# HOTRiDE: Hierarchical Ordered Task Replanning in Dynamic Environments

**N. Fazil Ayan** and **Ugur Kuter**
Dept. of Computer Science and
Institute for Advanced Computer Studies
Univ. of Maryland, College Park
College Park, MD 20742, USA
{nfa,ukuter}@cs.umd.edu

**Fusun Yaman**
Dept. of Computer Science and
Electrical Engineering
Univ. of Maryland, Baltimore County
1000 Hilltop Circle
Baltimore, MD 21250, USA
fusun@csee.umbc.edu

**Robert P. Goldman**
SIFT, LLC
211 North First Street, Suite 300
Minneapolis, MN 55401, USA
rpgoldman@sift.info

**Abstract**

One of the assumptions in classical planning is that the environment is *static*: i.e., the planner is the only entity that can induce changes in the environment. A more realistic assumption is that the environment is *dynamic*; that is, there are other entities in the world and the actions generated by the planner may fail due to the operations of these entitites. In this paper, we describe a planning system, called HOTRiDE (Hierarchical Ordered Task Replanning in Dynamic Environments), which interleaves plan generation, execution, and repair in order to work under such circumstances. Our approach is an extension of SHOP (Simple Hierarchical Ordered Planner) (Nau *et al.* 1999), which is a well-known Hierarchical Task Network planning system. Our experiments with HOTRiDE in an abstracted version of a military-planning domain demonstrates the potential of our approach.

## Introduction

One of the assumptions in classical planning is that the environment is *static*: i.e., the planner is the only entity that can induce changes in the environment. A more realistic assumption is that the environment is *dynamic*; that is, there are other entities in the world and the actions generated by the planner may fail due to the operations of these entitites.

In this paper, we describe a planning algorithm, called HOTRiDE (Hierarchical Ordered Task Replanning in Dynamic Environments), which interleaves plan generation, execution and repair. HOTRiDE is based on the well known *Hierarchical Task Network (HTN)* planning system, SHOP (Simple Hierarchical Ordered Planner) (Nau *et al.* 1999). An HTN planning system formulates a plan by decomposing tasks (i.e., symbolic representations of activities to be performed) into smaller and smaller subtasks until tasks are reached that can be performed directly. The basic idea was developed in the mid-70s (Sacerdoti 1975; Tate 1977), and the formal underpinnings were developed in the mid-90s (Erol, Hendler, & Nau 1996).

In HOTRiDE, we focused on the following issues:

- Adaptation of previous plan segments using SHOP-style HTN decompositions to account for problems introduced by executing plans in dynamic environments
- Replanning while preserving elements of the previous plan in order not to re-compute them over and over again.

Given a planning problem, HOTRiDE first generates a solution plan for that problem. Then, the system starts to execute this plan. If, during execution, an action fails unexpectedly, then HOTRiDE attempts to replan only the part of the original plan that is related to the failing action. However, replanning that segment of the plan may certainly affect the other parts of the original plan. To be able to determine and eliminate such side effects, HOTRiDE uses a data structure, called a *dependency graph*, which contains the derivation tree for the generated plan and the causal links between the nodes of that tree. The causal links show the relationship between the effects of an action to be executed now in the plan and the task decompositions occurring in future.

We have implemented a prototype of the HOTRiDE system and tested it on a Noncombatant Evacuation Operations (NEO) planning domain. The results of one set of experiments showed that HOTRiDE is able to generate plans and revise those plans if their executions fail until the goal tasks are successfully accomplished. We also did another set of experiments to investigate the quality of revisions made by HOTRiDE to the original plan and the results of those experiments confirmed that HOTRiDE does revisions to the original plan by preserving the actions accomplished so far as much as possible and modifying only the necessary parts of the original plan.

## Background

We use the same definitions for states, actions, primitive and nonprimitive tasks, task networks and planning problems as in (Nau *et al.* 1999). We summarize our definitions below.

We formalized a *planning domain* as a deterministic finite state-transition system $(S, A, \gamma)$ where $S$ is the finite set of all possible states of the world, $A$ is the finite set of all possible actions, and $\gamma$ is the *state-transition function* defined as $\gamma : S \times A \to S \cup \{\perp\}$.

An *HTN planning problem* description consists of the following: the initial state (a symbolic representation of the state of the world at the time that the plan executor will begin executing its plan) and the goal task network (a set of tasks to be performed, along with some constraints over those tasks that must be satisfied).

A solution to an HTN planning problem is a *plan*; i.e., a sequence of actions that, when executed in the initial state, performs the desired tasks. The execution of an action is *successful* if $\gamma(s, a) \neq \perp$ after we observe the state $s$ to execute $a$ in; otherwise, the execution of $a$ is not successful; i.e., we say that it is failed.

In order to generate solutions for the planning problems, an HTN planner uses an *HTN domain description* that contains two kinds of knowledge artifacts: *methods* and *operators*. HTN planners may have other kinds of knowledge artifacts as well. For example, the SHOP planner (Nau *et al.* 1999) also has axioms that can be used to infer conditions about the current state.

The operators are like the planning operators used in any classical planner. The names of these operators are designated as *primitive tasks* (i.e., tasks that we know how to perform directly). Any task that does not correspond to an operator name is a *nonprimitive* task.

Each method is a prescription for how to accomplish a non-primitive task by decomposing it into subtasks (which may be either primitive or non-primitive tasks). A method consists of three elements: (1) the task that the method can be used to accomplish, (2) the set of preconditions need to be satisfied for the method to be applicable, and (3) the subtasks to accomplish.

For example, consider the task of moving a collection of boxes from one location to another. One method might be to move them by car. For such a method, the preconditions might be that the car is in working order and is present at the first location. The subtasks might be to open the door, put the boxes into the car, drive the car to the other location, and unload the boxes.

One important assumption we made is that the preconditions of the planning operators or methods are conjunctions of literals. Based on this assumption, we define the notion of a *causal link* between two tasks in a similar way as in classical planning (Ghallab, Nau, & Traverso 2004). A causal link between a primitive task $a$ and another task (either primitive or nonprimitive) $t$ is defined as a pair $(e, p)$ such that $e$ is an effect of the action that corresponds to $a$ and $p$ is a precondition of the planning operator for $t$ if $t$ is a primitive task, or $p$ is a precondition of the HTN method for $t$ if $t$ is nonprimitive. A causal link between a nonprimitive task $t$ and another task (either primitive or nonprimitive) $t'$ is a pair $(e, p)$ such that $e$ is an effect of the action that corresponds to a primitive task that can be generated by successively decomposing $t$ using the available HTN methods and $p$ is a precondition of the planning operator for $t$ if $t$ is a primitive task, or $p$ is a precondition of the HTN method for $t$ if $t$ is nonprimitive. if there is a causal link $(p, e)$ between two tasks $t1$ and $t2$ as described above, then we say that $t1$ supports $t2$.

Given a task $t$, an *HTN trace* for $t$ consists of a plan $\pi$ that accomplishes $t$ and a subset of the methods from the input domain description that, when successively applied to $t$ and its subtasks, generates the plan $\pi$. An HTN trace is defined in terms of *HTN trace nodes*. An *HTN trace node* is a tuple $N = (t, \pi, A, D, Q, C)$ where $t$ is a task (primitive or nonprimitive), $\pi$ is a plan achieving the task $t$, $A$ is the cumulative additions and $D$ is the cumulative deletions made to the state while achieving the task $t$. $Q$ is defined as follows: (1) if $t$ is a primitive task then $Q$ is the preconditions of $t$, and (2) else if $t$ is a nonprimitive task then $Q$ is the preconditions of a method that can be applied to $t$ to yield $\pi$. $C$ is the set of pointers to the child nodes of $N$.

A *task-dependency graph* is defined as the triple $DG = (DT, CL, PL)$. In this definition, $DT$ is an HTN trace. $CL$, called the *causal-links list*, is defined as: for every possible ground atom, $p$, in the world, $CL(p)$ is a totally ordered list of heads of ground operator instances that add or delete $p$ to or from the state respectively. $PL$, called the *predicate-list*, is defined as follows: for every atom, $p$, that appears in the precondition list of a task (primitive or nonprimitive), $PL(p)$ is a totally ordered list of tasks that have $p$ as a precondition.

## HOTRiDE

In this section, we introduce HOTRiDE, an HTN planning, execution, and plan-repairing system. HOTRiDE is based on a modified version of the SHOP planning algorithm (Nau *et al.* 1999). Like SHOP, HOTRiDE does HTN-style task decompositions to generate the steps of a plan in the same order that they will be executed. In addition to a plan (i.e., a sequence of actions) for an input HTN planning problem, HOTRiDE also returns a task-dependecy graph as defined above.

The task-dependency graph generated by HOTRiDE consists of HTN trace nodes that are generated by the modified SHOP planner and it holds information about the dependencies among the tasks decomposed by SHOP during planning. We represent such dependencies between tasks via the causal links among them. Using these causal links, HOTRiDE can determine which parts of the plan are affected by the result of a certain operator application. If an action fails during execution, these causal links help HOTRiDE to find which decompositions in the HTN trace are not valid anymore and need to be replanned.

Once the dependency graph is computed, we assume that the plan is passed to a controller for execution. Because the environment is dynamic, at every execution step the state of the world is observed and the controller tries to execute the next plan action at the last observed state. Note that the observed state sequence might be different than the ones that are computed during the off-line planning (and replanning) phase. Thus HOTRiDE does not expect to follow the same state sequence as the planner projected but if at any point during the execution, an action fails (i.e., the current state of the world does not satisfy the preconditions of the action), then the execution stops and HOTRiDE attempts to generate a new plan from that state of the world to achieve the goal tasks given to it at the beginning.

Figure 1 shows a high-level description of the HOTRiDE plan generation, execution, and repair procedure. First, HOTRiDE checks every parent of the failed action using the dependency graph generated before. For example, suppose the dependency graph in Figure 2 represents the plan being executed and further suppose that the execution of the action $a$ fails during execution. The algorithm first identifies the *minimal failed parent*, a nonprimitive task $t$ in the dependency graph, and it attempts to generate a new decomposition for $t$. The minimal failed parent is found as follows:

```
Procedure HOTRiDE($g, M$)
  observe the state $s$
  $\langle \pi, D \rangle \leftarrow$ SHOP($s, g, M$)
  if $\pi =$ FAILURE then return FAILURE
  loop
    if $\pi = \emptyset$ then return SUCCESS
    select the action $a \in \pi$ that does not have any
          predecessors and remove it
    observe the state $s$
    execute $a$ in $s$ and observe the next state $s'$
    if $\gamma(s, a) = \perp$ then
        $\pi \leftarrow$ REPLAN($s', a, g, D, M$)
        if $\pi =$ FAILURE then return FAILURE
end-procedure

Procedure REPLAN($s, a, g, D, H$)
  $t \leftarrow$ the failed parent of $a$ that does not have any
            failed parents;
  if $t$ is a goal task in $g$ then return FAILURE
  $\langle \pi', D' \rangle \leftarrow$ SHOP($s, t, H$)
  if $\pi' =$ FAILURE then return FAILURE
  for each action $a' \in \pi'$ do
    let $t'$ be a task in the dependency graph $D$
          that is supported by $a'$
    update the dependency graph $D'$ to include
          a causal link from $a'$ to $t'$ of $D$
  $D'' \leftarrow D \cup D'$
  loop
    if every task in $D''$ is supported then return $\pi'$
    select a task $t'$ in $D''$ that is not supported
    $\pi'' \leftarrow$ REPLANNOPARENTS($s, t', g, D'', H$)
    if $\pi'' =$ FAILURE return FAILURE
    $\pi' \leftarrow \pi' \cup \pi''$
  return$\pi'$
end-procedure
```

Figure 1: A high-level description of the HOTRiDE procedure.

The parent task of a failed action in the HTN trace of the current dependency graph is a failed task since that task cannot be accomplished due to the failure. For example in Figure 2, this parent task is $t1$. Then, if $t1$ is the first subtask of its parent task $t2$, then HOTRiDE checks the preconditions of the method that decomposed $t2$ in the current HTN trace. If there is a precondition of that method that is not satisfied in the currently observed state of the world, then this means that $t2$ is also a failed task. This is because, by the way HTN planning is done in SHOP and therefore HOTRiDE, the preconditions of the method are the applicability conditions that must be satisfied in the state where the first subtask, in this case $t1$, of the method is to be accomplished.

HOTRiDE checks the parents of a failed task as described above until it generates a minimal failed task (i.e., a task that is marked as failed but whose parent is not) or it reaches to a goal task that does not have any parents in the HTN trace. This minimal failed task is the next task HOTRiDE attempts to generate a new HTN decomposition for.

If the failed task $t1$ is not the first subtask of its parent $t2$ as in the HTN trace shown in Figure 2, then HOTRiDE marks only $t1$ as the minimally failed task in the hierarchy and it attempts to decompose $t1$. If there are no decompositions for $t1$ other than the current one or all decompositions fail, then HOTRiDE marks $t1$ as well as its parent $t2$ as a failed tasks, and it attempts to replan for $t2$.

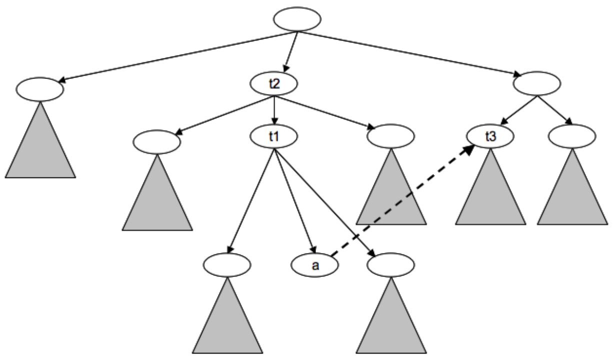

Figure 2: An illustration of a task-dependency graph.

After HOTRiDE identifies the nonprimitive task, say $t$, that needs to be replanned, it identifies the set of causal links in the current dependency graph that are supported by the plan originally generated for $t$. Then, HOTRiDE invokes SHOP again to generate a new plan for $t$ and its corresponding dependency graph. Next, HOTRiDE establishes all of the causal links between the any task in the new dependency graph and the tasks in the previous dependency graph that are not accomplished yet. After this operation, there may some tasks in the original dependency graph that are not supported by the current plan. For example, in Figure 2, suppose $t3$ is one such task after HOTRiDE replanned for $t1$. In other words, in the original dependency graph, $t3$ was supported by the action $a$, but when HOTRiDE replanned for $t1$, there is no support for $t3$ in the new plan any more. In this case, HOTRiDE considers $t3$ as a failed task and attempts to generate other possible decompositions for $t3$, if any. If there is no such decomposition, the algorithm tries the other possible decompositions for $t1$. Note that the algorithm does not attempt to replan for the parents of $t3$ since they are not relevant to the failure of the task $t1$ based on the observed state of the world. In Figure 1, the subroutine REPLANNOPARENTS is responsible for this operation.

HOTRiDE repeats the above process for every causal link that is not supported in the new plan generated for the failed task $t$. If the algorithm generates a new plan in which all the causal links are satisfied, it resumes the execution of this new plan, starting from first unexecuted action. Otherwise, HOTRiDE calls it self recursively on the parent task of $t1$ in the HTN hierarchy.

HOTRiDE repeats the plan repair process above until one of the following holds: either HOTRiDE generates a plan that is executed successfully in the world, or the plan-repair process marks a goal task as failed. In the latter case, HOTRiDE reports failure.

Note that it is possible that the repaired plan will include actions that were previously executed in the failed plan. As a result, the above algorithm allows for duplicate tasks/actions to be executed later in the process. The reason is that it is not clear to us at the moment how to differentiate whether an action in the repaired HTN and plan is redundant or not. One solution would be to have a set $M$ of HTN methods designed for the replanning aspect in mind, where $M$ includes additional methods that check for effects of already established subtasks and result in empty task decompositions (hence allow skipping some steps). Obviously this solution leaves all the burden on the domain designer. Another approach would be to include a mechanism for reasoning about how actions achieve the goals of the input planning problem so that the algorithm could infer, that for example, whether an action appearing a second time in the plan is necessary to establish a goal or subgoal condition in the world that is not apparent in the HTNs. One way to do this is to include dummy tasks and methods in the input HTNs so that the preconditions of such methods would need to be satisfied within the dependency graph generated by HOTRiDE. Finally, another way might be to generate more expressive explanations for the dependencies in the HTNs and the plans, similar to the approach described in (Warfield *et al.* 2007).

## Experiments

In this section, we describe our experiments with a prototype implementation of HOTRiDE. In our experiments, we used an abstract version of a Noncombatant Evacuation Operations (NEO) planning domain. The NEO planning domain for HTN planning was originally developed and reported by (Muñoz-Avila *et al.* 2001). This domain, called NEO Trans1.1, is an abstracted and simplified version of some of the transportation tasks that occur in real-world NEO planning. A NEO involves five primary locations: an assembly point, a headquarters, an intermediate staging base (ISB), the NEO site, and a safe haven. The objective is to generate a plan in order to get from each of these sites to the next one, while selecting the means of transportation and the route to be followed.

There are six possible means of transportation: plane, helicopter, ship, armored vehicle, car, and on foot. Plane transportation depends on the weather conditions, whether there are airports at both source and destination, and whether the environment is suitable for plane transportation or not. Helicopter transportation depends on the weather conditions and whether a helicopter is available or not. Ship transportation depends on the weather conditions, whether the destination location is accessible by sea and whether there exists a harbor there. An armored vehicle can be used any time if it is possible to use as a transportation means. Transportation can be done by car only if the environment in which NEO will be conducted is not hostile. If the conditions for any of the previous transportation methods cannot be satisfied then the transportation is performed by the troops without using any vehicles.

We have performed our experiments on a 700MHz Pentium III Compaq machine with 128MB RAM. We have created 100 random problems for both domains. We have run HOTRiDE 100 times on each problem and observed the average values for our measures.

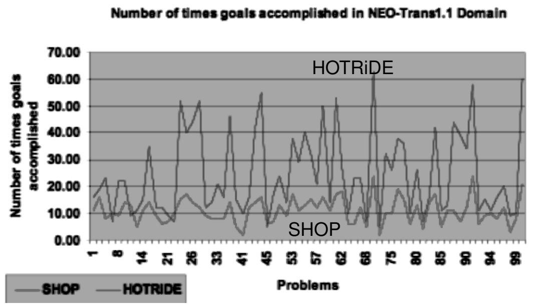

Figure 3: Number of times goals are accomplished in our NEO planning domain using SHOP and HOTRiDE to revise the original plan.

To simulate a dynamic environment and the plan-execution controller mentioned previously, we have implemented a simulator program that executes the plans generated by HOTRiDE. The execution of a plan is performed on the action basis and while executing the actions in a plan, the simulator randomly decides whether to fail a particular action or not. In our experiments, we assumed that the failure of an action means at least one of its preconditions becomes false, although it was true during the plan generation phase. Without loss of generality and for the sake of simplicity, we assumed that only one precondition of an action can be failed at a time by the simulator.

Figure 3 displays the number of times the goal tasks are accomplished in the dynamic environment. The two algorithms SHOP and HOTRiDE are used to revise the original plan whenever it fails. These results demonstrate that the number of times HOTRiDE can revise a plan is usually more than that of SHOP. The results show that SHOP is not designed to produce a new plan starting from the current state of the world. In the case of HOTRiDE whether it is the current state or the initial state does not make much difference since it first tries to find a local plan at the point of failure.

When a domain expert is available and the planning domains and the kinds of failures in those domains are simple enough, sometimes it might be possible to modify the HTNs used by SHOP to include "bookkeeping methods," which (1) allow SHOP to remember where the plan had failed during execution, (2) help SHOP initiate planning from that point. Under such circumstances, SHOP would be able to do replanning on its own without the dependency mechanisms we developed for HOTRiDE. In order to investigate this, we have created a simplified version of the NEO domain based on the original one described above and our assumption that actions fail in the domain only because one of their preconditions fail. Here, we encoded some bookkeeping HTN methods. As a very simple example for such methods, consider the following HTN method for transportation of a vehicle in the NEO world from a source location $?x$ to a destination $?y$:

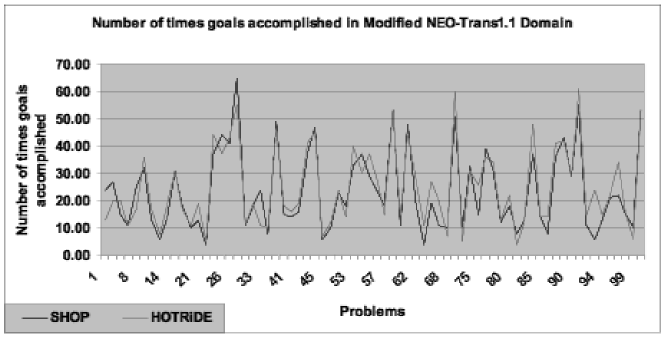

Figure 4: Number of times that goals are accomplished in our NEO planning domain using SHOP and HOTRiDE to revise the original plan. In our experiments, the number of goals accomplished by SHOP and HOTRiDE were generally the same; hence, the curves for both SHOP and HOTRiDE shown above.

```
(:method
   :task (planTransport-to ?x ?y)
   :preconditions
      ((airport ?x) (airport ?y) (tm ?x ?y FIXEDW)
        (weather TFP) (windStrengthOK ?x ?y FIXEDW))
   :subtasks
      ((planDistance ?x ?y FIXEDW)
        (planDuration ?x ?y FIXEDW)))
```

The following HTN method is a simple modification of the one above that does some bookkeeping checks in the state of the world:

```
(:method
   :task (planTransport-to ?y)
   :preconditions
      ((at ?x) (airport ?x) (airport ?y) (tm ?x ?y FIXEDW)
        (weather TFP) (windStrengthOK ?x ?y FIXEDW))
   :subtasks
      ((planDistance ?x ?y FIXEDW)
        (planDuration ?x ?y FIXEDW)))
```

The precondition $(at\ ?x)$ above is used the check the currently observed state of the world in order to verify if the vehicle is still at $?x$ as it was the vehicle's source location or if it is move somewhere else during planning without the control of the planner.

Figure 4 displays the number of times the goal tasks are accomplished in this modified NEO domain with bookkeeping methods for replanning in SHOP. These results demonstrate that in this domain HOTRiDE and SHOP have nearly same replanning power. The modifications to the domain and the bookkeeping HTN methods enabled SHOP to find a new plan by simply remembering the previously accomplished steps and bypassing some replanning steps that are sensitive to the initial state. In the case of HOTRiDE the modifications in the domain had no effect on its performance.

Finally we compared the number of actions performed to achieve the goals. Figure 5 demonstrates the average number of actions performed to achieve goals. In this particular version of the NEO domain, the optimum plan length is 12. In our experiments, HOTRiDE always accomplished the goals performing 12 actions. This shows that, HOTRiDE's plan repair procedure usually keeps the modifications to the original plan at a minimum level, while generating a successful plan at the end. In the case of SHOP the number of actions was generally greater than 12 because of replanning previously achieved goals and trying to re-achieve them or simply performing some bookkeeping actions.

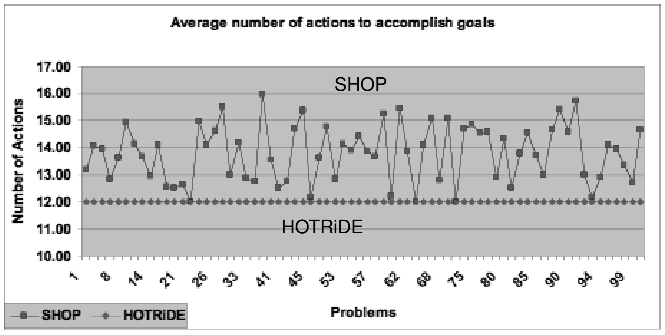

Figure 5: Average number of actions to accomplish goals in our NEO planning domain.

## Related Work

The HTN planning formalism is introduced in (Erol, Hendler, & Nau 1994; 1996). The SHOP algorithm is a domain-independent HTN-based planning algorithm that is presented in (Nau *et al.* 1999). Recently, there has been an interest from several research to develop planning system that include plan adaptation and replanning capabilities to deal with any unexpected events while the plans are executed in a real-world environments. Among those, the most related previous work is a recent study reported in (Warfield *et al.* 2007) on a planning algorithm called RepairSHOP, which is designed for replanning with HTNs. In terms of the basic plan-adaptation ideas, RepairSHOP is very similar to HOTRiDE since both approaches aim to generate plans using HTN task decompositions and replan when something unexpected happens during execution and the plan fails. The difference between the two approaches is the underlying techniques to deal with replanning. In HOTRiDE, we focused on simple, and hopefully, efficient ways to keep track of dependencies among the tasks that the SHOP planner would generate during planning and aimed to modify those dependencies as much as possible. RepairSHOP uses a more general and expressive data structure called, GoalGraph, to accomplish the same objective. Although using a GoalGraph would enable planner to produce explanations for task dependencies and replan based on those explanations, it is not clear to us how the two approaches would compare in terms of kinds of dependencies they can deal with and the efficiency they deal with them. We consider this comparison as an important next step in our research and leave it to a future study due to space and time limitations here.

The work reported in (Wang & Chien 1997) describes a planning algorithm that allows replanning using Hierarchical Task Networks (HTN) as formalized in(Erol, Hendler, & Nau 1994). This paper proposes an extension to the DPLAN algorithm (Chien *et al.* 1996) in order to perform replanning. Like HOTRiDE, the technique presented in the paper aims for the following: (1) to interleave plan generation and execution, (2) to be able to replan when unexpected changes occur in dynamic environments, and (3) to make minimal revisions to the original plan. Unlike HOTRiDE, this technique assumes that some of the predicates in the initial state of the world can be restored at the state the plan fails. We did not make this assumption since we believe that it may not be applicable in some real world domains such as the NEO-Trans-1.1 planning domain that we considered for our experiments and described above.

Continuous Planning and Execution Framework (CPEF) is introduced in (Myers 1999). CPEF is described as a first step in the development of a planning system that employs plan generation, execution, monitoring, and repair capabilities to solve complex tasks in unpredictable and dynamic environments. CPEF assumes that plans are dynamic, that is, they must be evolving in response to the changes in the environment. It is reported in the paper that CPEF leverages several AI techniques as components. Mainly, CPEF employs HTN planning and plan repair capabilities by the help of the SIPE-2 system (Wilkins 1988).

The Advisable Planner (AP) (Myers 1996) supports user provision of advice to guide the process of plan generation. The Procedural Reasoning System (PRS) (Georgeff & Ingrand 1989) is employed to have a reactive plan execution control that integrates goal-oriented and event-driven activity in a flexible hierarchical framework.

A survey of existing replanning techniques are described in (Russell & Norvig 2003). Other studies related with replanning and plan repair in planning dynamic environments include (Schoppers 1987), (Verfaillie & Schiex 1994), and (Bernard *et al.* 1998).

## Conclusion

In this paper, we have described an HTN-based planning system, called HOTRiDE, which is capable of plan generation, execution, and repair. Given an HTN planning problem, HOTRiDE uses a variant of the well-known planner SHOP (Nau *et al.* 1999). to generate a plan for that planning problem. Then, HOTRiDE starts to execute that plan in the world. If an action fails during execution, HOTRiDE attempts to revise the original plan by preserving the actions accomplished so far as much as possible and modifiying only the necessary parts of the original plan. The results of our preliminary experiments demonstrated such behavior: the length of the plans found by HOTRiDE in the experiments was always optimum which indicates that HOTRiDE preserves the goals accomplished previously and makes minimum number of changes in the original plan.

**Acknowledgments.** This work was supported in part by DARPA's Transfer Learning and Integrated Learning programs, NSF grant IIS0412812, and AFOSR grants FA95500510298, FA95500610405, and FA95500610295. The opinions in this paper are those of the authors and do not necessarily reflect the opinions of the funders.

## References

Allen, J. F.; Hendler, J.; and Tate, A., eds. 1990. *Readings in Planning*. Morgan Kaufmann.

Bernard, D.; Dorais, G.; Fry, C.; Jr, E.; Kanefsky, B.; Kurien, J.; Millar, W.; Muscettola, N.; Nayak, P.; Pell, B.; Rajan, K.; Rouquette, N.; Smith, B.; and Williams, B. 1998. Design of the remote agent experiment for spacecraft autonomy. In *IEEE Aerospace Conference*.

Chien, S.; Govindjee, A.; Estlin, T.; Wang, X.; and Jr., R. H. 1996. Integrating hierarchical task network and operator-based planning techniques to automate operations of communications antennas. IEEE Report.

Erol, K.; Hendler, J.; and Nau, D. S. 1994. HTN planning: Complexity and expressivity. In *AAAI*.

Erol, K.; Hendler, J.; and Nau, D. S. 1996. Complexity results for hierarchical task-network planning. *AMAI* 18:69–93.

Georgeff, M., and Ingrand, F. 1989. Decision-making in an embedded reasoning system. In *IJCAI*.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning: Theory and Practice*. Morgan Kaufmann.

Muñoz-Avila, H.; Aha, D. W.; Nau, D. S.; Weber, R.; Breslow, L.; and Yaman, F. 2001. SiN: Integrating case-based reasoning with task decomposition. In *IJCAI*.

Myers, K. 1996. Advisable planning systems. In Tate, A., ed., *Advanced Planning Technology*. AAAI Press.

Myers, K. L. 1999. A continuous planning and execution framework. *AI Magazine* 63–69.

Nau, D. S.; Cao, Y.; Lotem, A.; and Muñoz-Avila, H. 1999. SHOP: Simple hierarchical ordered planner. In Dean, T., ed., *IJCAI*, 968–973. Morgan Kaufmann Publishers.

Russell, S., and Norvig, P. 2003. *Artificial Intelligence, A Modern Approach (Second Edition)*. Upper Saddle River, NJ: Prentice-Hall.

Sacerdoti, E. 1975. The nonlinear nature of plans. In *IJCAI*, 206–214. Reprinted in (Allen, Hendler, & Tate 1990), pp. 162–170.

Schoppers, M. 1987. Universal plans for reactive robots in unpredictable environments. In *IJCAI*, 1039–1046.

Tate, A. 1977. Generating project networks. In *IJCAI*, 888–893.

Verfaillie, G., and Schiex, T. 1994. Solution reuse in dynamic constraint satisfaction problems. In *AAAI*, 307–312.

Wang, X., and Chien, S. 1997. Replanning using hierarchical task network and operator-based planning. In *ECP*.

Warfield, I.; Hogg, C.; Lee-Urban, S.; and Munoz-Avila, H. 2007. Adaptantion of hierarchical task network plans. In *FLAIRS-2007*.

Wilkins, D. E. 1988. *Practical Planning: Extending the Classical AI Planning Paradigm*. San Mateo, CA: Morgan Kaufmann.