Artificial Intelligence 55 (1992) 193–258
Elsevier

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 94305-4026, USA

James A. Hendler

Computer Science Department, University of Maryland, College Park, MD 20742, USA

Received June 1990
Revised August 1991

Abstract

Kambhampati, S. and J.A. Hendler, A validation-structure-based theory of plan modification and reuse, Artificial Intelligence 55 (1992) 193–258.

The ability to modify existing plans to accommodate a variety of externally imposed constraints (such as changes in the problem specification, the expected world state, or the structure of the plan) is a valuable tool for improving efficiency of planning by avoiding repetition of planning effort. In this paper, we present a theory of incremental plan modification suitable for hierarchical nonlinear planning, and describe its implementation in a system called PRIAR. In this theory, the causal and teleological structure of the plans generated by a planner are represented in the form of an explanation of correctness called the "validation structure". Individual planning decisions are justified in terms of their relation to the validation structure. 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 or changed planning situation. The repair of these inconsistencies involves removing unnecessary parts of the plan and adding new nonprimitive tasks to the plan to establish missing or failing validations. The result is a partially reduced plan with a consistent validation structure, which is then sent to the planner for complete reduction. We discuss this theory, present an empirical evaluation of the resulting plan modification system, and characterize the coverage, efficiency and limitations of the approach.

* Department of Computer Science and Engineering, Arizona State University, Tempe, AZ 85287, USA.

Correspondence to: S. Kambhampati, Department of Computer Science and Engineering, Arizona State University, Tempe, AZ 85287, USA.
E-mail: rao@asuvax.asu.edu.

1. Introduction

Although efficient domain-dependent planning systems (those which use knowledge particular to the domain of execution to limit planning search) can be designed for particular applications, the more general problem of domain-independent nonlinear planning has been shown to be intractable (NP-hard) [3].[1] Thus, approaches which may improve the cost of planning in general planning systems are being widely sought. One promising avenue for increasing efficiency is to investigate strategies that improve average-case planning efficiency by avoiding repetition of planning effort. Despite the long history of planning systems, most planning work has been concentrated on generating plans from scratch, without exploiting previous planning effort. Thus, much of the classical work in planning has been "ahistoric" and non-incremental—that is, asked to solve a problem very similar to one it has solved before, the planner performs no better than it did the first time. Recently, due both to interest in improving planning efficiency, as well as the gains being made in machine learning, the design of planning systems which can exploit previous planning experience has become a topic gaining wide interest in the planning community.

This paper focuses on an incremental plan modification strategy that allows a planner to construct a plan for a new planning problem by conservatively modifying a given plan—that is, by retaining as many of the applicable parts of the given plan as possible. Consider the following common scenario: Suppose a planner has generated a plan for a particular initial and goal state specification. It then encounters a change in the problem specification and wants to revise its plan to make it work in the new situation. One obvious possibility is for the planner to start from scratch again and find a plan for the changed problem specification. However, given the high cost of planning, it is worth investigating if the repetition of planning effort can be avoided by updating the existing plan to deal with the changes in the specification. Such an ability to incrementally modify existing plans, to make them conform to the constraints of a new or changed planning situation, can provide substantial computational advantages by avoiding repetition of computational effort and respecting previous commitments. In particular, it can support replanning to handle execution time failures or user-initiated specification changes, and reusing plans to exploit the typicality in the problem distribution. Finally, a planner's ability to incrementally modify its plans can also significantly improve its interactions with other modules in the problem solving environment whose analyses and commitments depend on the current state of the plan.

[1] This is a simplification of Chapman's results presented in [3]. A discussion of his results, and a review of AI planning systems and techniques, as well as definitions for many of the planning-related terms used in this paper, can be found in [10].

We are interested in the computational framework for supporting such incremental plan modification to accommodate changes in the problem specifications. Given an existing plan and a set of changes to be accommodated, there are two central decisions surrounding the modification process: which parts of the given plan can be salvaged, and in what ways should the other parts be changed. Two important desiderata for the plan modification capability are *flexibility* and *conservatism*. By "flexibility" we mean that the modification strategy should be able to deal with a wide variety of specification changes in a domain-independent fashion, reusing all applicable portions of the existing plan and gracefully degenerating into planning from scratch as the changes become more significant. By "conservatism" we mean that the strategy should modify the plan minimally (i.e., salvage as much of the old plan as possible) while accommodating the changes in the specification. The former is required for effective coverage of modification,[2] while the latter is needed to ensure efficiency. Designing modification strategies with these characteristics requires a systematic framework for maintaining and revising the dependencies underlying planning decisions, and the causal and teleological structure of the plans.

We have developed a theory of plan modification that allows the flexible and conservative modification of plans generated by a hierarchical nonlinear planner, and have implemented it in a system called PRIAR. In our theory, the causal and teleological structure of generated plans are represented in a form of explanation of plan correctness called a "validation structure". Individual planning decisions made during the generation of the plan are justified in terms of their relation to the validation structure. Modification is characterized as a process of detecting and removing inconsistencies in the validation structure resulting from the externally imposed constraints. Repair actions utilize the dependency structures to transform a completed plan with an inconsistent validation structure into a partially reduced plan with a consistent validation structure. This partially reduced plan is then sent to the planner for completion, and a completed plan is produced.

This formalization of the modification process provides a general unified framework for addressing many subproblems of planning and plan reuse, including plan modification, execution monitoring and replanning, incremental updating of plans (to accommodate externally imposed constraints), estimating the similarity of plans during the retrieval of plans for reuse, and controlling the choice of new actions to replace changed parts of existing plans. In this paper, we present the plan modification framework and evaluate its performance, coverage, correctness, efficiency and limitations.

[2] In other words, we do not want to be limited to some domain-dependent heuristics that may allow the planner to deal with only a prespecified set of specification changes in an efficient way.

1.1. Overview of the plan modification framework

The plan modification problem that is addressed by the PRIAR system 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^n),

(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.

In the PRIAR modification framework, the internal causal dependencies of generated plans are represented as a form of plan explanation, called a *validation structure*. The correctness of a plan is formally characterized in terms of the consistency of its validation structure. The individual steps of a plan as well as the planning decisions involved in the generation of that plan (e.g., task reductions), are justified in terms of their relation to the plan validation structure. In particular, the plan is annotated with information about the inconsistencies that will arise in the validation structure in the event these decisions have to be retracted. PRIAR provides efficient algorithms for automatically annotating and maintaining these justifications as a by-product of planning.

External changes in the plan specification are handled by computing the

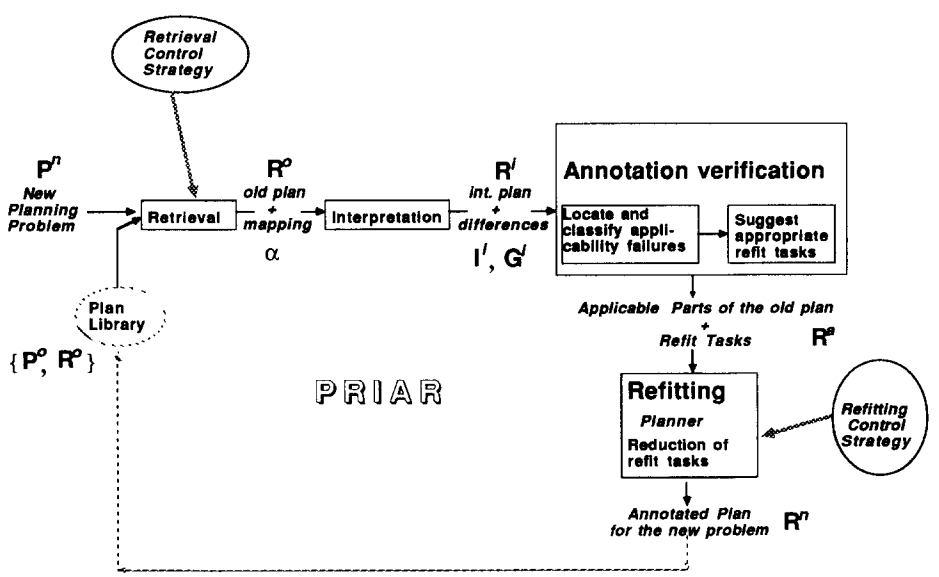

Fig. 1. Schematic overview of PRIAR.

ramifications of those changes on the plan validation structure, and repairing any resulting inconsistencies. In particular, modification 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 a 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 α 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 α. 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, based on their nature, a variety of repairs are suggested for removing them. The repairs include removing parts of R^i that are unnecessary and adding nonprimitive 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 subplanning 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 utilizes the plan validation structure to estimate the disturbance caused by various reduction choices to the applicable parts of R^a, and prefers the choices expected to cause least disturbance. Since the planner is allowed to backtrack over the refit tasks just as it would over other tasks during generative planning, the completeness of the underlying planner is not affected.

The computational savings of this approach stem from the fact that the complexity of solving the subplanning problems during refitting is on the average significantly less than the complexity of solving the entire planning problem from scratch. Furthermore, we will show that the overhead costs involved in augmenting generative planning with a plan modification capability are very low (of polynomial complexity compared to the exponential complexity of planning from scratch). Thus, our formalism provides an efficient framework for improving the run-time of domain-independent planning through plan modification.

The PRIAR modification framework has been completely implemented. The planner in the implementation is based on Tate's NONLIN [7, 26]. We modified NONLIN to handle partially reduced plans, and to automatically annotate the generated plans according to the validation-structure-based dependency repre-

sentation. The system has been tested in the blocks world domain and in a manufacturing planning domain [14]. Experiments in the blocks world certainly bear out the flexibility and efficiency of the incremental plan modification. Our results show that plan modification can provide up to 1–2 orders of magnitude reduction in planning cost, over a variety of specification changes (the details of these studies are provided in Section 7). Similar improvements have also been observed in a manufacturing planning domain [14, 15], where PRIAR's ability to incrementally modify an existing plan in response to changes in the specification also led to an improved interaction between the planner, and other specialists in the environment that make their commitments based on the plan.

1.2. Guide to this paper

The rest of this paper is organized as follows. The next section discusses previous research in plan modification, and motivates the PRIAR framework. Section 3 introduces some preliminary notation and terminology used throughout the paper. Section 4 presents the notion of plan validation structure, characterizes the correctness of a plan in terms of the consistency of its validation structure, and develops a scheme for justifying the various planning decisions in terms of their relation to the validation structure. Section 5 formally develops the modification processes for repairing various types of inconsistencies in the validation structure. Section 6 discusses applications of the modification framework to replanning and plan reuse. Section 7 discusses the empirical evaluation of the modification strategies, and Section 8 provides an analysis of the PRIAR plan modification techniques. Section 9 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.

2. Related work

Improving the efficiency of planning by exploiting previous planning experience has long been recognized as important in AI planning work; PRIAR draws a great deal from this tradition. The earliest research on learning from planning experience was done in conjunction with the STRIPS system [6], which was used to plan the motion of a robot called *Shakey* and to control it as it pushed a set of boxes through a number of interconnecting rooms. STRIPS had the capability to recover from simple execution time failures[3] and to improve

[3] A well-known SRI film shows *Shakey* following a STRIPS-generated plan using an execution monitor called PLANEX. Charley Rosen, the SRI AI Lab founder, dressed in a sinister cloak, appears and disrupts the position of the boxes during execution. PLANEX is then able to make use of information maintained by STRIPS to recover from this disruption and complete the plan.

its performance by utilizing previous planning experience. It did this by maintaining two types of information about the generated plans:

(i) *macro-operators* (also called *macrops*): generalized (variablized) sequences of operators, culled from the previous successful plans,
(ii) *triangle-tables*: data structures that recorded dependencies between the state of the world and the operator structure of the plans (and macro-operators).

However, this method of replanning left STRIPS incapable of modifying the internal structure of its stored macro-operators to suit new problem situations. Consequently the macro-operators could be used only when either the entire macrop or one of its subsequences was applicable in the current situation. Replanning in STRIPS consisted solely of attempts to restart the plan from an appropriate previously executed step.

An important reason for the inflexibility of macro-operator-based reuse in STRIPS was the impoverished plan representation which did not allow for hierarchical abstraction and least commitment. A recent hierarchical linear problem solver called ARGO [11] tries to partially overcome the former 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.

The inflexibility of macrops-based reuse led to the investigation of richer dependency structure representations (e.g. [5, 9]), and a larger variety of modification strategies (e.g. [31]). Hayes' [9] route planner was the first to advocate explicitly represented internal dependencies for guiding replanning. However, his framework was very domain-specific, and since his was a linear planner, the only replanning action allowed was deletion of some parts of the plan, thereby permitting the planner to re-achieve some higher-level independent subgoals in the hierarchical development of the plan.

NONLIN [26, 27] was the first hierarchical nonlinear planner to advocate explicit representation of goal dependencies to guide planning. Its GOST data structure was essentially a list of protection intervals associated with the plan which was used during planning to guide 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 the inter-decision dependencies induced by NONLIN.

Wilkins [31] extended the state of the art of domain-independent replanning significantly in his SIPE system. SIPE used context-layered world models, and a rich representation of plans to keep track of the dependencies between the

plan, the specification, and intermediate planning decisions. To deal with execution-time failures during replanning in SIPE, Wilkins proposed a taxonomy of repair actions, based on SIPE's dependency structures. The techniques were largely planner-specific, in that they were designed with the particulars of the SIPE program in mind.

The work described in this paper is to some extent a generalization of Wilkins' framework. However, an important difference is that Wilkins did not attempt to provide a formal basis for his dependency structures and the modification actions. This makes it very difficult to formally characterize the coverage or correctness of the modification strategies he employed. In contrast, our validation-structure-based theory presents a more formal representation of the internal dependencies used during plan modification. In particular, the correctness of the plan is defined in terms of the validation structure, and the individual planning decisions are justified in terms of their relation to the plan validation structure. This provides a clean framework to state, analyze and evaluate the modification strategies. In addition, our framework is somewhat more general than SIPE's in that it provides a unified basis for addressing several other subproblems of plan modification and reuse, including retrieval and mapping, and control of search during replanning—problems not addressed in Wilkins' model.

The framework described in this paper also relates to recent work in case-based reasoning, which addresses the issues involved in the adaptation of stored plans to new situations (e.g., [1, 2, 8]). In these systems the aim has been to make a majority of planning operations memory-based, so that plans are constructed by retrieving old plans, and applying appropriate patches to tailor them to specific circumstances. The emphasis of this work has typically been on heuristic modification strategies, with the planner assessing the correctness of modification only through external simulation or execution-time feedback. This is to some extent a reflection of the characteristics of the domains in which these systems were developed, where ensuring correctness of modification and avoiding executing-time failures were not as critical as the need to control access to planning knowledge.

Alterman's PLEXUS [1] system, for example, is an adaptive planner which starts with a highly structured plan library, and relies on the placement of a plan in the context of other plans in the library to guide adaptation. PLEXUS' primary mode of detecting applicability failures is through execution-time failures. When a failure is detected, the system attempts to exploit helpful cues from the new problem situation to trigger appropriate refitting choices to repair those applicability failures, and execute the result in turn. Similarly, Hammond's CHEF program [8] stores plans without an explicitly represented dependency structure. To be applied to a new situation, the plans 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 can lead to incorrect plans even relative to the domain knowledge contained in the case base and the modifier. CHEF relies on the assumption that the retrieval strategy and modification rules are robust enough to prevent frequent occurrence of such incorrect plans. To reduce the likelihood of failure of heuristically modified plans during execution, case-based reasoning systems test the modified plan with respect to an external domain model (simulator), and use the feedback to make repairs in the event of failure. For example, Hammond's CHEF system uses a domain-specific causal simulator to judge the correctness of a plan after it has been modified [8]. Dependency-directed debugging work in planning, such as Simmons' GORDIUS system [23, 25] also falls in this category. GORDIUS [23, 24] uses a generate–test–debug methodology, with an external simulator used for debugging the plans generated with the help of a set of heuristic associative rules.

In contrast to these approaches, our model is concerned with the flexibility of the modification strategy and the correctness of the modified plan relative to the planner's domain knowledge. Modification is closely integrated with generative planning, and is guided by the causal dependencies of the plan being modified, rather than by execution-time failures or by the results of external simulation. Thus, unlike the debugging strategies, which aim to compensate for the inadequacies of a generative planner, the primary motivation for modification in PRIAR is to improve the efficiency of planning without affecting the correctness of the plans produced.

The ability to improve efficiency without recourse to execution-time failure repair (or simulation thereof) is important in domains where execution-time failures can have significant costs or where a simulator doesn't exist. Furthermore, even if detailed external simulators are available, given the high cost of simulation and debugging (see [24]), it seems reasonable to ensure the correctness of modification with respect to the planner's own model before going to the simulation and debugging phase, as this increases the likelihood of the plan being correct with respect to the simulator.

To summarize, in relationship to previous research, our work has emphasized the provision of a formal domain-independent basis for flexible and conservative modification of plans. This approach allows us to better characterize the coverage and efficacy of the modification framework. In addition, our formalization also provides a unified framework for addressing a number of related subproblems of plan modification and reuse.

3. Preliminaries, notation and terminology

This paper develops a theory of plan modification in the context of hierarchical nonlinear planning. Hierarchical nonlinear planning (also known as hierar-

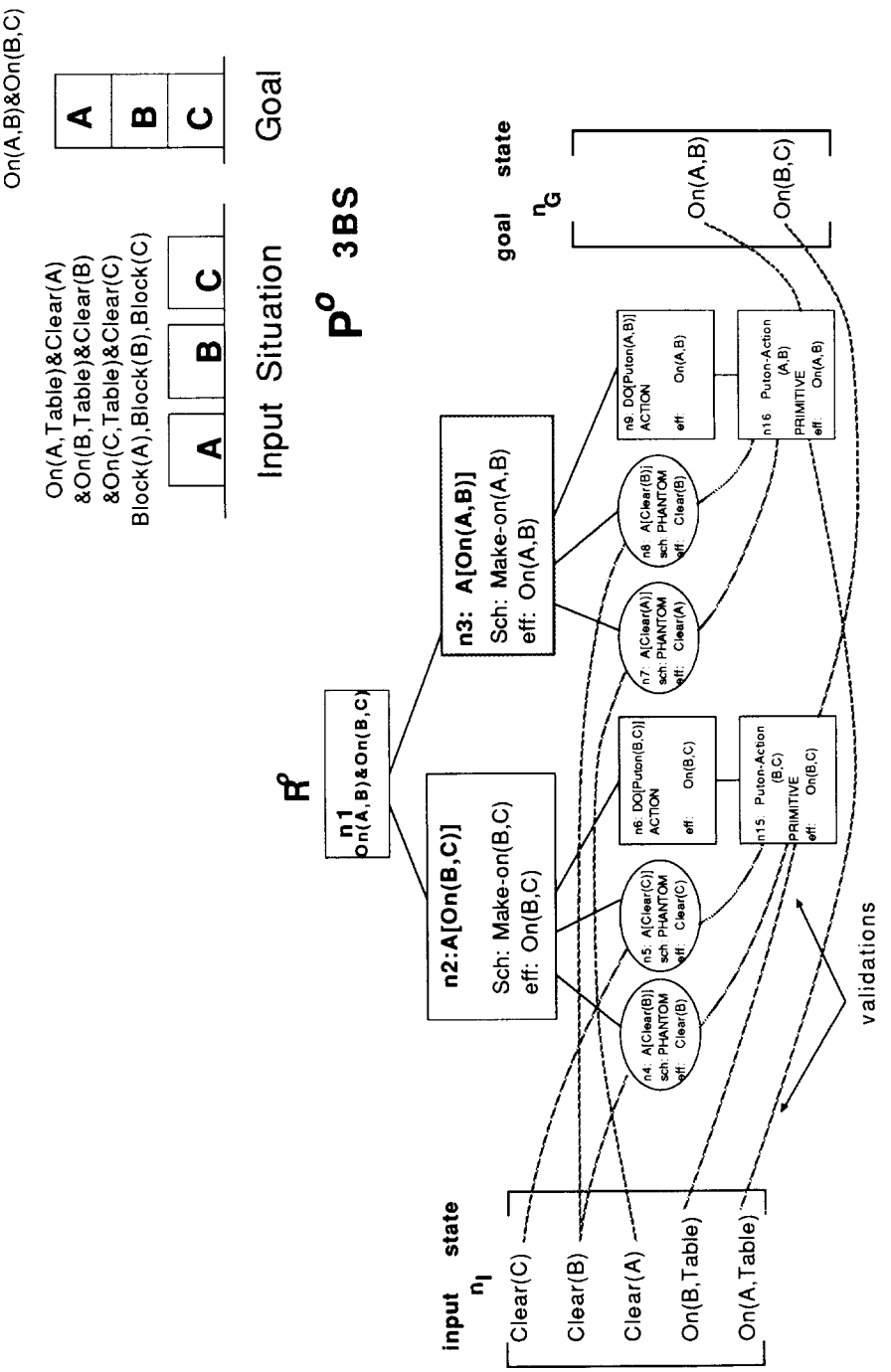

Fig. 2. Validation structure of 3BS plan.

chical planning) is the most prevalent 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 [22], NONLIN [27] and SIPE [30]. (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, shows the development of the plan and is called the "hierarchical task network" or "HTN" of the plan. Planning is considered complete when all the leaf nodes of the HTN are either primitive tasks (tasks that cannot be decomposed any further) or phantom goals (tasks whose intended effects are achieved as side-effects of some other tasks). The entire tree structure in Fig. 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.

3.1. Partial plans and task networks

A partial plan P is represented as a task network and can be formalized [33] as a 3-tuple $\langle T, O, \Pi \rangle$, where T is a set of tasks, O defines a partial ordering among elements of T, and Π is a set of conditions along with specifications about the ranges where the conditions must hold.

Task (action) representation

Each task T has a set of applicability conditions, denoted by *conditions*(T), and a set of expected postconditions, denoted by *effects*(T). In this paper, we will assume that both *effects*(T) and *conditions*(T) of each instantiated action consist of quantifier-free literals in first-order predicate calculus. The non-negated atomic formulas of *effects*(T) correspond to the operator "add-lists" in STRIPS terminology, while the negated atomic formulas correspond to the operator "delete-lists". It should be noted that this task representation does not allow conditional effects and deductive effects [3] (see Section 8.1).

Protection intervals

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.

3.2. 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 $\{C_i\}$ 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.[4] 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

$$P': \langle T \cup T_S - \{t\}, O' \cup O_S \cup O_1, \Pi' \rangle ,$$

where

(1) O' is computed by appropriately redirecting the ordering relations involving the reduced task t to its children;
(2) O_1 are the ordering relations introduced during the interaction resolution phase;
(3) finally, the protection intervals Π' are computed by (i) combining Π and Π_S, (ii) adding any protection intervals that were newly established to support the applicability conditions of the schema instance S_i, (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 interval where $t_2 = t$ to $\langle C, t_1, t_{Se} \rangle$ (where t_{Sb} and 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 [26] 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 tasks of T_S precedes t_{beg} or follows t_{end}) respectively. Other conventions might look at the effects and conditions of tasks belonging to T_S to decide t_{Sa}

[4] 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.

and t_{Sb}. For the purposes of this paper, either of these conventions is admissible.

Search control

The planner uses a backtracking control strategy to explore its search space. Thus, if the planner is unable to resolve any harmful interactions through addition of ordering relations as above, then it backtracks to explore other reduction choices.

3.3. Completed plan

A task network is said to represent a *completed plan* when none of its tasks have to be reduced further, and none of its protection intervals are violated. The planner cannot reduce certain distinguished tasks of the domain called *primitive tasks*. (It is assumed that the domain agent already knows how to execute such tasks.) Furthermore, 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 (such tasks are called *phantom goals* [4]).

3.4. 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) with HTN when the reference to P is unambiguous, and also refer to the members of T^* as the nodes of HTN.) The HTN of a plan 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 ordered 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 *subreduction* of n, and is denoted by $R(n)$. Following [4, 27], we also dis-

tinguish two types of operator 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 can test for, but cannot achieve. We shall use the notation "$F \vdash f$" to indicate that f directly follows from the set of facts in F. Finally, the modal operators "□" and "◇" denote necessary and possible truth of an assertion.

4. 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 a 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.

4.1. Validation structure

4.1.1. 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 the HTN must satisfy the relation $n_s < n_d$.

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 $|V| \leq \xi N_p$, where N_p is the length of the plan as defined above [12].)

The validation structure can be seen as an explanation of how each proposition that is either a precondition of a plan step, or a goal of the overall plan, satisfies the modal truth criterion used by the planner [3]. Given this, it is straightforward to construct the validation structure of a given partially ordered plan with the help of the relevant modal truth criterion, even if the information corresponding to the protection intervals is not maintained by the planner. In particular, for plans using TWEAK-like action representation, the validation structure can be computed in polynomial time (see [16] for details).

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 as the goal state n_G, and is provided by the effect $On(B, C)$ of node n_{15}.

Validations of a plan are distinguished by the type of conditions that they support, as well as level at which they were introduced into the plan during the reduction process. In particular, the *type* of a validation is defined as the type of the applicability condition that the validation supports (one of *filter condition*, *precondition*, or *phantom goal*). The *level* of a validation is defined as the reduction level at which it was first introduced into the HTN (see [12] for the formalization of this notion). For example, in Fig. 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.[5] The validation levels can be pre-computed efficiently at the time of annotation.

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 notion of consistency of validation structure, developed below, captures the effects of such changes on the plan validation structure.

4.1.2. Consistency of validation structure

Let V be the set of validations of the HTN $\langle P: \langle T, O, \Pi \rangle, T^*, D \rangle$, and I and G be the initial and goal state specifications of the HTN. We define a notion of correctness of a partially reduced HTN in terms of its set of validations in the following way:

[5] 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.

- For each $g \in G$, and each applicability condition C of the tasks $t \in T$, there exists a validation $v \in V$ supporting that goal or condition. If this condition is not satisfied, the plan is said to contain *missing validations*.
- None of the plan validations are violated. That is, $\forall v: \langle E, n_s, C, n_d \rangle \in V$,

$$E \in effects(n_s) \tag{1}$$

and

$$\nexists n \in T \text{ s.t. } \Diamond(n_s < n < n_d) \wedge effects(n) \vdash \neg C\,. \tag{2}$$

If this constraint is not satisfied, then the plan is said to contain *failing validations*.

In addition, we introduce a relevance condition as follows:

- For each validation $v: \langle E, n_s, C, n_d \rangle \in V$, there exists a chain of validations the first of which is supported by the effects of n_d and the last of which supports a goal of the plan. (More formally, $\forall v: \langle E, n_s, C, n_d \rangle \in V$ there exists a sequence $[v^1, v^2, \ldots, v^k]$ of validations belonging to V, such that (i) $v^k: \langle E^k, n_s^k, C^G, n_G \rangle$ supports a goal of the plan (i.e., $C^G \in G$) and (ii) $v^{k-1}: \langle E^{k-1}, n_s^{k-1}, C^k, n_s^k \rangle$ supports an applicability condition C^k on n_s^k, v^{k-2} supports an applicability condition of n_s^{k-1} and so on, with v^1 being supported by an effect of n_d.) If this constraint is not satisfied, then the plan is said to contain *unnecessary validations*.

A plan that satisfies all these conditions is said to have a *consistent validation structure*. The missing, failing and unnecessary validations defined above are collectively referred to as *inconsistencies* in the plan validation structure. Note that this definition of correctness is applicable to both completely and partially reduced HTNs. In particular, a completely reduced plan with a consistent validation structure constitutes a valid executable plan.

Let us consider the example of the 3BS plan shown in Fig. 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 $On(A, D)$, there will be no validation supporting the condition node pair $\langle On(A, D), n_G \rangle$. There is then 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 $\langle Clear(A), n_1, Clear(A), n_7 \rangle$ will fail, as $effect(n_s) \nvdash Clear(A)$.

4.2. Justifying planning decisions in terms of validation structure

To facilitate efficient reasoning about the correctness of a plan, and to guide incremental modification, we characterize the role played by the individual steps of the plan and planning decisions underlying its development in terms of their relation to the validation structure of the plan. We accomplish this by

annotating the individual nodes of the HTN of the plan with the set of validations that encapsulates the role played by the subreduction below that node in the validation structure of the overall plan. In particular, 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 $R(n)$.

4.2.1. 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 subreduction. Thus,

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

For example, the *e*-conditions of the node n_3 of the HTN of Fig. 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 subreduction. They can be used to decide when a subreduction is no longer necessary, or how a change in its effects will affect the validation structure of the parts of the plan outside the subreduction.

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

These 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 (see Section 4.3).

4.2.2. 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,

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

For example, the e-preconditions of the node n_3 in the HTN of Fig. 2 will include the validations

$$\langle \textit{Clear}(A), n_1, \textit{Clear}(A), n_7\rangle \quad \text{and} \quad \langle \textit{Clear}(B), n_1, \textit{Clear}(B), n_8\rangle .$$

The e-preconditions of a node can be used to locate the parts of rest of the plan that might become unnecessary or redundant, if the subreduction 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 $n_c \in \textit{children}(n)$, 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 $\exists n_c \in \textit{children}(n)$ such that v is an e-precondition of n_c. This follows from the fact that if $n_s \notin R(n)$ then $\forall n_c \in \textit{children}(n)$ $n_s \notin R(n_c)$ and that if $n_d \in R(n)$, then $\exists n_c \in \textit{children}(n)$ such that $n_d \in R(n_c)$.

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 of type $\langle I, n_1, C, n_i\rangle$ (where I is a fact that is true in the input state of the plan) are considered e-conditions of the input node n_1.

4.2.3. 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 subreduction $R(n)$ 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,

$$v_i: \langle E, n_s, C, n_d\rangle \otimes R(n)$$
$$\text{iff} \quad (\exists n' \in R(n)) \text{ s.t. } \textit{children}(n') = \emptyset \wedge \Diamond(n_s < n < n_d).$$

Given that $n_s < n_d$, the only cases in which $\Diamond(n_s < n' < n_d)$ are:

(i) n' is already totally ordered between n_s and n_d, i.e., $\Box(n_s < n' < n_d)$,
(ii) $n' < n_d \wedge n' \| n_s$,
(iii) $n_s < n' \wedge n' \| n_d$,
(iv) $n' \| n_s \wedge n' \| n_d$.

Using the transitivity of the "<" relation, we can simplify this disjunction to

$$n_s < n' < n_d \vee n_s \| n' \vee n_d \| n' .$$

Thus, we can re-express the "$\otimes$" relation as

$$v_i: \langle E, n_s, C, n_d \rangle \otimes R(n)$$
$$\text{iff} \quad \exists n' \in R(n) \text{ s.t. } \textit{children}(n') = \emptyset \wedge$$
$$(n_s < n' < n_d \vee n_s \| n' \vee n_d \| n') .$$

A validation $v_i: \langle E, n_s, C, n_d \rangle \in V$ 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,

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

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 Fig. 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 in gauging the effect of changes made at the subreduction below a node on the validations external to that subreduction. They are of particular importance in localizing the changes to the plan during refitting [17].

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

(1) $\textit{p-conditions}(n_1) = \textit{p-conditions}(n_G) = \emptyset$.
(2) When n is a leaf node, (i.e., children$(n) = \emptyset$), $R(n)$ will be $\{n\}$, and the definition of $\textit{p-conditions}(n)$ can be simplified as follows. From the definition of $\otimes$,

$$v_i: \langle E, n_s, C, n_d \rangle \otimes \{n\} \equiv n_s \| n \vee n_d \| n \vee (n_s < n < n_d)$$
$$\equiv \neg(n < n_s \vee n > n_d)$$

and, thus when n is a leaf node

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

(3) If $n_c \in children(n)$, and $v_c: \langle E, n_s, C, n_d \rangle \in p\text{-}conditions(n_c)$, then

$$v_c \in p\text{-}conditions(n) \quad \text{iff} \quad n_s, n_d \notin R(n) .$$

This follows from the fact that if $v_c \otimes R(n_c)$ then $\exists n' \in R(n_c)$ 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

$$\exists n_c \in children(n) \text{ s.t. } v \in p\text{-}conditions(n_c) .$$

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 subreduction, any leaf node of $R(n)$ should also have to be a leaf node of the subreduction of one of its children. So,

$$\exists n_c \in children(n) \text{ s.t. } n' \in R(n_c) .$$

Moreover, as the source and destination nodes of v do not belong to $R(n)$, they will also not belong to $R(n_c)$.

4.2.4. 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:

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

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

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, respectively, for the rest of the plan to have a consistent validation structure. In other words, the plan can be successfully executed from any state W of the world such that

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

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. In Section 6, we will show that the validation states also provide a clean framework for execution monitoring for partially ordered plans.

4.3. 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 n of the HTN is annotated with the following information:

(1) *Schema*(n), the schema instance that reduces node n,
(2) *Orderings*(n), the ordering relations that were imposed during the expansion of n (see Section 3.2),[6]
(3) e-preconditions(n),
(4) e-conditions(n),
(5) p-*conditions*(n).

Schema(n) and *Orderings*(n) are recorded in a straightforward 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 plan's set of validations,[7] V, and the partial ordering relations of the HTN. Next, using relations between the annotations of a 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 [12], 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 $O(N_P^2)$ [12]. Note that the ease of annotation computation reinforces the advantages to be gained by integrating planning and plan modification, as all the relevant information is available in the plan-time data structures. 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.

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* (shown in Figs. 3 and 4 respectively) to update node annotations consistently when incrementally

[6] Alternately, orderings can also be justified independently in terms of the validation structure; with each ordering relation, we can associate a set of validations that would be violated if that ordering is removed. This information is useful for undoing task reductions during plan modifications; see Section 5.2.1.

[7] 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. PRIAR's NONLIN-based planner maintains this information in its GOST data structure.

Procedure *Add-Validation*(v: $\langle E, n_s, C, n_d \rangle$, HTN: $\langle P: \langle T, O, \Pi \rangle, T^*, D \rangle$)

```
1  begin
2    V ← V ∪ {v}
3    foreach n' ∈ {n_s} ∪ ancestors(n_s) do
4      if n_d ∉ R(n')
5      then
6        e-conditions(n') ← e-conditions(n') ∪ {v} fi od
7    foreach n' ∈ {n_d} ∪ ancestors(n_d) do
8      if n_s ∉ R(n')
9      then
10       e-preconditions(n') ← e-preconditions(n') ∪ {v} fi od
11   foreach n ∈ T* s.t. n ≠ n_d ∧ n ≠ n_s ∧
12           children(n) = ∅ ∧ ¬(n < n_s ∨ n > n_d) do
13     foreach n' ∈ {n} ∪ ancestors(n) do
14       if n_s, n_d ∉ R(n')
15       then
16         p-conditions(n') ← p-conditions(n') ∪ {v} fi od od
17 end
```

Fig. 3. Procedure for incrementally adding validations to the HTN.

Procedure *Remove-Validation*(v: $\langle E, n_s, C, n_d \rangle$, HTN: $\langle P: \langle T, O, \Pi \rangle, T^*, D \rangle$)

```
1  begin
2    V ← V − {v}
3    foreach n' ∈ {n_s} ∪ ancestors(n_s) do
4      if n_d ∉ R(n')
5      then
6        e-conditions(n') ← e-conditions(n') − {v} fi od
7    foreach n' ∈ {n_d} ∪ ancestors(n_d) do
8      if n_s ∉ R(n')
9      then
10       e-preconditions(n') ← e-preconditions(n') − {v} fi od
11   foreach n ∈ T* do
12     p-conditions(n) ← p-conditions(n) − {v} fi od
13 end
```

Fig. 4. Procedure for incrementally removing validations from the HTN.

adding or deleting validations from the HTN.[8] PRIAR uses these procedures to re-annotate the HTN when changes are made to it 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 annotate the HTN. The time complexity of these algorithms is $O(N_P)$.

5. Modification by annotation verification

We will now turn to the plan modification process, and demonstrate the utility of the annotated validation structure in modifying a plan in response to a specification change. 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 Fig. 5 to produce a plan for a five-block-stacking problem S5BS1 (i.e., $P^n = S5BS1$),[9] shown on the right side. We shall refer to this as the 3BS→S5BS1 example.

5.1. 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 would be able to succeed with any of those mappings, selecting the right mapping can considerably reduce the cost of modification. The mapping and retrieval methodology used by PRIAR [12, 13] achieves this by selecting mappings based on the number and type of inconsistencies that would be caused in the validation structure of R^o.

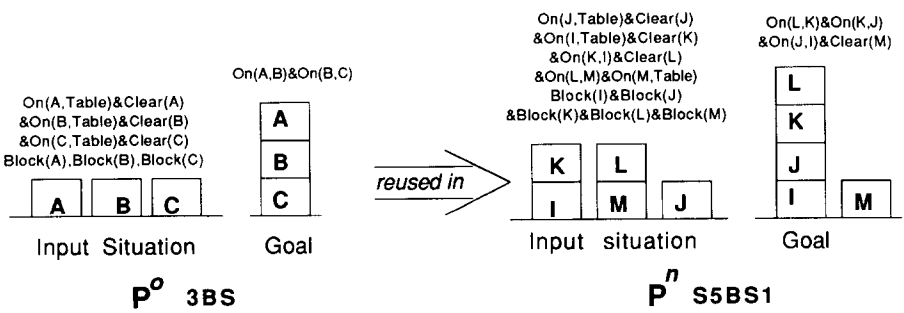

Fig. 5. The 3BS→S5BS1 modification problem.

[8] Note that any addition and removal of validations is always accompanied by a corresponding change to the set of protection intervals, Π, of the plan.

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

While the details of this strategy are beyond the scope of this paper, a brief discussion appears in Section 6.2. 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.)

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 5.2.1) on the inconsistencies in the validation structure of the interpreted plan. Let I^o and G^o be the description of the 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 α. Next, the interpreted initial state I^i, and the interpreted goal state, G^i, are computed as

$$I^i = I^n \cup I^o \cdot \alpha \quad \text{and} \quad G^i = G^n \cup G^0 \cdot \alpha$$

(where "$\cdot$" refers to the operation of object substitution). Finally, some facts of I^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

$$(f \in I^o \cdot \alpha) \wedge (i^n \nvdash f).$$

(2) A description (fact) $f \in I^i$ is marked a *new fact* iff

$$(f \in I^n) \wedge (I^o \cdot \alpha \nvdash f).$$

(3) A description (goal) $g \in G^i$ is marked an *extra goal* iff

$$(g \notin G^0 \cdot \alpha) \wedge (g \in G^n).$$

(4) A description (goal) $(g \in G^i)$ is marked an *unnecessary goal* iff

$$(g \in G^o \cdot \alpha) \wedge (g \notin G^n).$$

At the end of this processing, R^i, I^i and G^i are sent to the annotation verification procedure.

5.1.1. Example

Let us assume that the mapping strategy selects $\alpha = [A \rightarrow K, B \rightarrow J, C \rightarrow I]$ as the mapping from 3BS to S5BS1. Figure 6 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

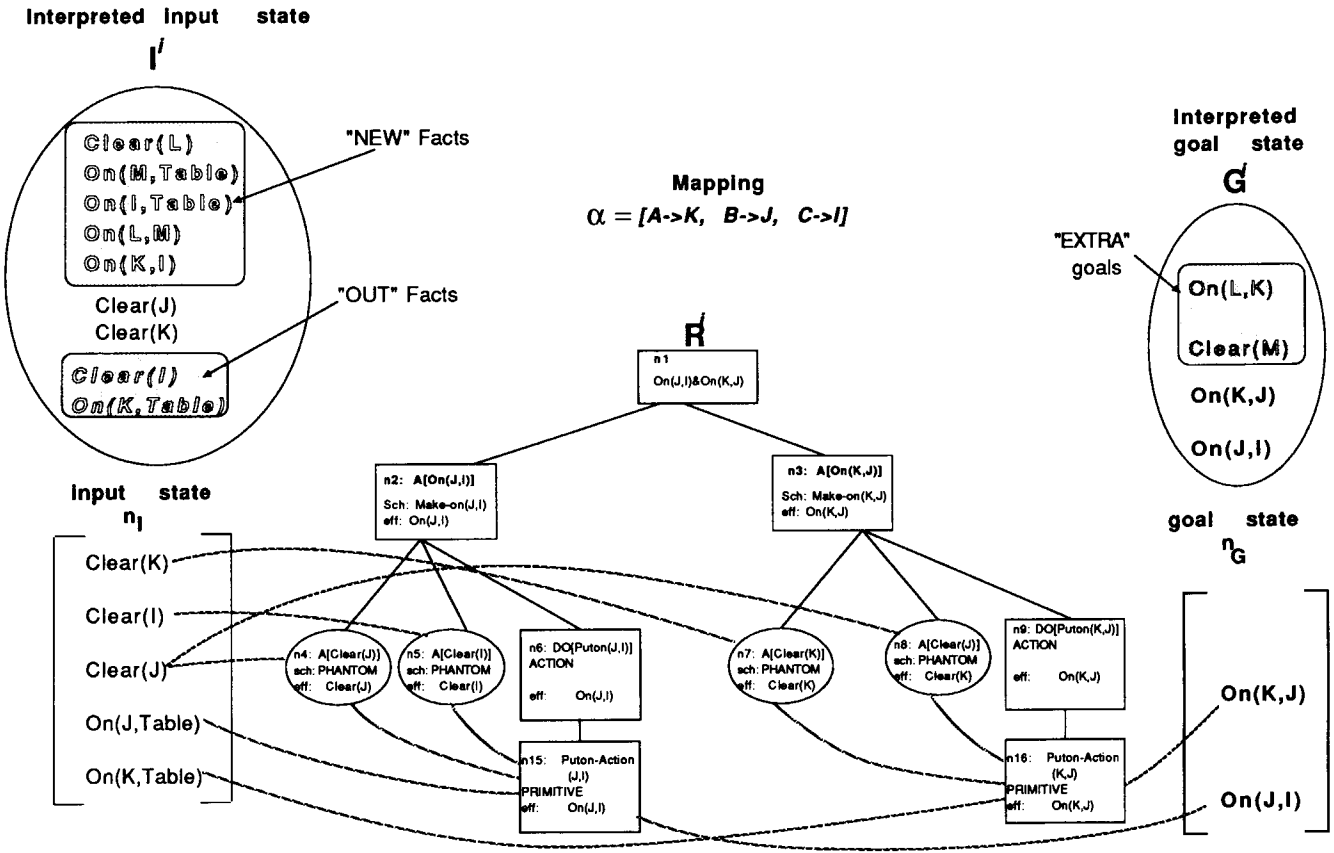

Fig. 6. Interpreted plan for 3BS→S5BS1.

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^i. Similarly, the goals *On*(L, K) and *Clear*(M) of S5BS1 are not goals of the interpreted 3BS plan. So, they are marked *extra* goals in G^i. There are no *unnecessary* goals.

5.2. Annotation verification and refit task specification

At the end of the interpretation procedure, R^i may not have a consistent validation structure (see Section 4.1.2) 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*; they 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 nonprimitive 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 7 provides the top-level control structure of the annotation verification process. The different subprocedures specialize in repairing specific types of inconsistencies in the validation structure. At the end of the annotation verification process, the HTN will be a partially (and perhaps totally) reduced task network with a consistent validation structure.

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.

5.2.1. 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 annotations on R^i. After removing an unnecessary validation from the HTN (which will also involve

```
Procedure Annotation-Verification( )
 1  input: R^i: interpreted plan,
 2         I^i: interpreted input state,
 3         G^i: interpreted goal state
 4  begin
 5    foreach g ∈ G^i s.t.
 6            g is marked as an unnecessary-goal do
 7      find v: ⟨E, n_s, C, n_G⟩ ∈ A^P(n_G) s.t. C = g
 8          Prune-Validation(v) od
 9    foreach g ∈ G^i s.t.
10            g is marked as an extra-goal do
11    Repair-Missing-Validation(g: condition, n_G: node) od
12    foreach f ∈ I^i s.t.
13            f is marked as an out-fact do
14      foreach v: ⟨E, n_1, C, n_d⟩ ∈ A^s(n_1) s.t. E = f do
15        if E' ∈ I^i s.t. E' is marked new ∧ E' ⊢ C
16          /*Verification*/
17        then do
18          Remove-Validation(v)
19          Add-Validation(v': ⟨E', n_1, C, n_d⟩) od
20        elseif type(C) = Precondition
21        then
22          Repair-Failing-Precondition-Validation(v)
23        elseif type(C) = Phantom
24              /*n_d is a phantom node*/
25        then
26          Repair-Failing-Phantom-Validation(v)
27        elseif type(C) = Filter-Condition
28        then
29          Repair-Failing-Filter-Condition-Validation(v) od od
30    foreach v: ⟨E, n_s, C, n_d⟩ ∈ V s.t.
31            n_s ≠ n_1 ∧ E ∈ I^i ∧ E is marked new in I^i
32            /*checking for serendipitous effects*/do
33      Exploit-p-Phantom-Validation(v) od
34  end
```

Fig. 7. Annotation verification procedure.

incrementally re-annotating the HTN, see Section 4.3), the HTN is searched for any node n_v that has no e-conditions. If such a node is found, then its subreduction, $R(n_v)$, 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 Fig. 8 gives the details of this process. After removing the unnecessary validation v from the plan, it checks to see if there are any subreductions that have no useful effects (lines 3–6). (Because of the explicit representation of the validation structure as annotations on the plan, this check is straightforward.) If there are such subreductions, they have to be removed from the HTN (lines 8–17). This involves removing all the internal validations of that subreduction from the HTN (lines 9–10), and recursively pruning the validations corresponding to the external preconditions of that subreduction (lines 11–12). This latter action is performed 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 10) not only removes the given validation, but also updates the validation structure (V) and the protection intervals (Π) of the HTN consistently. Finally, the subreduction is unlinked from the HTN (lines 13–16), and the partial ordering on the HTN (O) is updated so that the ordering relations that were imposed because of the expansions involved in $R(n)$ are retracted. This retraction is accomplished with the help of the *Orderings* field of each node in $R(n)$ (see Section 4.3) which stores the ordering relations that were imposed because of the expansion below that node. The procedure involves:

Procedure *Prune-Validation* (v: $\langle E, n_s, C, n_d \rangle$, HTN: $\langle P: \langle T, O, \Pi \rangle, T^*, D \rangle$)

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

Fig. 8. Procedure for repairing unnecessary validations.

(i) retracting from O all the ordering relations that are stored in the *Orderings* field of the removed nodes ($R(n)$),
(ii) appropriately redirecting[10] any remaining ordering relations of O involving the removed nodes (these correspond to the orderings that were inherited from the ancestors of n; see Section 3.2).

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 subreduction that has at least one e-condition (corresponding to some useful effect). 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, if the cost of execution of the plan were paramount, then it would be necessary to see if the remaining useful effects of the subreduction 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 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 going to the airport. This type of analysis can be done with the help of the "levels" of validations (see Section 4.1): we might decide to remove a subreduction $R(n)$ 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.

5.2.2. 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 $n_I < n_m < n_d$, 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 determine whether that validation introduces any

[10] To a sibling of n in case of pruned reduction, and to n in the case of a replaced reduction (see (3) in Section 3.2).

new failing validations into the plan (by causing harmful interactions with already established protection intervals of the plan). If it does, then those will be treated as additional inconsistencies to be handled by the annotation verification process. Finally, the incremental annotation procedures (Section 4.3) are 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.

5.2.3. Failing validations

The facts of I^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 would still 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, phantom goals,[11] and filter conditions—and discuss each of them in turn below.

5.2.3.1. 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 re-achieved. 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_v) = *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

[11] 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 [26], phantom goal validations support the "supervised conditions" of a schema (i.e., applicability conditions for which subgoals to establish the conditions are explicitly specified in the schema), while the precondition validations support the "unsupervised conditions" of a schema (i.e., applicability conditions for which no explicit subgoals are specified inside the schema).

introduce any further inconsistencies into the validation structure (see Section 8.1).) Finally, the annotations on other nodes of the HTN are adjusted incrementally to reflect this change.

5.2.3.2. Failing phantom validations

If a validation v_p: $\langle E, n_s, C, n_p \rangle$ 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 will know that it has to re-achieve that goal. This step essentially involves retracting the phantomization decision (by adding a refit task to achieve the phantom goal), and updating the HTN appropriately (similar to the process done in the *Prune-Validation* procedure (Fig. 8, lines 13–17)). Once this change is made, the failing validation v_p is no longer required by the node n_p, and so it is removed and the node annotations are updated appropriately.

5.2.3.3. Failing filter condition validations

In contrast to the validations supporting the preconditions and the phantom goals, the validations supporting failing filter conditions cannot be re-achieved 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 reduction will be until the refitting time, this processing is deferred until then (see Section 5.3).[12]

The procedure shown in Fig. 9 details the treatment of this type of validation failure during annotation verification. In lines 3–4, it finds the node n' that should be re-reduced by checking the filter conditions of the ancestors of n. Lines 6–18 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 7–11). Otherwise, such e-conditions represent internal validations of $R(n')$, and are removed from the validation structure (line 12). At the end of this processing, all the useful external effects of $R(n')$

[12] This type of applicability failure is very serious as it may require replacement of potentially large parts of the plan being reused, thereby increasing the cost of refitting. In [12, 13], we show that PRIAR's retrieval and mapping strategy tends to prefer reuse candidates that have fewer applicability failures of this type.

```
Procedure Repair-Failing-Filter-Condition-Validation
  (v_f: ⟨E, n_s, C, n_d⟩, HTN: ⟨P: ⟨T, O, Π⟩, T*, D⟩)
 1  begin
 2    Remove-Validation(v_f)
 3    find n' ∈ Ancestors(n_d) ∪ {n_d} s.t.
 4         C ∈ filter-conditions(n')
 5         /*replace reduction below n'*/
 6    foreach n_c ∈ R(n') s.t. children (n_c) = ∅ do
 7      foreach v': ⟨E', n'_s, C', n'_d⟩ ∈ e-conditions(n_c) do
 8        if v' ∈ e-conditions(n')
 9        then do
10          Remove-Validation(v')
11          Add-Validation(v": ⟨E', n', C', n'_d⟩) od
12        else Remove-Validation(v') fi od
13      foreach v': ⟨E', n'_s, C', n'_d⟩ ∈ e-preconditions(n_c) do
14        if v' ∈ e-preconditions(n')
15        then do
16          Remove-Validation(v')
17          Add-Validation(v": ⟨E', n'_s, C', n'⟩) od
18        else Remove-Validation(v') fi 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-Ordering(O, 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
```

Fig. 9. Procedure for repairing failing filter condition validations.

have n' as their source. Similar processing is done for the e-preconditions of the nodes of $R(n')$ (lines 13–18). Finally, all the descendents 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 Section 3.2). 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 subreduction are redirected rather than pruned.)

5.2.4. 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 $I^i \vdash E$. 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 reasoning 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 5.2.1). Currently, we do not allow PRIAR to add steps (cf. *white knights* [3]) or cause new interactions while establishing a p-phantom validation, and will thus exploit the serendipitous effects only if doing so will not cause substantial revisions to the plan.

5.2.5. Example

Figure 10 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 5.1. 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^i, causes the validation

$$\langle On(K, Table), n_I, On(K, Table), n_{16} \rangle$$

in R^i to fail. Since this is a failing filter condition validation,[13] 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 re-

[13] We follow the convention of [27] and classify *On*(*K*, ?*x*) as a filter condition rather than a precondition. Since some effects of the *Put-Block-on-Block* schema depend on the binding of ?*x* (in particular, it has an effect *Clear*(?*x*)), whenever the binding changes, it has to be propagated consistently throughout the plan. A way of doing this correctly is to treat this reduction-time assumption as a filter condition, and re-do the reduction at that level when the assumption fails to hold in a new situation.

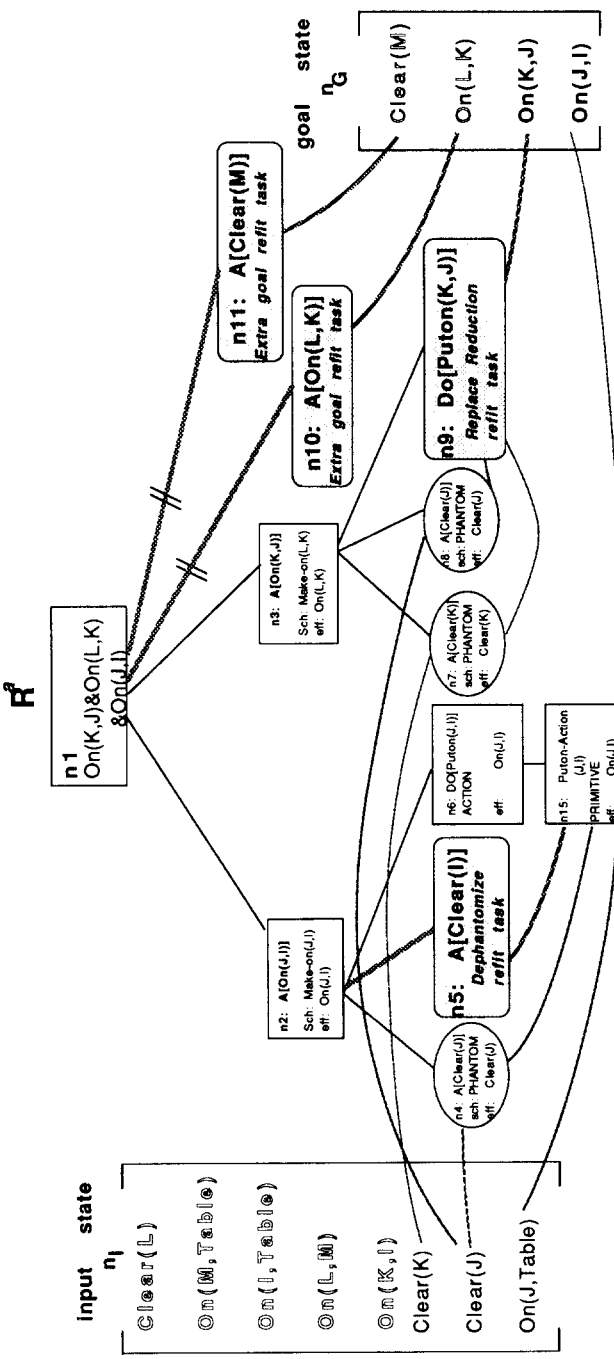

Fig. 10. Annotation-verified plan for 3BS → S5BS1.

moves $R(n_9)$ from the HTN, and adds a *replace reduction* refit task n_9: *Do*[*Puton*(K, J)]. The e-preconditions of 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 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 than 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^1 causes the validation $\langle$*Clear*(I), n_1, *Clear*(I), $n_5\rangle$ to fail. Since this validation supports the phantom goal node n_I, 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 no longer needs the failing validation from n_1, and it is removed.

Finally, the goals *Clear*(M) and *On*(L, K) of G^1 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 $n_1 < n_{10} < n_G$ and $n_1 < n_{11} < n_G$. 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.

5.2.6. Complexity of annotation verification

As we saw above, the core of the repair actions consists of tracking down validation dependencies, pruning inapplicable subreductions from the plan, adding new refit tasks and adjusting the node annotations. In [12], we show that the individual repair actions involved in the annotation verification process can each be carried out in $O(N_P^2)$ time, 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 $O(N_P^3)$ time [3]. Thus, the worst-case complexity of the repair actions is $O(N_P^3)$. Since there cannot be more than $|V|$ failing validations in a plan, the complexity of the overall annotation verification process itself is $O(|V|N_P^3)$ (where $|V| \leq \xi N_P$ as mentioned previously). Thus, the annotation verification process is of polynomial ($O(N_P^4)$) complexity in the length of the plan.

5.3. 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 planner treats the refit tasks in the same way as it treats other nonprimitive tasks—it attempts to reduce them with the help of the task reduction schemas. It is also allowed to backtrack over the refit tasks (including, ultimately, beyond R^a) to explore other parts of the search space, if required. The procedure used for reducing refit tasks is fairly similar to the one the planner normally uses for reducing nonprimitive tasks (see Section 3), with the following extension:

Refitting control

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, the refitting should leave any currently established protection intervals of the HTN unaffected. Of course, it may not be feasible to avoid interactions altogether and thus it is important to be able to compare the relative disturbances caused by various reduction choices, and select the best. Since the annotations on a node encapsulate the validations that were required to be preserved by the subreduction below a node to keep the validation structure of the HTN consistent, we can use them to estimate the relative disturbance caused by the various task reduction choices. Specifically, we measure the number and type of inconsistencies caused by the various reduction choices. In contrast to heuristics that are only concerned with minimizing the number of conflicts (cf. the "min-conflict" heuristics of Minton [21]), our strategy also weights the inconsistencies in terms of the estimated difficulty of repairing the inconsistency. The reduction choice ranked best by this strategy is used to reduce the refit task. A more detailed description of this control strategy can be found in [12, 17].

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 5.2.3.3), 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. PRIAR 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 re-achieving them during refitting, after first pruning them all during the annotation verification phase.
(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.

5.3.1. Example

Figure 11 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 Fig. 10. The top part of the figure shows the hierarchical structure of the task reductions underlying the development of the plan (abstract tasks are shown on the left, with their reductions shown to the right). The bottom part shows the chronological partial ordering relations among the leaf nodes of the HTN. The black nodes correspond to the parts of the interpreted plan that were salvaged by the reuse process, while the white nodes represent the refit tasks added during the annotation verification process and their subsequent reductions.

While reducing the refit tasks *Achieve*[*Clear*(*I*)], the planner has the choice

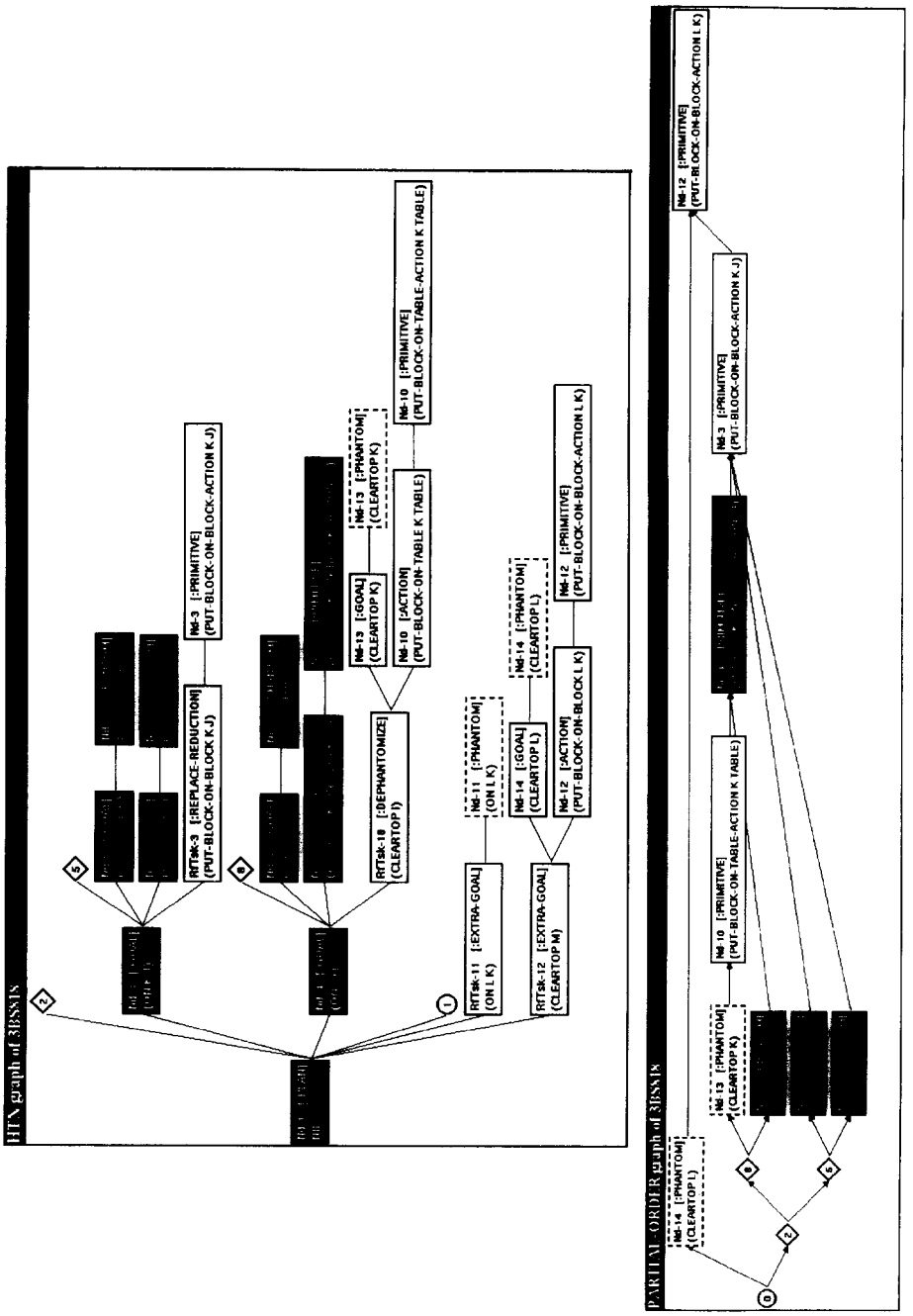

Fig. 11. The plan produced by PRIAR for 3BS → S5BS1.

of putting K on *Table*, L, M or J. The control strategy recommends putting K on *Table* since this causes the least number of conflicts with the validation structure of R^a.[14] Similarly, the refitting control strategy recommends that the extra goal refit tasks *Achieve*[*Clear*(M)] be reduced by putting L on K (rather than on *Table*, the other choice). At this point, the other extra goal refit task *Achieve*[*On*(L, K)] is achieved as a side-effect. As K is on *Table* by this point, the planner finds that the replace reduction refit task *Do*[*Puton*(K, J)] can, after all, be reduced by another instantiation of the same schema that was used to reduce it previously.[15]

6. Replanning and retrieval

An important contribution of PRIAR's modification strategy is that it provides a uniform framework for addressing various subproblems related to modification and reuse of plans. In this section we describe how the strategy can be used for dealing with failures that may arise during planning (replanning) and in helping to choose candidates (and mappings) for plan reuse.

6.1. Replanning

There are two facets of dealing with plan failures which arise in many typical planning domains: execution monitoring, in which the system executing a plan must notice that the state of the world deviates from that assumed during the planning process, and replanning, changing the plan to deal with such deviations.

6.1.1. Execution monitoring

The validation states (see Section 4.2.4) provide a precise framework for monitoring the state of the world during the execution of 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 P that may be executed next, $E(P, W, \text{EXEC})$, is computed as:

[14] Notice that putting K on J looks like a more optimal choice at this point. However, doing so at this juncture would lead to backtracking, as it affects the executability of the *Puton*(J, I) action. The locally suboptimal choice of putting K on *Table* is a characteristic of the Sussman Anomaly.

[15] If the planner chooses to reduce this refit task in the beginning itself, then it would have bound the location of K as being on top of I. In this case, since the location of K changes during the planning, the planner would have had to re-reduce that task. Such a re-reduction should not be surprising as it is a natural consequence of hierarchical promiscuity allowed in most traditional hierarchical planners (see [32] for a discussion).

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

As long as the agent executes any of the actions in $E(P, W, \text{EXEC})$ next, it is assured of following the plan, while taking into account any unexpected changes in the world state. In particular, when $E(P, W, \text{EXEC})$ contains the goal node n_G, we say that the plan has been executed successfully. Note that this execution model allows the planner to exploit parallelism in the plan: the primitive tasks can be executed in any way consistent with the partial ordering relations among them. The model also enables the planner to take advantage of any serendipitous effects to skip execution of parts of the plan that are rendered redundant, or to re-execute steps whose effects have been undone unexpectedly. Thus, we see that our model provides an extension of the STRIPS's triangle-table-based execution monitoring strategy which only worked for total orders, and could not be used for partially ordered plans.

6.1.2. Replanning

When the planner finds that *none* of its actions can be executed in the current world state ($E(P, W, \text{EXEC}) = \emptyset$), then replanning (or modification of the current plan P) is necessary. Efficiency considerations demand that as much of the existing plan be reused as possible to achieve the goals from the current situation. One obvious possibility is to modify the unexecuted portion of the plan to deal with the unexpected events; this is essentially the method used by previous replanning systems such as SIPE [31]. However, a replanning strategy that only attempts to salvage the unexecuted parts of the plan may be suboptimal in situations where the already executed parts of the plan are also reusable. For example, suppose that in the current world state some of the previously achieved goals have been undone. It may then be possible to re-achieve those goals efficiently by simply reusing (and appropriately modifying) some of the already executed portions of the plan, without having to plan for those goals again from scratch.

PRIAR's approach to this general replanning problem is to consider it as a special case of plan reuse. In particular, if R^o is the HTN being executed, G^o the set of original goals of the plan, and W the current state of the world (which necessitated the replanning), then PRIAR converts the replanning problem into the following plan reuse problem:

> Construct a plan to achieve G^o from the current world state W, reusing the plan R^o and retaining as many of its applicable parts as possible.

That is, instead of trying to repair the validation failures that are present in the

* See Note added in proof at the end of the paper.

validation state preceding the current execution point in the original plan, PRIAR tries to reuse the entire original plan and modify it to achieve the original goals starting from the current situation. The resultant plan will be a minimally modified version of the original plan that can be executed from the current world state to achieve the intended goals. This approach has been used successfully to model execution monitoring and replanning in the blocks world.

This model of conservative replanning is best suited to situations where it is reasonable to assume that the execution-time failures are caused by one-time-only accidents and unanticipated events. In particular, it is inadequate when the agent is faced with systematic repetitive failures (e.g. a greasy block keeps slipping from the robot's fingers), or failures arising from incorrectness and incompleteness of the planner's own domain models. In such cases, the scope of replanning needs to be broadened to include techniques such as debugging the planner's domain model (e.g. [23]), or planning to stop external interference. PRIAR's conservative replanning capability can however play an important role in these more general replanning frameworks.

6.2. Mapping and retrieval during reuse

While mapping is not a serious problem if the current plan itself is being modified due to some change in the specification (as in replanning), it becomes an important consideration in the case of modification during plan reuse. (In fact, the problem of choosing plans from a library and picking appropriate mappings to the current situation is the main problem being attacked in case-based planning [1, 8, 18].) 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 can considerably reduce the cost of modifying the chosen plan to conform to the constraints of the new problem. Such a selection requires an efficient similarity metric that is capable of estimating the expected cost of modifying R^o to solve P^n.

Using the framework we have described in this paper, 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 computational measure of similarity which ranks the different mappings based on the number and the type of inconsistencies they will introduce into the validation structure of the plan when it is reused in the new problem situation. The rationale behind this similarity metric—that the cost of refitting depends both on the number and on the type of validations of the old plan that have to be re-established in the new problem situation—follows from our discussion of annotation verification. In the 3BS→S5BS example, this strategy enables 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, as the former causes fewer inconsistencies in the validation structure of the interpreted 3BS plan.

In addition to being used to choose between mappings, this strategy can also be used to choose between several reuse candidates (i.e. different plans that might be modified for the current situation). By basing the retrieval on the estimated cost of modifying 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 on this retrieval and mapping strategy can be found in [12, 13].

7. Empirical evaluation

The modification techniques described in this paper have been completely implemented in the PRIAR system, which runs as compiled COMMON LISP code on a Texas Instruments EXPLORER-II Lisp Machine. The hierarchical planner used in this system is a version of NONLIN [27, 7] which has been re-implemented in COMMON LISP. PRIAR has been used to modify plans in an extended blocks world domain as well as in a manufacturing planning domain [14], where the objective is to construct a partially ordered sequence of machining operations for manufacturing simple machinable mechanical parts.

To empirically evaluate the performance of plan modification using the techniques described previously, we have conducted experiments using blocks world[16] planning problems. The evaluation trials consisted of solving blocks world problems by reusing a range of similar to dissimilar stored plans. In each trial, statistics were collected regarding the amount of effort involved in solving each problem from scratch versus solving it by modifying a given plan (in each case, PRIAR automatically computed mapping between R^0 and P^n). Approximately 80 sets of trials were conducted over a variety of problem situations and problem sizes. A comprehensive listing of these statistics can be found in [12].

Table 1 presents representative statistics from the experiments. The entries compare planning times (measured in cpu seconds), the number of task reductions (denoted by xn), and the number of detected interactions (denoted by xi), for from-scratch planning and for planning with reuse, in some representative experiments. The 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, the entry 3BS→S5BS1 in Table 1 corresponds to the example discussed in the previous sections. A complete listing of the test problem specifications can be found in [12]. The last

[16] See Appendix B for a specification of the axiomatization used.

Table 1
Sample statistics for PRIAR reuse

$R^0 \rightarrow P^n$	P^n from scratch	Reuse R^n	Savings (%)
3BS → 4BS1	[**4.0**s, 12n, 5i]	[**2.4**s, 4n 1i]	**39**
3BS → S5BS1	[**12.4**s, 17n, 22i]	]**5.2**s, 8n, 12i]	**58**
5BS → 7BS1	[**38.6**s, 24n, 13i]	[**11.1**s, 12n, 19i]	**71**
4BS1 → 8BS1	[**79.3**s, 28n, 14i]	[**22.2**s, 18n, 18i]	**71**
5BS → 8BS1	[**79.3**s, 28n, 14i]	[**10.1**s, 14n, 7i]	**87**
6BS → 9BS1	[**184.6**s, 32n, 17i]	[**18.1**s, 17n, 17i]	**90**
10BS → 9BS1	[**184.6**s, 32n, 17i]	[**6.5**s, 5n, 2i]	**96**
4BS → 10BS1	[**401.5**s, 36n, 19i]	[**52.9**s, 30n, 33i]	**86**
8BS → 10BS1	[**401.5**s, 36n, 19i]	[**14.5**s, 12n, 7i]	**96**
3BS → 12BS1	[**1758.6**s, 44n, 23i]	[**77.1**s, 40n, 38i]	**95**
5BS → 12BS1	[**1758.6**s, 44n, 23i]	[**51.8**s, 32n, 26i]	**97**
10BS → 12BS1	[**1758.6**s, 44n, 23i]	[**21.2**s, 13n, 7i]	**98**

column of the table presents the computational savings gained through reuse as compared to from-scratch planning (as a percentage of from-scratch planning time).

The entries in the table show that the overall planning times 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 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%.[17]

We also analyzed the variation in savings accrued by reuse in terms of the similarity between the problems and the size of the constructed plans. The plot in Fig. 12 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" in the figure shows the savings afforded by solving a particular seven-block problem by reusing several different blocks world plans (shown along the x-axis). Figure 13 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 the planning performance with respect to similarity between P^n and P^o, these plots bring out some other interesting characteristics of the PRIAR reuse behavior. As we pointed out earlier, a flexible and conservative modification strategy allows the planner to

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

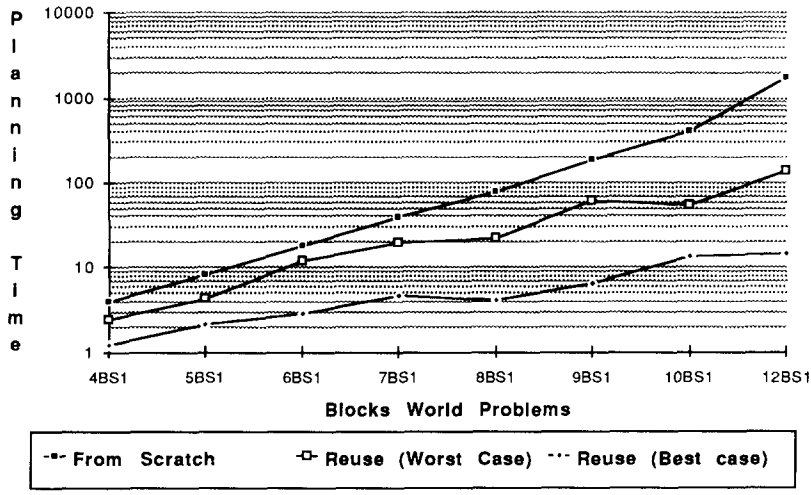

Fig. 12. Variation of performance with problem size and similarity.

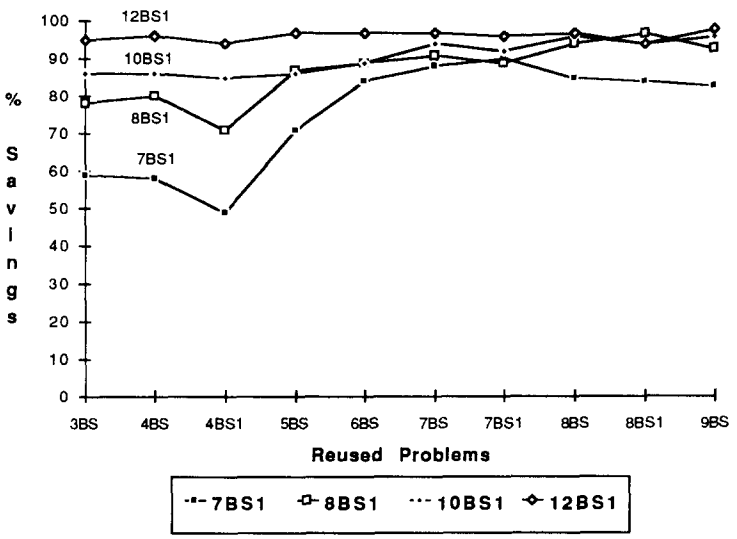

Fig. 13. From-scratch versus best and worst-case reuse performance.

effectively reuse any applicable parts of a partially relevant plan in solving a new planning problem. Because of this, reuse with such a strategy will be able to provide significant performance gains over a wide range of specification changes. The plots in Figs. 12 and 13 show that modification in PRIAR exhibits this property. Consider, for example, the plot for 12BS1 in Fig. 12. As we go from a dissimilar plan $R^{\circ} = 3BS$ to a more similar plan $R^{\circ} = 9BS$, the savings vary between 95% and 98% (corresponding to a variation in the speedup factor of 20 to 50).

Table 2
Variation of reuse performance with problem size

$R^o \rightarrow P^n$	P^n from scratch (cpu seconds)	Reuse R^o (cpu seconds)	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

A related pattern 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 versus 3BS→12BS1 in Fig. 12. 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 shows that as the complexity of the planning problems increases, the ability to solve new planning problems by flexibility modifying existing plans can lead to substantial computational savings.

In Section 8.3, we will provide a qualitative explanation for these empirical performance characteristics in terms of the search process in the space of the plans.

8. Discussion

Before we conclude this paper, we wish to present an assessment of the effectiveness of the modification framework. For example, we would like to characterize its coverage, correctness, and efficiency, in effect of the completeness of the underlying planning strategy, and, finally, discuss some of its limitations.

8.1. Coverage and correctness

Here we are interested in assessing the adequacy of PRIAR's repair actions in conservatively accommodating all possible specification changes in a given plan. As we have seen, PRIAR handles the changes in specifications by computing the ramifications of those changes on the validation structure of the plan, and modifying it to repair any resultant inconsistencies in the validation structure. The inconsistencies themselves are enumerated by characterizing the correctness of the plan in terms of its validation structure (Section 4.1.2). The validation structure of a plan can be shown to constitute an explanation of correctness of that plan with respect to the modal truth criterion [3, 16]. Thus,

the various inconsistencies in the validation structure correspond to the different ways in which a plan can fail to satisfy the modal truth criterion.[18] Since PRIAR provides repair actions for handling each of these types of inconsistencies (Section 5), we claim that it is capable of modifying any plan (describable within its action representation) to accommodate any changes in its specifications.

A related question concerns the expressiveness of the action representation itself. Our action representation (Section 3) largely covers the basic set of actions expressible by the hierarchical, nonlinear planners developed to date, although without modification it does not necessarily cover some of the more complex interactions found in certain planners, such as the resource bounds of SIPE [30] or the temporal windows of DIVISER [29]. Our experience in applying the system to the manufacturing planning domain [14] has shown that extending the modification framework to richer action representations (involving, for example, context-dependent effects, etc.) is in itself not as difficult as making the generative planner handle such actions efficiently and systematically. (In [12] a method for handling context-dependent effects during annotation verification is provided.[19])

Another important issue is the correctness and conservatism of the modification strategies. The modification strategies described here are "correct" in the sense that they do not introduce any new inconsistencies into the validation structure while repairing existing ones. In particular, 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 (to the ancestors of the source of destination nodes), or
(iii) some new validations are added.

We have pointed out in Section 5.2.1 that the repair actions remove only the validations supporting the subreductions of the plan that do not have any externally useful effects (specifically, no e-conditions). Thus, this removal does not introduce any inconsistencies into the validation structure. Next, from Section 3.2, it follows that, if a validation is holding in an HTN, then redirecting it to one of its ancestors does not introduce any new inconsistencies. Finally, new validations are added to the HTN either to re-establish failing validations (as in failing precondition and phantom validations), or to provide missing validations. Any failing precondition validation v: $\langle E, n_1, C, n_d \rangle$ is repaired by

[18] The definition of "failing validations" would however need to be modified slightly to admit plans that are correct with respect to Chapman's "white knight" conditions. In other words, the consistency of validation structure, as defined, is a sufficient, but not a necessary condition for the correctness of plans representable in TWEAK action representation.

[19] Of course, as we allow richer action representations, the complexity of annotation verification may also increase.

adding a new validation, v_r: $\langle E, n_r, C, n_d \rangle$, where n_r is a refit task added to achieve C, such that $n_I < n_r < n_d$ (Section 5.2.3). Since n_r has no expected effects other than C, its addition does not cause violation of any existing validations. Thus, the only possible inconsistency that could be caused by this change is the violation of v_r itself, and we can show that v_r will not be violated.[20] Similarly, when new validations are introduced into HTN to take care of missing validations or p-phantom validations, 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 none of the repair actions introduce new inconsistencies into the HTN.

The modification strategies are "conservative" in the sense that they do not remove any portion of the plan that can be reused in the new situation. In particular, the *Prune-Validation* procedure never removes a subreduction that has a non-empty set of e-conditions. Similarly, the *Repair-Failing-Filter-Condition-Validation* procedure replaces exactly the subreduction that was dependent on the failing filter (unachievable) condition. Finally, as we noted in Section 5.3, conservatism is also fostered by the refitting control strategy.

What all this amounts to is that, given a plan and a new problem specification, PRIAR can return a partially reduced HTN with a consistent validation structure for the new problem, which retains all the applicable portions of the given plan and contains refit tasks to establish any required validations. The planner can treat this HTN as if it were an intermediate point encountered during its search process, and can proceed to reduce it to find a complete plan for the new problem.

8.2. Effect on the completeness of the underlying planner

An important consideration in augmenting a generative planner with the PRIAR modification strategies is the effect of such an augmentation on the completeness of the planner. We will start by addressing the effect on the class of problems that are solvable by the planner. If a given problem P^n is solvable by the planner from scratch, then it is easy to see that the addition of modification strategies does not affect its solvability. This is because whatever may be the plan R^o that PRIAR starts with, in the worst case, the planner can always backtrack over the annotation-verified HTN, R^a, to other parts of the overall search space for solving P^n (in other words, the overall search space during plan modification is the same as the search space during planning from scratch; no portions of the search space are pruned from consideration *a priori*).

[20] For v_r to fail, there should exist a node n such that $\Diamond(n_r < n < n_d)$ and $effects(n) \vdash \neg E$. Since $n_I < n_r < n_d$ (see Section 5.2.3.1), this will also imply that $\Diamond(n_I < n < n_d)$. That is, v itself could not have been established originally. Since v was established previously, by refutation we know that v_r cannot be failing.

A more interesting question is regarding the "quality" of the plan constructed. Note that depending upon the domain theory, there may be many correct plans for solving a problem P^n, and if the planner is solving this problem from scratch, it may find any of those correct plans. The effect of modification strategies (and the refitting control strategy) is to bias the planner's search in such a way that it will find a plan that is structurally closer to the plan R^o that it is asked to modify. There is, however, no guarantee that the plan found by modifying a particular plan R^o will be as preferable as a plan that might have been found by solving P^n from scratch. This is not surprising, as the validation structure and the modification strategies do not capture any information about optimality considerations (such as relative preferences among plans). This is, however, not a serious limitation *vis à vis* hierarchical nonlinear planning, as it (as well as much of the rest of research work on domain-independent planning) concentrates on "satisficing" rather than "optimizing" search strategies.

8.3. Efficiency

The techniques described in this paper provide a methodology by which a planner can improve its performance over time—by reusing existing plans at plan generation time, or by modifying the current plan in response to specification changes during planning or execution. A planner which uses these techniques can do no better than a from-scratch planner if it is provided an inappropriate plan to reuse, or if the changes in the specification are so drastic as to render most of the current plan inapplicable. Thus, the worst-case complexity of planning with modification will still be the same as that of planning from scratch. What we do expect from the use of this incremental modification strategy is an improvement of average-case planning behavior—in particular the ability to handle a variety of specification changes incrementally, reusing any applicable portions of the given plan, while gracefully degrading to from-scratch planning performance as the differences become significant.

It is difficult to characterize average-case improvement formally without knowing the precise distribution of specification changes that can be expected in typical planning situations. Another complication is that while the consistency of the annotation-verified plan R^a allows the planner is 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 would have to satisfy the "downward solution" property [28] (where the existence of an abstract plan implies the existence of specializations of the abstract plan at each lower level). However, guaranteeing this property for a

domain formalization, while still retaining an effective abstraction hierarchy, may turn out to be infeasible in practice for most domains [28, 32].

Although formal characterization of average-case improvement is difficult, we can get a practical understanding of the effectiveness of our incremental strategy by looking at its coverage, the overhead involved in facilitating it, and finally the empirical performance gains afforded by the incremental strategy. We have already discussed the coverage of PRIAR's plan modification strategies (Section 8.1). In terms of the overhead incurred in facilitating incremental modification, we have shown that the dependency structures can be annotated efficiently as a by-product of planning, and that the storage requirements for maintaining the dependency structures is not significant (Section 4.3). Furthermore, generating the annotation-verified plan is inexpensive since all of the repair procedures can be run in polynomial time (Section 5.2.6). Considering the exponential complexity of from-scratch planning and the performance gains promised by the incremental modification framework, the overhead involved in augmenting the generative planner with an incremental modification capability seems eminently justifiable.

Finally, in terms of performance, by starting with a partially reduced plan containing all the applicable parts of an existing plan, and conservatively controlling the search such that the already-reduced (applicable) parts of R^a are left undisturbed, PRIAR attempts to minimize the repetition of planning effort in response to specification changes. We can develop a qualitative understanding of the effect of this on the search process in the space of plans. If β is the effective branching factor of the search space,[21] Δ is the operator distance between the problem specification P^n (which itself can be seen as an abstract plan) and the plan R^n, and Δ' 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 $O(\beta^{\Delta-\Delta'})$ when $\Delta \leq \Delta'$ [12, 19]. Since annotation verification tries to retain all parts of R^o that are applicable in P^n, we typically have $\Delta \leq \Delta'$. Thus, any similarity between P^n and $[P^o, R^o]$ can lead to a possibly exponential reduction in explored search space.

Of course, this expected exponential reduction may not always be realized for two reasons (see the discussion in Section 8.2):

(i) If the domain reduction schemas do not have the downward solution property (as is typically the case), the interactions between the reductions of the refit tasks and the rest of R^a can, in theory, force the planner to backtrack over R^a and explore the whole search space.

[21] Note that here β corresponds to both the choice in reducing tasks and the choice in resolving interactions. In terms of Chapman's model of nonlinear planning [3], this can be best understood as the average number of ways of interpreting the modal truth criterion to achieve a goal for that particular problem and domain.

(ii) If the domain is such that there are multiple correct solutions to a given planning problem, then left to itself, the planner might find a plan $R^{n'}$ that is closer to P^n, in terms of operator distance, than R^n.

In both these cases, the cost of solving the new problem through incremental modification can exceed that of solving it from scratch.

However, as the results described in Section 7 show, the modification strategies do, on the average, afford a significant reduction in the explored search space, resulting in high performance gains for a variety of specification changes. Furthermore, as problem size increases, the effective branching factor of the search space also increases. (One way of understanding this is that as the size of the planning problem increases, the number of ways of interpreting the modal truth criterion to achieve a goal also increases.) As β increases, so will the relative reduction in the search space. Thus, the savings afforded by reuse tend to become more significant with increase in problem size (as demonstrated by the entries in Table 2).

8.4. *Limitations, extensions and future work*

Viewed as an integrated theory of learning to improve planning performance from experience, PRIAR has some obvious shortcomings. In particular, such a theory would have to account for the issues of storage and organization of generated plans—i.e., *when* it is worth storing a generated plan for future rescue and *how* best to organize stored plans in memory for efficient retrieval in the future. Indiscriminate storage of plans can degrade a planner's performance by making it spend an inordinate amount of time in the retrieval phase. (Recent work in machine learning (e.g. [20]) has amply demonstrated the practical importance of this utility problem.)

While PRIAR provides an integral building block for any scheme to integrate planning and learning—viz., a framework for incrementally modifying existing plans to solve new problems—it does not directly address the issues of storage and organization. We believe, however, that the flexibility of the PRIAR modification framework may facilitate significantly simpler solutions to these problems. For example, typically, much of the complexity of retrieval is due to the insistence on best-match retrieval. By being able to flexibly reuse *any* partially applicable portions of a retrieved plan, PRIAR can mitigate the criticality of best-match retrieval. Similarly, the PRIAR modification framework, based as it is on a systematic characterization of the explanation of correctness of the plan, is amenable to seamless integration with explanation-based generalization techniques. In particular, we have recently developed provably sound algorithms for generalizing partially ordered plans based on the validation structure representations discussed in this paper [16]. Such strategies constitute a first step towards addressing the storage issues, as they allow PRIAR to avoid storing two instances of the same general plan.

Next, while PRIAR's strategy of handling specification changes ensures correctness of the modified plan with respect to the planner, it cannot guarantee the actual executability of the plan in the real world, any more than a from-scratch planner itself can. In particular, failures arising from the incompleteness of the planner's domain model cannot be rectified in this framework. To handle such failures, the planning strategy would still have to be complemented by simulation and debugging strategies such as those used in GORDIUS [23] and CHEF [8]. However, as we remarked in Section 2, the PRIAR framework *can* be gainfully integrated with these debugging strategies, as it increases the likelihood of generated plans being correct.

A related issue is that sometimes it is not enough to ensure correctness of the modified plan with respect to the planner. For example, in many domains, planning is best characterized as a hybrid activity involving interaction between a general purpose planner and a set of specialized reasoners. The ability to incrementally modify plans can play a very important role in such hybrid planning architectures as it can provide substantial computational advantages both by avoiding repetition of computational effort and by respecting previous commitments. However, here, we are no longer concerned solely with the internal consistency of the revised plan, but with the global consistency—both the planner and the specialists must be satisfied with the current state of the overall plan. In particular, to avoid costly ripple effects, the planner must keep track of any implicit constraints imposed by the specialists, through appropriate interfaces, and respect them during any plan revision. We are exploring these issues in our ongoing work in hybrid planning architectures [15].

Finally, a minor limitation of the current implementation is the relation between the planner's search control strategy and the modification framework. In the current implementation, PRIAR's NONLIN-based planner uses a chronological backtracking regime to explore its search space during planning. As has been noted in the literature [26], chronological backtracking is a very inefficient control strategy in hierarchical nonlinear planning. The plan modification framework we describe can be seen as introducing a dependency-directed component into the backtracking. Given a set of inconsistencies in a validation structure, the modification strategy essentially allows the planner to restart with the part of the plan that has a consistent validation structure. However, in the current system, validation structures are not used for that purpose. A logical extension to PRIAR would be to augment the planner such that the modification component would be invoked whenever the planner has to do backtracking. This can be easily accomplished by making the planner utilize the incremental annotation algorithms to maintain the internal dependency structures of the plan throughout planning. Every time the planner reaches a backtracking point, instead of using chronological backtracking, it can use PRIAR's modification strategies to remove the inconsistencies in the validation structure that led to the failure and then resume planning.

9. Conclusion

In this paper, we have presented a formal basis for flexible and conservative modification of existing plans to handle changes in problem specifications. This ability can provide substantial computational advantages by respecting previous commitments and avoiding repetition of planning effort. 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. We have discussed the development of a planning system based on this theory, PRIAR, and characterized its coverage, efficiency and limitations. In addition, we have described empirical experimentation showing significant performance gains using this system. We have also shown that this theory provides a general unified framework for addressing several subproblems of planning and plan reuse, particularly execution monitoring and replanning, estimating the similarity of plans during the retrieval of plans for reuse, and controlling the choice of new actions to replace changed parts of existing plans.

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 Fig. 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 failing filter condition validations, unnecessary validations, and p-phantom validations (the capabilities that were not brought out in the example 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 1–11). The mapping and retrieval strategy (Section 6.2) prefers one of those candidates

$$\langle 6\text{BS}, \alpha = [A \rightarrow L, C \rightarrow O, B \rightarrow P, D \rightarrow M, E \rightarrow N] \rangle$$

as better suited for solving the 5BP problem (lines 13–19).

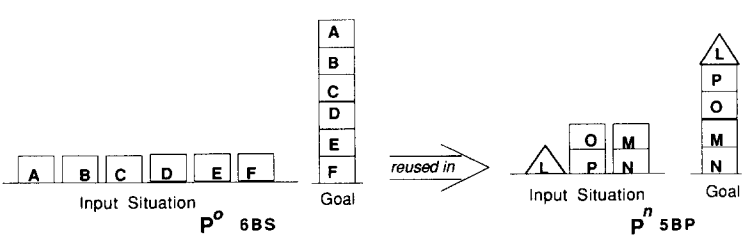

Fig. A.1. The 6BS→5BP modification problem.

```
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 O) (ON M N) (ON L P) (ON O M))
5    :INPUT ((BLOCK P) (CLEARTOP O) (ON O P) (ON L TABLE)
6              (ON P TABLE) (BLOCK O) (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⟩} (L A) (N E) (M D) (O C) (P B)))
11  ({⟨Plan::6BS⟩} ((L B) (N F) (M E) (O D) (P C))))
12                                                          . . .
13 *******PLAN-KERNEL-BASED-ORDERING
14 The Plan Choices 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⟩} ((L A) (N E) (M D) (O 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)
32   [Prenodes:(21 22)] [Succnodes: (6 1)]
33 6: :PRIMITIVE (PUT-BLOCK-ON-BLOCK-ACTION D E)
34   [Prenodes:(18 19 7)] [Succnodes: (5 1)]
35 5: :PRIMITIVE (PUT-BLOCK-ON-BLOCK-ACTION C D)
36   [Prenodes:(15 16 6) [Succnodes: (4 1)]
37 4: :PRIMITIVE (PUT-BLOCK-ON-BLOCK-ACTION B C)
38   [Prenodes:(12 13 5)] [Succnodes: (3 1)]
39 3: :PRIMITIVE (PUT-BLOCK-ON-BLOCK-ACTION A B)
40   [Prenodes:(9 10 4)] [Succnodes: (1)]
41 **********

42       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 the 5BP problem (line 58).

Figure A.2 shows the HTN of the 6BS plan after the interpretation process.

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

Next, PRIAR starts the annotation verification process; Fig. A.3 shows the HTN after this process is complete. During the annotation verification process, PRIAR first considers the unnecessary validation supporting the *unnecessary* goal $On(N, F)$ (lines 60–66). 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 subreduction below the intermediate level node ND0110: $On(N, F)$ (the node with label "I" in Fig. A.2) will have to be removed from the plan to take care of this unnecessary validation. Consequently, the annotation-verified plan, shown in Fig. A.3, does not contain any nodes of this subreduction.

```
60 **********Annotation Verification**********
61 ANNOT-VERIFY: Start

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

Next, 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 subreduction below the node for achieving $On(M, N)$ (see the node with the label "II" in Fig. A.2) from the interpreted plan. Consequently, the annotation-verified plan, shown in Fig. A.3, does not contain any nodes of this subreduction.

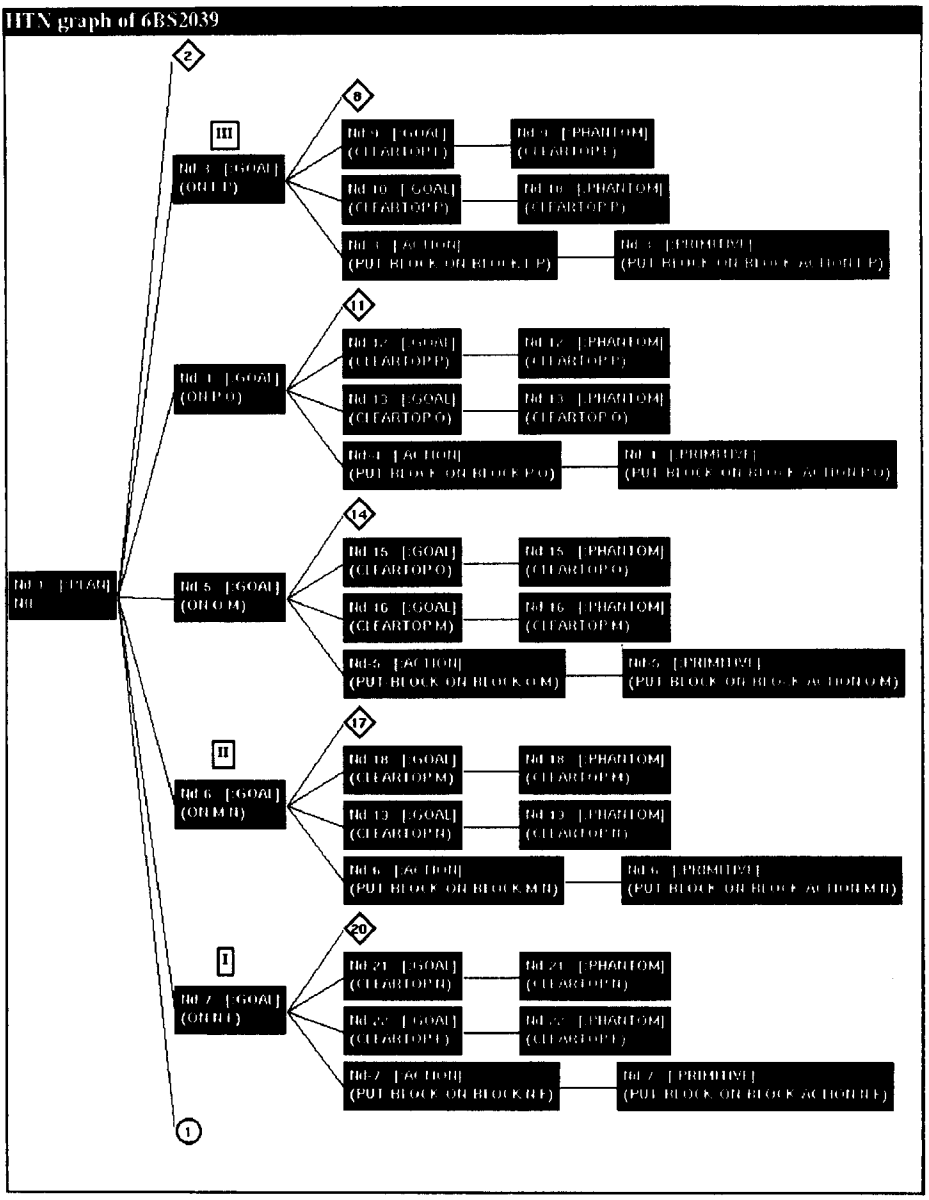

Fig. A.2. 6BS plan after interpretation.

```
67  ANNOT-VERIFY: Processing p-phantom validations (if any)
68      The goal (ON M N) is supported by a p-phantom validation
69      Checking to see if it can be phantomized
70    Check-p-Phantom-Validation: the condition (ON M N)
71      can be established from new initial state!!!
72    Check-p-Phantom-Validation: Pruning the other contributor
73      {⟨6::ND0268⟩[:PRIMITIVE(PUT-BLOCK-ON-BLOCK-ACTION M N)]...}
74       from the HTN
```

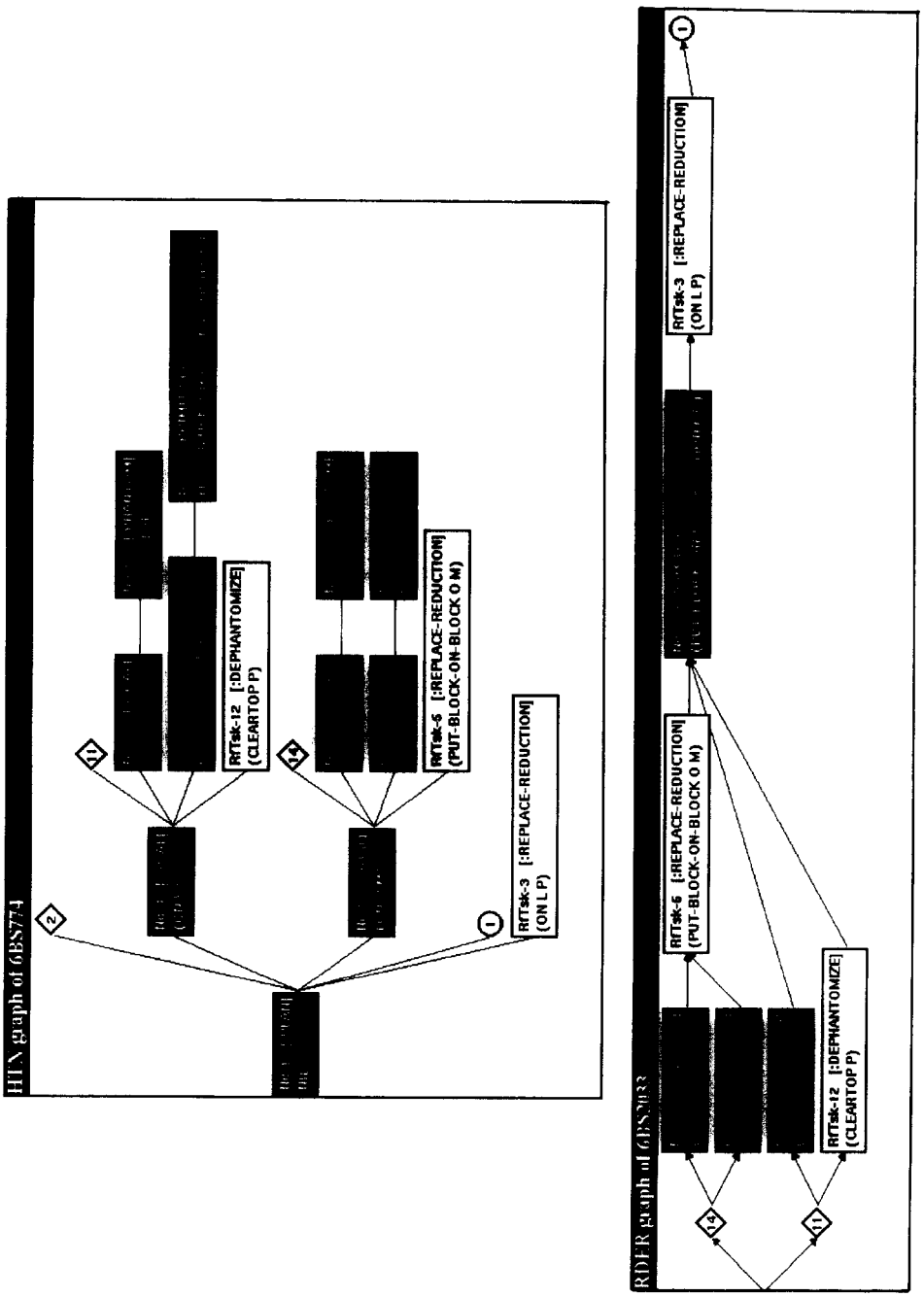

Fig. A.3. 6BS plan after annotation verification.

```
75      Pruning the reduction below the node
76    {⟨6::ND0109⟩[:GOAL(ON M N)]...}
77      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 subreduction below the node which first posted that filter condition. In this case, PRIAR finds that the node for achieving the goal *On*(L, P) (see the node with the label "III" in Fig. A.3) which is an ancestor of the node with the failing condition validation, first posted the filter condition *Block*(L) into the plan. So it decides to replace the subreduction below this node. Consequently, the annotation verified plan in Fig. A.3 contains a refit task to re-achieve the goal *On*(L, P) in place of the replaced subreduction.

```
78  ANNOT-VERIFY: Processing extra goals (if any)
79  ANNOT-VERIFY: Looking for failed validations.
80    The FILTER (:use-when) condition (BLOCK L) at node
81        {⟨3::ND0232⟩[:PRIMITIVE(PUT-BLOCK-ON-BLOCK-ACTION L P)]...}
82        is failing because of :out fact (BLOCK L) in ⟨INIT-STATE⟩
83
84      REFIT-FILTER-COND-FAILURE: Adding a refit-task
85          {⟨REFIT-TASK004⟩[:REPLACE-REDUCTION(ON L P)]...}
86          to re-reduce the node
87          {⟨3::ND0106⟩[:GOAL(ON L P)]...}
88      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 89–106). It repairs them by adding a second replace reduction refit task and a de-phantomize refit task to the annotation-verified plan. Figure A.3 shows the partially reduced HTN after the annotation verification process. The top part of the figure shows the hierarchical structure of the task reduction while the bottom part shows the chronological partial ordering relations among the leaf nodes of the HTN. The black nodes correspond to the parts of the interpreted plan while the white nodes represent the refit tasks added during the annotation verification process. This partially reduced HTN is then sent to the planner for refitting.

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

92    REFIT-FILTER-COND-FAILURE: Adding a refit-task
93        {⟨REFIT-TASK0002⟩[:REPLACE-REDUCTION(PUT-BLOCK-ON-
                                                BLOCK O M)]...}
94        to re-reduce the node
95      {⟨5::ND0176⟩[:ACTION(PUT-BLOCK-ON-BLOCK O M)...}
96    REFIT-FILTER-COND-FAILURE: Removing the replaced reduction from the plan
```

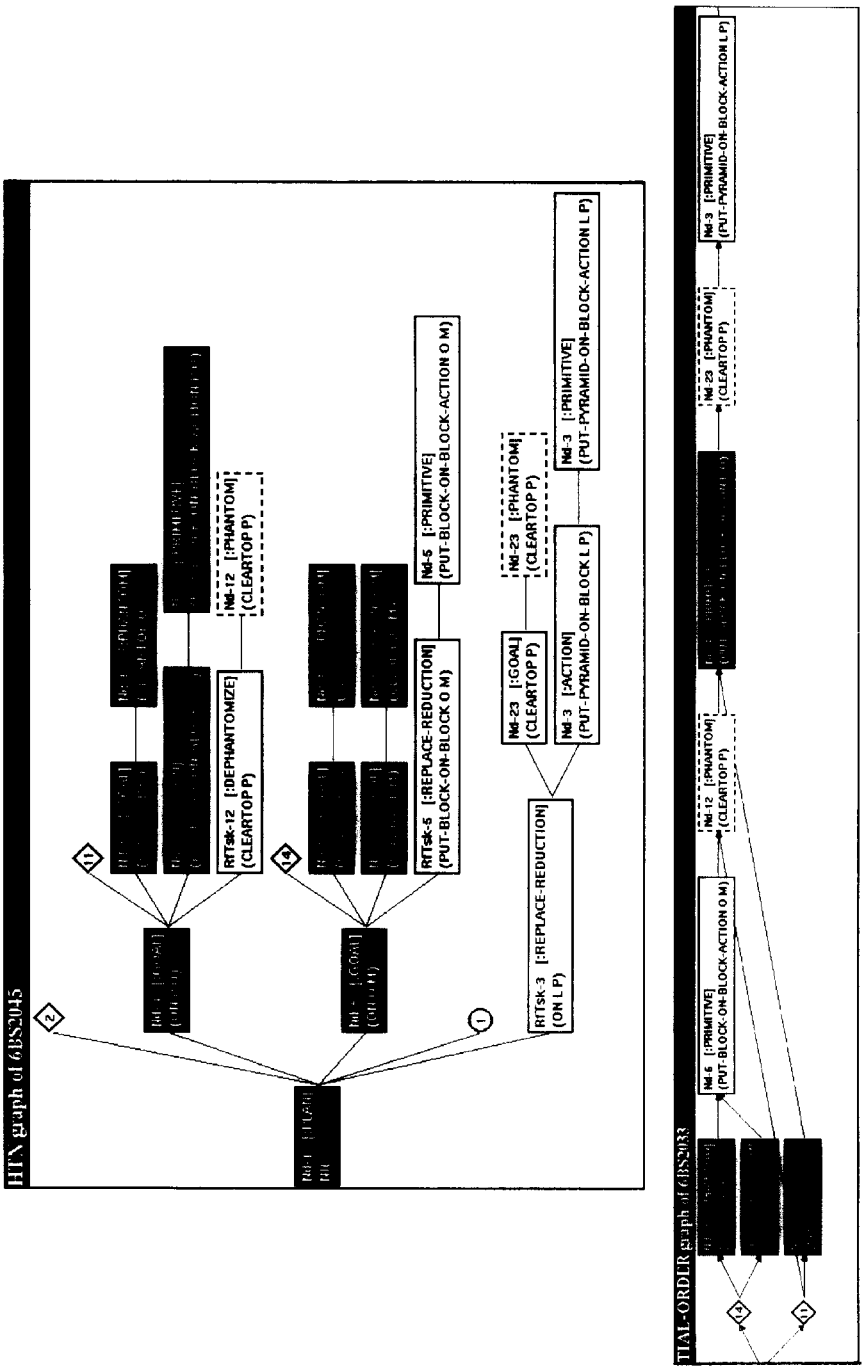

Fig. A.4. Result of refitting 6BS plan to 5BP problem.

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

107 Entering {⟨REFIT-TASK0004⟩[:REPLACE-REDUCTION(ON L P)]...}
108 Entering {⟨REFIT-TASK0002⟩[:REPLACE-REDUCTION(PUT-BLOCK-ON-BLOCK
                                                              O M)]...}
109 Entering {⟨REFIT-TASK0006⟩[:DEPHANTOMIZE(CLEARTOP P)]...}
110 ANNOT-VERIFY: END
```

The planner starts by reducing the replace-reduction refit task corresponding to $On(L, P)$ (lines 112–130). Since L is a pyramid, the planner finds that the only appropriate schema instance for reducing this refit task is MAKE-PYRAMID-ON-BLOCK(L, P). Next, since the refit task is a replace-reduction refit task, PRIAR finds during installation (Section 5.3) 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 136–142).

Figure A.4 shows the result of refitting, which is a completely reduced HTN for solving the 5BP problem. The black nodes represent the parts of the 6BS plan that remained applicable to the 5BP problem and the white nodes represent the reductions of refit tasks. There is no separate subplan for achieving the goal $On(M, N)$ in this HTN since this is made true from the initial state 5BP problem. The bottom part of the figure shows the partial ordering relations among the steps of the developed plan.

```
111 **********Calling Generative Planner**********

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

113 PLANNER: The schema choices to reduce the refit task are:
114 ({SCH0021}: MAKE-PYRAMID-ON-BLOCK00140018::(ON L P)
115   BY {⟨1::ND0020⟩[:ACTION(PUT-PYRAMID-ON-BLOCK L P)]...}
116 The chosen schema is:
117 {SCH0021}
118   MAKE-PYRAMID-ON-BLOCK00140018::(ON L P)
119     Expansion:
120       0 {⟨0::ND0019⟩[:GOAL(CLEARTOP P)]}
121       1 {⟨1::ND0020⟩[:ACTION(PUT-PYRAMID-ON-BLOCK L P)]}
122     Conditions:
123       《SC5125》 :PRECOND (CLEARTOP P) :at 1 :from (0)
```

```
124        《SC5126》 :USE-WHEN (PYRAMID L) :at 0 :from (-24)
125        《SC5127》 :USE-WHEN (BLOCK P) :at 1 :from (-24)
126                                                    . . .

127  Install Choice: Installing the schema ({SCH0021}
128    MAKE-PYRAMID-ON-BLOCK00140018::(ON L P) BY
129      {⟨1::ND0020⟩[:ACTION(PUT-PYRAMID-ON-BLOCK L P)] . . .}
130    to Re-reduce the task ({⟨REFIT TASK0004⟩[:REPLACE-REDUCTION(ON L P)]})

131    The c-precondition (CLEARTOP L) of the task
132      ({⟨REFIT-TASK0004⟩[:REPLACE-REDUCTION(ON L P)]})
133      is not required by the chosen schema
134
135    So, pruning the validation corresponding to this ncondition
136  PLANNER: Expanding Refit-task Achieve [(PUT-BLOCK-ON-BLOCK O M)]
137  PLANNER: The schema choices to reduce the refit-task are:
138  ({SCH0027}::PUT-BLOCK-ON-BLOCK00220025::(PUT-BLOCK-ON-BLOCK O M)
139    BY {⟨0::ND0026⟩[:PRIMITIVE(PUT-BLOCK-ON-BLOCK-ACTION O M)]})

140  PLANNER: Expanding Refit-task Achieve [(CLEARTOP P)]
141    The refit-task is PHANTOMIZED with an effect of the node(s)
142    ({⟨5::ND0026⟩[:PRIMITIVE(PUT-BLOCK-ON-ACTION O M)]})

143  ****The planning is OVER
144  The plan is . . .

145  ******************************
146
147  5: :PRIMITIVE (PUT-BLOCK-ON-BLOCK-ACTION O M)
148    [Prenodes:(6 15 16)] [Succnodes: (3 1 4)]
149  4: :PRIMITIVE (PUT-BLOCK-ACTION P O)
150    [Prenodes:(12 5 13)] [Succnodes: (23 1)]
151  3: :PRIMITIVE (PUT-PYRAMID-ON-BLOCK-ACTION L P)
152    [Prenodes:(23 0 5)] [Succnodes: (1)]

153  **********GOAL STATE**********
154    《SC0062》 :PRECOND (ON O M) :at 1 :from (5)
155    《SC0061》 :PRECOND (ON P O) :at 1 :from (4)
156    《SC0060》 :PRECOND (ON L P) :at 1 :from (3)
157    《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 step1)
                (: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 step1))
  :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 step1)
                (:filter (block ?y) :at step 2))
  :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 (cleartop ?x))
                (step2 :action (put-block-on-table ?x table)))
  :conditions  ((:filter (block ?x) :at step 1))
  :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 step 1)
                (:filter (block ?y) :at step2)
                (:filter (on ?y ?x) :at step2))
  :effects     ((step2 :assert (cleartop ?x))
                (step2 :assert (on ?y table)))
  :variables   (?x ?y))
```

```
(opschema  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 step 1)
                (:filter (block ?y) :at step1)
                (:filter (block ?z) :at step1)
                (:filter (on ?y ?x) :at step2)
                (:filter (cleartop ?z) :at step2)
                (:filter (not (equal ?z ?y)) :at step1)
                (:filter (not (equal ?x ?z)) :at step1))
  :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 step1)
                (:filter (block ?y) :at step 1)
                (:filter (cleartop ?x) :at step1)
                (:filter (cleartop ?y) :at step1)
                (:filter (on ?x ?z) :at step1))
  :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 step1)
                (:filter (block ?y) :at step1)
                (:filter (cleartop ?y) :at step1)
                (:filter (on ?x ?z) :at step1))
  :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 step1)
                (:filter (cleartop ?x) :at step1)
                (:filter (on ?x ?z) :at step1))
  :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 step1)
                (:filter (on ?x ?z) :at step1))
  :effects     ((step1 :assert (on ?x table))
                (step1 :assert (cleartop ?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
```

Acknowledgement

Bulk of this research was done while the first author was a graduate research assistant at the Center for Automation Research, University of Maryland, College Park. Lindley Darden and Larry Davis have significantly influenced the early development of this work. Mark Drummond, Amy Lansky, Jack Mostow, Austin Tate, David Wilkins, and the reviewers of IJCAI-89, AAAI-90 have provided several helpful comments on previous drafts. The paper also benefited from the extensive comments from one of the AI Journal referees.

Dr. Kambhampati has been supported in part by 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), the 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". Dr. Hendler is also affiliated with the UM Systems Research Center (an NSF supported engineering research center) and the UM Institute for Advanced Studies, and support for this research comes from ONR grant N00014-88-K-0560 and NSF grant IRI-8907890.

References

[1] R. Alterman, Adaptive planning. *Cogn. Sci.* **12** (1988) 393–421.
[2] J.G. Carbonell, Derivational analogy and its role in problem solving, in: *Proceedings AAAI-83*, Washington, DC (1983) 64–69.
[3] D. Chapman, Planning for conjunctive goals, *Artif. Intell.* **32** (1987) 333–377.
[4] E. Charniak and D. McDermott, Managing plans of actions, in: *Introduction to Artificial Intelligence* (Addison-Wesley, Reading, MA, 1984) Chapter 9, 485–554.
[5] L. Daniel, Planning: modifying non-linear plans, DAI Working Paper 24, University of Edinburgh, Edinburgh, Scotland (1977); also as: Planning and operations research, in: *Artificial Intelligence: Tools, Techniques and Applications* (Harper and Row, New York, 1983).
[6] R.E. Fikes, P.E. Hart and N.J. Nillsson, Learning and executing generalized robot plans, *Artif. Intell.* **3** (1972) 251–288.
[7] S. Ghosh, S. Kambhampati and J.A. Hendler, Common Lisp implementation of a NONLIN-based hierarchical planner: a user manual, Tech. Report (in preparation).
[8] K.J. Hammond, Explaining and repairing plans that fail, *Artif. Intell.* **45** (1990) 173–228.
[9] P.J. Hayes, A representation for robot plans, in: *Proceedings IJCAI-75*, Tblisi, Georgia (1975).
[10] J.A. Hendler, A. Tate and M. Drummond, AI planning: systems and techniques, *AI Mag.* **11** (2) (1990).
[11] M.N. Huhns and R.D. Acosta, ARGO: a system for design by analogy, *IEEE Expert* (Fall 1988) 53–68; also in: *Proceedings of 4th IEEE Conference on Applications of AI* (1988).
[12] S. Kambhampati, Flexible reuse and modification in hierarchical planning: a validation structure based approach, Ph.D. Dissertation, CS-Tech. Report 2334, CAR-Tech. Report

469, Center for Automation Research, Department of Computer Science, University of Maryland, College Park, MD (1989).
[13] S. Kambhampati, Mapping and retrieval during plan reuse: a validation-structure based approach, in: *Proceedings AAAI-90*, Boston, MA (1990) 170–175.
[14] S. Kambhampati and M.R. Cutkosky, An approach toward incremental an interactive planning for concurrent product and process design, in: *Proceedings ASME Winter Annual Meeting on Computer Based Approaches to Concurrent Engineering*, Dallas, TX (1990).
[15] S. Kambhampati, M.R. Cutkosky, J.M. Tenenbaum and S.H. Lee, Combining specialized reasoners and general purpose planners: a case study, in: *Proceedings AAAI-91*, Anaheim, CA (1991) 199–205.
[16] S. Kambhampati and S. Kedar, Explanation based generalization of partially ordered plans, in: *Proceedings AAAI-91*, Annaheim, CA (1991) 679–685.
[17] S. Kambhampati and J.A. Hendler, Control of refitting during plan reuse, in: *Proceedings IJCAI-89*, Detroit, MI (1989) 943–948.
[18] J.L. Kolodner, Maintaining organization in a dynamic long-term memory, *Cogn. Sci.* **7** (1983) 243–280.
[19] R.E. Korf, Planning as search: a quantitative approach, *Artif. Intell.* **33** (1987) 65–88.
[20] S. Minton, Quantitative results concerning the utility of explanation-bassed learning, *Artif. Intell.* **42** (1990) 363–391.
[21] S. Minton, A.B. Philips, P. Laird and M.D. Johnston, Solving large-scale constraint-satisfaction and scheduling problems using a heuristic repair method, in: *Proceedings AAAI-90*, Boston, MA (1990) 17–24.
[22] E.D. Sacerdoti, *A Structure for Plans and Behavior* (Elsevier North-Holland, New York, 1977).
[23] R. Simmons, A theory of debugging plans and interpretations, in: *Proceedings AAAI-88*, St. Paul, MN (1988) 94–99.
[24] R. Simmons and R. Davis, Generate, test and debug: combining associational rules and causal models, in: *Proceedings IJCAI-87*, Milan, Italy (1987) 1071–1078.
[25] G.J. Sussman, HACKER: a computational model of skill acquisition, Memo 297, AI Lab, MIT, Cambridge, MA (1973).
[26] A. Tate, Project planning using a hierarchic non-linear planner, Research Report 25, Department of AI, University of Edinburgh, Edinburgh, Scotland (1976).
[27] A. Tate, Generating project networks, in: *Proceedings IJCAI-77*, Cambridge, MA (1977) 888–893.
[28] J. Tenenberg, Abstraction in planning, Doctoral Dissertation, Tech. Report 250, Rochester University, Rochester, NY (1988).
[29] S.A. Vere, Planning in time: windows and durations for activities and goals, *IEEE Trans. Pattern Anal. Mach. Intell.* **5** (3) (1983) 246–247.
[30] D.E. Wilkins, Domain-independent planning: representation and plan generation, *Artif. Intell.* **22** (1984) 269–301.
[31] D.E. Wilkins, Recovering from execution errors in SIPE, *Comput. Intell.* **1** (1985).
[32] D. Wilkins, *Practical Planning* (Morgan Kaufmann, San Mateo, CA, 1989).
[33] Q. Yang, Improving the efficiency of planning, Doctoral Dissertation, Department of Computer Science, University of Maryland, College Park, MD (1989).

Note added in proof

After the paper has gone to the printers, we have noticed a minor error in our formulation of validation-structure-based execution monitoring in Section 6.1.1. The definition of $E(P, W, \text{EXEC})$ has to be modified to account for the fact that parallel steps can be executed in any order. The correct formulation would be

$$E(P, W, \text{EXEC}) = \{n \mid primitive(n) \wedge matches(A^{p}(n), W)\}$$

where $matches(A^{p}(n), W)$ is true as long as both the following clauses are satisfied:

(1) for all the validations v: $\langle E, n_s, C, n_d \rangle \in A^{p}(n)$ such that $\Box(n_s < n < n_d)$, $holds(v, W)$ is true;
(2) for each node n_p such that $n_p \| n$,
either $holds(v, W)$ is true for every validation v: $\langle E, n_s, C, n_d \rangle \in A^{p}(n)$ such that $n_s = n_p$, and $\Box(n < n_d)$.
or $holds(v, W)$ is true for every validation v: $\langle E, n_s, C, n_d \rangle \in A^{p}(n)$ such that $n_d = n_p$, and $\Box\ (n_s < n)$;
(where $holds(v: \langle E, n_s, C, n_d \rangle, W)$ is true if and only if $W \vdash C$).

The second clause captures the notion that n can be executed if for every action n_p that is parallel to n:

- either n_p has already been executed successfully (in which case each of its e-conditions supporting the applicability conditions of any of the successors of n must hold in W),
- or n_p can still be executed successfully after n, without re-executing any step that is necessarily before n in the plan (and thus logically could already have been executed).

(When the plan P is totally ordered, the second clause will not be applicable, as no two nodes in the plan will be unordered, and the formulation reduces to that of STRIPS/PLANEX triangle-table-based execution monitoring framework [6].)

Intuitively, $matches(A^{p}(n), W)$ tries to capture the notion that all the validation links in at least one *cutset* of the plan graph that separates n from all its predecessor nodes, must hold in the current world state.

Note that this formulation of $E(P, W, \text{EXEC})$ allows for the possibility that the parallel nodes that have already been executed may have to be re-executed, if their intended effects are not holding. It can be used to define a simple nondeterministic automaton that represents all the possible behaviors (sequences of world states) that one can get out of the partially ordered plan P. Given a world state W, this automaton nondeterministically selects and executes an action from the set of executable actions, given by $E(P, W, \text{EXEC})$. The automaton terminates with failure when the set $E(P, W, \text{EXEC})$ is empty, and with success when this set contains the goal node n_G.