A Critical Look at Critics in HTN Planning*

Kutluhan Erol kutluhan@cs.umd.edu
James Hendler hendler@cs.umd.edu
Dana S. Nau nau@cs.umd.edu
Reiko Tsuneto reiko@cs.umd.edu

Computer Science Department,
Institute for Systems Research, and
Institute for Advanced Computer Studies
University of Maryland, College Park, MD 20742

Abstract

Detecting interactions and resolving conflicts is one of the key issues for generative planning systems. Hierarchical Task Network (HTN) planning systems use *critics* for this purpose. Critics have provided extra efficiency and flexibility to HTN planning systems, but their procedural –and sometimes domain-specific – nature has not been amenable to analytical studies. As a result, little work is available on the correctness or efficiency of critics. This paper describes a principled approach to handling conflicts, as implemented in UMCP[1], an HTN planning system. Critics in UMCP have desirable properties such as systematicity, and the preservation of soundness and completeness.

1 Introduction

Detecting interactions and resolving conflicts is one of the key issues for planning systems. The importance of this issue was realized as long ago as the 1970s in early AI planning systems such as STRIPS [Fikes and Nilsson, 1971] and HACKER [Sussman, 1990]. The introduction of task networks and task decomposition in NOAH [Sacerdoti, 1977] provided an even richer set of interactions and resolution methods, and a component of NOAH called the *critic mechanism* was designed for handling these interactions. Critics helped prune the search space by detecting dead ends in advance and by resolving many types of conflicts as soon as they appeared. Critics could also draw upon domain-specific information to do their job more efficiently. The power of the critic mechanism was quickly realized and adopted by hierarchical task network (HTN) planning systems [Tate, 1977; Vere, 1983; Wilkins, 1988].

*This work was supported in part by NSF Grants DDM-9201779, IRI-9306580 and NSF EEC 94-02384, AFOSR (F49620-93-1-0065), the ARPA/Rome Laboratory Planning Initiative (F30602-93-C-0039), and ONR grant N00014-91-J-1451. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or ONR.

[1]Universal Method-Composition Planner

Some of the critics identified by Sacerdoti [1977] (based in part on Sussman's [1990] earlier work) include:

- *Resolve Conflicts.* The conflicts handled by this critic, later referred to as "deleted-condition" interactions, have received the bulk of the attention in the literature.
- *Eliminate redundant preconditions.* This critic both handled "phantom" conditions and found cases where two different procedural networks added the same primitive prior to usage.

In addition to the interactions handled by these critics, several other situations that can arise in planning have been identified in the literature:

- For his DEVISER system, Vere [1983] has discussed temporal interactions between the times at which actions must occur.[2] He has used temporal windowing and performed an analysis thereof to eliminate possible reductions.
- Wilkins' SIPE system [Wilkins, 1988] has added several different mechanisms for recognizing resource interactions and for allowing user preferences to be considered when making a choice among reductions.
- Yang, Nau, and Hendler [Yang et al., 1993] have discussed a general "action-precedence" interaction that, while less general than deleted-condition interactions, can be exploited in some planning situations. They also have discussed a "simultaneous action" interaction that arises in some domains.
- To handle iteration in plans, Drummond [1985] has proposed several extensions to the procedural net, and an extension to Sacerdoti's Resolve-conflicts critic.
- NONLIN [Tate, 1977] and O-Plan2 [Tate et al., 1994] provide various condition types which can be used to reduce the search space. In O-Plan2, Constraint Managers support decision making of the planner by providing complete information about the constraints they are managing.
- A number of special-purpose "domain dependent" planning systems have identified interactions occurring only in the particular domain for which the sys-

[2]See also [Dean, 1983].

tem is being developed. Typically special-purpose heuristics are introduced to exploit this knowledge.

As can be seen, the many interactions which need to be handled during planning go beyond the (relatively) well-understood deleted-condition interaction. To handle these interactions, implemented planning systems usually use critics or similar mechanisms. Unfortunately it is difficult for a user to exploit these planning systems effectively (i.e. reasonably efficiently and correctly) without an in-depth understanding of the implementation details of the critic mechanisms. To reason about analytical properties of such mechanisms (i.e. systematicity, soundness, completeness), a general model of interactions and critics is clearly needed.

The work described in [Erol et al., 1994a; 1994b] presents a formal model for HTN planning, which provides a constraint-based representation for interactions among tasks and enables principled approaches to conflict detection and handling in HTN planning. This paper presents conflict management and constraint handling techniques based on that framework. Among the properties of these techniques are soundness, completeness and systematicity. These techniques have been implemented in UMCP, an HTN planning system.

2 An Overview of HTN planning

Here is a brief informal description of HTN planning. For a precise formal description, see [Erol et al., 1994a; 1994b].

HTN planning representations for actions and states of the world are similar to those used in STRIPS-style planning.[3] Each state of the world is represented by the set of atoms true in that state. Actions, which in HTN planning are usually called *primitive tasks*, correspond to state transitions; i.e., each action is a partial mapping from the set of states to the set of states.

The primary difference between HTN planners and STRIPS-style planners is in what they plan for, and how they plan for it. In STRIPS-style planning, the objective is to find a sequence of actions that will bring the world to a state that satisfies certain conditions or "attainment goals." Planning proceeds by finding operators that have the desired effects and by making the preconditions of those operators into subgoals. In contrast, HTN planners search for plans that accomplish *task networks*, which can include things other than just attainment goals; and they plan via task decomposition and conflict resolution, which shall be explained shortly.

A task network is a collection of tasks that need to be carried out, together with constraints on the order in which tasks can be performed, the way variables are instantiated, and what literals must be true before or after each task is performed. Unlike STRIPS-style planning, the constraints may or may not contain conditions on what must be true in the final state. Formally, a *task network* is a syntactic construct of the form $[(n_1 : \alpha_1) \ldots (n_m : \alpha_m), \phi]$, where

- each α_i is a task;
- n_i is a label for α_i (to distinguish it from any other occurrences of α_i in the network);
- ϕ is a boolean formula constructed from variable binding constraints such as $(v = v')$ and $(v = c)$, ordering constraints such as $(n \prec n')$, and state constraints such as (initially l), (n, l), (l, n), and (n, l, n'), where v, v' are variables, l is a literal, c is a constant, and n, n' are node labels.[4] Intuitively, $(n \prec n')$ means that the task labeled with n must precede the one labeled with n'; (n, l), (l, n) and (n, l, n') mean that l must be true in the state immediately after n, immediately before n, and in all states between n and n', respectively. (initially l) means that l must be true in the initial state. Both negation and disjunction are allowed in the constraint formula.

$[(n_1 : achieve[clear(v_1)])(n_2 : achieve[clear(v_2)])$
$(n_3 : do[move(v_1, v_3, v_2)])$
$(n_1 \prec n_3) \wedge (n_2 \prec n_3) \wedge (n_1, clear(v_1), n_3)$
$\wedge (n_2, clear(v_2), n_3) \wedge \ (on(v_1, v_3), n_3) \wedge \neg(v_1 = v_2)$
$\wedge \neg(v_1 = v_3) \wedge \neg(v_2 = v_3)]$

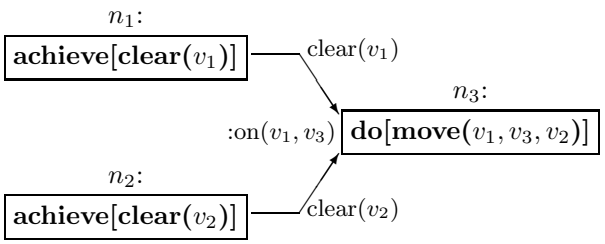

Figure 1: A task network, and its graphical representation.

As an example, Fig. 1 shows a blocks-world task network and its graphical representation. In this task network there are three tasks: clearing v_1, clearing v_2, and moving v_1 to v_2. The task network also includes the constraints that moving v_1 should be done last, v_1 and v_2 should remain clear until we move v_1, and that the variable v_3 is bound to the location of v_1 before v_1 is moved.

A task network that contains only primitive tasks is called a *primitive task network*. Such a network might occur, for example, in a scheduling problem. In this case, an HTN planner would try to find a schedule (task ordering and variable bindings) that satisfies all the constraints.

In the more general case, a task network can contain *non-primitive tasks*, which the planner needs to figure out how to accomplish. Non-primitive tasks cannot be

[3] The term "STRIPS-style" planning is used to refer to any planner (either total- or partial-order) in which the planning operators are "STRIPSoperators" (i.e., operators consisting of three lists of atoms: a precondition list, an add list, and a delete list).

[4] We also allow node labels in the constraints to be of the form $first[n_i, n_j, \ldots]$ or $last[n_i, n_j, \ldots]$ so that we can refer to the task that starts first and to the task that ends last among a set of tasks, respectively.

1. Input a planning problem **P**.
2. If **P** contains only primitive tasks, then
   resolve the conflicts in **P** and return the result.
   If the conflicts cannot be resolved, return
   failure.
3. Choose a non-primitive task t in **P**.
4. Choose an expansion for t.
5. Replace t with the expansion.
6. Use critics to find the interactions among the
   tasks in **P**, and suggest ways to handle them.
7. Apply one of the ways suggested in step 6.
8. Go to step 2.

Figure 2: The Standard HTN Planning Procedure.

1. Input a planning problem $\mathbf{P} =< d, I, \mathcal{D} >$.
2. Initialize OPEN-LIST to contain only d.
3. If OPEN-LIST is empty, then
   halt and return "NO SOLUTION."
4. Pick a task network tn from the OPEN-LIST.
5. If tn is primitive, its constraint formula is TRUE,
   and tn has no committed-but-not-realized
   constraints, then return tn as the solution.
6. Pick a refinement strategy R for tn .
7. Apply R to tn and insert the resulting set of
   task networks into OPEN-LIST.
8. Go to step 3.

Figure 3: High-level Refinement-Search in UMCP

executed directly, because they represent activities that may involve performing several other tasks. For example the task of traveling to New York can be accomplished in several ways, such as flying, driving or taking the train. Flying would involve tasks such as making reservations, going to the airport, buying ticket, boarding the plane; and flying would only work if certain conditions were satisfied: availability of tickets, being at the airport on time, having enough money for the ticket, etc.

Ways of accomplishing non-primitive tasks are represented using constructs called *methods*. A method is a syntactic construct of the form (α, d) where α is a non-primitive task, and d is a task network. It states that one way to accomplish the task α is to achieve all the tasks in the task network d without violating the constraints in d. For example, the task network in Figure 1 presents one possible way of accomplishing $\mathbf{on}(v_1, v_2)$, thus $(achieve[on(v_1, v_2)], d)$ is a method for Blocks world domain, where d is the task network in Figure 1.

An HTN planning problem is represented as a triple $\mathbf{P} = \langle d, I, \mathcal{D} \rangle$, where d is the task network we need to plan for, I is the initial state, and $\mathcal{D}$ is the set of operators and methods associated with the planning domain.

A number of different systems that use heuristic algorithms have been devised for HTN planning [Tate, 1977; Vere, 1983; Wilkins, 1988], and several recent papers have tried to provide formal descriptions of these algorithms [Yang, 1990; Kambhampati and Hendler, 1992]. Figure 2 presents the essence of these algorithms: HTN planning works by expanding tasks and resolving conflicts iteratively, until a conflict-free plan can be found that consists only of primitive tasks.

Expanding or *reducing* each non-primitive task (steps 3–5) is done by finding a method capable of accomplishing the non-primitive task, and replacing the non-primitive task with the task network produced by the method. Details of how to do the task expansion is presented in [Erol et al., 1994a; 1994b].

The task network produced in Step 5 may contain conflicts caused by the interactions among tasks. The job of finding and resolving such interactions is performed by critics. This is reflected in Steps 6 and 7 of Figure 2: after each reduction, a set of critics is checked so as to recognize and resolve interactions between this and any other reductions. Thus, critics provide a general mechanism for detecting interactions early, so as to reduce the amount of backtracking.

3 Planning in UMCP

One way of finding solutions to HTN planning problems is to generate all possible expansions of the input task network to primitive task networks, then generate all possible ground instances (assignment of constants to variables) and total orderings of those primitive task networks and finally output those whose constraint formulae evaluate to true. However, considering the size of the search space, it is more appropriate to try to take advantage of the structure of the problem, and prune large chunks of the search space by eliminating in advance some of the variable bindings, orderings or methods that would lead to dead-ends. To accomplish this UMCP uses a branch-and-bound approach [Kanal and Kumar, 1988].

A task network can be thought of as an implicit representation for the set of solutions for that task network. UMCP works by refining a task network into a set of task networks, whose sets of solutions together make up the set of solutions for the original task network. Those task networks whose set of solutions are determined to be empty are filtered out. In this aspect, UMCP nicely fits into the general refinement search framework described in [Kambhampati et al., 1995].

Figure 3 contains a sketch of the high-level search algorithm in UMCP. Search is implemented by keeping an OPEN-LIST of task networks in the search space that are to be explored; by altering how task networks are picked from the OPEN-LIST and how they are inserted, depth-first, breadth-first, best-first and various other search techniques can be employed. Step 5 checks whether tn is a solution node; if all tasks in tn are primitive, the constraint formula is the atom TRUE, and the list of constraints that have been committed to be made true but not yet made true is empty, then all task orderings and variable assignments consistent with the auxiliary data structures associated with tn are plans for the original problem.[5] Those plans can be easily enu-

[5] Constraints and the data structures will be discussed in

merated. If tn is *not* a solution node, then it is refined by some refinement strategy R, and the resulting task networks are inserted back into the OPEN-LIST.

Three types of refinement strategies used in UMCP are task reduction, constraint refinement, and user-specific critics. Task reduction involves retrieving the set of methods associated with a non-primitive task in tn, expanding tn by applying each method to the chosen task and returning the resulting set of task networks. User-specific critics is one of the places where UMCP can be tailored for specific domains. If a domain-specific refinement strategy is available, it can be used to improve the performance of the planner. This paper will focus on constraint refinement.

4 Constraint Handling in UMCP

4.1 Overview

This section contains an overview of the constraint handling mechanisms in UMCP, which serve as domain independent critics. They are designed to preserve soundness, completeness, and systematicity. Details for each type of constraint are summarized in the next section. For a full description, see [Erol, 1995].

The three types of decisions in HTN planning are the choice of method for each non-primitive task, the choice of constant to assign to each variable, and the orderings of tasks. Of those three, the choice of method is directly reflected in the task network (i.e. in the list of tasks and the constraint formula). Auxiliary data structures are required for the other two. Thus, along with each task network, UMCP keeps a list of possible values for each variable values until the size of the list exceeds a threshold, and a partial order graph of task nodes. Both of those structures will be referred to as commitments. Dealing with some constraints might not be possible at the current level of detail in a task network; dealing with those constraints has to be postponed until the task network is refined further. Until then, those constraints are stored in a list called the Promissory List (the list of constraints the planner has committed to make true, but has not done so yet).

The four phases of constraint refinement in UMCP are constraint selection, constraint update, constraint propagation and constraint simplification, which are carried out sequentially.

Constraint selection involves deciding which constraints in the constraint formula or in the commitments to work on. Constraint selection returns a list of constraint formulae, where each formula is a conjunct of atomic constraints. The list of constraint formulae is selected in such a way that (a) the formulae in the list are mutually inconsistent in the presence of commitments (in order to preserve systematicity), and (b) the list covers all possibilities (in order to preserve completeness). Some examples of constraints that may be selected include (i) an atomic constraint[6] and its negation, (ii) a conjunct of unit clauses from the constraint formula, or (iii) a set of possible constraints for a variable – i.e. if the set of possible values for a variable v are $\{truck_1, truck_2, truck_3\}$, UMCP may branch out on the constraints $(v = truck_1), (v = truck_2), (v = truck_3)$.

For each constraint formula in the list computed in the constraint selection phase, the *constraint update phase* computes a task network for every possible way of making the selected formula true by further restricting the commitments of the task network. For each atomic constraint in the selected constraint formula, the following steps are executed: first it is evaluated; if it evaluates to true, it can be ignored, if it evaluates to false, the task network fails, otherwise further restrictions are placed on the commitments (variable bindings, orderings etc.) to make the constraint necessarily true in all further refinements of the task network. There might be multiple possible ways of accomplishing that, thus even atomic constraint update computes a list of task networks rather than a single task network. However, UMCP does constraint update in such a way that there is no overlap among the set of solutions to the task networks in this list. For some constraints, at the current level of detail in the task network, it might not be possible via restrictions on commitments to ensure that the constraint will be true in all further refinements. Those constraints are simply recorded in the Promissory List.

In the *constraint propagation phase*, UMCP evaluates and simplifies the constraints in the Promissory List of each task network produced during the constraint update phase. If any of those constraints evaluate to false, the task network fails; those that evaluate to true are removed from the Promissory List. Constraint update is performed on the remaining simplified constraints if possible at the current level of detail in the task network. This phase is repeated until no more propagation is possible.

UMCP contains evaluation and simplification routines for every type of constraint, as described in the next section. These routines are used in the *constraint simplification phase* to evaluate and simplify the constraint formulae of the task networks produced during the propagation phase. For instance if part of a conjunct evaluates to true, that part is dropped, if it evaluates to false, the whole conjunct evaluates to false. Disjuncts are treated analogously. Those task networks whose constraint formulae evaluate to false are pruned.

4.2 Details

This section describes how constraint evaluation and update is done for each type of constraint. Update always involves evaluating the constraint and it fails whenever the constraint evaluates to false. This is omitted from the explanations below for brevity.

Variable Binding Constraints

Type: $(v = a)$
Evaluation: Return true if constant a is the only possible value for variable v; return false if a is not a possible value for v; return $(v = a)$ otherwise.
Update: To make it true, set the possible value list for v to a, replace v with a throughout the task network. To

detail in the next section.

[6] An atomic constraint refers to any instance of the types of constraints discussed in Section 2.

make it false, remove a from the possible value list for v. If v has only one possible value left, substitute that value for v throughout the task network.

Type: $(v_1 = v_2)$
Evaluation: Return true if both v_1 and v_2 have the same one possible value; return false if they do not have any common possible values or if negation of the constraint is in the Promissory List; return the constraint itself otherwise.
Update: To make it true, set the possible value list for v_2 to the intersection of possible values for v_1 and v_2, set the possible value list for v_1 to v_2, and replace v_1 with v_2 throughout the task network. To make it false, insert its negation in the Promissory List.

Ordering Constraints
Ordering constraints are handled by querying and modifying the partial order graph. They are of the form (node-expression $\prec$ node-expression).

A node-expression can be in either of the 2 forms: $first[n_1, n_2, \ldots, n_k]$, or $last[n_1, n_2, \ldots, n_k]$. Thus, a total of four types of ordering constraints need to be considered.[7]

Let $A_i = \{n_{i1}, \ldots, n_{ik}\}$ be lists of node labels for $i = 0, 1$. Recall that $first[A_0]$ and $last[A_0]$ refer to the node which is ordered to be the first and the last among the nodes in the expansion of nodes $\{n_{01}, \ldots, n_{0k}\}$, respectively.

Let's define $O_i, I_i \subseteq A_i$ as the nodes that are not ordered after (respectively before) any node in $A_0 \cup A_1$.

Let's define $B_i, C_i \subseteq A_i$ as the nodes that are not ordered after (respectively before) *some* node in A_i and are not ordered after(respectively before) *all* nodes in $A_{(i+1) \bmod 2}$.

Type: $(first[A_0] \prec first[A_1])$
Evaluation: Return true if O_1 is empty; return false if O_0 is empty; return $(first[O_0] \prec first[O_1])$ otherwise.
Update: If O_0 contains a single primitive task put links from the node in O_0 to each node in O_1 in the partial order graph; otherwise insert $(first[O_0] \prec first[O_1])$ into the Promissory List.

Type: $(first[A_0] \prec last[A_1])$
Evaluation: Return false if B_0 or C_1 is empty; return true if some node in B_0 is ordered before one in C_1; otherwise return $(first[B_0] \prec last[C_1])$.
Update: If both B_0 and C_1 contain single primitive tasks, put a link between them, otherwise insert $(first[B_0] \prec last[C_1])$ into the Promissory List.

Evaluations and updates of $(last[A_0] \prec first[A_1])$ and $(last[A_0] \prec last[A_1])$ are done analogously.

State Constraints
The set of atoms in the initial state are stored in a discrimination tree for fast querying. Negative literals need not be stored due to the closed world assumption. Note that "¬ (initially l)" and "(initially $\neg l$)" have the same meaning.

Type: `(initially` l`)`
Evaluation: Return true if all ground instances of l that are consistent with possible values lists in commitments are in initial state; return false if none of the ground instances of l that are consistent with possible values lists are in initial state.
Update: To make it true (false) do the following: If l has only one variable, restrict that variable to values for which l is true (false) in the initial state. If l contains $k > 1$ variables, for each combination of values for the first $k - 1$ variables, output a task network by assigning those variables the corresponding constants, and restricting the value of the last variable so as to make l true (false) in the initial state. Combinations of values for which this is not possible need not be considered. For example, in order to make **(initially type[**v**,truck])** true in transport logistics domain, it suffices to restrict possible values for v to constants of type truck.

The state constraints of the form (l, n) are evaluated and updated by computing the effectors. An effector of l is either (i) a primitive task with an effect that can unify with l or $\neg l$, (ii) the initial state, (iii) a compound task whose expansion might contain such a primitive task, or (iv) a committed-to state constraint that is stored in Promissory List whose literal can unify with l or $\neg l$. Those effectors that are ordered after n or shadowed by[8] some other effector of l are ignored. If all effectors of l are positive then (l, n) is true, if all are negative than it is false, otherwise it is unknown yet. Update is delayed if some of the positive effectors are compound tasks, and the constraint is recorded in Promissory List. Otherwise for each positive effector e a new task network is created where e is the establisher of l with added constraints $(preserve\ l\ e\ n) \wedge (preserve\ \neg l\ e\ n)$preventing any action between e and n from denying or asserting l, respectively.

Evaluating and/or updating constraints of the form (n, l) is delayed until n refers to a single primitive task symbol. In that case (n, l) evaluates to true if the action labeled by n asserts l, it evaluates to false if that action denies l, otherwise it evaluates to (l, n).

Constraints of the form (n, l, n') are converted to $(n' \prec n) \vee [(n \preceq n') \wedge (n, l) \wedge (preserve\ l\ n\ n')]$ and handled as such.

4.3 Properties of Constraint Refinement

As discussed in Section 3, constraint refinement takes a task network tn as input, and returns a list of task networks $R(tn)$. We can now demonstrate the properties of constraint refinement in UMCP:

- Soundness: Any solution to any task network in $R(tn)$ is also a solution for tn. Thus constraint refinement does not introduce invalid solutions. UMCP satisfies this property because commitments in task networks grow monotonically, and constraints in the Promissory List are removed only as they become necessarily true.

[7] Ordering constraints which contain negation or refer to single node labels can be converted to one of those four forms.

[8] An effector x shadows an effector y if x is ordered between y and n, and whenever an effect of y codesignates with l, so does an effect of x.

- Completeness: Any solution for tn is also a solution for some task network in $R(tn)$. Thus constraint refinement does not eliminate any valid solutions. UMCP satisfies this property because any time a constraint is selected in constraint selection phase, its negation is also selected (unless it contradicts with the commitments or the constraint formula), and all possible ways of making a constraint true are tried in the constraint update phase.
- Systematicity: The set of candidate solutions for each task network in $R(tn)$ are mutually disjoint. Thus UMCP does not examine the same candidates multiple times. The way systematicity is accomplished in UMCP is by making sure (a) the branches in constraint selection are mutually exclusive (i.e. any two conjuncts have a common literal, positive in one, negated in the other); (b) there is no overlap among the solution sets to the task networks produced in update phase.

5 Related Work

Causal links are used by POCL planners such as SNLP [McAllester and Rosenblitt, 1991] to establish preconditions and to detect threats. Causal links are also employed by UMCP in the form of special state constraints stored in the Promissory List. SNLP's threat removal process is similar to how UMCP handles those special constraints in its constraint propagation phase.

[Chapman, 1987] introduced the MTC (modal truth criterion) to tell whether a literal is true at a given point in a partially-ordered plan. In order to evaluate state constraints, UMCP uses an extended version of the MTC that also accounts for compound tasks. UMCP's extended MTC algorithm runs in quadratic time—and it is directly applicable for computing Chapman's MTC, for which the other known algorithms run in cubic time.

NOAH [Sacerdoti, 1977] employs its *resolve conflicts* critic to deal with deleted-condition interactions, which are explicitly represented by state constraints in UMCP. The constraint refinement techniques of UMCP guarantees these interactions will be handled without sacrificing soundness or completeness.

UMCP evaluates each constraint before trying to make it true, and skips those constraints that are already true, and hence it emulates NOAH's *eliminate redundant preconditions* critic.

HTN planners often allow several types of conditions in methods. How to deal with those conditions has been a topic of debate.

NONLIN [Tate, 1977] evaluates filter conditions as soon as they are encountered, using the Q&A mechanism. Q&A returns false unless it can verify those conditions to be necessarily true, even if the conditions are possibly true. Thus, NONLIN often backtracks over filter conditions which would have been achieved by actions in later task expansions or by more ordering and variable binding commitments. As a result, NONLIN may fail to find a solution when a solution exists, or may miss a short and simple solution and do much more work to find a longer and more complicated solution.

At first glance, the problems such as these might seem to argue against the use of filter conditions: at one extreme, using filter conditions immediately to prune the search space sacrifices completeness, and at the other extreme, postponing their use until the plan is complete (so as to preserve completeness) is inefficient.[9]

Although the above argument is partially correct, it ignores a third possibility that lies between the two extremes. In general, to preserve completeness, a planner cannot use a filter condition to prune the search space unless the filter condition evaluates to "necessarily false"—but this does not necessarily require that the task network has been expanded into a primitive and totally-ordered plan. Instead, UMCP simply records the filter conditions in the Promissory List and prunes the task network only when one of them becomes necessarily false.

More specifically, UMCP handles filter conditions and other constraints as follows:

- Some instances of variable binding, ordering, and state constraints can be dealt with immediately. For example, conditions (e.g., an object's type) that are not affected by the actions are represented by constraints of the form **(initially l)**. Such constraints can be evaluated at any time by querying the initial state, and they can be committed to by appropriately restricting the possible values for the variables in l.[10]
- Those constraints that cannot be dealt with immediately are stored in the Promissory List, and are processed in the constraint propagation phases.

Constraints in UMCP go through three stages: they first appear in constraint formula; then possibly in the Promissory List if they cannot be dealt with at the time they are selected in constraint selection phase; and finally they are reflected in restrictions on possible values for variables and task orderings. This three-stage approach facilitates dealing with the disjunctions in the constraint formula, and by postponing its processing of some types of constraints, UMCP preserves completeness without sacrificing efficiency.

6 Conclusion

Dealing with numerous types of interactions is an important aspect of planning systems. The work described in [Erol et al., 1994a; 1994b] has provided a formal framework for representing interactions and conflicts via constraints, and in this paper we have introduced techniques for constraint handling as a means for detecting interactions and resolving conflicts. Those techniques preserve soundness, completeness, systematicity, and they have been implemented in UMCP, an HTN planning system.

[9] In fact, Collins and Pryor [1992] have made a similar argument against filter conditions in the context of planning with STRIPS-style operators.

[10] SIPE [Wilkins, 1988] uses a "sort hierarchy" for this purpose, the only difference in UMCP is that UMCP allows arbitrary boolean formulae constructed from all types of constraints, instead of a conjunct of constraints as in SIPE.

By instantiating the constraint selection strategy in different ways, various commitment strategies discussed in the literature can be used by UMCP. For example, variable instantiation can be done before anything else (as in NONLIN), all primitive tasks can be totally ordered as soon as they appear in task networks, or task expansions can be deferred until all conflicts have been resolved (least commitment). Currently, we are designing experiments to empirically evaluate these techniques.

UMCP's constraint-handling mechanism provides the capabilities of many domain-independent critics discussed in the literature, and UMCP's user-specific critics module can be used to incorporate domain-specific critics as well. The modular and formal nature of UMCP makes it readily extensible. We are currently exploring ways of extending UMCP's constraint-handling mechanism to handle numerical and complex temporal constraints so that it can do deadline and resource management, and provide the capabilities of other domain-independent critics.

References

[Allen et al., 1990] Allen, J.; Hendler, J. and Tate, A. editors. *Readings in Planning.* Morgan-Kaufmann, San Mateo, CA, 1990.

[Chapman, 1987] Chapman, D. Planning for conjunctive goals. *Artificial Intelligence*, 32:333–378, 1987.

[Collins and Pryor, 1992] Collins, G. and Pryor, L. Achieving the functionality of filter conditions in a partial order planner In *Proc. AAAI-92*, 1992, pp. 375—380.

[Dean, 1983] Dean, T. Time map maintenance. Technical Report 289, Yale University, October 1983.

[Drummond, 1985] Drummond, M. Refining and Extending the Procedural Net. In *Proc. IJCAI-85*, 1985.

[Erol et al., 1994a] Erol, K.; Hendler, J. and Nau, D. Semantics for Hierarchical Task Network Planning. CS-TR-3239, UMIACS-TR-94-31, ISR-TR-95-9, University of Maryland, March 1994.

[Erol et al., 1994b] Erol, K.; Hendler, J. and Nau, D. Complexity results for hierarchical task-network planning. To appear in *Annals of Mathematics and Artificial Intelligence* Also available as Technical report CS-TR-3240, UMIACS-TR-94-32, ISR-TR-95-10 Computer Science Dept., University of Maryland, March 1994.

[Erol, 1995] Erol, K. HTN Planning: Formalization, Analysis, and Implementation. Ph.D. Dissertation, Computer Science Dept., University of Maryland, 1995. *In preparation.*

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

[Kambhampati and Hendler, 1992] Kambhampati, S. and Hendler, J. "A Validation Structure Based Theory of Plan Modification and Reuse" *Artificial Intelligence*, May, 1992.

[Kambhampati, 1992] Kambhampati, S. "On the utility of systematicity: understanding trade-offs between redundancy and commitment in partial-ordering planning," unpublished manuscript, Dec., 1992.

[Kambhampati et al., 1995] Kambhampati, S.; Knoblock, C. and Yang, Q. Planning as refinement search: A unified framework for evaluating design tradeoffs in partial order planning. To appear in *Artificial Intelligence*, Special Issue on Planning.

[Kanal and Kumar, 1988] Kanal, L. and Kumar, V. Search in Artificial Intelligence, Springer-Verlag, 1988.

[McAllester and Rosenblitt, 1991] McAllester, D. and Rosenblitt, D. Systematic nonlinear planning. In *Proc. AAAI-91*, 1991.

[Minton et al., 1991] Minton, S.; Bresina, J. and Drummond, M. Commitment strategies in planning. In *Proc. IJCAI-91*, 1991.

[Sacerdoti, 1977] Sacerdoti, E. D. *A Structure for Plans and Behavior,* Elsevier-North Holland. 1977.

[Sussman, 1990] Sussman, G.J., HACKER: a computational model of skill acquisition, MIT AI Lab Memo 1973, also in Allen, 1990, *Readings in Planning.* Morgan Kaufmann.

[Tate et al., 1990] Tate, A.; Hendler, J. and Drummond, D. AI planning: Systems and techniques. *AI Magazine*, (UMIACS-TR-90-21, CS-TR-2408):61–77, Summer 1990

[Tate, 1977] Tate, A. Generating Project Networks In *Proc. IJCAI-77*, 1977. pp. 888–893.

[Tate et al., 1994] Tate, A.; Drabble, B. and Dalton, J. The Use of Condition Types to Restrict Search in an AI Planner. *Proc. AAAI-94.* pp. 1129-1134.

[Vere, 1983] Vere, S. A. Planning in Time: Windows and Durations for Activities and Goals. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-5(3):246–247, 1983.

[Wilkins, 1988] Wilkins, D. *Practical Planning: Extending the classical AI planning paradigm,* Morgan-Kaufmann, CA. 1988.

[Yang, 1990] Yang, Q. Formalizing planning knowledge for hierarchical planning *Computational Intelligence* Vol.6., 12–24, 1990.

[Yang et al., 1993] Yang, Q.; Nau, D.S.; and Hendler, J. Merging separately generated plans with restricted interactions. *Computational Intelligence*, 9(1), February 1993.