Abstract
We propose the use of hierarchical (HTN) planning techniques to encode strategies that one or more Bots should execute while acting in highly dynamic environments such as Unreal Tournament© games. Our approach allows the formulation of a grand strategy but retains the ability of Bots to react to the events in the environment while contributing to the grand strategy.

Motivation
First person-shooter are a very popular kind of game due to their fast pace and reflexes required by their players. Researchers have observed that such games have also the potential to become testbed for different AI algorithms (Laird & van Lent, 1999).

One of the games that has caught the attention from the AI research community is Unreal Tournament (from now on UT) developed by Epic Megagames, Inc. Server-client architecture has been developed allowing different programs to control the behavior of UT Bots. This has led to programming environments that follow an event-driven paradigm, by which the Bot reacts to the changes in the environment. These changes are received as messages from the UT server.

One of the issues with event-driven paradigms is the difficulty of formulating a grand strategy while individual Bots must react to an ever-changing environment. We advocate the use of HTN planning techniques to accomplish the goals of formulating a grand strategy and assigning tasks for the individual Bots to accomplish this strategy. At the same time we retain the event-driven programming of each individual Bots. By doing so, Bots are able to react in highly dynamic environments while contributing to the grand strategy.

In the next section we will study current approaches for controlling Bots in the client-server architecture. Next, we discuss how HTN planning can be used to formulate strategies and still allow the individual Bots to react to events. In the next section we describe some technical difficulties we encountered and how we deal with them. Finally we make some remarks.
switch( state ) {
    default:
    case EXPLORING:
        explore();
        break;
    case HEALING:
        heal();
        break;
    case HUNTING:
        hunt();
        break;
}

Figure 1: Excerpt of the code from a JavaBot

Event handlers are used to detect relevant events that may require interrupting the current action been executed to select a new one. For example, while performing the exploring action, the Bot may interrupt the explore action and start a hunting action if it detects an enemy in the surrounding area.

Soar Bots

The Soar Bot project was developed at the University of Michigan. Soar Bot is based on the Soar Architecture (Laird et al., 1987). Soar uses operators, which define basic deliberative acts. Operators consist of preconditions and effects. This is the standard representation of operators in AI Planning (Fikes & Nilsson, 1972). Preconditions indicate the conditions that must be valid in the state of the world to apply the operator and the effects indicate the changes in the state of the world when the operator is applied. For example, an operator may have as condition that an enemy is within visual range of the Bot and as effect to start hunting at the enemy. This hunt effect may be the condition for another operator. This is the same kind of behavior that can be encoded by event handlers and conditional branching such as the one represented in Figure 1. However, operators allow a more declarative representation of the behavior of the Bots.

Soar also defines rules, which select and apply operators. They can compare and terminate the execution of operators. Soar uses a mechanism to select the more suitable rule for a particular situation, which in turn will determine which operator is selected. For example, one rule may select the operator for hunting an enemy in the event of sighting it. Another rule may select to run from the enemy also in the event of sighting the enemy. Soar will choose the rule (and as a result the operator) that is more suitable according to their utility in the current situation (Laird & Duchim, 2000). Since rules are evaluated continuously, Soar can react quickly to changing situations by interrupting the execution of the current operator and selecting a new one. Soar Bot uses a Tcl wrapper to deal with communication issues with the server. Soar Bots work only for one Bot (i.e., it can’t coordinate multiple Bots).

Hierarchical (HTN) Representations of Bot Strategies

Several variants have been proposed for hierarchical planning (e.g., (Wallace; 2003)). The particular variant we follow in this paper is the one described in (Nau et al., 1999). This variant has been used successfully in several real-world applications including the Bridge Baron game (Smith et al., 1998). We will show how this variant can be used to encode high-level strategies while coordinating individual UT bots.

HTNs (for: Hierarchical Task Networks) is a formalism for representing hierarchical plans. HTNs refine high-level tasks into simpler tasks. In the context of UT Bots, high-level tasks indicate complex goals such as Domination\(X\), where \(X\) is the list of objects of type location that must be controlled. In domination games, when team members steps into one of the locations in \(X\), the team gets a point for every five seconds it remains under the control of the team. The game is won by whoever team gets a certain amount of points first.

Low-level tasks range from intermediate goals such as capturing a certain location to concrete actions such as attacking an enemy in the surrounding area. Tasks representing concrete actions are called primitive tasks since they cannot be decomposed into other subtasks. Compound tasks are tasks that can be further decomposed into simpler subtasks.

Formally, a hierarchical Task Network is a set of tasks and their ordering relations, denoted as \(N=(t_1,…,t_m),<\) (\(m \geq 0\)), where \(<\) is a binary relation expressing temporal constraints between tasks. One of the most important properties of HTNs, and of particular interest for representing UT Bots strategies, is that HTNs are strictly more expressive than operator representations (Erol et al., 1994), which use preconditions and effects.

<table>
<thead>
<tr>
<th>Method</th>
<th>Head: Domination(X)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Preconditions:</strong></td>
<td>1. numberPlayersTeam(N_{\text{team}}), 2. numberLocations(X,N), 3. (N_{\text{team}} &gt; N/2 + 2), 4. SelectLocsGeographTogether(X,P,N/2+1), 5. Divide3Groups(N/2+1,T1,T2,T3), 6. RemainingLocations(RP,X,P)</td>
</tr>
<tr>
<td><strong>Subtasks:</strong></td>
<td>1. CoverLocations(T1,P), 2. PatrolLocations(T2,P), 3. HarrassLocations(T3,RP)</td>
</tr>
<tr>
<td><strong>Orderings:</strong></td>
<td>none</td>
</tr>
</tbody>
</table>

Figure 2: Example of a method for decomposing the task domination

The knowledge artifacts for representing under which conditions a compound task can be decomposed are called
methods. Methods encode strategies for accomplishing compound tasks. A method is an expression of the form \( M=(h,P,ST,<) \), where \( h \) (the method’s head) is a compound task, \( P \) is a set of preconditions, and \( ST \) is the set of \( M \)'s (children) subtasks. Figure 2 shows an example of a method for UT Bots. The task decomposed by method is Domination(X). This method states a strategy that divides the team into three groups \((T_1, T_2, T_3)\). Group \( T_1 \) will cover half of the locations plus one \((P \) denotes these locations). Each location in \( P \) is covered by one member of \( T_j \) (Subtask 1). Group \( T_2 \) will patrol throughout the points in \( P \) (Subtask 2). Group \( T_3 \) will harass the members of the opposing team in the remaining locations (Subtask 3).

A method is applicable to decompose a task if the preconditions are valid in the current state of the world. The method in Figure 2 requires that the team to be as large as at least half of the locations plus two (Preconditions 1-3). The \( 4^{th} \) precondition sets \( P \) to be half of the locations plus one that are geographically together. The \( 5^{th} \) precondition divides the team in 3 groups, \( T_1, T_2 \) and \( T_3 \). Group \( T_1 \) will have \( N/2 + 1 \) members. The remaining elements of the team are distributed evenly among \( T_2 \) and \( T_3 \). The last precondition sets \( RP \)'s to be the locations in \( X \) that are not in \( P \).

Figure 3 shows an example map. The three white squares represent the domination locations, \( X \). If there are 4 Bots in the team, the method shown in Figure 2 becomes applicable. In this situation, \( P \) could consist either of the middle and the upper right locations or the middle and the lower left locations. \( T_1 \) consists of 2 Bots, each of which will be assigned to one location in \( P \). \( T_2 \) consists of a single Bot, which will be in charge of patrolling the 2 locations in \( P \). \( T_3 \) consists of the remaining Bot and will harass any enemy Bot in the location not in \( P \).

An important characteristic of HTN planning is that method decomposition does not change the state of the world. The compound tasks represent high level goals and the methods capture strategies to achieve (decompose) them. The actual changes in the world are done when accomplishing primitive tasks. These tasks are accomplished by operators, which differ from the standard operators in that they have no preconditions, only effects. The reason is that the actual conditions are evaluated when selecting the adequate strategy (in the method selection). Once the primitive tasks are reached, the strategy has been selected and it is executed by the Bots performing concrete actions (i.e., the operator’s effects).

Formally an operator is an expression of the form \( O=(h, effects) \), where \( h \) (the operator’s head) is a primitive task, and \( effects \) indicate how the world changes. Figure 4 shows an example of an operator achieving the primitive task \( \text{CoverLocation}(B,L) \). In this task an specific Bot, \( B \), is assigned to cover an specific location \( L \). The effects of this operator are to move \( B \) to location \( L \) and defend it.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Head: CoverLocation(B,L)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Effects:</td>
<td>Move(B,L)</td>
</tr>
<tr>
<td></td>
<td>Defend(B,L)</td>
</tr>
</tbody>
</table>

**Built-In Preconditions and Effects**

Initially, our plan was to represent the methods and operators using a declarative syntax such as the one exemplified in Figures 2 and 4. We found quickly two problems with this approach. First, preconditions such as the ones described in Figure 2 can be difficult to express. Take for example the \( 4^{th} \) precondition, which selects half plus one domination locations in \( X \) that are geographically together. Expressing such a condition in a declarative language involves making complex expressions. Furthermore, even if we could develop a complete declarative language, most likely processing such expressions would take prohibitive long time in a very fast-paced environment such as UT. Second, effects such as \( \text{Move}(B,L) \) also represents complex executions. Any definition of \( \text{Move} \) will have to consider a path to get there and contingencies that may occur (e.g., finding an enemy along the way).

Our solution for both problems was to use Java Bot methods to define a method’s preconditions and operator’s effects. This allows for a rapid evaluation of preconditions and executing effects (commanding Bots to execute actions). These Java Bot functions are built-in functions that preserve the principles of strategic planning resulting from the HTN task decomposition process. Figure 5 shows part of the declaration of the method presented in Figure 2. It declares the method’s head and all the parameters. The
preconditions are evaluated in the build-in function `evalCondHalfPlusOne`. One of the subtasks, `CoverLocations`, is also shown with its associated parameters.

```
<ooba_method task="Domination">
  <ooba_listparameter>
    <ooba_parameter id = "X">
      <ooba_parameter id = "T1">
        <ooba_parameter id = "T2">
          <ooba_parameter id = "T3">
            <ooba_parameter id = "P">
              <ooba_parameter id = "RP">
                </ooba_listparameter>
              </ooba_task>
            </ooba_listparameter>
          </ooba_task>
        </ooba_listparameter>
      </ooba_task>
    </ooba_listparameter>
  </ooba_routine def="evalCondHalfPlusOne"/>
</ooba_method>
```

Figure 5: representation of the Method of Figure 2 in the XML language description

Operators are defined similarly, with their effect being the call to a Java Bot routine that uses the standard event-driven paradigm to control the behavior of the Bot. An important restriction is that each operator affects a single Bot. Thus, the coordination of the Bots is reflected in the hierarchy and not in the specific actions they undertake.

### Strategy Change versus Strategy Modification

Once the strategy is selected, it is pursued until the strategy is changed (i.e., a new strategy is selected). While the Bots react to the changes in the environment, the strategy is not modified. The main advantage is that this scheme ensures that a unified strategy will be pursued. The main drawback is that conditions may change so dramatically the current strategy may not be adequate anymore. We will extend our approach to continuously evaluate applicability conditions of the high-level strategies. When the applicability of the current strategy falls below a pre-defined threshold, a new strategy is selected for execution.

We will also explore replanning techniques as an alternative to selecting a new strategy. In replanning, parts of the current strategy are modified to account for changes in the current environment (Petrie, 1991). The main advantage over selecting a new strategy is that, by modifying the current strategy, some tasks may not need to change at all, and Bots performing these tasks can continue performing them. In contrast, a change in strategy will result in a change of the task that every Bot is performing.

### Final Remarks

One of the issues with event-driven paradigms typically used to control the behavior of the Bots is the difficulty of coping with seemingly contradictory goals. On the one hand the Bot needs to react quickly in a highly dynamic environment. On the other hand the Bot must contribute to the grand strategy to win the game. We advocate the use of HTN planning techniques to accomplish these goals. A grand strategy is laid out and event-driven programming allows the Bots to react in highly dynamic environments while contributing to the grand task.

### References


