Plan Recovery in Reactive HTNs Using Symbolic Planning

Lydia Ould Ouali[1], Charles Rich[2] and Nicolas Sabouret[1]

[1] LIMSI-CNRS, UPR 3251, Orsay, France
Université Paris-Sud, Orsay, France
{ouldouali, nicolas.sabouret}@limsi.fr
[2] Worcester Polytechnic Institute
Worcester, Massachusetts, USA
rich@wpi.edu

Abstract. Building formal models of the world and using them to plan future action is a central problem in artificial intelligence. In this work, we combine two well-known approaches to this problem, namely, reactive hierarchical task networks (HTNs) and symbolic linear planning. The practical motivation for this hybrid approach was to recover from breakdowns in HTN execution by dynamically invoking symbolic planning. This work also reflects, however, on the deeper issue of tradeoffs between procedural and symbolic modeling. We have implemented our approach in a system that combines a reactive HTN engine, called Disco, with a STRIPS planner implemented in Prolog, and conducted a preliminary evaluation.

1 Introduction

Hierarchical task networks (HTNs) are widely used for controlling intelligent agents and robots in complex, dynamic environments. There are many different formalizations and graphical notations in use for HTNs. In this paper we use the simple tree notation shown in Fig. 1, which we will explain in detail in Section 4.1. HTNs are typically hand-authored and can be quite large, with five or more levels of task hierarchy and dozens or even hundreds of tasks at the leaves.

All HTNs share the basic structure of decomposing tasks into sequences (or sometimes partially ordered sets) of subtasks, with alternative decompositions (sometimes called recipes) for different situations. In addition to the decomposition tree structure, most HTNs also have conditions, such as preconditions and postconditions, associated with nodes in the tree to control execution of the HTN.

HTNs were originally a hierarchical extension of classical linear (e.g., STRIPS [4]) plans, and as in classical plans, the conditions associated with tasks were *symbolic*, i.e., they were written in some kind of formal logic and logical inference was used to reason about them. Later, in response to the difficulties of symbolic modeling (see Section 3) a variant, called *reactive* HTNs, was developed in which the conditions are *procedural*, i.e., they are written in a programming language and evaluated by the appropriate programming language interpreter. The idea

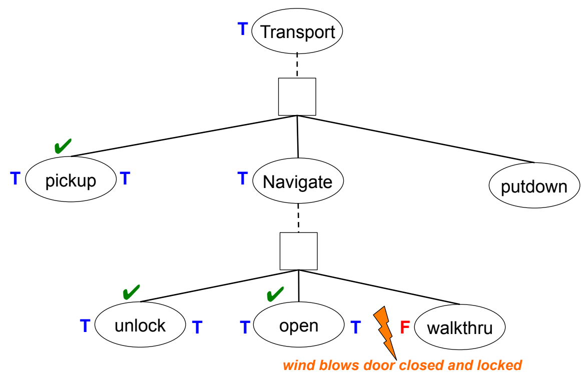


Fig. 1: Breakdown in HTN execution after wind blows door closed and locked. Check marks indicate successfully executed tasks; "T" indicates a condition that has been evaluated and returned true; "F" indicates a condition that has returned false.

of reactive HTNs has also been used in game development, where they are called behavior trees.[3]

This work is focused on reactive HTNs, and specifically on recovering from breakdowns in their execution. The basic idea is to add a small proportion of symbolic conditions to a reactive HTN in order to support a linear planner performing local plan recovery. Section 2 below starts with a simple, motivating example.

The problem of plan recovery has been studied in symbolic HTNs (see [1, 2, 9, 10]). This work is inspirational, but not directly relevant, because these plan repair techniques rely upon *all* of the conditions in the HTN being symbolically expressed, which obviates the use of a reactive HTN.

Others have proposed adding some kind of symbolic planning to reactive HTNs. For example, Firby [5] proposed using a planner to reorder tasks in the HTN execution or to help choose between alternative decompositions. Brom [3] proposed using planning to help execute tasks with time constraints. However, no one has yet developed a complete hybrid procedural/symbolic algorithm (see Section 4.2) similar to ours.

Finally, this work is preliminary because, although we have implemented and tested our algorithm on synthetically generated data (see Section 5), how well it will work in practice is still an open question.

2 A Motivating Example

To further introduce and motivate our work, we first consider a small, intuitive example of reactive HTN execution breakdown and recovery. The basic idea of this example, shown in Fig. 1 is that a robot has been programmed using an HTN to transport an object through a locked door. In this HTN, the toplevel

[3] See http://aigamedev.com/open/article/popular-behavior-tree-design

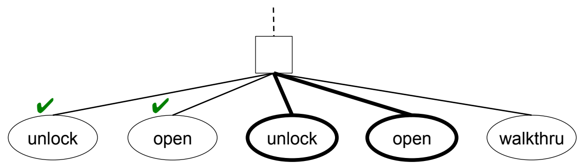


Fig. 2: Sequence of two primitive tasks (in bold) added to plan for Navigate to recover from breakdown in Fig. 1.

task, Transport, is decomposed into three steps: pickup, Navigate and putdown. Navigate is further decomposed into three steps: unlock, open and walkthru. Each of these tasks is represented by an oval in Fig. 1. (The square boxes in the HTN are there to support alternative decompositions, which can be ignored in this example.)

At the moment in time depicted in Fig. 1, the robot has successfully picked up the object, unlocked the door and opened it. However, before the precondition of the walkthru step is evaluated, the wind blows the door closed and the door locks. The walkthru precondition checks that the door is open and thus returns false. At this point, there are then no executable tasks in the HTN, which is what we call a *breakdown*.

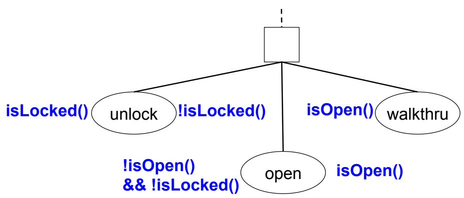
Such breakdowns are not unusual in reactive HTNs, especially when they are executing in complex, dynamic environments. In fact, something similar to this actually happened recently to the winning robot in the prestigious DARPA Robotics Challenge[4] (emphasis added): "However, team Schaft lost points when *a gust of wind blew a door out of the robot's hand and the robot was unable to exit a vehicle* after navigated a driving course successfully." It can be hard to anticipate all possible things that can go wrong; and trying to incorporate all possible recovery plans into the HTN in advance can lead to an explosion of programming effort.

However, looking at this breakdown in particular, the recovery solution, shown in Fig. 2, is obvious, namely to unlock and open the door. Furthermore, this would be a trivial problem for a symbolic linear (e.g., STRIPS) planner to solve if only the pre- and postconditions of the relevant primitives were specified symbolically.

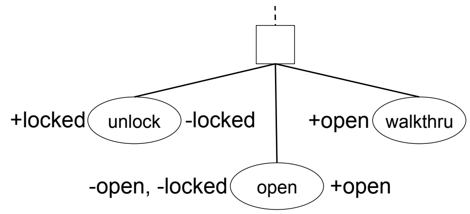
In a reactive HTN, pre- and postconditions are written in a procedural (programming) language and evaluated by the appropriate programming language interpreter. Fig. 3a shows the relevant procedural conditions in the Navigate plan as they might typically be written, for example, in JavaScript. For example, "isOpen()" would call code in the robot's sensory system to check whether the door is currently open. In comparison, Fig. 3b shows how the same primitive tasks would typically be formalized for a STRIPS planner using symbolic features.

Suppose that when the breakdown in Fig. 1 occurred, the HTN execution engine somehow had available the symbolic modeling knowledge shown in Fig. 3b. Recovering from the breakdown could then be formulated as a STRIPS plan-

[4] https://herox.com/news/148-the-darpa-robotics-challenge



(a) Procedural conditions



(b) Symbolic features for conditions

Fig. 3: Procedural versus symbolic conditions for Navigate plan.

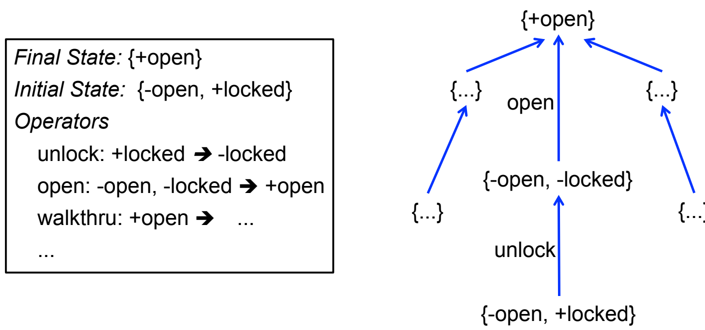


Fig. 4: Recovery from breakdown in Fig. 1 as a STRIPS planning problem.

ning problem (see Fig. 4) in which the initial state is the *current* world state, i.e., the door is not open and locked, and the final state is the failed precondition of walkthru, i.e., the door is open. Simple backward-chaining would then quickly find the solution sequence of operators, namely unlock followed by open. This recovery plan could then be spliced into the HTN as shown in Fig. 2 and execution could continue.

The goal of this paper is to generalize the solution to this example problem into an application-independent plan recovery algorithm and associated modeling methodology for reactive HTNs, as described in the next two sections.

3 Procedural versus Symbolic Modeling

A reasonable first question to ask after reading the motivating example above is why not just use a symbolic planner instead of an HTN to control the robot in the first place, and apply well-known replanning approaches when a breakdown occurs?

Answering this question leads directly to the issue of modeling. Symbolic planners, such as STRIPS, require *complete and correct* symbolic descriptions of all of the primitive tasks (operators) in the problem domain. Different planners use different symbolic formalisms to represent this knowledge, such as the add/delete lists shown in Fig. 3b, PDDL [6], and others. However, what all symbolic planners have in common is that if these symbolic descriptions are incorrect or incomplete (relative to reality), then the generated plans will fail—the poorer the correspondence to reality, the less reliable the plans will be.

Unfortunately, artificial intelligence research has shown that producing complete and correct symbolic descriptions of complex real-world domains is extremely hard and, for practical purposes, often impossible. Certainly the textbook example in Fig. 4 is easy to model symbolically, but no one participating in the DARPA Robotics Challenge seriously considered symbolically modeling the complete task domain.

The difficulty of symbolic modeling is why reactive HTNs were invented. The knowledge in a reactive HTN is encoded in two places: in the decomposition structure of the tree and in the code for the procedural conditions (especially the applicability conditions for decomposition choices, which will be explained in the next section).

So is it easier to model complex real-world domains using reactive HTNs than symbolically? As a practical matter, the answer appears to be yes, since reactive HTNs are commonly used in such applications. Our guess is that there are two main reasons for this. First, it is well known that hierarchy helps people organize their thinking to deal with complexity. Second, a procedural condition, such as a precondition, is only applied to the *current* world, whereas symbolic descriptions are essentially axioms that must be true in all possible worlds.

But of course, as we saw above, reactive HTNs can break down, which leads us to the hybrid approach, described in the next section, in which a reactive HTN is augmented with some symbolic conditions to aid specifically with recovery.

4 A Hybrid Approach

In this section we generalize the motivating example in Section 2 in two ways by considering: (1) other types of breakdowns and (2) a larger set of possible final states. We will first present a general plan recovery algorithm and then discuss the modeling methodology that goes along with it.

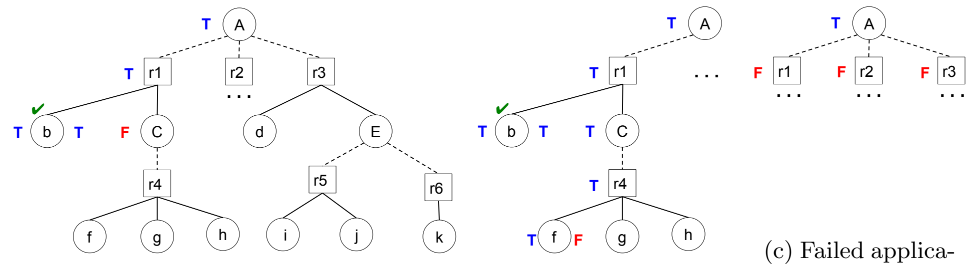
4.1 Reactive HTNs

For the purpose of this work, a reactive HTN, such as the general example in Fig. 5a, is formally a bipartite tree with alternating levels of *task* nodes (shown as circles or ovals) and *decomposition* (recipe) nodes (shown as squares), with a task node at the root and task nodes at the leaves. The tasks at the leaves are called *primitive*; the other tasks are called *abstract*.

Associated with the nodes of the tree are three kinds of boolean-valued procedures, each of what are evaluated in the current world state. Task nodes have an optional *precondition* and/or *postcondition*. Every decomposition node has an *applicability condition*.

Reactive HTNs are basically a kind of and/or tree, where the task nodes are "and" and the decomposition nodes are "or." Execution is a depth-first, left-to-right traversal of the tree starting at the root, with various conditions being evaluated along the way, as described below.

If the current execution node is a task, then its precondition, if any, is first evaluated. If the precondition returns false, then execution is halted (a breakdown); otherwise execution continues. If the task is primitive, then it is directly



(a) Failed precondition (b) Failed postcondition (c) Failed applicability conditions

Fig. 5: Examples of the three types of breakdown in reactive HTN execution.

executed (typically changing the world state); otherwise (i.e., for abstract tasks) the applicability conditions of the children (decomposition) nodes are evaluated in order until the first one that returns true—execution continues with this decomposition node. If all of the applicability conditions of the children return false, then execution is halted (a breakdown).

When execution of a task node is completed, its postcondition, if any, is evaluated. If the postcondition returns false, then execution is halted (a breakdown); otherwise execution continues.

If the current execution node is a decomposition, then the children (task) nodes are executed in order.

Fig. 5 summarizes the three types of execution breakdowns that are possible in reactive HTN execution. The motivating example in Section 2 was a failed precondition, as in Fig. 5a. Notice that this taxonomy does not distinguish between different possible underlying causes of a breakdown. A breakdown can be caused by an external, i.e., unmodeled, agency unexpectedly changing the environment (e.g., the wind in Section 2); or it can be due to a programming bug, such as incorrect tree structure or an incorrectly coded condition. The fact these these different causes are indistinguishable in the breakdown is an inherent limitation of reactive HTNs.

4.2 Plan Recovery Algorithm

The most significant generalization of the plan recovery algorithm over the motivating example in Section 2 concerns the choice of final state for the linear planning problem. In the example (see Fig. 4), the final state is the failed precondition of the walkthru primitive. However, there are other recovery possibilities that might also make sense.

For example, suppose there was a symbolic postcondition on walkthru that specified that the robot is located in the room on the other side of the door. After the breakdown, another way to recover could be for the robot to successfully find a plan (using other operators than just unlock and open) to achieve this condition, e.g., by going out another door of the current room and taking a different hallway to the original destination room. We will call this process making the postcondition of walkthru the *target* of a candidate recovery planning problem.

Continuing with this line of thought, suppose that there was no symbolic postcondition provided for walkthru, but the symbolic postcondition of Navigate specified the desired location of the robot. In that case, the postcondition of Navigate would be a good candidate recovery target.

Similarly, suppose the original breakdown in the example had instead occurred due to the postcondition of unlock failing. In this situation, the symbolic precondition of walkthru and the symbolic postconditions of walkthru and Navigate, if they are provided, are still good recovery targets.

Based on this reasoning, in the algorithm below we consider the largest possible set of pre- and postconditions in the tree as candidate recovery targets, excluding only those that have already been used in the execution process and have evaluated to true. We suspect this is an over-generalization, but need more practical experience to determine a better approach.

The recovery target issue for applicability conditions is somewhat different. The only time that an applicability condition should be a recovery target is when its and all of its siblings' conditions have evaluated to false, as in Fig. 5c.

Fig. 6 shows the pseudocode for the hybrid system we have designed. The toplevel procedure, EXECUTE, executes an HTN until either it is successfully completed, or there is a breakdown with no possible recovery. The plan recovery algorithm starts at line 5. The main subroutine, FINDCANDIDATES, recursively walks the HTN tree, accumulating candidate target conditions for the recovery planning. Notice that SYMBOLICPLANNER is not defined here, since any symbolic linear planner can be used (our implementation is described in Section 5). Notice also that since the set of operators used for symbolic planning doesn't change during execution of the HTN, it is not an explicit argument to the symbolic planner (see further discussion regarding symbolic operators in Section 4.3).

In more detail, notice on line 6 that our approach requires a method for computing from the current world state an initial state representation in the formalism used by the symbolic planner. For example, for the STRIPS planner in Section 2 this means that for every feature, such as "open," there must be an associated procedure, such as "isOpen()," to compute its value in the current world state. This association is a basic part of the hybrid modeling methodology discussed in next section.

Notice on line 8 that the candidate conditions are sorted by distance from the current node in the tree (closest first), using a simple metric such as the length of the shortest path between them in the undirected graph. The reason for this is to give preference to recovery plans that keep more closely to the structure of the original HTN. We do not yet have any experience with how well this heuristic works in practice.

Finally in EXECUTE, notice on line 12 that when a recovery plan is found, it must be properly spliced into the HTN. In the simple example in Fig. 2, this is merely a matter of inserting a sequence of nodes as children of the common parent between the initial and final nodes. However, if the initial and final nodes are more distant in the tree, more complicated changes are needed to replace the intervening tree structure with the new plan.

```
procedure Execute(htn)
    while htn is not completed do
        current ← next executable node in htn
        if current ≠ null then execute current
        else  [breakdown occurred]
            initial ← symbolic description of current world state
            candidates ← FindCandidates(htn)
            sort candidates by distance from current
            for final ∈ candidates do
                plan ← SymbolicPlanner(initial, final)
                if plan ≠ null then
                    splice plan into htn between current and final
                    continue while loop above
            Recovery failed!

procedure FindCandidates(task)
    conditions ← ∅
    pre ← symbolic precondition of task
    if pre ≠ null ∧ procedural prec of task has not evaluated to true then
        add pre to conditions
    post ← symbolic postcondition of task
    if post ≠ null ∧ procedural postc of task has not evaluated to true then
        add post to conditions
    applicables ← ∅
    allFalse ← true
    for decomp ∈ children of task do
        for task ∈ children of decomp do FindCandidates(task)
        if allFalse then
            if procedural appl condition of decomp has evaluated to false then
                app ← symbolic applicability condition of decomp
                if app ≠ null then add app to applicables
            else allFalse ← false
    if allFalse then add applicables to conditions
    return conditions
```

Fig. 6: Pseudocode for hybrid reactive HTN execution and recovery system.

The code in the definition of FindCandidates closely follows the discussion above regarding possible target conditions.

4.3 Modeling Methodology

The hybrid system described above tries to take advantage of whatever symbolic knowledge is provided by the HTN author. Notice that for the pre- or postcondition of each task node there are four possibilities: no specified condition, only a procedural condition, only a symbolic condition, or both. Since applicability conditions are not optional, there are only two possibilities for decomposition nodes: only a procedural condition, or a procedural and a symbolic condition.

As we have argued earlier, symbolic modeling is very difficult in general. The reason an author is using a reactive HTN in the first place is likely because it was not practical to fully model the domain symbolically. Two key methodological issues are therefore where to best invest whatever effort is available for symbolic modeling, and how to make the overall process of mixed procedural and symbolic modeling as convenient as possible for the HTN author. We will share some ideas below, but this is the area in which our work is also preliminary.

Our initial intuition, illustrated by the example in Section 2, is to concentrate symbolic modeling on the primitive tasks. This is because we expect symbolic plan recovery to be most successful as a local planning strategy.

Which pre- and postconditions are provided symbolically also has implications for the set of operators used by the symbolic planner. Only a task with both a symbolic precondition and postcondition can be included in the operator set. However, the planning operators need not be restricted to primitive tasks. If an abstract task is fully specified symbolically, it can in principle be included in a linear recovery plan by using its already defined decompositions (and hoping they are appropriate in the new context).

Finally, we believe there is the opportunity for tools to make the hybrid modeling process less onerous. For example, we are planning a simple design-time tool that recognizes common coding conventions, such as in Fig. 3a, and automatically produces corresponding symbolic conditions, such as in Fig. 3b. Run-time tools can also keep track of breakdowns and advise where additional symbolic knowledge may be useful.

5 Implementation and Evaluation

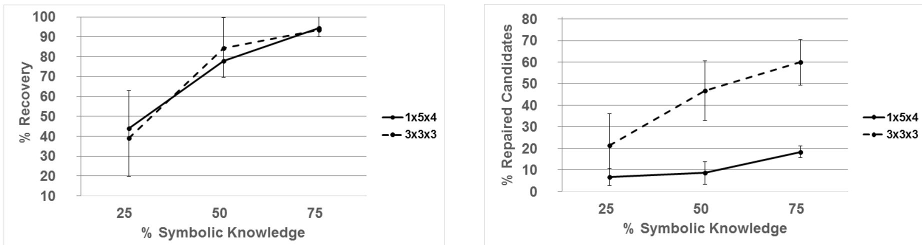
Fig. 7: Discolog implementation.

We have implemented the hybrid system described above in pure Java (see Fig. 7) using the ANSI/CEA-2018 standard [7] for reactive HTNs and Disco [8] as the HTN execution engine. For the symbolic planner, we have used a simple implementation of STRIPS running in a pure Java implementation of Prolog.[5] Using Prolog will facilitate adding additional symbolic reasoning rules to the planning process.

The ultimate evaluation of our proposed new approach is to build several agents that operate in complex, dynamic real-world environments and to see how easy they were to build and how robustly they performed. In the meantime, however, we tested our system on synthetically generated HTNs with different levels of symbolic knowledge. Our simple hypothesis was that the more symbolic knowledge provided, the better the recovery algorithm would perform.

Fig. 8 shows the results of our experiments, which confirmed this hypothesis. We tested trees of two RxSxD sizes, 3x3x3 and 1x5x4, where R is the decomposition (recipe) branching factor, S is task (step) branching factor, and D is the task depth (see Fig. 9). For each test, we randomly sampled from (the very large space of) all possible combinations of symbolic knowledge at three overall levels:

[5] See http://tuprolog.apice.unibo.it



(a) Recovery/breakdown ratio (b) Proportion of candidates repaired

Fig. 8: Results of testing on synthetically generated HTNs with different levels of symbolic knowledge.

25%, 50% and 75% (percentage of conditions in the tree that are symbolically specified). We did not test larger trees because the experimental running times became too long.

Fig. 8a graphs how the proportion of breakdowns that are successfully recovered increases as the symbolic knowledge increases. In Fig. 8b, we were interested in the proportion of planning problems submitted to the planner which it solved, which also increased as the symbolic knowledge increased. (For this experiment, we made a small modification to the algorithm to prevent it from stopping at the first solved problem.)

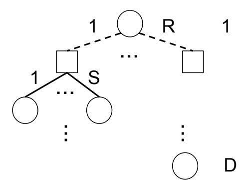


Fig. 9: RxSxD

References

1. N. F. Ayan, U. Kuter, F. Yaman, and R. Goldman. HOTRiDE: Hierarchical ordered task replanning in dynamic environments. In *Planning and Plan Execution for Real-World Systems–Principles and Practices for Planning in Execution: Papers from the ICAPS Workshop. Providence, RI*, 2007.
2. G. Boella and R. Damiano. A replanning algorithm for a reactive agent architecture. In *Artificial Intelligence: Methodology, Systems, and Applications*, pages 183–192. Springer, 2002.
3. C. Brom. Hierarchical reactive planning: Where is its limit. In *Proceedings of MNAS: Modelling Natural Action Selection*, Edinburgh, Scotland, 2005.
4. R. E. Fikes and N. J. Nilsson. STRIPS: A new appraoch to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.
5. R. Firby. An investigation into reactive planning in complex domains. In *AAAI*, pages 202–206, 1987.
6. M. Ghallab et al. PDDL–The planning domain definition language. 1998.
7. C. Rich. Building task-based user interfaces with ANSI/CEA-2018. *IEEE Computer*, 42(8):20–27, August 2009.
8. C. Rich and C. L. Sidner. Using collaborative discourse theory to partially automate dialogue tree authoring. In *Proc. Int. Conf. on Intelligent Virtual Agents*, pages 327–340, Santa Cruz, CA, September 2012.
9. R. Van Der Krogt and M. De Weerdt. Plan repair as an extension of planning. In *ICAPS*, pages 161–170, 2005.
10. I. Warfield, C. Hogg, S. Lee-Urban, and H. Munoz-Avila. Adaptation of hierarchical task network plans. In *FLAIRS*, pages 429–434, 2007.