specification

At the end of the interpretation procedure, R^i may not have a consistent validation structure (see §2.2) as the differences between the old and the new problem situations (as marked in I^i and G^i) may be causing inconsistencies in the validation structure of R^i. These inconsistencies will be referred to as *applicability failures,* as these are the reasons why R^i cannot be directly applied to P^n. The purpose of the annotation verification procedure is to modify R^i such that the result, R^a, will be a partially reduced HTN with a consistent validation structure.

The annotation verification procedure achieves this goal by first localizing and characterizing the applicability failures caused by the differences in I^i and G^i, and then appropriately modifying the validation structure of R^i to repair those failures. It groups the applicability failures into one of several classes depending on the type of the inconsistencies and the type of the conditions involved in those inconsistencies. Based on this classification, it then suggests appropriate repairs. The repairs involve removal of unnecessary parts of the HTN and/or addition of non-primitive tasks (called "refit tasks") to establish missing and failing validations. In addition to repairing the inconsistencies in the plan validation structure, the annotation verification process also uses the notion of *p* -phantom-validations (see below) to exploit any serendipitous effects to shorten the plan. Figure 5 provides the top level control structure of the annotation verification process.

The individual repair actions taken to repair the different types of inconsistencies are described below; they make judicious use of the node annotations to modify R^i appropriately. The specifications of the exact procedures used by all these modification actions can be found in [13].

3.2.1. Unnecessary Validations-Pruning Unrequired Parts

If the supported condition of a validation is no longer required, then that validation can be removed from the plan along with all the parts of the plan whose sole purpose is supplying those validations. The removal can be accomplished in a clean fashion with the help of the

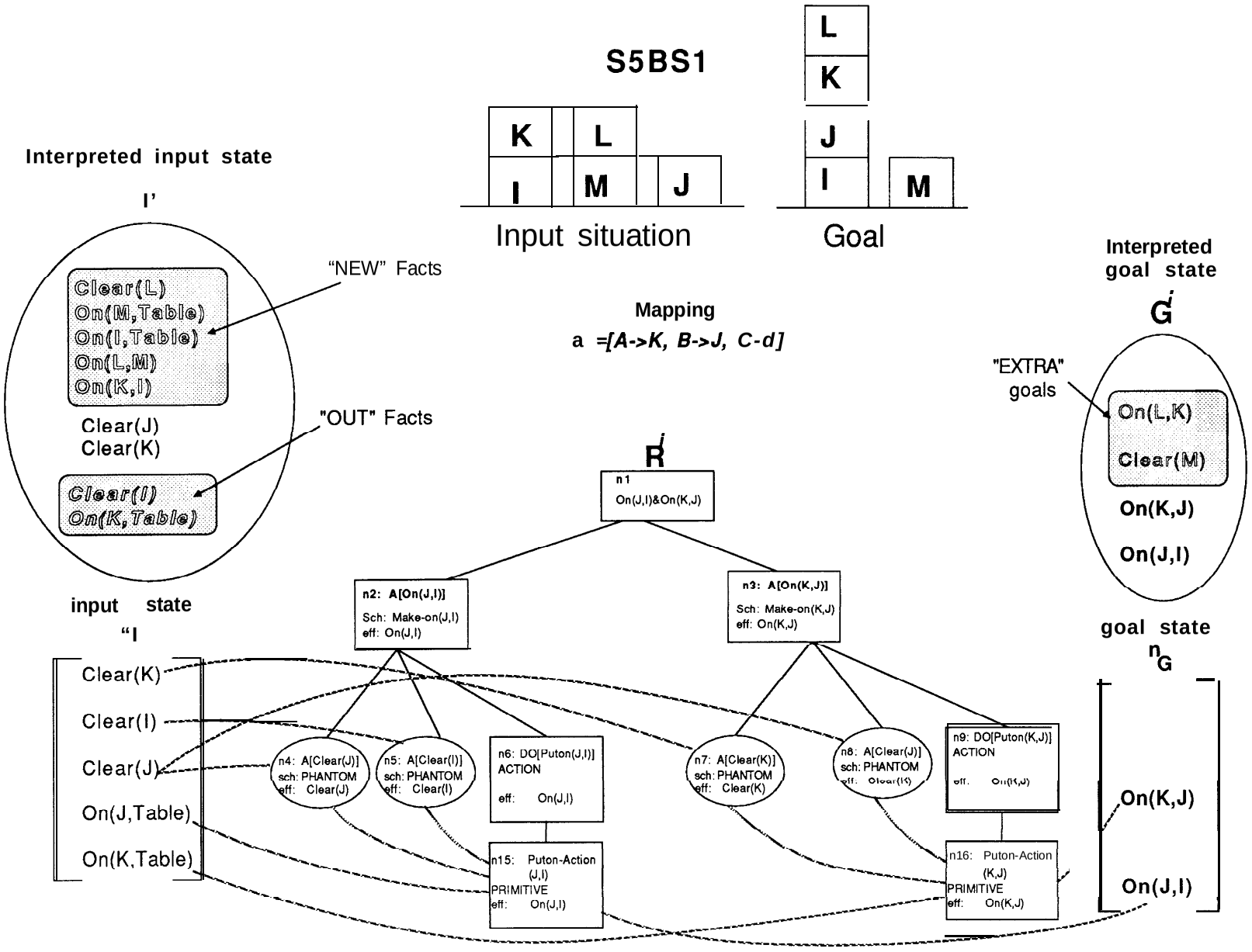

Figure 4. Interpreted Plan for 3BS->S5BS1

```
Procedure Annotation–Verif ication ( )
1     input: R^i : Interpreted plan, I': Interpreted input state, G^i : Interpreted goal state
2     begin
3       foreach g ∈ G^i s.t. g is marked as an unnecessary-goal
4         do find v:⟨E,n_s,C,n_G⟩ ∈ A^p(n_G) s.t. C=g
5                 Prune-Validation(v ) od
6       foreach g ∈ G^i s.t.g is marked as an extra-goal
7         do Repair-Missing-Validation(g:condition, n_G:node ) od
8       foreach f ∈ I' s.t. f is marked as an out-fact
9         do foreach v :⟨E,n_I,C,n_d⟩ ∈ A^s(n_I) s.t. E=f
10           do if E' ∈ I^i s.t. E' is marked new ∧ E' ⊢ C   /*Verification*/
11                then do Remove-Validation(v )
12                        Add-Validation(v':⟨E',n_I,C,n_d⟩) od
13                elseif type (C )=Precondition
14                      then Repair-Failing-Precondition-Validation(v )
15                elseif type (C)=Phantom /*n_d is a phantom node*/
16                      then Repair-Failing-Phantom-Validation(v )
17                elseif type (C )=Filter-Condition
18                      then Repair-Failing-Filter-Condition-Validation(v ) od od
19       foreach v :⟨E,n_s,C,n_d⟩ ∈ V s.t.
20           n_s≠n_I ∧ E ∈ I^i ∧ E is marked new in I'/*checking for serendipitous effects*/
21         do Exploit-P-Phantom-Validation(v ) od
22     end
```

Figure 5. Annotation-Verification Procedure

annotations on R^i: After removing an unnecessary validation from the HTN (which will also involve incrementally re-annotating the HTN, see section 2.3), the HTN is searched for any node n_v that has no e-conditions. If such a node is found, then its sub-reduction, ***R (n,),*** has no useful purpose, and thus can be removed from the HTN. This removal turns the e-preconditions of n_v into unnecessary validations, and they are handled in the same way recursively.

The procedure ***Prune-Validation*** in Figure 6 gives the details of this process. After removing the unnecessary validation v from the plan, it checks to see if there are any sub-reductions that have no useful effects (lines 3-5). (Because of the explicit representation of the validation structure as annotations on the plan, this check is straightforward.) If there are such sub-reductions, they have to be removed from the HTN (lines 6-l 6). This involves removing all the internal validations of that sub-reduction from the HTN (lines 7-8), and recursively pruning the validations corresponding to the external preconditions of that sub-reduction (lines 9-10). This latter action is to ensure that there won't be any parts of the **HTN** whose sole purpose is to supply validations to the parts that are being removed. The ***Remove-Validation*** procedure (line 8) not only removes the given validation, but also updates the validation structure **(V)** and the protection intervals (II) of the ***HTN*** consistently. Finally, the sub-reduction is unlinked from the **HTN** (lines 12-14), and the partial ordering on the HTN (0) is updated so

```
Procedure Prune-Validation (v:⟨E,n_s,C,n_d⟩,HTN:⟨P:⟨T,O,Π⟩,T*,D⟩)
    begin
2     Remove-Validation(v)
3     if' e-conditions(n_s)=∅
4        then do find n ∈ (n,) ∪ ancestors (n,) s. t.
5                 e -conditions(n) = 0 ∧ e -conditions(parent (n)) ≠ 0
6                    /*Remove the sub-reduction below n'*/
7           foreach n' ∈ R (n) s.t. children (n')=∅
8              do foreach v' ∈ e -conditions(n')
9                    do Remove-validation(v') od
10                foreach v'' ∈ e -preconditions(n)
11                   do Prune -Validation (v'') od
12                /* unlinking R (n) from HTN */
13             T* ← T*-R(n)
14             T ← T-R(n)
15             D ← D−{d | d ∈ D ∧ d ⊆ R(n)}
16             Update-Orderings(O) R (n)) od fi
17    end
```

Figure 6. Procedure for repairing unnecessary validations

that the ordering relations that were imposed because of the expansions involved in $R(n)$ are retracted. This backtracking is accomplished with the help of the *orderings* field of each node in R(n) *(see* section 2.3) which stores the ordering relations that were imposed because of the expansion below that node. The procedure involves: *(i)* Retracting from 0 all the ordering relations that are stored in *the orderings* field of the removed nodes ($R(n)$), and *(ii)* Appropriately redirecting[6] any remaining ordering relations of 0 involving the removed nodes (these correspond to the orderings that were inherited from the ancestors of n *; see* § 1.2).

The structure of the HTN at the end of this procedure depends to a large extent on the importance of the validation that is being removed (that is, how much of the HTN is directly or indirectly present solely for achieving this validation). *The Prune-Validation* procedure removes exactly those parts of the plan that become completely redundant because of the unnecessary validation. It will not remove any sub-reduction that has at least one e-condition (corresponding to some useful effect). Many previous plan modification strategies (such as [6, 9]) did not have this flexibility. Explicit representation of the validation structure makes this possible in PRIAR's framework. There is, however, a trade-off involved here: the strategy adopted by the *Prune-Validation* procedure is appropriate as long as the goal is to reduce the cost of planning (refitting). However it should be noted that if the cost of execution of the plan were paramount, then it would be necessary to see if the remaining useful effects of the sub-reduction could be achieved in an alternate way that would incur a lower cost of execution To take an extreme example, suppose the plan R^o achieves two of its goals, taking a flight and reading a paper, by buying a paper at the airport. If R^o is being reused in a situation where

[6] To a sibling of n in case of pruned reduction, and to n in the case of a replaced reduction (see 3.2.3).

the agent does not have to take a flight, it will be better to satisfy the goal of buying the paper in an alternate way, rather than by going to the airport. This type of analysis can be done with the help of the 'levels' of validations (see section 2.1): We might decide to remove a sub-reduction $R(n)$ and achieve its useful effects in an alternate way if the levels of e-conditions of n which are removed are 'significantly' higher than the levels of the remaining e-conditions of n. PRIAR currently does not do this type of analysis while pruning a validation.

3.2.2. Missing Validations—Adding Tasks for Achieving Extra Goals

If a condition G of a node n_d is not supported by any validation belonging to the set of validations of the plan, V, then there is a missing validation corresponding to that condition-node pair. Since, an extra goal is any goal of the new problem that is not a goal of the old plan, it is un-supported by any validation in R^i. The general procedure for repairing missing validations (including the extra goals, which are considered conditions of n_G) is to create a refit task of *the* form n_m :*Achieve* [G], and to add it to the HTN in such a way that $n_l < n_m < n_d$, and $parent(n_m) = parent(n_d)$. The new validation $v_m : \langle G, n_m, G, n_d \rangle$ will now support the condition G. Before establishing a new validation in this way PRIAR uses the planner's truth criterion (interaction detection mechanisms) to make sure whether that validation introduces any new failing validations into the plan (by causing harmful interactions with the already established protection intervals of the plan). The incremental annotation procedures are then used to add the new validation *to the* HTN. *Notice* that no *a priori* commitment is made regarding the order or the way in which the condition G would be achieved; such commitments are made by the planner itself during the refitting stage.

3.2.3. Failing Validations

The facts of I' which are marked *"out"* during the interpretation process, may be supplying validations to the applicability conditions or goals of the interpreted plan R^i. For each failing validation, the annotation verification procedure first attempts to see if that validation can be re-established locally by a new effect of the same node. If this is possible, the validation structure will be changed to reflect this. A simple example would be the following: Suppose there is a condition *Greater* $(B, 7)$ on some node, and the fact *Equal* $(B, 10)$ in the initial state was used to support that condition. Suppose further that in the new situation *Equal* $(B, 10)$ is marked *out* and *Equal* $(B, 8)$ is marked *new*. In such a case, it should be possible to establish the condition just by redirecting the validation to *Equal* $(B, 8)$.

When the validations cannot be established by such local adjustments, the structure of the HTN has to be changed to account for the failing validations. The treatment of such failing validations depends upon the types of the conditions that are being supported by the validation. We distinguish three types of validation failures—validations supporting preconditions, phan-

tom goals[7] and filter conditions respectively-and discuss each of them in turn below.

3.2.3.1. Failing Precondition Validations

If a validation v :$\langle E, n_s, C, n_d \rangle$ supporting a precondition of some node in the HTN is found to be failing, because its supporting effect *E* is marked *out,* it can simply be reachieved. The procedure involves creating a refit task, n_v :*Achieve [E*], to re-establish the validation v , and adding it to *the HTN* in such a way that $n_s < n_v < n_d$ and *parent (n,) = parent* (n_d). The validation structure of the plan is updated so that the failing validation v is removed and an equivalent validation v':$\langle E, n_v, C, n_d \rangle$ is added. (This addition does not introduce any further inconsistencies into the validation structure (see section 4.2.1).) Finally, the annotations on the other nodes of the HTN are adjusted incrementally to reflect this change.

3.2.3.2. Failing Phantom Validations

If a validation v_p :($E, n_s, C, n_p \rangle$ is found to be failing and n_p is a phantom goal, then v_p is considered a failing phantom validation. If the validation supporting a phantom goal node is failing, then the node cannot remain phantom. The repair involves undoing the phantomization, so that the planner would know that it has to re-achieve that goal. This step essentially involves backtracking over the phantomization decision and updating the HTN appropriately (similar to *the* process done in the *Prune-Validation* procedure (Figure 6, lines 12-16). Once this change is made, the failing validation v_p is no longer required by the node n_p, and so it is removed (updating **V** and II).

3.2.3.3. Failing Filter Condition Validations

In contrast to the validations supporting the preconditions and the phantom goals, the validations supporting failing filter conditions cannot be reachieved by the planner. Instead, the planning decisions which introduced those filter conditions into the plan have to be undone. That is, if a validation v_f :$\langle E, n_s, C_f, n_d \rangle$ supporting a filter condition C_f of a node n_d is failing, and *n'* is the ancestor of n_d whose reduction introduced C_f into the HTN originally, then the subreduction *R (n')* has to be replaced, and n' has to be re-reduced with the help of an alternate schema instance. So as to least affect the validation structure of the rest of the HTN, any new reduction of n' would be *expected* to supply (or consume) the validations previously supplied (or consumed) by the replaced reduction. Any validations not supplied by the new reduction would have to be re-established by alternate means, and the validations not consumed by the new reduction would have to be pruned. Since there is no way of knowing what the new

[7] The difference between a precondition validation and a phantom goal validation is largely a matter of how the corresponding conditions are specified in the task reduction schemas. In NONLIN terminology [29], the precondition validations support the ''unsupervised conditions'' of a schema, while the phantom goal validations support the ''supervised conditions'' of a schema.

reduction will be until the refitting time, this processing is deferred until then.[8]

The procedure shown in Figure 7, details the treatment of this type of validation failure during annotation verification. In line *3*, it finds the node n' that should be re-reduced by checking the filter conditions of the ancestors of n. Lines 5-l 8 detail changes to the validation structure of the HTN. Any e-conditions of the nodes belonging to $R(n')$ are redirected to n', if they support nodes outside $R(n')$ (lines 6-9). Otherwise, such e-conditions represent internal validations of $R(n')$, and are removed from the validation structure (line 10). At the end of this processing, all the useful external effects of $R(n')$ have n' as their source. Similar processing is done for the e -preconditions of the nodes of $R(n')$ (lines 12-18). Finally, all the descendants of n' are removed from the HTN (lines 20-22), and the partial orderings of HTN are updated to reflect this removal (line 23). Apart from removing the orderings imposed by the expansions of nodes in $descendents(n')$, this step also involves redirecting any ordering relations that were inherited from ancestors of n' back to n' (see the discussion in section 3.2.1).

Procedure **Repair-Failing-Filter-Condition-Validation** $(v_f:\langle E,n_s,C,n_d\rangle, \text{HTN}:\langle P:\langle T,0,\Pi\rangle,T^*,D\rangle)$

```
1  begin
2    Remove-Validation(v_f)
3    find n' ∈ Ancestors(n_d) ∪ {n_d} s.t. C ∈ filter-conditions(n')
4         /*replace reduction below n'*/
5    foreach n_c ∈ R(n') s.t. children(n_c) = ∅
6      do foreach v':⟨E',n'_s,C',n'_d⟩ ∈ e-conditions(n_c)
7           do if v' ∈ e-conditions(n')
8                then do Remove-Validation(v')
9                        Add-Validation(v'':⟨E',n'_s,C',n'_d⟩ od
10               else Remove-Validation(v')
11          fi od
12       foreach v':⟨E',n'_s,C',n'_d⟩ ∈ e-preconditions(n_c)
13         do
14           if v' ∈ e-preconditions(n')
15             then do Remove-Validation(v')
16                     Add-Validation(v'':⟨E',n'_s,C',n'⟩ od
17             else Remove-Validation(v') fi
18         od od
19        /* unlinking descendents(n') from HTN */
20   T* ← T* − descendents(n')
21   T ← T-descendents(n')
22   D ← D-{d | d ∈ D ∧ d ⊆ descendents(n')}
23   Update-Orderings(0, descendents(n)) od fi
24        /*Mark n' as a refit-task of type replace-reduction*/
25   T ← T ∪ {n'}
26   refit task-type(n') ← "replace-reduction"
27 end
```

Figure 7. Procedure for repairing failing filter condition validations

[8] This type of applicability failure is very serious as it may require replacement of potentially large parts of the plan being reused, there by increasing the cost of refitting. In [13, 17], we show that PRIAR's retrieval and mapping strategy tends to prefer reuse candidates that have fewer applicability failures of this type.

Finally, *n'* now constitutes an unreduced refit-task and so it is added to T (lines 25-26). (Notice that a difference between this and *the Prune-Validation* procedure is that in this case the e -preconditions of the replaced sub-reduction are redirected rather than pruned.)

3.2.4. P -Phantom-Validations-Exploiting Serendipitous Effects

When R^o is being reused in the new planning situation of P^n, it is possible that after the interpretation, some of the validations that R^i establishes via step addition can now be established directly from the new initial state. Such validations are referred to as p -phantom validations. More formally, a validation $v_p:\langle E, n_s, C, n_d\rangle$ is considered a p -phantom-validation of R^i if $n_s \neq n_I$ and $I^n \vdash E$. Exploiting such serendipitous effects and removing the parts of the plan rendered redundant by such effects can potentially reduce the length of the plan. Once the annotation verification procedure locates such validations, **PRIAR** checks to see if they can actually be established from the new initial state. This analysis involves reasoning over the partially ordered tasks of the **HTN** to see if through possible introduction of new ordering relations, an effect of n_I can be made to satisfy the applicability condition supported by this validation. The facilities of typical nonlinear planners can be used to carry out this check. When a p-phantom validation v_p is found to be establishable from n_I, the parts of the plan that are currently establishing this validation can be pruned. This is achieved by pruning v_p (see section 3.2.1). Currently, we do not allow PRIAR to add steps ***(cf.** white knights* [3]) or cause new interactions while establishing a p-phantom validation, and exploit the serendipitous effects only if doing so will not cause substantial revisions to the plan.

3.2.5. Example

Figure 8 shows R^a, the HTN produced by the annotation verification procedure for the 3BS→S5BS1 example. The input to the annotation verification procedure is the interpreted plan R^i discussed in section 3.1. In this example, R^i contains two missing validations corresponding to extra goals, a failing phantom validation and a failing filter condition validation. The fact *On (K ,Table)*, which is marked *out* in I', causes the validation $\{On(K,Table), n_I, On(K,Table), n_{16}\rangle$ in R^i to fail. Since this is a failing filter condition validation', the reduction that first introduced this condition into the **HTN** would have to be replaced. In this case, the condition *On (K ,Table)* came into the **HTN** during the reduction of node n_9:*Do [Puton(K ,J)*]. Thus, the annotation verification process removes $R(n_9)$ from the **HTN,** and adds a *replace reduction* refit task n_9:*Do* [*Puton (K* ,J)]. The e -preconditions of the replaced reduction, $\langle Clear(K), n_7, Clear(K), n_{16}\rangle$ and $\langle Clear(J), n_8, Clear(J), n_{16}\rangle$, are redirected such that the refit task n_9 becomes their destination. Similarly the e -condition of the replaced

[9] We follow the convention of [30] and classify On $(K,?x)$ as a filter condition rather than a precondition. Some effects of the plan depend on the binding of $?x$ and one way of correctly propagating the effects when the binding of $?x$ changes is to treat this reduction-time assumption as a filter condition.

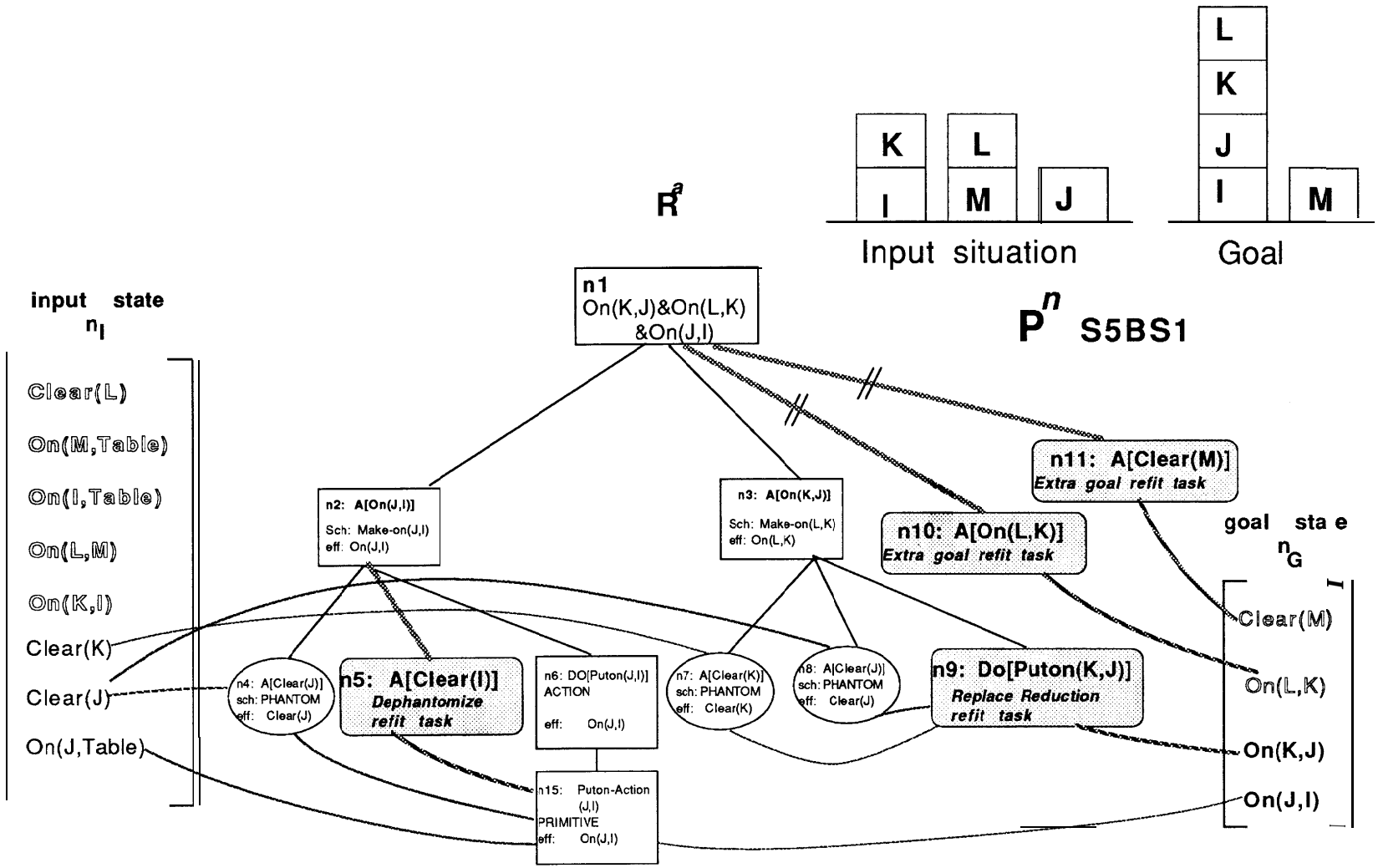

Figure 8. Annotation-verified Plan for 3BS->S5BS1

reduction, $\langle On(K,J), n_{16}, On(K,J), n_G \rangle$ is redirected such that n_9 becomes the source. These last two steps ensure that any possible reduction of n_9 will be aware of the fact that it is expected to supply the e -conditions and consume e -preconditions of the replaced reduction.

Next, the fact *Clear* (I), which is marked *out* in I^i causes the validation $\langle Clear(I), n_I, Clear(I), n_5 \rangle$ to fail. Since this validation supports the phantom goal node n_t, *the* annotation verification procedure undoes the phantomization and converts n_5 into a refit task n_5:*Achieve [Clear(I)]* to be reduced. Once this conversion is made, n_5 longer needs the failing validation from n_I, and it is removed.

Finally, the goals $Clear(M)$ and $On(L,K)$ of G^i are extra goals, and are not supported by any validation of the **HTN.** So, the refit tasks n_{10}:*Achieve [On (L ,K)]* and n_{11}:*Achieve [Clear(M)]* are created and added to the **HTN,** in parallel to the existing plan such that $n_I < n_{10} < n_G$ and $n_I < n_{11} < n_G$. The node n_{10} now supports the validation $\langle On(L,K), n_{10}, On(L,K), n_G \rangle$ and the node n_{11} supplies the validation $\langle Clear(M), n_{11}, Clear(M), n_G \rangle$.

Notice that the HTN shown in this figure corresponds to a partially reduced task network which consists of the applicable parts of the old plan and the four refit tasks suggested by the annotation verification procedure. It has a consistent validation structure, but it contains the unreduced refit tasks n_{10}, n_{11}, n_9 and n_5.

3.2.6. Complexity of Annotation Verification

In [13], we show that the repair actions involved in the annotation verification process can all be carried out in $\mathbf{0}(N_P^2)$, except for the steps involving interaction detection when new validations are introduced during the repair of missing validations and p -phantom validations. This latter step essentially involves checking for the truth of an assertion in a partially ordered plan. It is known that under the TWEAK representation (which does not allow conditional effects and state independent domain axioms), this step can be carried out in $\mathbf{0}(N_P^3)$ time [3]. Thus, the worst case complexity of the repair actions is $\mathbf{0}(N_P^3)$. Since there cannot be more than $|\mathbf{V}|$ failing validations in a plan, the complexity of the overall annotation verification process itself is $\mathbf{0}(\mathrm{IV}|N_P^3)$ (where $\mathbf{IV\ I} \leq \xi N_P$ as mentioned previously). Thus, the annotation verification process is of polynomial $(\mathbf{0}(N_P^4))$ complexity in the length of the plan.

3.3. Refitting

At the end of the annotation verification, R^a represents an incompletely reduced HTN with a consistent validation structure. To produce an executable plan for P^n, R^a has to be completely reduced. This process, called refitting, essentially involves reduction of the refit tasks that were introduced into R^a during the annotation verification process. The responsibility of reducing the refit tasks is delegated to the planner by sending R^a to the planner. An important difference between refitting and from-scratch (or generative) planning is that in refitting, the

planner starts with an already partially reduced HTN. For this reason, solving P^n by reducing R^a is less expensive on the average than solving P^n from scratch.

The procedure used for reducing refit tasks is fairly similar to the one the planner normally uses for reducing non-primitive tasks (see section 1.4), with one important extension. An important consideration in refitting is to minimize the disturbance to the applicable parts of R^a during the reduction of the refit tasks. Ideally, it should leave any already established protection intervals of HTN unaffected. To ensure this *conservatism* of refitting, the default schema selection procedure is modified in such a way that for each refit task, n_r, it selects a schema instance that is expected to give rise to the least amount of disturbance to the validation structure of R^a. The annotations on n_r guide this selection by estimating the effect of the reduction of n_r on the rest of the plan. (Section 3.4.1 contains a brief discussion of this heuristic control strategy.) Once the planner selects an appropriate schema instance by this strategy, it reduces the refit task by that schema instance in the normal way, detecting and resolving any interactions arising in the process.

A special consideration arises during the reduction of refit tasks of type *replace-reduction.* After selecting a schema instance to reduce such refit tasks, **PRIAR** might have to do some processing on the **HTN** before starting the task reduction. As we pointed out during the discussion of failing filter condition validations (section 3.2.3.3), when a node *n* is being re-reduced it is expected that the new reduction will supply all the e-conditions of n and will consume all the e-preconditions of n. If the chosen schema instance does not satisfy these expectations, then the validation structure of the plan has to be re-adjusted. **PRTAR** does this by comparing the chosen schema instance, S_i, and the e-conditions and e-preconditions of node *n* being reduced, to take care of any validations that S_i does not promise to preserve. It will *(i)* add refit tasks to take care of the e-conditions of n that are not guaranteed by S_i, and *(ii)* prune parts of the HTN whose sole purpose is to achieve e-preconditions of n that are not required by S_i.

An alternative way of treating the failing filter condition validations, which would obviate the need for this type of adjustment, would be to prune the e-preconditions of *n* at the time of annotation verification itself, and add separate refit tasks to achieve each of the e-conditions of *n* at that time. However, this can lead to wasted effort on two counts:

(1) Some of the e-preconditions of n might actually be required by any new reduction of *n* , and thus the planner might wind up reachieving them during refitting, after first pruning them all during annotation verification.

(2) Some of the e-conditions of n might be promised by any alternate reduction of *n* , and thus adding separate refit tasks to take care of them would add unnecessary overhead of reducing the extra refit tasks.

In contrast, the only possible wasted effort in the way **PRIAR** treats the failing filter condition validations is that the annotation verification procedure might be adding refit tasks to achieve

validations (say to support the conditions of the parts of the plan which provide e - preconditions to the replaced reduction) that might eventually be pruned away during this latter adjustment.

3.3.1. Example

Figure 9 shows the hierarchical task reduction structure of the plan for the S5BS1 problem that **PRIAR** produces by reducing the annotation-verified task network (shown in Figure 8). (The top down hierarchical reductions are shown in left to right fashion in the figure. The dashed arrows show the temporal precedence relations developed between the nodes of the HTN.) The shaded nodes in the figure correspond to the parts of the interpreted plan R^i that survive after the annotation verification and refitting process, while the white nodes represent the refit tasks added during the annotation verification process, and their subsequent reductions.

During refitting, the planner reduces the refit task *Achieve [Clear* (I)] by putting *K* on *Table,* realizing that even though putting *K* on *I* looks locally optimal, it causes more disturbance to the validation structure of R^a (see below). The extra goal refit task *Achieve [Clear(M)]* is reduced by putting *L* on *K;* and this decision leads to the achievement of the other extra goal refit task *Achieve [On (L ,K)]* as a side-effect. As *K* is on *Table* by this point, the nlanner finds that the replace reduction refit task *Do* [*Puton (K J)]* can after all be

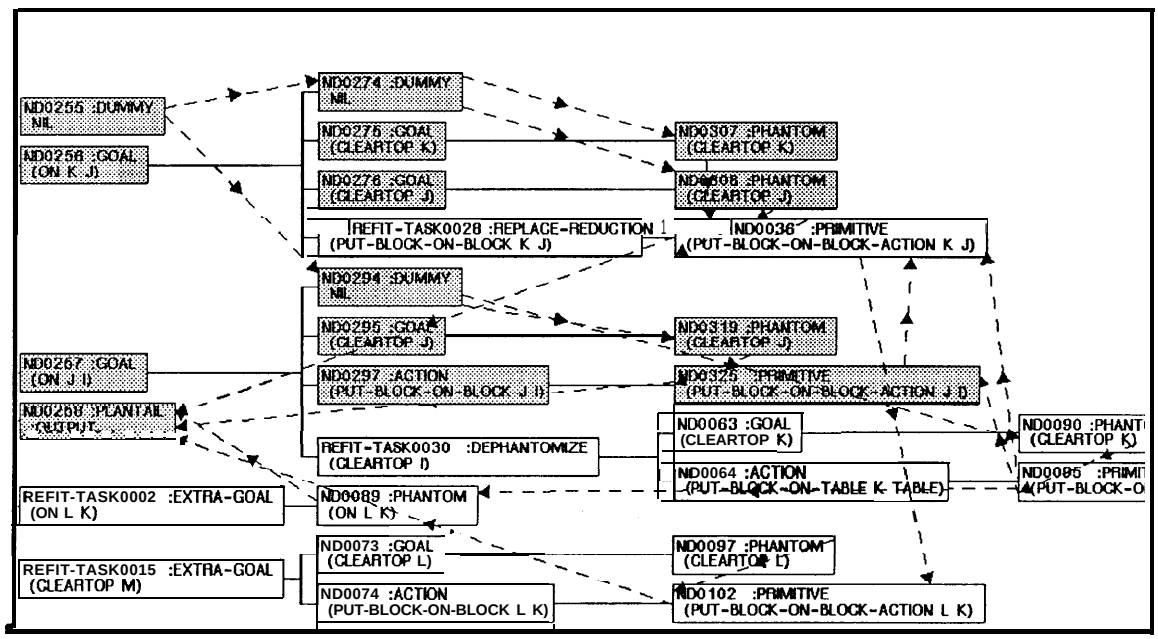

Figure 9. The plan produced by PRIAR for 3BS→S5BS1

reduced by another instantiation of the same schema that was used to reduce it previously[10].

3.4. Issues of Control

In this section we will briefly address the issues of control in **PRIAR'S** plan modification process. The purpose is to explain the role played by the plan validation structure in controlling refitting and retrieval. For the detailed development of these control strategies, the reader is referred to [15, 17, 13].

3.4.1. Conservative Control of Refitting

To derive maximum benefits from modification and reuse, and to prevent the possibility of the refitting process degenerating into from-scratch planning, care must be taken to ensure that the reduction of the refit tasks would cause minimum disturbance to the parts of the plan that are already applicable in the new situation. **PRIAR** exploits the annotated validation structure of the plan to estimate the disturbance caused by the reductions of refit-tasks to the rest of the plan, and uses this estimate to select among the schema instance choices for reducing the refit tasks.

To estimate the disturbance caused by individual task reduction choices, **PRIAR** develops the notion of the *task kernel* of a refit task. The task kernel encapsulates the set of validations that have to be preserved by any reduction of that node to leave the validation structure of R^a undisturbed; it is defined in terms of the node annotations. The reduction choices are ranked by the degree to which their applicability conditions and effects preserve the validations of the task kernel of the refit task. In the 3BS→S5BS1 example above, this control strategy recommends that the planner reduce the refit task *A [Clear(I)]* by putting K on *Table* rather than on L, M, or J (even though the last choice would appear locally optimal as it achieves the extra goal *On* (K, J)[11]), because this causes the least amount of disturbance to the validation structure of R^a. Similarly, for the refit task *Achieve [Clear* (M)], the control strategy recommends reduction by putting L on K rather than on *Table,* the other available choice. This allows it to achieve the second *extra goal* refit task as a side effect. A detailed description of this control strategy is beyond the scope of this paper, and can be found in [15, 13].

3.4.2. Controlling Mapping

While mapping is not a serious problem if the current plan itself is being modified due to some change in the specification, it becomes an important consideration in the case of modification during plan reuse. There are typically several semantically consistent mappings between objects of the two planning situations, P^o and P^n, and the selection of the right mapping could

[10] If the planner chooses to reduce this refit task in the beginning, then it would have bound the location of K is on I at that time. Then, since the location of K changes during the planning, the task would have to be re-reduced. Such a re-reduction should not be surprising as it is a natural consequence of hierarchical promiscuity allowed in most traditional hierarchical planners (see [36] for a discussion).

[11] This locally inoptimal choice is the characteristic of the sussman anomaly. Putting K on J at this juncture would lead to backtracking, as it affects the executability of the *Puton* (J, I) action.

considerably reduce the cost of modifying the chosen plan to conform to the constraints of the new problem. To do such selection, the matching metric should be able to estimate the expected cost of modifying R^o to solve P^n. **In** PRIAR modification framework, the cost of refitting R^o to P^n can be estimated by analyzing the degree of match between the validations of R^o and the specification of P^n, for various mappings $\{\alpha_i\}$. We have developed a heuristic ordering strategy which ranks the different mappings based on the number and the type of validations of the old plan that are dependent on the input state and goal state features of the old planning situation, which will be preserved in the new problem situation. The rationale behind this heuristic-that the cost of refitting depends both on the number and type of validations of the old plan that have to be re-established in the problem situation-should be intuitively obvious given our discussion of annotation verification. In our example, this strategy allows **PRIAR** to choose the mapping $[A \rightarrow L, B \rightarrow K, C \rightarrow J]$ over the mapping $[A \rightarrow K, B \rightarrow J, C \rightarrow I]$ while reusing the 3BS plan to solve the S5BS1 problem. In is instructive to note that while this strategy is used to choose between two reuse candidates corresponding to the same plan with different mappings in the current example, in general the strategy can also choose between reuse candidates using different plans. By basing retrieval on the appropriateness of using the old plan in the new problem situation, this strategy strikes a balance between purely syntactic feature-based retrieval methods, and methods which require a comparison of the solutions of the new and old problems to guide the retrieval (e.g. [2]). Further details of this retrieval and mapping strategy can be found in [17,131.

4. Analysis and Evaluation of PRIAR

4.1. Empirical Evaluation

The **PRIAR** modification framework described in this paper has been completely implemented in **COMMON LISP** and runs as compiled code on a Texas Instruments **EXPLORER-II** Lisp Machine. The hierarchical planner used in **PRIAR** is a reimplemented version of **NONLIN** [30, 7]. Performance evaluation experiments were conducted in an extended blocks world domain (see Appendix B for the domain specification) to quantify the savings in planning effort afforded by the modification framework. **(PRIAR** is also being adapted to provide an incremental planning capability for process planning in concurrent engineering environments. A prototype version is currently operational; see [21] for details.) The evaluation experiments consisted of solving several blocks world problems by reusing a range of similar to dissimilar stored plans. In each experiment, statistics were collected for solving the new problem from scratch and for solving it by modifying a given plan. A comprehensive listing of these statistics can be found in [13].

The cost of retrieval was factored out in all these experiments by providing PRIAR with a specific existing plan R^o to be reused while solving the new problem P^n. However, the appropriate mapping, a, between R^o and P^n is still chosen by the retrieval procedure. Such a

testing strategy is motivated by our desire to measure the flexibility of the modification framework by forcing **PRIAR** to solve P^n by reusing different R^o 's.

The problems used in these experiments are all from the blocks world. Problems 3BS, 4BS, 6BS, 8BS etc. are block stacking problems with three, four, six, eight, etc. blocks respectively on the table in the initial state, and stacked on top of each other in the final state. Problems 4BS1, 5BS1, 6BS1 etc. correspond to blocks world problems where all the blocks are in some arbitrary configuration in the initial state, and stacked in some order in the goal-state. In particular, S5BS1 corresponds to the example that we discussed in the previous sections. A complete listing of the test problem specifications can be found in [13].

Table 1 presents representative statistics from the experiments. It compares planning times (measured in *cpu seconds),* the number of task reductions, and the number of detected interactions, for from-scratch planning and for planning with reuse, in some representative experiments. The second entry in Table 1 corresponds to the 3BS→5SBS1 example discussed in the previous sections. The last column of the table presents the computational savings gained through reuse as compared to from-scratch planning (as a percentage of the from

$R^o \rightarrow P''$	P'' From Scratch	Reuse R^o	Savings (%)
3BS→4BS1	[**4.0***s*, 12*n*, 5*i*]	[**2.4***s*, 4*n*, 1*i*]	*39*
3BS→S5BS1	[**12.4***s*, 17*n*, *22i*]	[**5.2***s*, 8*n*, 12i]	58
5BS→7BS1	[**38.6***s*, 24*n*, 13i]	[**11.1***s*, 12*n*, 19i]	71
4BS1→8BS1	[**79.3***s*, 28*n*, 14i]	[**22.2***s*, 18*n*, 18i]	71
5BS→8BS1	[**79.3***s*, 28*n*, 14i]	[**10.1***s*, 14*n*, *7i*]	*87*
6BS→9BS1	[**184.6***s*, 32*n*, 17i]	[**18.1***s*, 17*n*, 17i]	**90**
10BS→9BS1	[**184.6***s*, 32*n*, 17i]	[**6.5***s*, 5*n*, *2i*]	*96*
4BS→10BS1	[**401.5***s*, 36*n*, 19i]	[**52.9***s*, 30*n*, *33i*]	*86*
8BS→10BS1	[**401.5***s*, 36*n*, 19i]	[**14.5***s*, 12*n*, *7i*]	*96*
3BS→12BS1	[**1758.6***s*, 44*n*, *2 3 i*]	[**77.1***s*, 40*n*, 38*i*]	*95*
5BS→12BS1	[**1758.6***s*, 44*n*, *2 3 i*]	[**51.8***s*, 32*n*, 26*i*]	*97*
10BS→12BS1	[**1758.6***s*, 44*n*, *2 3 i*]	[**21.2***s*, 13*n*, 7*i*]	*98*

Table 1. Sample statistics for PRIAR reuse

scratch planning time).

The entries in the table show that the overall planning times as well as the number of task reductions improve significantly with reuse. This confirms that reuse and modification in the PRIAR framework can lead to substantial savings over generative planning alone. The relative savings over the entire corpus of (approximately 70) experiments ranged from 30% to 98% (corresponding to speedup factors of 1.5 to 50), with the highest gains shown for the more difficult problems tested. The average relative savings over the entire corpus was 79%[12].

We also analyzed the variation in the savings accrued by reuse in terms of the similarity between the problems and the size of the constructed plans. Figure 10 shows the plot of this variation. It plot shows the computational savings achieved when different blocks world problems are solved by reusing a range of existing blocks world plans. For example, the curve marked 7BS1 shows the savings afforded by solving a particular seven-block problem by reusing several different blocks world plans (shown on the x-axis). Figure 11 summarizes all the individual variations by plotting (in logarithmic scale) the from-scratch planning time, and the best and worst case reuse planning times observed for the set of blocks world problems used in our experiments. It shows an observed speedup of one to two orders of magnitude.

Apart from the obvious improvement in reuse performance with respect to similarity between P^n and P^o, these plots bring out two other interesting characteristics of the PRIAR

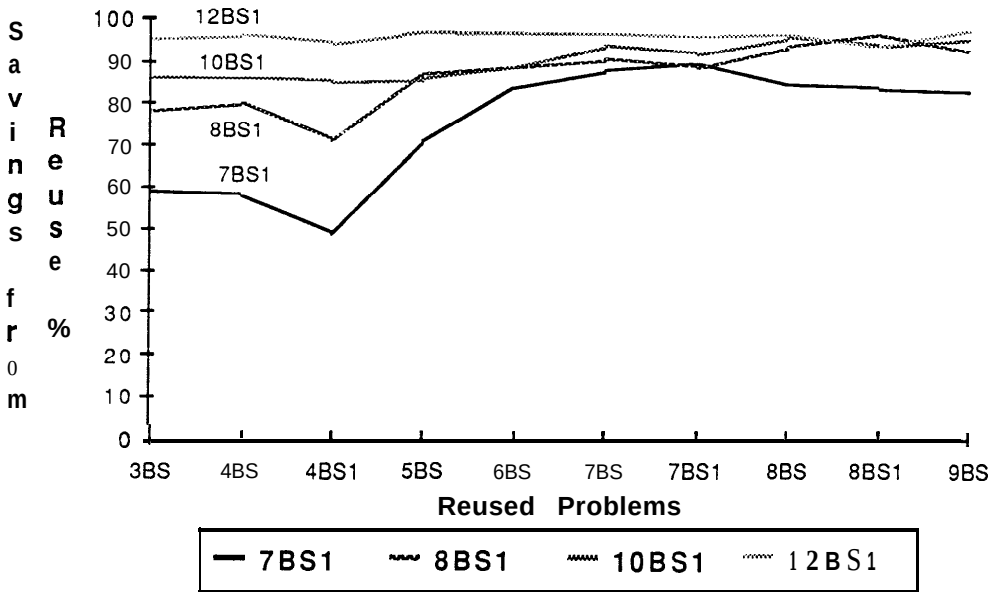

Figure 10. Variation of performance with problem size and similarity

[12] The cumulative savings were much higher, but they are biased by the higher gains of the more difficult problems.

reuse behavior:

1. Flexibility and Conservatism of Modification:

As we pointed out earlier, a flexible and conservative modification strategy provides the capability to effectively reuse any applicable parts of a partially relevant plan in solving a new planning problem. An important characteristic of such a modification strategy is that is that as the size of P^n increases, the computational savings afforded by PRIAR stay very high for a wide range of reused plans with varying similarity.- This behavior is brought out by the plots in Figures 10 and 11. Consider, for example, the plot for the 12BS 1 in Figure 10. As we go from a dissimilar plan R^o = 3BS to a very similar plan R^o = 9BS, the savings vary between 95% and 98% (corresponding to a variation in the speedup factor of 20 to 50). One of the important benefits of a flexible reuse framework is that the best match retrieval may not be critical for the utility of plan reuse. This may allow the use of simple and computationally efficient retrieval strategies [173.

2. Performance improvement with respect to the size of the planning problem:

An interesting pattern observed in PRIAR's performance is that when it modifies the same plan R^o to solve several different problems, the computational savings increase with the size of the problem being solved. Consider for example the cases of 3BS→7BS1 vs. 3BS→12BS1 in Figure 10. The improvement with size is further characterized by the statistics in Table 2, which lists the performance statistics when the 3BS plan is used to solve a set of increasingly complex blocks world problems. This can be explained in

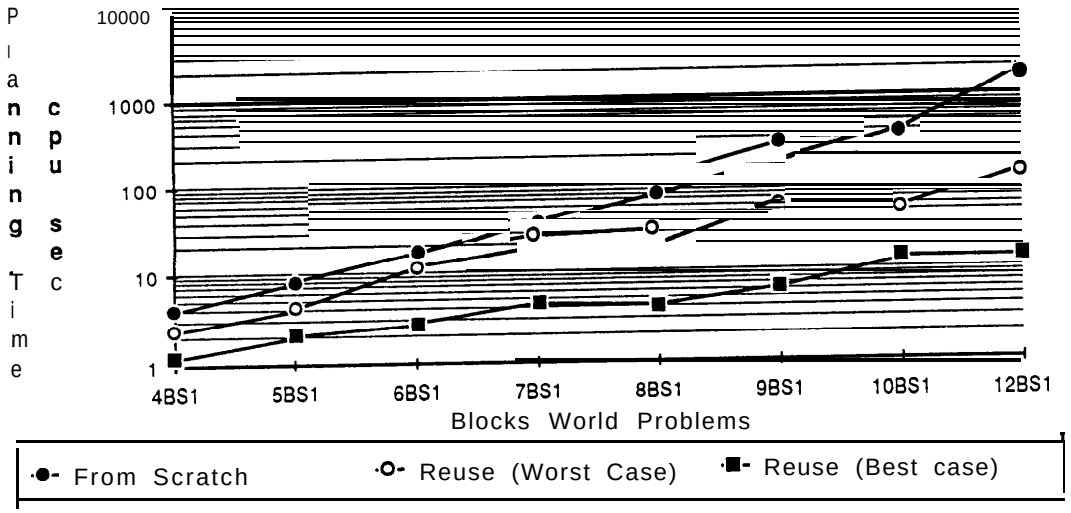

Figure 11. From-scratch vs. best and worst case reuse performance

$R^o \rightarrow P^n$	P^n From Scratch (*cpu sec.*)	Reuse R^o (*cpu sec.*)	Savings (%)	speedup
3BS→4BS1	**4.0**	**2.4**	**39**	**1.6**
3BS→5BS1	**8.4**	**4.3**	**49**	**1.9**
3BS→7BS1	**38.6**	**15.6**	**59**	**2.5**
3BS→8BS1	**79.3**	**17.4**	**78**	**4.6**
3BS→10BS1	**401.5**	71.4	**86**	**5.6**
3BS→12BS1	1758.6	**77.1**	**95**	**22.8**

Table 2. Variation of reuse performance with problem size

terms of the search process in the space of the plans. In hierarchical planning, as the size of a planning problem increases, the effective branching factor of the search space also increases. For example, for a g goal problem, where the average number of choices for reducing a goal in the domain is Ψ, the branching factor at the first level will be proportional to $g \times \Psi$; i.e., the branching factor increases with g [13]. If β is the branching factor of the search space, A is the operator distance between the problem specification P^n and the plan R^n, and A' is the operator distance between the R^a and R^n, then we can quantify the *relative reduction* in the explored search space during plan reuse as $\beta^{\Delta-\Delta'}$ [23, 13]. Thus, as β increases, so will the relative reduction in the search space. Thus, as problem size increases, the savings afforded by reuse tend to become more significant.

4.2. Analysis

In this section we shall analyze the completeness, coverage, flexibility and efficiency of the PRIAR framework.

4.2.1. Completeness

To demonstrate completeness, we must show that PRIAR can solve any new planning problem by correctly modifying any plan, whose validation structure is describable within its representation language. If we assume that the underlying planning strategy is complete, the completeness of PRIAR can be established by demonstrating that for any given plan R^o and a new

[13] Another way of understanding this is that as the size of the the planning problem increases, the number of ways of interpreting the modal truth criterion to achieve a goal (in Chapman's model of nonlinear planning [3]) also increases.

problem P^n, PRIAR provides an HTN with a consistent validation structure to the planner. The validation structure based modification is complete, in that it will correctly handle all types of applicability failures that may arise during plan modification, and provide the planner with a partially reduced HTN with a consistent validation structure. In particular, our definition of inconsistencies (see §2.2) captures all types of applicability failures that can arise due to a change in the specification of the problem; and our annotation verification procedure provides methods to correctly modify the plan validation structure to handle each type of inconsistency (see section *3.2).*

Proposition: ***The*** HTN ***at the end of the Annotation-verification procedure is a partially reduced plan with a consistent validation structure.***

When a plan is being reused in a new problem situation, the inconsistencies in the validation structure originate from the differences in the initial and final state specification; the interpretation procedure marks these differences. The overall plan can be seen as a black-box, which consumes the validations in the initial validation state $A^s(n_I)$ and supplies the validations in the final validation state $A^p(n_I)$. Thus the only way the differences in the problem specifications can cause inconsistencies in the validation structure of the plan is by affecting the validations in $A^s(n_I)$ and $A^p(n_G)$[14]. Thus, the annotation-verification procedure would only have to check these validations.

The only ways in which the validations of $A^s(n_I)$ and $A^p(n_G)$ can be affected by the changes in the problem specifications are: (i) some validations of $A^s(n_I)$ fail because of the disappearance of their supporting effects, *(ii)* some validations of $A^p(n_G)$ are not required because they are supporting unnecessary goals and finally *(iii)* some goals of the new problem are not supported by any validations of $A^p(n_G)$. These are precisely the cases that are defined as the inconsistencies in the validation structure of a plan (in section 2.1). We have seen that the annotation-verification process modifies the plan validation structure to take care of each of these three possibilities, and also to exploit any serendipitous effects. The repair actions involve either removing some parts of the plan, or adding high level non-primitive tasks to the plan to re-establish missing or failing validations. To prove that the resulting partially reduced HTN has a consistent validation structure, we need only show that the repair actions themselves do not introduce any inconsistencies.

There are three kinds of changes made to the validation structure of R^i during these repair tasks: *(i)* some existing validations are removed, *(ii)* some existing validations are re-directed, or ***(iii)*** some new validations are added. We can easily show that PRIAR's methods for removal of unnecessary validations, and redirecting validations (to the ancestors of the source

[14] Of course, while taking care of some of the affected validations, the annotation verification procedure might prune or redirect some internal validations of the plan (see the procedures for pruning validations and repairing failing filter condition validations).

or destination nodes) do not introduce any new inconsistencies. Thus the only remaining case is the addition of a new validation. Here too, there are two possibilities:

(1) When a failing precondition validation v :$\langle E, n_I, C, n_d \rangle$ is repaired by adding a new validation, v_r:$\langle E, n_r, C, n_d \rangle$, such that $n_I < n_r < n_d$. In this case, the only possible inconsistency could be failure of v_r. For v_r to fail, there should exist a node n such that $\Diamond (n, < n < n_d)$ and effects $(n) \vdash \neg E$. Since, $n_I < n_r < n_d$ (see section 3.2.3. 1), this will also imply that $\Diamond (n_I < n < n_r)$. That is, v itself could not have been established. Since v was established previously, by refutation we know that v_r cannot be failing.

(2) When completely new validations are introduced into HTN to take care of missing validations or *p* -phantom-validations. In these two cases, we have seen that the repair actions invoke the planner's truth criterion to make sure that the new validation does not lead to the failure of any existing validations.

Thus, all the repair actions remove the inconsistencies in R^i, without adding any new inconsistencies. Consequently, the HTN after annotation verification, R^a, has a consistent validation structure. Cl

To summarize, the annotation-verification based reuse framework presented here is com*plete* in the sense that if P^n is a problem that PRIAR's planner can solve from scratch, then **PRIAR** can take any arbitrary previously developed plan, R^o, a new problem P^n and provide R^a which can then be reduced by the planner to give a plan for P^n. This is because we are able to list with certainty all the possible inconsistencies that can arise in the validation structure of a plan during reuse and provides methods to remove the inconsistencies without introducing any new inconsistencies.

Notice, however, that while the consistency of annotation-verified plan R^a allows the planner to try to solve for P^n by reducing R^a rather than starting from scratch, it cannot by itself ensure that a plan for P^n can be found without backtracking over R^a. For this latter property to hold, the abstraction used in the task reduction schemas representing the domain should have the *"downward solution"* property [31] where the existence of an abstract plan implies the existence of specializations of this solutions at each lower level (see below).

4.2.2. Coverage

Here we discuss how well the modification capability provided by our theory covers the range of possible plan modification tasks. The validation structure developed here covers the internal dependencies of the plans produced by most traditional hierarchical planners. The captured dependencies can be seen as a form of explanation of correctness of the plan with respect to the planner's own domain model. By ensuring the consistency of the validation structure of the modified plan, **PRIAR** guarantees correctness of the modified plan with respect to the planner. However, it should be noted that as the dependencies captured by the validation structure do not represent any optimality considerations underlying the plan, the optimality of modification

is not guaranteed. Further, since the modification is integrated with the planner, failures arising from the incorrectness or incompleteness of the planner's own domain model will not be detected or handled by the modification theory" [18]. Of course, these should not be construed as limitations of the theory, as the goal of the theory is to improve the average case efficiency of the planner.

4.2.3. Flexibility and Efficiency

Computational savings in modifying plans in the **PRIAR** framework stem from the fact that the annotation verification process expends a polynomial amount of processing on R^t to produce a partially reduced HTN, R^a, which can, on the average, be reduced with exponentially less effort compared to planning for P^n from scratch. While we cannot expect a reduction in the theoretical complexity of planning unless the domain schemas have the "downward solution property" (see above), typically there is a strong performance improvement by starting the planner off with R^a. The empirical results discussed in section 4.2.1. provide support to this.

PRIAR reuse strategy is flexible in that it can effectively modify any existing plan to solve any new problem. Flexibility, however, is a double-edged sword-while it improves the coverage of the modification strategy by allowing a plan to be reused in a wide variety of new situations, it also leads to situations where the plan is reused in a totally inapplicable situation. In **PRIAR,** however, this does not pose a serious problem because the annotation verification procedure is of polynomial complexity. In the worst case, when none of the steps of R^o are applicable in the new situation, annotation verification will return a degenerate **HTN** containing refit tasks for all the goals of P^n. In such extreme cases PRIAR may wind up doing a polynomial amount of extra work compared to a pure generative planner? In other words, the worst case complexity of plan modification remains the same as the worst case complexity of generative planning. However, on the average, **PRIAR** will be able to minimize the repetition of planning effort (thereby accruing possibly exponential savings in planning time) by providing the planner with a partially reduced HTN that contains all the applicable parts of the plan being modified, and conservatively controlling refitting such that the already reduced (applicable) parts of R^a are left undisturbed. The claims of flexibility and average case efficiency are also supported by the empirical evaluation experiments that were conducted on **PRIAR,** as discussed in section 4.1.

---

[15] In [19] we discuss some preliminary ideas about dealing with failure of validations established by modules external to the planner.

[16] It should also be noted that the mapping and retrieval strategy developed in [17, 13] helps in ruling out such degenerate cases to a large extent.

5. Comparison to Previous Work

Early research in plan reuse and replanning was done in conjunction with the work on **STRIPS** planner [6]. The **STRIPS'** triangle-table based approach to replanning suffered from many limitations. As we pointed out in section 1, **STRIPS** was unable to modify the internal structure of its remembered macro-operators to suit new problem situations, and consequently could reuse them only when either the entire macrop or one of its subsequences was applicable in the current situation. Its only response to execution time failures was restarting the plan from an appropriate previously executed step. Such a capability is in general not sufficient to provide a robust replanning capability, as it is very rare that the execution time failures are so benign as to be repaired by restarting the plan from an earlier point. A recent hierarchical linear problem solver called **ARGO** [11] tries to partially overcome the inflexibility of the macro-operator based reuse by remembering macro-operators for each level of its hierarchical plan. However, it too lacks the capability to modify the intermediate steps of a chosen macro-operator, and is consequently unable to reuse all the applicable portions of a plan.

Hayes [9] was the first to suggest the idea of explicitly represented internal dependencies for guiding replanning. However, his framework was very domain-specific and the only replanning action allowed in it was to delete a part of a plan, thereby permitting the planner to reachieve some higher level goals in the hierarchical development of the plan. **NONLIN** [30, 29] was the first hierarchical planner to advocate explicit representation of goal dependencies to guide planning. Its **GOST** data structure is essentially a list of protection intervals associated with the plan, and is used during the planning to guide the interaction detection and resolution. Daniel [5] exploited NONLIN's plan structure to develop a framework for representing decision dependencies to aid in backtracking during planning. The intent was to enable **NONLIN** to do dependency directed backtracking during plan generation. While Daniel's research did not explicitly consider replanning or reuse problems, it generalized Hayes' notion of decision graphs significantly to capture inter-decision dependencies induced by **NONLIN.** However, here again, the development was very planner specific. There was neither a formal characterization of the remembered dependencies, nor a systematic exploration of their utility in plan modification. Recently, Morris et al [24] started exploring the utility of TMS-based data dependency methods for representing these decision-graph structures to provide a dependency-directed backtracking capability during planning. In the following we discuss the relation between **PRIAR** modification framework and these data dependency methods:

Any dependency directed plan transformation scheme must be able to handle the following three distinct issues: *(i*) What choice points would have to be revoked to handle the change in the specification or the environment, *(ii)* How to effectively retract the decisions that were made in the context of those choice points, and *(iii)* How best to guide the planning after the retraction, to satisfy the overall goals. While decision graphs, context layered world-models [34] and TMS based data dependency frameworks provide strategies for handling *ii, they* do not provide guidance on *i* and *iii.* In contrast, we have shown that the explicit planner-

independent representation of the causal dependencies of a plan (as its validtaion structure) provides a powerful medium for deliberating on what types of modifications are required and how to guide the planner in carrying out those modifications.

Wilkins' framework for guiding replanning and execution monitoring in SIPE [35] comes closest to PRIAR's plan reuse and modification framework in its treatment of applicability failures. (For a detailed discussion of how PRIAR's modification framework is used to guide and control execution monitoring and replanning, see [20].) SIPE's domain-independent replanning actions are similar to the repairs to the plan validation structure that are suggested by PRIAR'sannotation-verification process. However, SIPE does not attempt to explicitly characterize the role played by the individual tasks of the HTN in the validation of the rest of the plan. Consequently, some of its replanning actions are planner dependent, and are not stated formally. In contrast, PRIAR'S annotated validation structure gives a clean framework to state the replanning actions precisely and explicitly. Another important difference between the modification strategies of PRIAR and SIPE is that the latter does not attempt to control the replanning once the appropriate replanning actions were suggested to SIPE. As we discussed briefly in section 3.4.1 PRIAR employs a heuristic control strategy grounded in the plan validation structure for this purpose.

In contrast to the dependency directed debugging strategies such as [27, 8, 28] which aim to compensate for the inadequacies of the generative planner by debugging the generated plans, PRIAR aims to improve the efficiency of planning by ensuring the correctness of modification with respect to the planner. The plan debugging strategies proposed in GORDIUS and CHEF use an explanation of the correctness of the plan with respect to an external (deeper) domain model-generated through a causal simulation of the plan to guide the debugging of the plan-to compensate for the inadequacies of the planner's own domain model. In contrast, the plan modification strategy proposed in PRIAR utilizes the plan validation structure, an automatically generated explanation of correctness of the plan with respect to the planner's own domain model, to integrate planning and plan modification and to ensure correctness of plan with respect to the planner. Since the cost of debugging tends to be very high[17], a fruitful avenue of research might be to combine these strategies such that PRIAR's strategies are used to efficiently generate plans that are correct with respect to the planner, and the debugging strategies are used to test and debug these plans with respect to external domain models. In this sense, PRIAR's strategies are complementary to these debugging strategies.

A significant amount of research in case-based reasoning addressed the issues involved in the adaptation of stored plans to new situations (e.g., [1, 8, 32]). In contrast to PRIAR, typically these modification strategies are not integrated with a generative planner, are not concerned with correctness and conservatism of modification, and are typically heuristic in nature. This is

[17] In [26], Simmons notes that the success of GORDIUS's Generate-Test-Debug paradigm rests on the presence of a robust generator since debugging is very costly.

to a large extent a reflection of the characteristics of the domains in which these systems were developed, where the need to avoid execution time failures is not as critical as the need to control access to planning knowledge. For example, PLEXUS [1], an adaptive planner, starts with a highly structured plan library, and relies on the place of a plan in the background of other plans in the library to guide adaptation. **PLEXUS** works as an interpretive planner, and its primary mode of detecting applicability failures is through execution time failures. When a failure is detected, **PLEXUS** attempts to exploit the helpful cues from the new problem situation to trigger appropriate refitting choices to repair those applicability failures, and execute the result in turn. Similarly, **CHEF'S** [8] stored plans do not have explicitly represented dependency structure, and they are modified by domain dependent modification rules to make the old plan satisfy all the goals of a new problem. These modification strategies do not consider the internal causal dependency structure of the plan, and thus may lead to incorrect plans even relative to the domain knowledge contained in the case-base and the modifier. **CHEF** presumes that its retrieval strategy and modification rules are robust enough to prevent frequent occurrence of such incorrect plans (as we discussed above, **CHEF** does test the correctness of its modification through a simulation with respect to an external domain model). In contrast to **PLEXUS** and **CHEF, PRIAR** is concerned with the correctness of the modified plan relative to the planner's own domain knowledge, and uses the plan validation structure to ensure this. This capability is important both because debugging itself is a very costly operation (see [26, 27]) and because domain characteristics may put a very high premium on postponing all debugging to the execution time.

Finally, **PRIAR'S** approach to plan reuse is in the spirit of Carbonell's [2] proposed methodology for "problem solving by derivational analogy" which recommends remembering a full derivational history along with every problem solution, and using it to guide its analogical transformation later. **PRIAR** can be seen as a step towards the systematic exploration of the utility of including one class of information-the plan validation structure-in the stored derivational trace.

6. Conclusion

We presented a theory of plan modification that utilizes the validation structure of the stored plans to yield a flexible and conservative modification framework. The validation structure, which constitutes a hierarchical explanation of correctness of the plan with respect to the planner's own knowledge of the domain, is annotated on the plan as a by-product of the initial planning. Plan modification is characterized as a process of removing inconsistencies in the validation structure of a plan, when it is being reused in a new (changed) planning situation. Annotation verification, a polynomial time process, carries out the repair of these inconsistencies. The repairs involve removing unnecessary parts of the **HTN,** adding new high-level tasks to it to re-establish failing validations, and exploiting any serendipitous effects to shorten the plan. The resultant partially reduced HTN with a consistent validation structure is given to the

planner for complete reduction. As the planner starts with a partially reduced HTN, it takes significantly less time on the average to produce a complete plan. This is supported by the results of the empirical studies in blocks world, which demonstrated 20-98 % savings (corresponding to speedup factors of 1.5 to 50) over pure generative planning, with the highest gains shown for the most complex problems tested.

We discussed the development of this theory in **PRIAR,** and characterized its completeness, coverage, efficiency and limitations. PRIAR's modification theory enables a planner to conservatively modify its plan in response to incremental changes in the specification, to reuse its existing plans in new problem situations, and to efficiently replan in response to execution time failures. While the plans made by **PRIAR are at the same level** of correctness as the ones that are made by the planner from scratch, in practical terms, **PRIAR** allows the planner to solve more problems in a "reasonable amount" of time and computational resources. This is very significant, since it enlarges the set of problems that are practically solvable by the planner. Currently, we are exploring the application of PRIAR modification strategy to more realistic domains [21], and investigating the methodology of plan modification in complex domains where the planner does not have access to all the domain knowledge and has to interact with other specialized domain modules [19].

Acknowledgements

Lindley Dar-den and Larry Davis have significantly influenced the development of this work. Jack Mostow and Austin Tate provided useful comments on previous drafts. Mark Drummond, David Wilkins, Nils Nilsson, Marty Tenenbaum, Quiang Yang and reviewers of AAAI-90 and IJCAI-89 provided several useful and pointers. To all, **our thanks.**

Appendix A. Trace output by PRIAR

This appendix contains an annotated trace of the **PRIAR** program as it plans for a blocks world problem by reusing an existing plan. Specifically, it follows PRIAR in solving the 5BP problem shown on the right in Figure A.1 by reusing an existing plan for solving the 6BS problem shown on the left. This example is specifically designed to show how **PRIAR** handles the failing filter condition validations, unnecessary validations and *p* -phantom validations (the capabilities that were not brought out in the example that was discussed in the paper).

In this example, PRIAR's partial unification procedure generates two plausible reuse candidates for solving the 5BP problem from the 6BS plan (lines l-l 1). The plan kernel based ordering then prefers one of those candidates $\langle 6BS, \alpha=[A \rightarrow L, C \rightarrow O, B \rightarrow P, D \rightarrow M, E \rightarrow N]\rangle$ as better suited for solving the 5BP problem (lines 13-19).

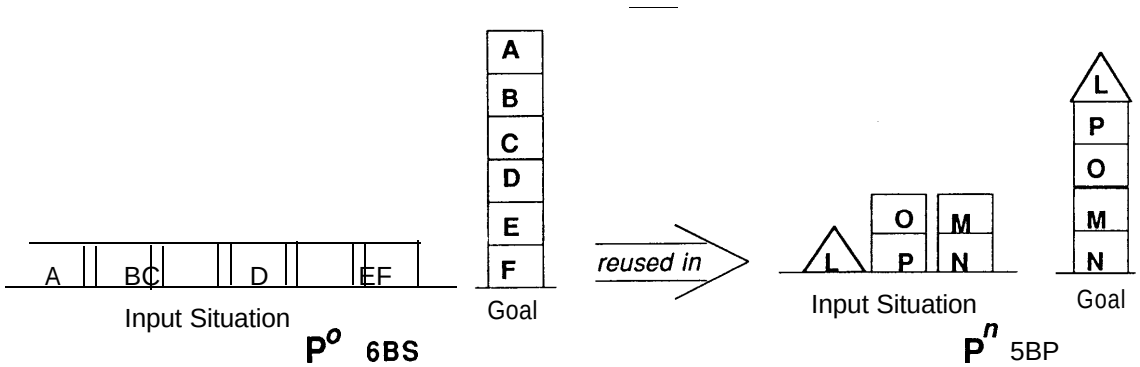

Figure A.l. 6BS→5BP Modification problem

```
1 PRIAR> (plan-for :problem '5bs-phantom-pyramid :reuse t)
2 Trying to solve the problem by reusing old plans

3  Calling...
4 (REUSE-PLAN GOALS ((ON P 0) (ON M N) (ON L P) (ON 0 M))
5             :INPUT ((BLOCK P) (CLEARTOP 0) (ON 0 P) (ON L TABLE)
6                     (ON P TABLE) (BLOCK 0) (BLOCK N) (BLOCK M)
7                     (PYRAMID L) (CLEARTOP M) (ON M N)) )
8 **********************Retrieving similar old plan*****************************

9 RETRIEVE: There are 2 possible Complete Matches. They are...
10 (({<Plan::6BS>} ((LA) (N E) (MD) (0 C) (P B)))
11  ({<Plan::6BS>} ((L B) (N F) (M E) (0 D) (P C))))
12                                                              . . .
13  *******PLAN-KERNEL-BASED-ORDERING
14 The Plan Choice-s ranked best by the Plan-kernel based retrieval Process are
15          ([({<Plan::6BS>} ((L A) (N E) (M D) (O C) (P B)))]{18})
16  Choosing
17 [({<Plan::6BS>} ((LA) (N E) (M D) (0 C) (P B)))]{ 18)
18  to be reused to solve the current problem
19 Copying and Loading plan into memory

20 using the following plan
21 Plan Name: 6BS
```

```
22      Goals:          ((ON B C) (ON C D) (ON D E) (ON E F) (ON A B))
23      Initial State: ((BLOCK D) (BLOCK B) (BLOCK A) (CLEARTOP A)
24                              (BLOCK C) (CLEARTOP D) (CLEARTOP C) (CLEARTOP B)
25                              (ON D TABLE) (ON C TABLE) (ON B TABLE) (ON A TABLE)
26                              (BLOCK F) (BLOCK E) (CLEARTOP F) (CLEARTOP E)
27                              (ON E TABLE))
28      Plan Kernel: #<PLANKERNEL 10733054>
29 The plan is...
30 ********************************
31 7: :PRIMITIVE (PUT-BLOCK-ON-BLOCK-ACTION E F) [Prenodes:(21 22)] [Succnodes: (6 1)]
32
33 6: :PRIMITIVE (PDT-BLOCK-ON-BLOCK-ACTION D E) [Prenodes:(18 19 7)] [Succnodes: (5 1)]
34
35 5: :PRIMITIVE (PDT-BLOCK-ON-BLOCK-ACTION C D) [Prenodes:(15 16 6)] [Succnodes: (4 1)]
36
37 4: :PRIMITIVE (PUT-BLOCK-ON-BLOCK-ACTION B C) [Prenodes:(12 13 5)] [Succnodes: (3 1)]
38
39 3: :PRIMITIVE (PDT-BLOCK-ON-BLOCK-ACTION A B) [Prenodes:(9 10 4)] [Succnodes: (1)]
40 ********************************

41          The mapping is [A→L E→N D→M C→O B→P ]
```

Next, the 6BS plan is interpreted in the 5BP problem situation with the chosen mapping. The interpretation process, apart from marking various facts as *in* and *out,* finds that one of the goals of the 6BS problem, *On (N ,F)*, is *unnecessary* for solving 5BP problem (line 57). Figure A.2 shows the **HTN** of the 6BS plan after the interpretation process.

```
42 **********Interpretation**********
43 Mapping the retrieved plan into the current problem
44 The mapping used is: [A→L E→N D→M C→O B→P ]
45 INTERPRET: adding fact (ON 0 P) to the initial state
46 INTERPRET: adding fact (PYRAMID L) to the initial state
47 INTERPRET: adding fact (ON M N) to the initial state
48 INTERPRET: Marking the fact (BLOCK L) in init-state :out
49 INTERPRET: Marking the fact (CLEARTOP L) in init-state :out
50 INTERPRET: Marking the fact (CLEARTOP P) in init-state :out
51 INTERPRET: Marking the fact (ON M TABLE) in init-state :out
52 INTERPRET: Marking the fact (ON 0 TABLE) in init-state :out
53 INTERPRET: Marking the fact (BLOCK F) in init-state :out
54 INTERPRET: Marking the fact (CLEARTOP F) in init-state :out
55 INTERPRET: Marking the fact (CLEARTOP N) in init-state :out
56 INTERPRET: Marking the fact (ON N TABLE) in init-state :out
57 INTERPRET: Marking the goal (ON N F) in goal-state :unnecessary
58 INTERPRETation is over
```

Next, **PRIAR** starts the annotation verification process; figure A.3 shows the **HTN** after this process. During the annotation verification process, PRIAR first considers the unnecessary validation supporting *the unnecessary* goal *On* (N ,*F*) (lines 59-65). The appropriate repair action is to recursively remove the parts of the plan whose sole purpose is to achieve this validation. In this case, PRIAR finds that the sub-reduction below the intermediate level node ND01 10: *On* (N ,*F*) (the node with a single asterisk in Figure A.2) will have to be removed from the plan to take care of this unnecessary validation. Consequently, the annotation verified plan, shown in Figure A.3, does not contain any nodes of this sub-reduction.

```
59 **********Annotation Verification**********
60 ANNOT-VERIFY: start

61 ANNOT-VERIFY: Processing unnecessary goals (if any)
62              The goal (ON N F) is UNNECESSARY
63    Remove Unnecessary Goal: Pruning the reduction below the node
64                             {<7::ND0110>[:GOAL(ON N F)] . ..)
65                                  To take care of this unnecessary goal.
```

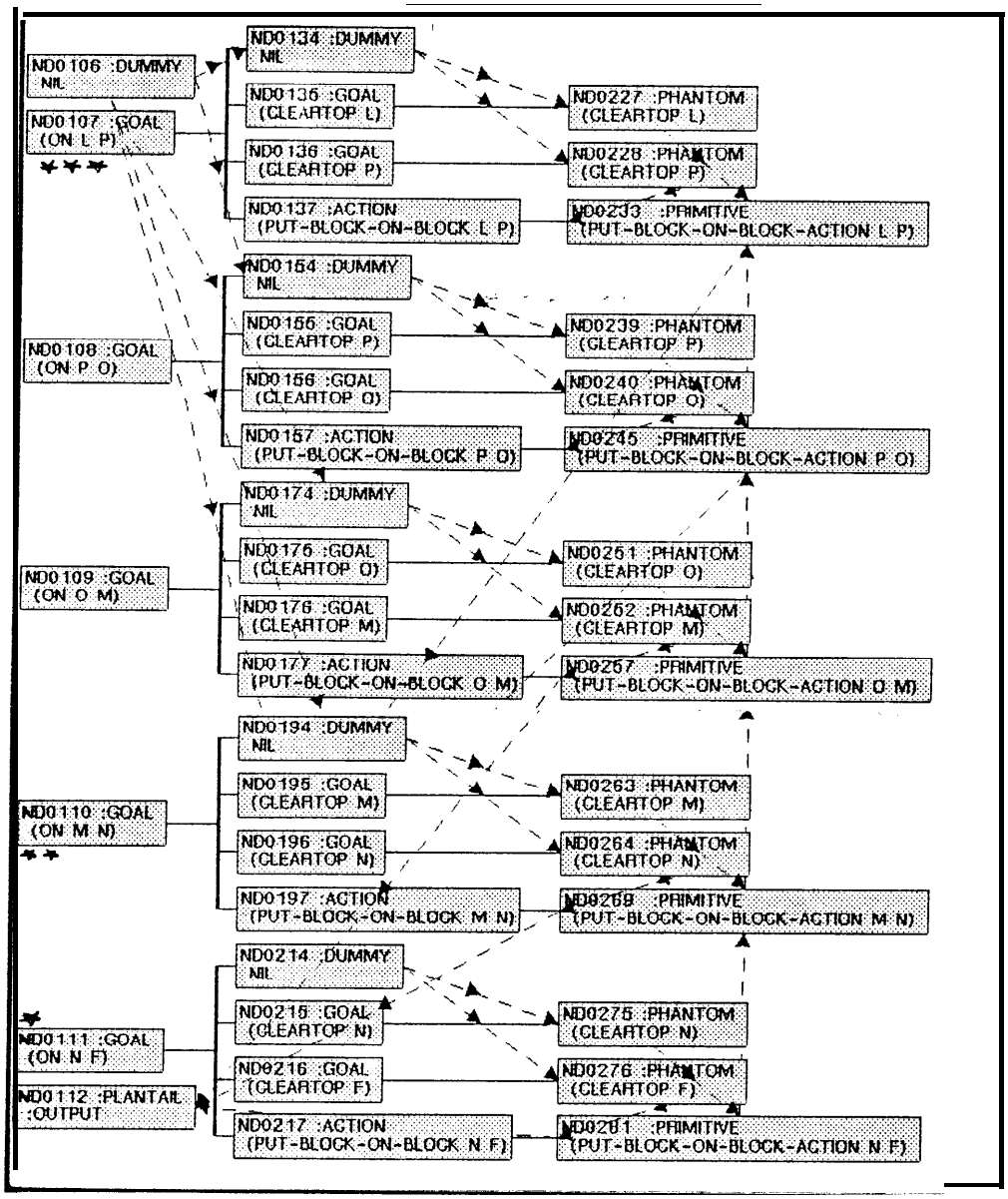

Figure A.2. 6BS plan after interpretation

Next, the annotation verification checks for any *p-phantom* validations. It finds that the validation supporting the goal *On* (*M* ,*N*) is a *p* -phantom validation since *On* (*M* ,*N*) was achieved through task reduction in the 6BS plan, while it is now true in the initial state of the new problem situation. **PRIAR** uses the planner's goal achievement procedures to check whether *On* (*M* ,*N*) can now be established from the initial state. As this check is successful, PRIAR decides to shorten the plan by pruning the validation that is currently supporting the goal *On* (*M* ,N), and to support *On* (*M* ,N) by the *new* fact from the initial state. This pruning will remove the sub-reduction below the node ND0109: *On* (*M* ,*N*) (see the double-asterisked node in Figure A.2) from the interpreted plan. Consequently, the annotation verified plan, shown in Figure A.3, does not contain any nodes of this sub-reduction.

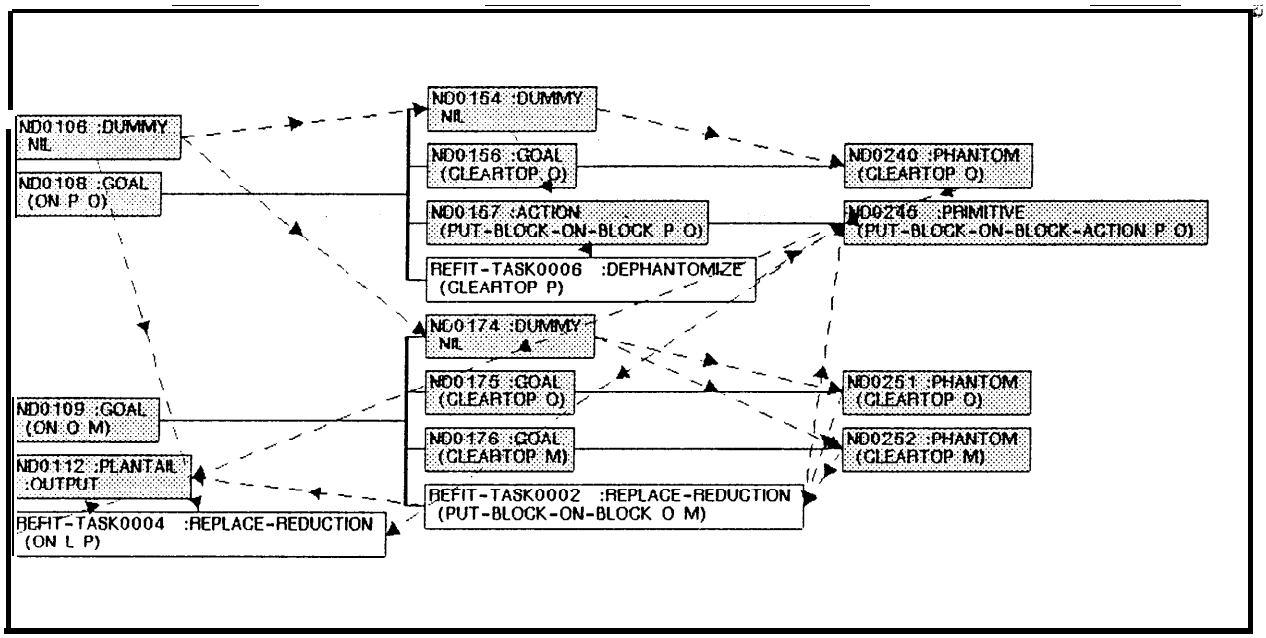

Figure A.3. 6BS plan after annotation verification

```
66 ANNOT-VERIFY: Processing p-phantom validations (if any)
67               The goal (ON M N) is supported by a p-phantom validation
68               Checking to see if it can be phantomized
69   Check-p-Phantom-Valldatlon: the condition (ON M N)
70                     can be established from new initial state! ! !
71   Check-p-Phantom-Valldatlon: Pruning the other contributor
72                          {<6::ND0268>[:PRIMITIVE(PUT-BLOCK-ON-BLOCK-ACTION M N)] . ..}
73                          from the HTN
74                          Pruning the reduction below the node
75                          {<6::ND0109>[:GOAL(ON M N)] . ..)
76                                        To take care of this p-phantom validation
```

After taking care of unnecessary and *p-phantom* validations, the annotation verification procedure finds that the validation supporting the filter condition *Block* (*L*) is failing, because *L* is a *Pyramid* in the new problem situation. The appropriate repair action is to replace the sub-reduction below the node which first posted that filter condition. In **this** case, PRIAR finds that the node ND0106: *On* (*L* ,*P*), which is an ancestor of the node with the failing filter condition validation, first posted the filter condition *Block* (*L*) into the plan. So it decides to replace the sub-reduction below this node. Consequently, the annotation verified plan in Figure A.3 contains a refit task **REFIT-TASK0004:** *Achieve [On* (*L* ,*P*)] in place of the replaced sub-reduction.

```
77 ANNOT-VERIFY: Processing extra goals (if any)
78 ANNOT-VERIFY: Looking for failed validations..
79     The FILTER (:use-when) condition (BLOCK L) at node
80               {<3::ND0232>[:PRIMITIVE(PUT-BLOCK-ON-BLOCK-ACTION L P)] . ..}
81               is failing because of :out fact (BLOCK L) in <INIT-STATE>
82
83          REFIT-FILTER-COND-FAILURE: Adding a refit-task
84                    {<REFIT-TASK0004>[:REPLACE-REDUCTION(ON L P)] . ..}
85                    to re-reduce the node
86               {<3::ND0106>[:GOAL(ON L P)]...)
```

```
87      REFIT-FILTER-COND-FAILURE: Removing the replaced reduction from the plan
```

The annotation verification procedure goes on to find a second failing filter condition validation and a failing phantom condition validation (lines 88-105). It repairs them by adding a second replace reduction refit task and a dephantomize refit task to the annotation-verified plan. Figure A.3 shows the partially reduced HTN after the annotation-verification process. This is then sent to the planner for refitting.

```
88      The FILTER (:use-when) condition (ON 0 TABLE)
89              at node {<5::ND0251>[:PRIMITIVE(PUT-BLOCK-ON-BLOCK-ACTION 0 M)] . ..) is failing
90          because of :out fact (ON 0 TABLE) in <INIT-STATE>

91      REFIT-FILTER-COND-FAILURE: Adding a refit-task
92                      {<REFIT-TASK0002>[:REPLACE-REDUCTION(PUT-BLOCK-ON-BLOCK 0 M)] . . . }
93                       to m-reduce the node
94                  {<5::ND0176>[:ACTION(PUT-BLOCK-ON-BLOCK 0 M)] . ..)
95      REFIT-FILTER-COND-FAILURE: Removing the replaced reduction from the plan

96      The :PRECOND condition (CLEARTOP P) at node
97          {<4::ND0106> [:PRIMITIVE(PUT-BLOCK-ON-BLOCK-ACTION P 0)]
98                  is failing because of :out fact (CLEARTOP P) in <INIT-STATE>
99
100     DEPHANTOMIZE.COAL: Adding refit-task
101                 {<REFIT-TASK0006>[:DEPHANTOMIZE(CLEARTOP P)]... }
102                 in the place of the phantom goal
103                 {<12::ND0154>[:GOAL(CLEARTOP P)] . ..)
104
105 annot-verify: Entering refit-tasks into the planners TASK-QUEUE in correct order

106 Entering {<REFIT-TASK0004>[:REPLACE-REDUCTION(ON L P)]...)
107 Entering {<REFIT-TASK0002>[:REPLACE-REDUCTION(PUT-BLOCK-ON-BLOCK 0 M)]...}
108 Entering {<REFIT-TASK0006>[:DEPHANTOMIZE(CLEARTOP P)]...)

109 ANNOT-VERIFY: END
```

The planner starts by reducing the replace-reduction refit task corresponding to $On(L,P)$ (lines 11 l-129). Since L is a pyramid, the planner finds that the only appropriate schema instance for reducing this refit task is **MAKE-PYRAMID-ON-BLOCK@** $,P$). Next, since the refit task is a replace-reduction refit task, during installation, **PRIAR** finds that the e-precondition of the refit task that was supporting the condition Clear(L), is no longer required by the new schema instance (the reason being that L, which is a pyramid, is always clear). So the e-precondition is pruned from the HTN. After this, the planner goes on to reduce the refit task with the chosen schema. The other two refit tasks are also reduced in turn by a similar process (lines 135-141).

Figure A.4 shows the result of refitting, which is a completely reduced HTN for solving the 5BP problem. The shaded nodes represent the parts of the 6BS plan that remain applicable to the 5BP problem, and the white nodes represent the reductions of refit tasks. There is no separate sub-plan for achieving the goal $On(M,N)$ in this **HTN** since this is made true from the initial state of 5BP problem.

```
110 **********Calling Generative Planner**********

111 PLANNER: Expanding refit task Achieve [(ON L P)]

112 PLANNER: The schema choices to reduce the refit task are:
113 ({SCH0021}: MAKE-PYRAMID-ON-BLOCK00140018::(ON L P)
114               BY (cl ::ND0020>[:ACTION(PUT-PYRAMID-ON-BLOCK L P)]...)
115 The chosen schema is :
116 {SCH0021}
117   MAKE-PYRAMID-ON-BLOCK00140018::(ON L P)
118     Expansion:
119         0 {<0::ND0019>[:GOAL(CLEARTOP P)]
120         1 {<1::ND0020>[:ACTION(PUT-PYRAMID-ON-BLOCK L P)])
121     Conditions:
122         <<SC5125>> :PRECOND (CLEARTOP P) :at 1 :from (0)
123         <<SC5126>> :USE-WHEN (PYRAMID L) :at 0 :from (-24)
124         <<SC5127>> :USE-WHEN (BLOCK P) :at 1 :from (-24)
125                                                    . . .

126 Install Choice: Installing the schema ({ SCHO021)
127   MAKE-PYRAMID-ON-BLOCK00140018::(ON L P) BY
128     {<1::ND0020>[:ACTION(PUT-PYRAMID-ON-BLOCK L P)] ..}
129   to Re-reduce the task ({<REFIT TASK0004>[:REPLACE-REDUCTION(ON L P)]})

130   The c -precondition (CLEARTOP L) of the task
131     ({<REFIT-TASK0004>[:REPLACE-REDUCTION(ON L P)]))
132     is not required by the chosen schema
133
134   So, pruning the validation corresponding to this ncondition
```

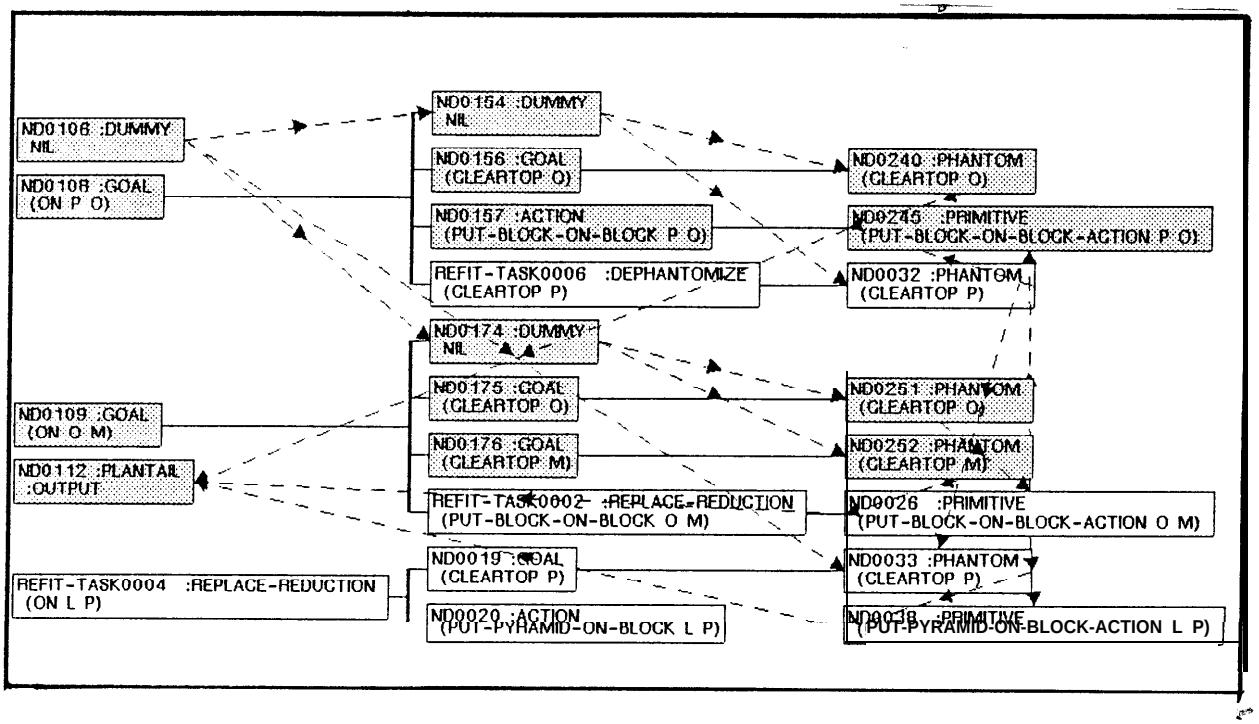

Figure A.4. Result of refitting 6BS plan to 5BP problem

```
PLANNER: Expanding Refit-task Achieve [(PUT-BLOCK-ON-BLOCK 0 M)]
PLANNER: The schema choices to reduce the refit-task are:
      ({ SCH0027}::PUT-BLOCK-ON-BLOCK00220025::(PUT-BLOCK-ON-BLOCK 0 M) BY
                  {<0::ND0026>[:PRIMITIVE(PUT-BLOCK-ON-BLOCK-ACTION 0 M)]}

PLANNER: Expanding Refit-task Achieve [(CLEARTOP P)]
        The refit-task is PHANTOMIZED with an effect of the node(s)
        ({<5::ND0026>[:PRIMITIVE(PUT-BLOCK-ON-BLOCK-ACTION 0 M)]})

****The Planning is OVER
The plan is...

**********a****

 5: :PRIMITIV E (PUT-BLOCK-ON-BLOCK-ACTION 0 M) [Prenodes:(6 1 5 16)] [Succnodes: (3 1 4)]

4: :PRIMITIVE (PUT-BLOCK-ON-BLOCK-ACTION P 0) [Prenodes:(12 5 13)] [Succnodes: (23 1)]

 3: :PRIMITIVE (PUT-PYRAMID-ON-BLOCK-ACITON L P) [Prenodes:(23 0 5)] [Succnodes: (1)]


**********GOAL STATE**********
                    <<SC0062>> :PRECOND (ON 0 M) :at 1 :from (5)
                    <<SC0061>> :PRECOND (ON P 0) :at 1 :from (4)
                    <<SC0060>> :PRECOND (ON L P) :at 1 :from (3)
                    <<SC0059>> :PRECOND (ON M N) :at 1 :from (0)
```

Appendix B. The Blocks World Domain Specification

```
(setf *autocond* t)
 ;;;Automatically fill in sub-goals as preconditions of main goal steps

(opschema  make-pyramid-on-block
   :todo        (on ?x ?y)
   :expansion  ((step1 :goal (cleartop ?y))
                  (step2 :action (put-pyramid-on-block ?x ?y)) )
   :orderings  ((step1 → step2))
   :conditions ((:filter (pyramid ?x) :at stepl)
                  (:filter (block ?y) :at step2))
   :effects     ((step2 :delete (cleartop ?y))
                  (step2 :assert (on ?x ?y)))
   :variables  (?x  ?y))

(opschema  make-pyramid-on-table
   :todo        (on ?x table)
   :expansion  ((step1 :action (put-pyramid-on-table  ?x ?y)))
   :conditions ((:filter (pyramid ?x) :at stepl))
   :effects     ((step1 :assert (on ?x table)))
   :variables  (?x  ?y))

(opschema make-block-on-block
   :todo        (on ?x ?y)
   :expansion  ((step1 :goal (cleartop ?x))
                  (step2 :goal (cleartop ?y))
                  (step3 :action (put-block-on-block ?x ?y)))
   :orderings   ((step1 → step3) ( step2 → step3))
   :conditions ((:filter (block ?x) :at stepl)
                  (:filter (block ?y) :at step2))
   :effects     ((step3 :delete (cleartop ?y))
                  (step3 :assert (on ?x ?y)))
   :variables  (?x  ?y))

(opschema  make-block-on-table
   :todo        (on ?x table)
   :expansion  ((step1 :goal (clear-top ?x))
                  (step2 :action (put-block-on-table ?x table)))
   :conditions ((:filter (block ?x) :at stepl))
   :orderings  ( (step1 → step2))
   :effects     ((step2 :assert (on ?x table)))
   :variables  (?x  ?y))


(opschema make-clear-table
   :todo        (cleartop ?x)
   :expansion  ((step1 :goal (cleartop ?y))
                  (step2 :action (put-block-on-table ?y table)))
   :orderings  ((step1 → step2))
   :conditions ((:filter (block ?x) :at stepl)
                  (:filter (block ?y) :at step2)
                  (:filter (on ?y ?x) :at step2) )
   :effects     ((step2 :assert (cleartop ?x))
                  (step2 :assert (on ?y table)))
```

```
   :variables (?x ?y))

(opscbema makeclear-block
   :todo       (cleartop ?x)
   :expansion ((step1 :goal (cleartop ?y))
                    (step2 :action (put-block-on-block ?y ?z)) )
   :orderings ((step1 → step2))
   :conditions ((:filter (block ?x) :at stepl)
                    (:filter (block ?y) :at stepl)
                    (:filter (block ?z) :at stepl)
                    (:filter (on ?y ?x) :at step2)
                    (:filter (cleartop ?z) :at step2)
                    (:filter (not (equal ?z ?y)) :at stepl)
                    (:filter (not (equal ?x ?z)) :at stepl) )
   :effects    ((step2 :assert (cleartop ?x))
                    (step2 :assert (on ?y ?z))
                    (step2 :delete (cleartop ?z)) )
   :variables (?x ?y ?z))

(actschema put-block-on-block
    :todo       (put-block-on-block ?x ?y)
    :expansion ((step1 :primitive (put-block-on-block-action ?x ?y)))
    :conditions ((:filter (block ?x) :at stepl)
                    (:filter (block ?y) :at stepl)
                    (:filter (cleartop ?x) :at stepl)
                    (:filter (cleartop ?y ) :at stepl)
                    (:filter (on ?x ?z) :at stepl) )
    :effects    ((step1 :assert (on ?x ?y))
                    (step1 :assert (cleartop ?z))
                    (step1 :delete (cleartop ?y))
                    (step1 :delete (on ?x ?z)) )
   :variables (?x ?y ?z)
    )

(actschema put-pyramid-on-block
    :todo       (put-pyramid-on-block ?x ?y)
    :expansion ((step1 :primitive (put-pyramid-on-block-action ?x ?y)))
    :conditions ((:filter (pyramid ?x) :at stepl)
                    (:filter (block ?y) :at stepl)
                    (:filter (cleartop ?y ) :at stepl)
                    (:filter (on ?x ?z) :at stepl) )
    :effects    ((step1 :assert (on ?x ?y))
                    (step1 :assert (cleartop ?z))
                    (step1 :delete (cleartop ?y))
                    (step1 :delete (on ?x ?z)) )
   :variables (?x ?y ?z))

(actschema put-block-on-table
    :todo       (put-block-on-table ?x table)
    :expansion ((step1 :primitive (put-block-on-table-action ?x table)))
    :conditions ((:filter (block ?x) :at stepl)
                    (:filter (cleartop ?x) :at stepl)
                    (:filter (on ?x ?z) :at stepl) )
    :effects    ((step1 :assert (on ?x table))
                    (step1 :assert (cleartop ?z))
                    (step1 :delete (on ?x ?z)) )
```

```
    : variables     (?x ?z))

(actschema  put-pyramid-on-table
     :todo          (put-pyramid-on-table ?x table)
     :expansion ((step1 :primitive (put-pyramid-on-table-action ?x table)))
     :conditions ((:filter (pyramid ?x) :at stepl)
                        (:filter (on ?x ?z) :at stepl) )
     :effects       ((step1 :assert (on ?x table))
                        (step1 :assert (cleat-top ?z))
                        (step1 :delete (on ?x ?z)))
   :variables     (?x ?z))

(domain-axioms
   (← (cleartop table)
        t)
   ;;(cleartop table) is always derivable
   (← (not (cleartop ?x))
        (on ?y ?x)) ;;if ?y is on ?x then ?x cannot be clear
   (← (not (on ?other ?x))
        (and (block ?x)(on ?z ?x)))
   ;;if ?x is a block and ?z is on top of ?x, nothing else is on its top
   (← (not (on ?z ?other))
        (on ?z ?x))
  ;;if ?z is on ?x it is not on any other block

   (← (not (on ?x ?y))
        (pyramid ? y))
     ;;nothing can be on the top of a pyramid
   (← (equal ?x ?x)
        t)
   ;;equality  axiom)
(closed-world-predicate 'equal :set t)
;;record that equality is a closed-world predicate
```

References

1. R. Alterman, "An Adaptive Planner", *Proceedings of 5th AAAI, 1986, 65-69.*
2. J. G. Carbonell, "Derivational Analogy and its Role in Problem Solving", *Proceedings* of *AAAI,* Washington D.C., 1983, 64-69.
3. D. Chapman, "Planning for Conjunctive Goals", *Artificial Intelligence 32* (1987), *333-*377.
4. E. Chamiak and D. McDermott, ' 'Chapter 9: Managing Plans of Actions", in *Introduction to Artificial Intelligence,* Addison-Wesley Publishing Company, 1984, 485-554.
5. L. Daniel, "Planning: Modifying non-linear plans", DAI Working paper 24, University of Edinburgh, December 1977. (Also appears as ' 'Planning and Operations Research, '' in *Artificial Intelligence: Tools, Techniques and Applications,* Harper and Row, New York, 1983).
6. R. Fikes, P. Hart and N. Nilsson, "Learning and Executing Generalized Robot Plans", *Artificial Intelligence 3* (1972), *251-288.*
7. S. Ghosh, S. Kambhampati and J. Hendler, "Common Lisp Implementation of a NONLIN-based hierarchical planner: A User Manual", Technical Report (under preparation), Department of Computer Science, University of Maryland, College Park.
8. K. J. Hammond, "CHEF: A Model of Case-Based Planning", *Proceedings* ***of*** *5th AAAI,* 1986, 267-271.
9. P. J. Hayes, "A Representation for Robot Plans", *Proceedings of 4th IJCAI*, 1975.
10. J. Hendler, A. Tate and M. Drummond, "AI Planning: Systems and Techniques", *AI Magazine,* Summer, 1990 (To appear).
11. M. N. Huhns and R. D. Acosta, "ARGO: A System for Design by Analogy", *IEEE Expert,* Fall 1988, 53-68. (Also appears in Proc. of 4th IEEE Conf. on Appln. of AI, 1988).
12. S. Kambhampati and J. A. Hendler, "Adaptation of Plans via Annotation and Verification' ', *Ist Intl. Conf. on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems, 198 8,* 164- 170.
13. S. Kambhampati, "Flexible Reuse and Modification in Hierarchical Planning: A Validation Structure Based Approach", CS-Tech. Rep.-2334, CAR-Tech. Rep.-469, Center for Automation Research, Department of Computer Science, University of Maryland, College Park, MD 20742, October 1989. (Ph.D. Dissertation).
14. 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.

15. S. Kambhampati and J. A. Hendler, "Control of Refitting during Plan Reuse", *11 th International Joint Conference on Artificial Intelligence,* Detroit, Michigan, USA, August 1989, 943-948.

16. S. Kambhampati, "A Theory of Plan Modification", *Proceedings of Eighth AAAI,* Boston, MA, 1990.

17. S. Kambhampati, "Mapping and Retrieval during Plan Reuse: A Validation-Structure Based Approach", *Proceedings of Eighth AAAI,* Boston, MA, 1990.

18. S. Kambhampati, "A Classification of Plan Modification Strategies Based on their Information Requirements", *AAAI Spring Symposium on Case-Based Reasoning,* March 1990.

19. S. Kambhampati and J. M. Tenenbaum, "Towards a Paradigm for Planning in Interactive Domains with Multiple Spcialized Domain Modules", *Proceedings of AAAI workshop on Automated Planning for Complex Domains,* Boston, MA, August 1990.

20. S. Kambhampati, "A Framework for Replanning in Hierarchical Nonlinear Planning", *AAAI Spring Symposium on Planning in Uncertain, Unpredictable or Changing Environments,* March 1990.

21. S. Kambhampati and A. Philpot, "Incremental Planning for Concurrent Product and Process Design' ', Technical Report (under preparation), Center for Design Research and Department of Computer Science, Stanford University.

22. J. L. Kolodner, "Case-Based Problem Solving", *Proceedings of the Fourth International Workshop on Machine Learning,* University of California, Irvine, June 1987, 167-178.

23. R. Korf, "Planning as Search: A Quantitative Approach", *Artificial Intelligence 33* (1987), 65-88.

24. P. Morris, R. Feldman and B. Filman, *Use of Truth Maintenance in Automatic Programming,* Intellicorp Inc., 1975 El Camino Real West, Mountain View, CA 94040, March 1990.

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

26. R. Simmons and R. Davis, "Generate, Test and Debug: Combining Associational Rules and Causal Models", *Proceedings of 10th IJCAI* IO (1987), 1071-1078.

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

28. *G.* J. Sussman, in *HACKER: a computational model of skill acquisition,* American Elsevier, New York, NY, 1977.

29. A. Tate, "Project Planning Using a Hierarchic Non-Linear Planner", Research Report 25, Department of AI, University of Edinburgh, 1976.

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

31. J. Tenenberg, "Abstraction in Planning", Rochester Tech. Rep. 250 (Doctoral Dissertation), May 198 8.

32. 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.

33. D. E. Wilkins, "Domain-independent planning: representation and plan generation", *Artificial Intelligence 22* (1984), *269.*

34. D. E. Wilkins, "Recovering from execution errors in SIPE'', *Computational Intelligence* I (1985).

35. D. E. Wilkins, "Causal reasoning in planning", *Computational Intelligence 4 (1988).*

36. D. Wilkins, in *Practical Planning*, Morgan Kaufmann Publishers, Inc., 1989.

37. Q. Yang, ''Improving the Efficiency of Planning'', Doctoral Dissertation, Department of Computer Science, University of Maryland, College Park, 1989.