# 12

# Exploring HTN Planners through Example

*Troy Humphreys*



## 12.1 Introduction

As programmers we may find ourselves perpetually looking for that "better solution" to whatever problems we've encountered—better performance, maintainability, or usability. It's only after we implement those solutions that we understand some of the nuances that come with them. Often, these nuances might be the deciding factor in what solution we go with.

In AI development, a common problem to solve is behavior selection. There are many solutions to this problem, such as finite-state machines, behavior trees, utility-based selection, neural networks, and planners. This article aims to explore the nuances of a type of planner called *hierarchical task networks* (HTN) by using real world examples that one can run into during development.

Planning architectures such as HTN take a problem as input and supply a series of steps that solves it. In HTN terms, the series of steps is called a *plan*. What makes hierarchical

task networks unique to other planners is that it allows us to represent the problem as a very high level task, and through its planning process, recursively breaks this task into smaller tasks. When this process is completed, we are left with a series of atomic tasks that represent a plan. Breaking up high level tasks into smaller ones is a very natural way of solving many sorts of problems. In our case, the problem is simply "figuring out what to do." With a high degree of modularity and fast run time execution, HTNs make an attractive choice as a solution. For those of you that are familiar with behavior trees, these benefits might also seem familiar. Unlike behavior trees, however, HTN planners can reason about the *effects* of possible actions. This ability to reason about the future allows HTN planners to be incredibly expressive in how they describe behavior.

There have been many different systems used for HTN planning [Erol 95]. The system we will be exploring is the system that we used on *Transformers: Fall of Cybertron* [HighMoon 12], which is based on a *total-order forward decomposition* planner. The following example will walk through some of the challenges we faced and the benefits we received during development by using a simplified, fictional example.

For our example, we will use a troll NPC called a "Trunk Thumper." The designer's initial description is that he's a big, nasty, lumbering troll that patrols its numerous bridges and attacks passing enemies with a large tree trunk. And just like development in the real world, this design is bound to change.

## 12.2 Building Blocks of HTN

Before building the behavior for our Trunk Thumper, it's important to go over the basic building blocks of hierarchical task networks so you can get an idea of how it all works. An NPC, in our case the Trunk Thumper, has a *planner* that uses a *domain* and *world state* to build a sequence of tasks called a *plan*. This plan will be run by the Trunk Thumper's *plan runner.* The world state is updated by the NPC's sensors and by the successfully completed tasks executed by the plan runner. A diagram of the system is Figure 12.1.

### 12.2.1 The World State

Like any type of behavior algorithm, hierarchical task networks need some type of knowledge representation that describes the current problem space. In the case of our Trunk Thumper, this would be a representation that describes what our troll knows about the world and himself in it. Other types of behavior algorithms might query the actual state of different objects in the world. For example, query an object's location or their health. But with HTN, this information needs to be encoded into something it can understand, called the *world state*. The world state is essentially a vector of properties that describe what our HTN is going to reason about. Here is some simple pseudocode.

```
enum EHtnWorldStateProperties
{
    WsEnemyRange,
    WsHealth,
    WsIsTired,
    …
}
```

```
enum EEnemyRange
{
    MeleeRange,
    ViewRange,
    OutOfRange,
    …
}
vector<byte> CurrentWorldState;
EEnemyRange currentRange = CurrentWorldState[WsEnemyRange];
CurrentWorldState[WsEnemyRange] = MeleeRange;
```

As you can see from the pseudocode, world state can simply be an array or vector indexed by an `enum` such as `EhtnWorldStateProperties`. Each entry in the world state can have its own set of values. In the case of `WsIsTired`, the byte can represent the Boolean values zero and one. With `WsEnemyRange`, the values in the `enum` `EEnemyRange` are used. It's important to note that the world state only needs to represent what is needed for the HTN to make decisions. That's why `WsEnemyRange` is represented by abstract values, instead of the actual range. The goal of the world state isn't to represent every possible state of every possible object in the game. It only needs to represent the problem space that our planner needs to make decisions. What this means for our example, of course, is that it only needs to represent what the Trunk Thumper needs to make decisions.

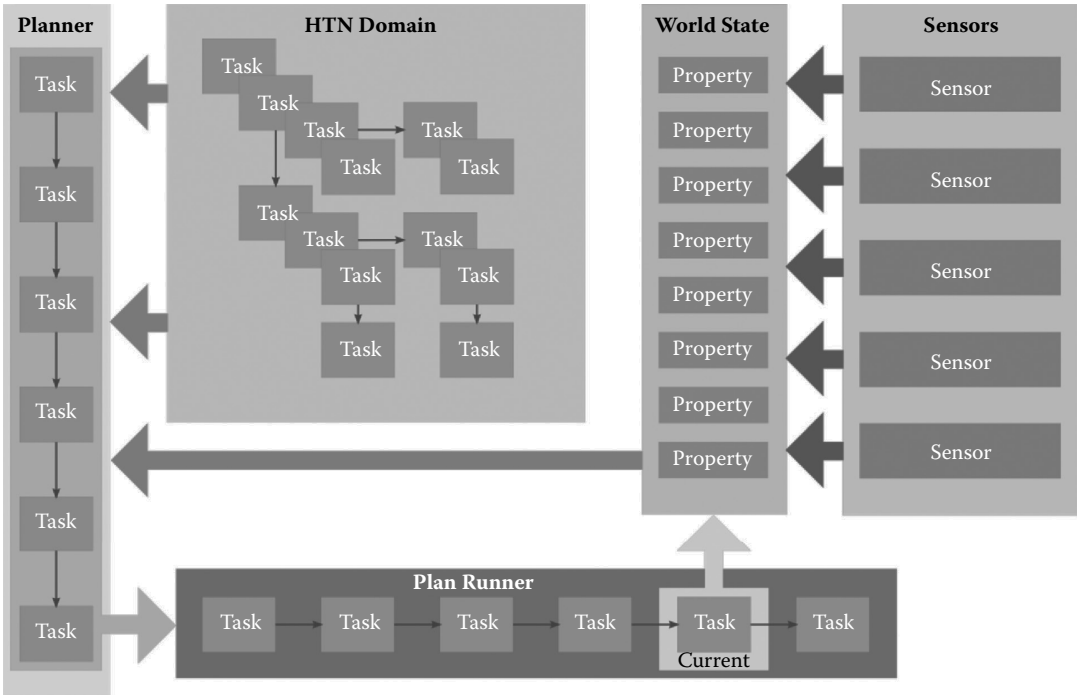


Figure 12.1

Overview of the HTN system.

### 12.2.2 Sensors

If you recall, an HTN outputs a plan or sequence of tasks. These tasks will have an effect on the world state as it is executed. There are outside influences such as the player or other NPCs, however, that will affect the world state as well. For example, both the enemy and the troll can affect the world state property, `WsEnemyRange`. The tasks executed by the troll could update this property if they were to move the troll. There is nothing in the HTN planner to handle changes produced by the enemy moving, however.

There are many different ways these changes can be translated into the world state. One preferable way is a simple sensor system that manages a set of time-sliced sensors. Each sensor can manage different world state properties. Examples of some different sensors include vision, hearing, range, and health sensors. These sensors would work the same as in any other AI system, with an added step of encoding their information into the world state that our HTN can understand.

### 12.2.3 Primitive Tasks

As we mentioned already, a hierarchical task network is made up of tasks. There are two types of tasks that are used to build a HTN, called *compound tasks* and *primitive tasks*. Primitive tasks represent a single step that can be performed by our NPC. In our Trunk Thumper example, uprooting a tree or attacking with a trunk slam would be examples of primitive tasks. A set of primitive tasks is the *plan* that we are ultimately getting out of the HTN. Primitive tasks are comprised of an *operator* and sets of *effects* and *conditions*.

In order for a primitive task to execute, its set of conditions must be valid. This allows the task's implementer to ensure the correct conditions are met for the task to run. It's important to note that a primitive task's conditions are not a requirement for the implementation of HTN. They are, however, recommended to reduce the redundancy of checks that would be needed higher in the HTN hierarchy. In addition, doing so will avoid potential bugs that can arrive from having to do these checks in multiple places.

A primitive task's effects describe how the success of the task will affect the NPC's world state. For example, the task `DoTrunkSlam` executes the troll's tree trunk melee attack and results in the troll becoming tired. The `DoTrunkSlam`'s *effects* are the manner in which we describe this result. This allows the HTN to reason about the "future" as was mentioned earlier. Since the effect of "being tired" is represented, our Trunk Thumper is able to make a better decision of what to do after `DoTrunkSlam` or if it's even worth doing so at all.

The *operator* represents an atomic action that a NPC can do. This might sound exactly like the primitive task itself. The difference being that the primitive task along with its effects and conditions describe what the operator means in terms of the HTN we are building.

As an example, let's take the two tasks `SprintToEnemy` and `WalkToNextBridge`. Both of these tasks use the `MoveTo` operator, but the two tasks change the state of our NPC in different ways. On the successful completion of `SprintToEnemy`, our NPC will be at the enemy and tired, specified by the task's effects. `WalkToNextBridge` task's effects would set the NPC's location to the bridge and he'd be a little more bored. As you can see, we are able to use the same *operator* but describe two different uses for it in terms of our network. Here is the notation we will use to describe a primitive task going forward along with the `SprintToEnemy` and `WalkToNextBridge` tasks as an example.

Primitive Task [*TaskName*(*term1*, *term2*,...)]
    Preconditions [*Condition1*, *Condition2*, …]//optional
    Operator [*OperatorName*(*term1*, *term2*,...)]
        Effects [*WorldState op value, WorldState = value, WorldState += value*]//optional
Primitive Task [SprintToEnemy]
    Preconditions [WsHasEnemy == true]
    Operator [NavigateTo(EnemyLoc, Speed_Fast)]
        Effects [WsLocation = EnemyLoc, WsIsTired = true]
Primitive Task [WalkToNextBridge]
    Operator [NavigateTo(BridgeLoc, Speed_Slow)]
        Effects [WsLocation = BridgeLoc, WsBored += 1]

### 12.2.4 Compound Tasks

Compound tasks are where HTN get their "hierarchical" nature. You can think of compound task as a high level task that has multiple ways of being accomplished. Using the Trunk Thumper as an example, he may have the task `AttackEnemy`. Our Thumper may have different ways of accomplishing this task. If he has access to a tree trunk, he may run to his target and use it as a melee weapon to "thump" his enemy. If no tree trunks are available, he can pull large boulders from the ground and toss them at our enemy. He may have a multitude of other approaches if the conditions are right.

In order to determine which approach we take to accomplish a compound task, we need to select the right *method*. Methods are comprised of a set of conditions and tasks. In order for the method to be the selected approach, the conditions are validated against the world state. The set of tasks, or *subtasks*, represent the method's approach. This subtask set can be comprised of primitive tasks as well as compound. The ability to put compound tasks into the methods of other compound tasks is where hierarchical task networks get their hierarchical nature. Here is an example of the notation we will use to describe a compound task going forward.

Compound Task [*TaskName*(*term1*, *term2*,...)]
    Method *0* [*Condition1*, *Condition2*,...]
        Subtasks [*task1*(*term1*, *term2*,...). *task2*(*term1*, *term2*,...),...]
    Method *1* [*Condition1*, *Condition2*,...]
        Subtasks [*task1*(*term1*, *term2*,...). *task2*(*term1*, *term2*,...),...]

In our previous example, using the tree trunk as a melee weapon and throwing boulders are both methods to the `AttackEnemy` compound task. The conditions in which we decide which method to use depend on whether the troll has a tree trunk or not. Here is an example of the AttackEnemy task using the notation above.

Compound Task [AttackEnemy]
    Method 0 [WsHasTreeTrunk == true]
        Subtasks [NavigateTo(EnemyLoc). DoTrunkSlam()]
    Method 1 [WsHasTreeTrunk == false]
        Subtasks [LiftBoulderFromGround(). ThrowBoulderAt(EnemyLoc)]

By understanding how compound tasks work, it's easy to imagine how we could have a large hierarchy that may start with a `BeTrunkThumper` compound task that is broken down into sets of smaller tasks—each of which are then broken into smaller tasks, and so on. This is how HTN forms a hierarchy that describes how our troll NPC is going to behave.

It's important to understand that compound tasks are really just containers for a set of methods that represent different ways to accomplish some high level task. There is no compound task code running during plan execution.

## 12.3 Putting Together an HTN Domain

Now that we have an overview of the main building blocks of HTN, we can build a simple *domain* for our Trunk Thumper to illustrate how it works. A *domain* is the term used to describe the entire task hierarchy. As we mentioned before, our troll has numerous bridges that he actively patrols and attacks enemies with a large tree trunk. We start with a compound task called `BeTrunkThumper`. This root task encapsulates the "main idea" of what it means to be a Trunk Thumper.

```
Compound Task [BeTrunkThumper]
    Method [WsCanSeeEnemy == true]
        Subtasks [NavigateToEnemy(), DoTrunkSlam()]
    Method [true]
        Subtasks [ChooseBridgeToCheck(), NavigateToBridge(), CheckBridge()]
```

As you can see with this root compound task, the first method defines the troll's highest priority. If he can see the enemy, he will navigate using `NavigateToEnemy` task and attack his enemy with the `DoTrunkSlam` task. If not, he will fall to the next method. This next method will run three tasks; choose the next bridge to check, navigate to that bridge, and check the bridge for enemies. Let's take a look at the primitive tasks that make up these methods and the rest of the domain.

```
Primitive Task [DoTrunkSlam]
    Operator [AnimatedAttackOperator(TrunkSlamAnimName)]
Primitive Task [NavigateToEnemy]
    Operator [NavigateToOperator(EnemyLocRef)]
        Effects [WsLocation = EnemyLocRef]
Primitive Task [ChooseBridgeToCheck]
    Operator [ChooseBridgeToCheckOperator]
Primitive Task [NavigateToBridge]
    Operator [NavigateToOperator(NextBridgeLocRef)]
        Effects [WsLocation = NextBridgeLocRef]
Primitive Task [CheckBridge]
    Operator [CheckBridgeOperator(SearchAnimName)]
```

The first task `DoTrunkSlam` is an example of how a primitive task can describe an operator in terms of the HTN domain. Here, the task is really executing an animated attack operator and the animation name is being passed in as a term. The next task

NavigateToEnemy is also an example of this, but on the successful completion of this task, the world state WsLocation is set to EnemyLocRef via the primitive task's effect.

## 12.4 Finding a Plan

With a domain made up of compound and primitive tasks, we are starting to form an image of how these are put together to represent an NPC. Combine that with the world state and we can talk about the work horse of our HTN, the *planner*. There are three conditions that will force the planner to find a new plan: the NPC finishes or fails the current plan, the NPC does not have a plan, or the NPC's world state changes via a sensor. If any of these cases occur, the planner will attempt to generate a plan. To do this, the planner starts with a root compound task that represents the problem domain in which we are trying to plan for. Using our earlier example, this root task would be the BeTrunkThumper task. This root task is pushed onto the TasksToProcess stack. Next, the planner creates a copy of the world state. The planner will be modifying this *working world state* to "simulate" what will happen as tasks are executed.

After these initialization steps are taken, the planner begins to iterate on the tasks to process. On each iteration, the planner pops the next task off the TasksToProcess stack. If it is a compound task, the planner tries to decompose it—first, by searching through its methods looking for the first set of conditions that are valid. If a method is found, that method's subtasks are added on to the TaskToProcess stack. If a valid method is not found, the planner's state is rolled back to the last compound task that was decomposed. We will go into more detail about restoring the planner's state later.

If the next task is primitive, we need to check its preconditions against the working world state. If the conditions are met, the task is added to the final plan and its effects are applied to the working world state. The effects are applied because the planner assumes that task is going to succeed. This allows future methods to consider that new state. If the primitive task's conditions are not met, the planner's state is rolled back such as was done for the compound task. This iteration process is continued until the TasksToProcess stack is empty. Upon completion, the planner will either end up with a list of primitive tasks or the planner will have rolled back far enough that the result was no plan. Below is the example pseudocode that shows this process.

```
WorkingWS = CurrentWorldState
TasksToProcess.Push(RootTask)
while TasksToProcess.NotEmpty
{
    CurrentTask = TasksToProcess.Pop()
    if CurrentTask.Type == CompoundTask
    {
        SatisfiedMethod = CurrentTask.FindSatisfiedMethod(WorkingWS)
        if SatisfiedMethod != null
        {
            RecordDecompositionOfTask(CurrentTask, FinalPlan, DecompHistory)
            TasksToProcess.InsertTop(SatisfiedMethod.SubTasks)
        }
        else
        {
            RestoreToLastDecomposedTask()
        }
    }
```

```
    else//Primitive Task
    {
        if PrimitiveConditionMet(CurrentTask)
        {
            WorkingWS.ApplyEffects(CurrentTask.Effects)
            FinalPlan.PushBack(CurrentTask)
        }
        else
        {
            RestoreToLastDecomposedTask()
        }
    }
}
```

There is a bit of magic going on in the `RecordDepositionOfTask` and `RestoreToLastDecomposedTask` functions that should be explained in more detail. The record function records the planner's state onto the `DecompHistory` stack. This includes the `TasksToProcess` and `FinalPlan` containers as well as the method chosen for the decomposition and its owning compound task. By popping off this recorded state to the planner via the restore function, the planner can backtrack either when a compound task cannot be decomposed or when a primitive's conditions aren't satisfied.

As you might have realized, the planner uses a depth-first search to find a valid plan. This does mean that you may have to explore the whole domain to find a valid plan. However, it's important to remember that you are traversing a *hierarchy* of tasks. This hierarchy allows the planner to cull large sections of the network via the compound task's methods. Because we aren't using a heuristic or cost—such as with A* and Dijkstra searches—we can skip any kind of sorting. These features allowed the HTN planner in *Transformers: Fall of Cybertron* to be considerably faster than our GOAP system used in *Transformers: War for Cybertron* [HighMoon 10].

Now that the planner has been explained, we can expand our example and see how a modified version of the Trunk Thumper domain might decompose (Figure 12.2). This domain's root task is still `BeTrunkThumper`, but the `DoTrunkSlam` is now a compound task. `DoTrunkSlam` has two methods—each doing a different version of the trunk slam. The method's conditions for both compound tasks have been omitted for simplicity. Underneath the domain you can see the planner's iterations going from top to the bottom. For each iteration, you can see the left-most task in the `TasksToProcess` stack being processed.

## 12.5 Running the Plan

Running an HTN plan is pretty straightforward. The NPC's *plan runner* will attempt to execute each primitive task's operator in sequence. As it successfully completes each task, the planner applies the task's effects to the world state. If the task fails for some reason that is specific to the operator it's running, the plan also fails and forces a re-plan.

The plan can also fail if the current or any of the remaining task's conditions become invalid. The plan runner monitors these tasks' preconditions against a "working world state" much like the planner. As it confirms each task's preconditions, its effects are applied

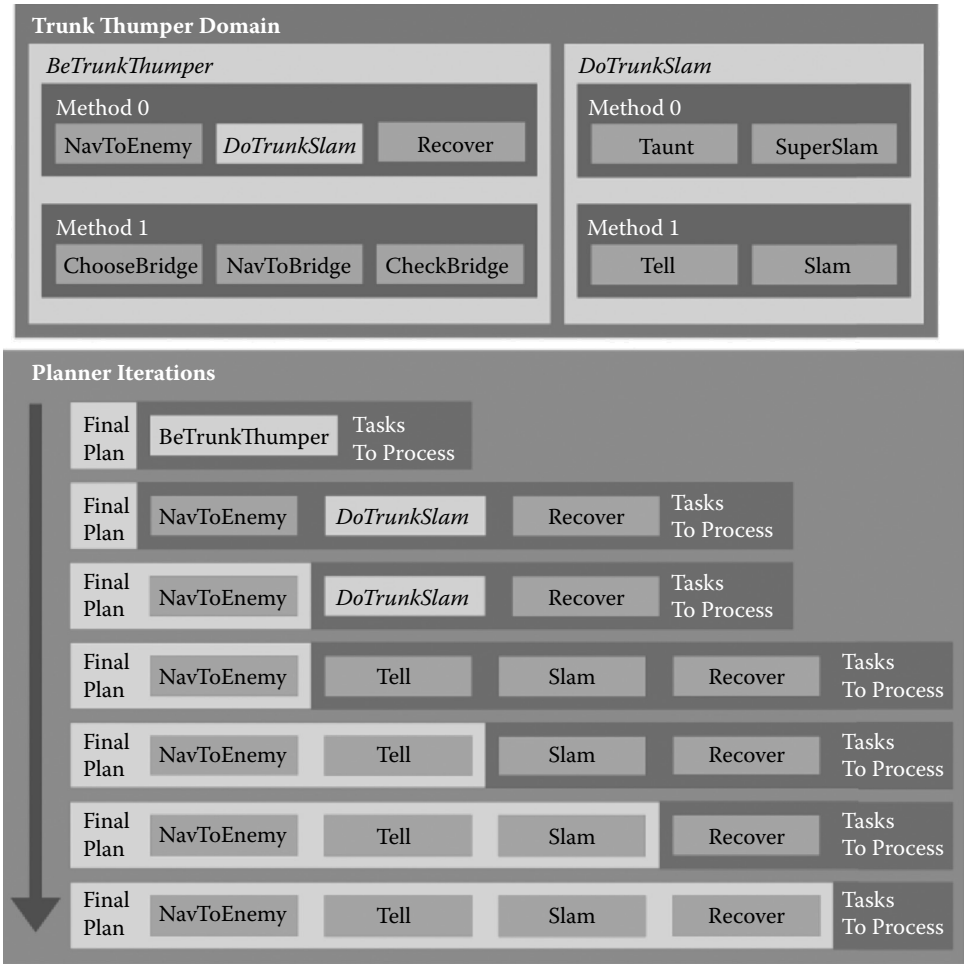


Figure 12.2

Decomposition of the Trunk Thumper domain, showing the resulting plan if BeTrunkThumper.Method0 and DoTrunkSlam.Method.1 were chosen.

to the working world state. It's important that it applies the effects because following task's preconditions might rely on these effects being applied in order to be valid. This plan validation allows the HTN domain to be a bit more expressive and reactive to the changes of the world state.

## 12.6 Using Recursion for Greater Expressiveness

After seeing our troll in game, the designers think that the tree trunk attack is a little overpowered. They suggest that the trunk breaks after three attacks, forcing the troll to search for another one. First we can add the property `WsTrunkHealth` to the world state.

By wrapping up the attack method into its own compound task and adding a little recursion, we will be able to modify the troll's attack behavior. The changed domain would now be:

Compound Task [BeTrunkThumper]
  Method [ WsCanSeeEnemy == true]
    ***Subtasks [AttackEnemy()]*** *// using the new compound task*
  Method [true]
    Subtasks [ChooseBridgeToCheck(), NavigateToBridge(), CheckBridge()]
**Compound Task [AttackEnemy]**//new compound task
  **Method [WsTrunkHealth > 0]**
    Subtasks [NavigateToEnemy(), DoTrunkSlam()]
  **Method [true]**
    **Subtasks [FindTrunk(), NavigateToTrunk(), UprootTrunk(), AttackEnemy()]**
Primitive Task [DoTrunkSlam]
  Operator [DoTrunkSlamOperator]
    **Effects [WsTrunkHealth += -1]**
**Primitive Task [UprootTrunk]**
  **Operator [UprootTrunkOperator]**
    **Effects [WsTrunkHealth = 3]**
**Primitive Task [NavigateToTrunk]**
  **Operator [NavigateToOperator(FoundTrunk)]**
    **Effects [WsLocation = FoundTrunk]**

When our troll can see the enemy, he will attack just as before—only now, the behavior is wrapped up in a new compound task called `AttackEnemy`. This task's high priority method performs the navigate and slam like the original domain, but now has the condition that the trunk has some health. The change to the `DoTrunkSlam` task will decrement the trunk's health every successful attack. This allows the planner to drop to the lower priority method if it has to accommodate a broken tree trunk.

The second method of `AttackEnemy` handles getting a new tree trunk. It first chooses a new tree to use, navigates to that tree, and uproots it, after which it is able to `AttackEnemy`. Here is where the recursion comes in. When the planner goes to decompose the `AttackEnemy` task again it can now consider the methods again. If the tree trunk's health was still zero, this would cause the planner to infinite loop. But the new task `UprootTrunk`'s effect sets `WsTrunkHealth` back to three, allowing us to have the plan `FindTrunk` → `NavigateToTrunk` → `UprootTrunk` → `NavigateToEnemy` → `DoTrunkSlam`. This new domain allows us to reuse methods already in the domain to get the troll back to thumping.

## 12.7 Planning for World State Changes not Controlled by Tasks

So far all of the plans we have been building depend on the primitive task's effects changing the world state. What happens when the world state is changed *outside* the control of primitive tasks, however? To explore this, let's modify our example once again. Let us assume that a designer notices that when the troll can't see the enemy, he simply goes back

to patrolling the bridges. The designer asks you to implement a behavior that will chase after the enemy and react once he sees the enemy again. Let's look at the changes we could make to the domain to handle this issue.

```
Compound Task [BeTrunkThumper]
    Method [ WsCanSeeEnemy == true]
        Subtasks [AttackEnemy()]
    Method [ WsHasSeenEnemyRecently == true]//New method
        Subtasks [NavToLastEnemyLoc(), RegainLOSRoar()]
    Method [true]
        Subtasks [ChooseBridgeToCheck(), NavigateToBridge(), CheckBridge()]
Primitive Task [NavToLastEnemyLoc]
    Operator [NavigateToOperator(LastEnemyLocation)]
        Effects [WsLocation = LastEnemyLocation]
Primitive Task [RegainLOSRoar]
    Preconditions[WsCanSeeEnemy == true]
    Operator [RegainLOSRoar()]
```

With this rework, if the Trunk Thumper can't see the enemy, the planner will drop down to the new method that relies on `WsHasSeenEnemyRecently` world state property. This method's tasks will navigate to the last place the enemy was seen and do a big animated "roar" if he once again sees the enemy. The problem here is that the `RegainLOSRoar` task has a precondition of `WsCanSeeEnemy` being true. That world state is handled by the troll's vision sensor. When the planner goes to put the `RegainLOSRoar` task on the final task list it will fail its precondition check, because there is nothing in the domain that represents what the expected world state will be when the navigation completes.

To solve this, we are going to introduce the concept of *expected effects*. Expected effects are effects that get applied to the world state only during planning and plan validation. The idea here is that you can express changes in the world state that *should* happen based on tasks being executed. This allows the planner to keep planning farther into the future based on what it believes will be accomplished along the way. Remember that a key advantage planners have at decision making is that they can reason about the future, helping them make better decisions on what to do next. To accommodate this, we can change `NavToLastEnemyLoc` in the domain to:

```
Primitive Task [NavToLastEnemyLoc]
    Operator [NavigateToOperator(LastEnemyLocation)]
        Effects [WsLocation = LastEnemyLocation]
        ExpectedEffects [WsCanSeeEnemy = true]
```

Now when this task gets popped off the decomposition list, the working world state will get updated with the expected effect and the `RegainLOSRoar` task will be allowed to proceed with adding tasks to the chain. This simple behavior could have been implemented a couple of different ways, but expected effects came in handy more than a few times during the development of *Transformers: Fall of Cybertron*. They are a simple way to be just a little more expressive in a HTN domain.

## 12.8 How to Handle Higher Priority Plans

To this point, we have been decomposing compound tasks based on the order of the task's methods. This tends to be a natural way of going about our search, but consider these attack changes to our Trunk Thumper domain.

```
Compound Task [AttackEnemy]
    Method [WsTrunkHealth > 0, AttackedRecently == false,
CanNavigateToEnemy == true]
        Subtasks [NavigateToEnemy(), DoTrunkSlam(), RecoveryRoar()]
    Method [WsTrunkHealth == 0]
        Subtasks [FindTrunk(), NavigateToTrunk(), UprootTrunk(), AttackEnemy()]
    Method [true]
        Subtasks [PickupBoulder(), ThrowBoulder()]
Primitive Task [DoTrunkSlam]
    Operator [DoTrunkSlamOperator]
        Effects [WsTrunkHealth += -1, AttackedRecently = true]
Primitive Task [RecoveryRoar]
    Operator [PlayAnimation(TrunkSlamRecoverAnim)]
Primitive Task [PickupBoulder]
    Operator [PickupBoulder()]
Primitive Task [ThrowBoulder]
    Operator [ThrowBoulder()]
```

After some play testing, our designer commented that our troll is pretty punishing. It only lets up on its attack against the player when it goes to grab another tree trunk. The designer suggests putting in a recovery animation after the trunk slam and a new condition not allowing the slam attack if the troll has attacked recently. Our designer has also noticed that our troll behaves strangely if he could not navigate to his enemy (due to an obstacle, for example). He decided to put in a low priority attack to throw a boulder if this happened.

Everything about these behavior changes seems fairly straightforward, but we need to take a closer look at what could happen while running the trunk slam plan. After the actual slam action, we start running the `RecoveryRoar` task. If, while executing this roar, the world state were to change and cause a re-plan, the `RecoveryRoar` task will be aborted. The reason for this is that, when the planner gets to the method that handles the slam, the `AttackRecently` world state will be set to true because the `DoTrunkSlam` completed successfully. This will cause the planner to skip the "slam" method tasks and fall through to the new "throw boulder" method, resulting in a new plan. This will cause the `RecoveryRoar` task to be aborted mid-execution, even though the currently running plan is still valid.

In this case, we need a way to identify the "priority" of a running plan. There are a couple ways of solving this. Since HTN is a graph, we can use some form of a cost-based search such as A* or Dijkstra, for example. This would involve binding some sort of cost to our tasks or even methods. Unfortunately, tuning these costs can be pretty tricky in practice. Not only that, we would now have to add sorting to our planner, which will slow its execution.

Instead we would like to keep the simplicity and readability of "in-order priority" for our methods. The problem is a plan does not know the decomposition order of compound tasks that the planner took to arrive at the plan—it just executes primitive tasks' operators.

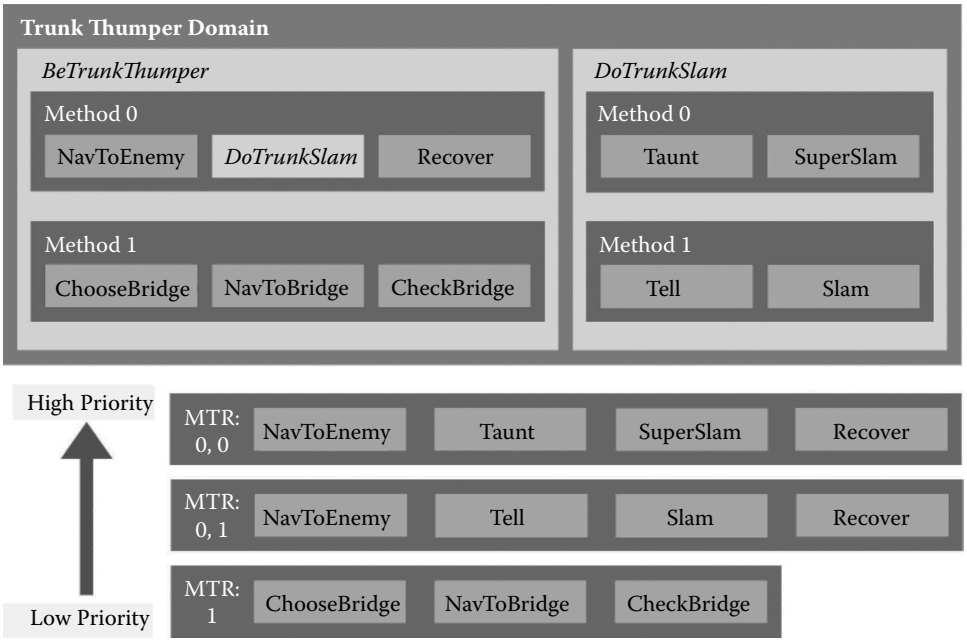


Figure 12.3

All possible plans with the Trunk Thumper domain and the Method Traversal Record for each plan, sorted by priority.

The order of a compound task's methods are what we want to use to define priority—yet the plan isn't aware of what a compound task is. To get around this, we can encode our traversal through the HTN domain as we search for a plan. This *method traversal record* (MTR) simply stores the method index chosen for each compound task that was decomposed to create the plan. Now that we have the MTR we can use it in two different ways to help us find the better plan. The simplest method would be to plan normally and compare the newly found plan's MTR with the currently running plan's MTR. If all of the method indexes chosen in the new plan are equal or higher priority, we found our new plan. An example is shown in Figure 12.3.

We can also choose to use the current plan's MTR during the planning process, as we decompose compound tasks in the new search. We can use the MTR as we search for a valid method only allowing methods that are equal to or higher priority. This allows us to cull whole branches of our HTN based on the current plan's MTR. Our first method is the easier of the two, but if you find you're spending a lot of your processing time in your planner, the second method could help speed that up.

Now that we have the ability to abort currently running plans for higher priority plans, there is a subtle implementation detail that can cause unexpected behaviors in your NPCs. If you set up your planner to re-plan on world state changes, the planner will try to re-plan when tasks apply their effects on successful execution. Consider this altered subsection of the Trunk Thumper's domain below.

```
Compound Task [AttackEnemy]
    Method [WsPowerUp = 3]
        Subtasks [DoWhirlwindTrunkAttack(), DoRecovery()]
    Method [WsEnemyRange > MeleeRange,]
        Subtasks [DoTrunkSlam(), DoRecovery()]
Primitive Task [DoTrunkSlam]
    Operator [AnimatedAttackOperator(TrunkSlamAnimName)]
        Effects [WsPowerUp += 1]
Primitive Task [DoWhirlwindTrunkAttack]
    Operator [DoWhirlwindTrunkAttack()]
        Effects [WsPowerUp = 0]
Primitive Task [DoRecover]
    Operator [PlayAnimation(TrunkSlamRecoveryAnim)]
```

This new behavior is designed to have the troll do the `DoWhirlwindTrunkAttack` task, after executing the `DoTrunkSlam` three times. This is accomplished by having the `DoTrunkSlam` task's effect increase the `WsPowerUp` property by one each time it executes. This might seem fine at first glance, but you will have designers at your desk informing you that the troll now combos a trunk slam directly into a whirlwind attack every time. The problem arises on the third execution of `DoTrunkSlam`. The task's effects are applied and the planner forces a re-plan. With `WsPowerUp` equal to three, the planner will pick the higher priority Whirlwind attack method. This cancels the `DoRecovery` task that is designed to break the attacks up, allowing the player some time to react.

Normally, the whirlwind method should be able to cancel plans of lower priority. But the currently running plan is still valid, and the only reason this bug is occurring is that the planner is replanning on all world state changes, including changes by successfully completed primitive task's effects. Simply not replanning when the world state changes via effects being applied from a primitive tasks will solve this problem—which is fine, because the plan was found with those world state changes in mind anyway. While this is a good change to make, it won't be the full solution. Any world state changes *outside* of the tasks the plan runner is executing will force a replan and cause the bug to resurface.

The real problem here is the domain and how it's currently setup. There are a couple of different ways we can solve this, and it really matters how you view it. One could say that the recovery animation is part of the attack, so it might be worth incorporating that animation into the attack animation. That way the recovery *always* plays after the slam attack. This hurts the modularity of the domain. What if the designers want to chain three slams then do a recovery?

A better way would be to use world state to describe the reason that `DoRecovery` is needed. Consider the change below:

```
Compound Task [AttackEnemy]
    Method [WsPowerUp = 3]
        Subtasks [DoWhirlwindTrunkAttack(), DoRecovery()]
    Method [WsEnemyRange > MeleeRange,]
        Subtasks [DoTrunkSlam(), DoRecovery()]
```

Primitive Task [DoTrunkSlam]
    Operator [AnimatedAttackOperator(TrunkSlamAnimName)]
        Effects [WsPowerUp += 1, **WsIsTired = true]**
Primitive Task [DoWhirlwindTrunkAttack]
    **Preconditions [WsIsTired == false]**
    Operator [DoWhirlwindTrunkAttack()]
        Effects [WsPowerUp = 0]
Primitive Task [DoRecover]
    Operator [PlayAnimation(TrunkSlamRecoveryAnim)]
        **Effects [WsIsTired = false]**

Using the `WsIsTired` world state, we can properly describe the reason we need the `DoRecovery` task. The `DoTrunkSlam` task now makes the Trunk Thumper tired, and he can't execute `DoWhirlwindTrunkAttack` until he gets a chance to recover. Now, when the world state changes, the `DoRecovery` task won't be interrupted and yet we save the modularity of `DoTrunkSlam` and `DoRecovery`. When implementing priority plan picking, these subtle details can really throw a wrench in your HTN behaviors. It's important to ask yourself if you are properly representing the world when you run into these types of behavior issues. As we saw in this case, a simple world state is all that was needed.

## 12.9 Managing Simultaneous Behaviors

A lot of different behavior selection algorithms are very good at doing one thing at a time, but complications arise when it comes time to do two things at once. Luckily, there are a couple ways you can handle this problem with HTN.

One's first reaction might be to roll multiple operators into one. This will work, but this has a couple pitfalls: it removes the ability to reuse operators we have already developed, the combining of multiple operators brings an added complexity that hurts maintainability, and any variation to this combined operator can force us to duplicate code if not handled correctly. Chances are you are going to run into behavior that will need to do multiple things at once, often enough that you are going to want to avoid this method.

A more intuitive way to handle this is to build a separate HTN domain to handle different components of your NPC. Using our troll example, we might have a behavior where we need him to navigate towards his enemy but guard himself from incoming range attacks. We can break this up into multiple operators that control different parts of the body—a navigation operator that would handle the lower body and a guard operator to handle the upper body. Knowing that, we can build two domains and use two planners to deal with the upper and lower bodies.

You may find early on that this can be tricky to implement. The issue that arises is that you need to sync up the tasks in each planner. You can accomplish this by making sure you have world state that describes what's going on in each planner. In our troll example, we can have a world state called *Navigating* that will be set to true when any lower body navigation task is running. This will allow the upper body planner to make decisions based on this information. Below is an example of how these two domains might be set up.

```
Compound Task [BeTrunkThumperUpper]//Upper domain
    Method [WsHasEnemy == true, WsEnemyRange <= MeleeRange]
        Subtasks [DoTrunkSlam()]
    Method [Navigating == true, HitByRangedAttack == true]
        Subtasks [GuardFaceWithArm()]
    Method [true]
        Subtasks [Idle()]
Compound Task [BeTrunkThumperLower]//Lower domain
    Method [WsHasEnemy == true, WsEnemyRange > MeleeRange]
        Subtasks [NavigateToEnemy(), BeTrunkThumperLower()]
    Method [true]
        Subtasks [Idle()]
Primitive Task [DoTrunkSlam]
    Operator [DoTrunkSlamOperator]
Primitive Task [GuardFaceWithArm]
    Operator [GuardFaceWithArmOperator]
Primitive Task [NavigateToEnemy]
    Operator [NavigateToOperator(Enemy)]
        Effects [WsLocation = Enemy]
Primitive Task [Idle]
    Operator [IdleOperator]
```

Now this works great, but there are a couple minor problems with it. A second planner will add a bit of performance hit. Keeping these domains synchronized will hurt their maintainability. Lastly, you will not gain any friends when other programmers run into the debugging headache you just created with your multiple planners—trust me.

There is another alternative for our troll shielding example that does not involve two planners. Currently, navigation tasks complete after successfully *arriving* at the destination. Instead, we can have the navigation task start the path following and complete *immediately*, since the path following is happening in the background and not as a task in the plan runner. This frees us to plan during navigation, which allows us to put an arm up to shield the troll from incoming fire. This works as long as we have a world state that describes that we are navigating and the current distance to the destination. With this we can detect when we arrive and plan accordingly. Below is an example of how the domain would look.

```
Compound Task [BeTrunkThumper]
    Method [WsHasEnemy == true, WsEnemyRange <= MeleeRange]
        Subtasks [DoTrunkSlam()]
    Method [WsHasEnemy == true, WsEnemyRange > MeleeRange]
        Subtasks [NavigateToEnemy()]
    Method [Navigating == true, HitByRangedAttack == true]
        Subtasks [GuardFaceWithArm()]
    Method [true]
        Subtasks [Idle()]
Primitive Task [DoTrunkSlam]
    Operator [DoTrunkSlamOperator]
```

```
Primitive Task [GuardFaceWithArm]
    Operator [GuardFaceWithArmOperator]
Primitive Task [NavigateToEnemy]
    Operator [NavigateToOperator(Enemy)]
        Effects [Navigating = true]
Primitive Task [Idle]
    Operator [IdleOperator]
```

As you can see, this domain is similar to our dual domain approach. Both approaches rely on world state to work correctly. With the dual domain, the *Navigating* world state was used to keep the planners in sync. In the later approach, world state was used to represent the path following happening in the background, but without the need of two domains and two planners running.

## 12.10 Speeding up Planning with Partial Plans

Let us assume that we have built the Trunk Thumper's domain into a pretty large network. After optimizing the planner itself, you have found the need to knock a couple milliseconds off your planning time. There are a couple of ways we can still eek more performance out of it. As we explained, HTN naturally culls out large portions of the search space via the methods in compound tasks. There may be instances, however, where we can add a few more methods to cull more search space. In order to do this, we need to have the right world state representation.

If those techniques don't get you the speed you need, *partial planning* should. Partial planning is one of the most powerful features of HTN. In simplest terms, it allows the planner the ability to not fully decompose a complete plan. HTN is able to do this because it uses forward decomposition or forward search to find plans. That is, the planner starts with the *current* world state and plans *forward* in time from that. This allows the planner to only plan ahead a few steps.

GOAP and STRIPS planner variants, on the other hand, use a *backward* search [Jorkin 04]. This means the search makes its way from a desired goal state toward the current world state. Searching this way means the planner has to complete the entire search in order to know what *first* step to take. We will go back to a simple version of our Trunk Thumper domain to demonstrate how to break it up into a partial plan domain.

```
Compound Task [BeTrunkThumper]
    Method [WsCanSeeEnemy == true]
        Subtasks [NavigateToEnemy(), DoTrunkSlam()]
Primitive Task [DoTrunkSlam]
    Operator [DoTrunkSlamOperator]
Compound Task [NavigateToEnemy]
    Method […]
        Subtasks […]
```

Here, we have a method that will expand both the `NavigateToEnemy` and `DoTrunkSlam` tasks if `WsCanSeeEnemy` is true. Since whatever tasks that make up

`NavigateToEnemy` might take a long time, it would make this a good option to split into a partial plan. There isn't much point to planning too far into the future since there is a good chance the world state could change, forcing our troll to make a different decision. We can convert this particular plan into a partial plan:

```
Compound Task [BeTrunkThumper]
    Method [WsCanSeeEnemy == true, WsEnemyRange > MeleeRange]
        Subtasks [NavigateToEnemy()]
    Method [WsCanSeeEnemy == true]
        Subtasks [DoTrunkSlam()]
Primitive Task [DoTrunkSlam]
    Operator [DoTrunkSlamOperator]
Compound Task [NavigateToEnemy]
    Method […]
        Subtasks […]
```

Here, we have broken the previous method into two methods. The new high priority method will navigate to the enemy only if the troll is currently out of range. If the troll is not outside of melee range, he will perform the trunk slam attack. Navigation tasks are also prime targets for partial plans, since they often take a long time to complete. It's important to point out that splitting this plan is only doable if there is a world state available to differentiate the split.

This method of partial planning requires the author of the domain to create the split themselves. But there is a way to automate this process. By assigning the concept of "time" to primitive tasks, the planner can keep track of how far into the future it has already planned. There are a couple issues with this approach, however. Consider the domain.

```
Compound Task [BeTrunkThumper]
    Method [WsCanSeeEnemy == true]
        Subtasks [NavigateToEnemy(), DoTrunkSlam()]
Primitive Task [DoTrunkSlam]
    Preconditions[WsStamina > 0]
    Operator [DoTrunkSlamOperator]
Compound Task [NavigateToEnemy]
    Method […]
        Subtasks […]
```

With this domain, assume the primitive tasks that make up the navigation cross the time threshold that is set in the planner. This would cause the troll to start navigating to the enemy. But if the world state property `WsStamina` is zero, the troll can't execute the `DoTrunkSlam` anyway because of its precondition. The automated partial plan split removed the ability to validate the plan properly. Of course the method can be written to include the stamina check to avoid this problem. But since both ways are valid, it is better to insure both will produce the same results. Not doing so will cause subtle bugs in your game.

Even if you feel that this isn't a real concern, there is also the question of how to continue where the partial plan left off. We could just replan from the root, but that would

require us to change the domain in some way to understand that it's completed the first part of the full plan. In the case of our example, we would have to add a higher priority method that checks to see if we are in range to do the melee attack. But if we have to do this, what's the point of the automated partial planning?

A better solution would be to record the state of the unprocessed list. With that we can modify the planner to start with a list of tasks, instead of the one root task. This would allow us to continue the search where we left off. Of course, we would not be able to roll back to *before* the start of the second part of the plan. Running into this case would mean that you've already run tasks that you should not have. So if the user runs into this case, they can't use partial planning because there are tasks later in the plan that need to be validated in order to get the correct behavior.

With *Transformers: Fall of Cybertron*, we simply built the partial plans into the domains. For us, the chance of putting subtle bugs into the game was high and we found that we were naturally putting partial plans in our NPC domains anyway when full plan validation wasn't necessary. A lot of our NPCs were using the last example from Section 12.9 for navigation, which is also an example of partial planning.

## 12.11 Conclusion

Going through the process of creating a simple NPC can be a real eye-opener to the details involved with implementation of any behavior selection system. Hopefully we have explored enough of hierarchical task networks to show its natural approach to describing behaviors, the re-usability and modularity of its primitive tasks. HTN's ability to reason about the future allows an expressiveness only found with planners. We have also attempted to point out potential problems a developer may come across when implementing it. Hierarchical task networks were a real benefit to the AI programmers on *Transformers: Fall of Cybertron* and we're sure it will be the same for you.

## References


[Erol et al. 94] K. Erol, D. Nau, and J. Henler, "HTN planning: Complexity and expressivity." *AAAI-94 Proceedings*, 1994.

[Erol et al. 95] K. Erol, J. Henler, and D. Nau. "Semantics for Hierarchical Task-Network Planning." Technical report TR 95-9. The Institute for Systems Research, 1995.

[Ghallab et al. 04] M. Ghallab, D. Nau, and P. Traverso, *Automated Planning*. San Francisco, CA: Elsevier, 2004, pp. 229–259.

[HighMoon 10] *Transformers: War for Cybertron*, High Moon Studios/Activision Publishing, 2010.

[HighMoon 12] *Transformers: Fall of Cybertron*, High Moon Studios/Activision Publishing, 2012.

[Jorkin 04] Jeff Orkin. "Applying goal-oriented action planning to games." In *AI Game Programming Wisdom 2*, edited by Steve Rabin. Hingham, MA: Charles River Media, 2004, pp. 217–227.