Planning to Coordinate:
Using HTN to Coordinate Unreal® Tournament Bots

By

Hai Hoang

A Thesis

Presented to the Graduate and Research Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Computer Science and Engineering

Lehigh University

May of 2005

This thesis is accepted and approved in partial fulfillment of the requirements for the Master of Science.

_____

Date

_____

Thesis Advisor

_____

Chairperson of Department

Acknowledgements

Special thanks to Dr. Hector Muñoz-Avila for his inspiration, guidance, critiques, advice, and unyielding support throughout this project. This project and thesis would have not been realized without him.

Thanks to DARPA for sponsoring this research project.

Thanks to all my friends at the Intelligent Systems and Technologies Lab (InSyTe). Thank you Stephen Lee-Urban, Marc Ponsen, Ian Warfield, and Ke Xu for all your help and company.

And a special thanks to Stephen M. Lee-Urban for all your help with this thesis.

Table of Contents

List of Figures

Abstract.

Using Hierarchical Task Network (HTN) planning representations to encode strategic game AI in a first-person-shooter (FPS) game has been proposed by (Munoz-Avila & Fisher, 2004) but this was preliminary work and no specific application or implementation of HTN in games were provided. This thesis presents a research project, which was funded by the Defense Advanced Research Projects Agency (DARPA), showing that it is possible to use HTN to model team-based strategies in a FPS game by providing an actual implementation. The HTN planner used to create the HTN plans is SHOP (Nau *et al*., 1999) and the client used to control the bots in UT is Javabot (Javabot, 2005). It also presents the results of the experiments of this implementation where HTN representations are used to encode strategies that coordinate a team of bots in an Unreal Tournament© (UT) Domination game. The experiment consists of two simple strategies encoded in HTN and a third that is a combination of the two and the bot dynamically selects which of the two strategies is more applicable to execute. A paper describing the two simple strategies and the results of the experiments has been accepted by the First Annual Artificial Intelligence for Interactive Digital Entertainment Conference (AIIDE-05). The results from that paper will also be presented in this thesis along with the result for the third strategy. The results are very promising and they should stimulate further research that utilizes HTN in game AI.

1. Introduction

Unreal Tournament is a multiplayer, First-Person Shooter (FPS) game developed by Epic Games, Inc. It uses a client/server architecture that allows different client programs to control the behaviors of bots, and one such client is the Javabot which could control multiple bots at the same time. Currently, Javabot uses a finite state machine (FSM) to control the individual behaviors of a bot. Each bot is controlled individually so these bots could not effectively communicate with each other even in a team based game.

The AI for the bots that came with UT is very good with fragging games such as Deathmatch because the bot's ability to navigate, run, and shoot is very good. A frag, usually associated with multiplayer Deathmatch games, refers to a kill of an opponent. GameArea.com said that "their basic death-matching skills have been virtually perfected; their understanding of all the different game rules is perfect; their ability to fully navigate levels is uncanny; and their threat to your existence is very real" (GameAreana, 2005). The AI is good, but it is only good for games such as Deathmatch where fast reflex and shooting skills are needed and team coordination is not as important. In team-oriented games such as Capture-the-Flag and Domination, these bots do not perform too well because of hard-coded coordinations. For example, a team of two human players were to play the Capture-the-Flag game; once the flag has been captured, one team member will carry the flag back to the base while the other player will go in front of his team member to defend him. With the UT bots, one bot would carry the flag back to the base while the other one just follow the bot with the flag around. This behavior is part of the hard-coded behavior that is built into the bot for Domination games. Ideally, the bots should

coordinate in such a way that the second bot should defend the bot with the flag by going in front of him to defend him not just merely following him.

In order to coordinate team-based strategies, Munoz-Avila & Fisher have proposed using HTNs to encode team-based strategies. This approach "allows the formulation of a grand strategy but retains the abilities of the bots to react to the events in the environment while contributing to the grand strategy" (Munoz-Avila & Fisher, 2004). The basic idea behind this implementation is that the planner would run every few seconds and query the UT server for information about the current state of the game and the bots. Based on the information from the UT server, the planner develops a plan and tells each of the bots what to do. The plan created by SHOP, which is represented with HTN techniques, is the grand strategy that is used to coordinate the bot. While each bot is executing the plan, it also reacts to events in the game. When there is no plan for the bots to execute, each both would follow its own FSM. For example, as part of the plan, the bot is supposed to go to location A. While on its path to location A, the bot sees an enemy in front of it, the bot would shoot at its enemy. Shooting at its enemy is part of the bot's ability to react to events in the game not part of the strategy encoded in HTN.

The goal of this project is to demonstrate that HTN planning techniques can be used to effectively coordinate team-based strategies in a FPS game. With this goal in mind, a lot of the implementations have been kept as simple as possible. This ensures that the performance gain is from the coordination of the bots using HTN planning and not from making a better bot. The performance metric for the experiments is to win the domination games against the bots with the same capabilities except without HTN planning. This performance metric is used to measure the success of research project.

2. Unreal Tournament

UT is a first-person shooter game developed by Epic Games, Inc. It was released in 1999 as a sequel to Unreal that focused more on the multiplayer aspects of the game. It earned the title "Game of the Year" from various publications such as Gamespot, CNET Gamecenter, GameSpy.com, and Computer Gaming World. UT has been praised for its many features including over 10 weapons, over 50 maps and more could be added or created with the level editor, UnrealEd, that comes with the game, and over 5 game modes. Below is a description of each game mode:

1) Deathmatch/ Team Deathmatch - In this mode, every man is for himself. The one that hits the pre-set frag limit first or has the most frags at the end of the game wins. Team Deathmatch is the team-based equivalent of the Deathmatch game. The team that hits the pre-set frag limit first or has the most frags at the end of the game wins.
2) Last Man Standing – Each player or team starts with a certain number and each time a player is fragged, that number is decremented. When that number reaches "0", that player is out. The last one standing with a number wins.
3) Capture the Flag – Players must capture the opposing team's flag and bring it back to their own base.
4) Domination – Each map has a certain number of "control points" called Dompoints. Players must occupy the control point to take them over, and for every five seconds that control point remains under the control of that player's team, that team gets a point.
5) Assault – Players join one of the two teams, the attackers or defenders. The attackers have a final objective to complete while the defenders must stop the attackers from completing the objective within a preset time limit. If the attackers achieve the objective before the timer reaches zero, the attacker

wins; otherwise, the defenders win. The game restarts and the roles are reversed.

These game modes are the default game modes that came with the original UT. The Game of the Year release of UT came with a few more additional game modes. These game modes all support multiplayer and team-based game plays as well as single player games.

UT also comes with a single player campaign which is a comprehensive tutorial for each of its game play modes, designed from the ground up to make learning to play the game a fun and challenging process.

Figure 1. A bot from blue team just has dominated a Dompoint (which is now in blue). On the right hand side the current score of the game is shown; blue team has 17 points whereas red ream has 23.

2.1 Unreal Tournament Bots

One of the most praised features of UT is the Artificial Intelligence (AI) that controls the bots. Bots are AI-controlled players. All of the game modes mentioned above is multiplayer and team-based. What happens if the user cannot go online to play with other players? In single player mode, other players are needed to make the game fun and exciting. In order to get other players in the game, UT bots are needed in order to add players to the game to provide the user with team play experiences. These bots come with eight difficulty levels, from novice to godlike and can be changed during the course of the game. This is good since users can change the difficulty level during the course of a practice session to test their skills.

UT bots are programmed with UnrealScript, developed by Epic Games, Inc. for the Unreal series to program the game logics. An UnrealScript file is just a plain text file like a C++ or Java source code file. Once compiled, it will produce an Unreal bytecode file that the Unreal Engine (like the Java Virtual Machine) could understand. These Unreal bytecode files are executed during a game play by the Unreal Engine to control the bots in the game. When the designer designed the UnrealScript, he had a few goals for UnrealScript in mind. Those goals were:

- To provide the development team and the third-party Unreal developers with a powerful, built-in programming language that maps naturally onto the needs and nuisances of game programming
- To support the major concepts of time, state, properties, and networking which traditional programming languages do not address.

- To enable rich, high level programming in terms of game objects and interactions rather than bits and pixels. (UnrealScript Language Reference, 2005)

Basically, UnrealScript was created to make it easier for people to create bots for UT. It was successful in creating a scripting language that is easy to use since it models other Object Oriented Programming languages. The problem with UnrealScript is that even though it is easy to use, "there is no manual for UnrealScript, no complete language references, no complete introduction to the language, and no complete documentation for how the game has actually been built with the language" (UnrealScript for Dummies, 2005). This makes it hard for people to learn and use the UnrealScript to modify the AI that controls the bots, even though the language has been designed to be easy to use.

2.2 Gamebot/JavaBot

For AI researchers that want to integrate their decision support systems with UT, they must somehow use UnrealScript to integrate with their decision support systems written in another language. This is not easy because the original intent of UnrealScript is to allow programmers to easily create UT bots for UT with no intention for UT bots to communicate with other programs. A group of people at the University of Southern California's Information Sciences Institute (USC-ISI) and Carnegie Mellon University (CMU) has started a project to tackle this problem.

Gamebot is a project that was started at USC-ISI and jointly developed by CMU to create an AI test-bed for research in multi-agents systems (Gamebot, 2005). It is an API (Application Programming Interface) that allows characters in the game to be controlled

over the network by other programs. Once these external programs are connected to the UT server, the server then sends information about the game states and the bots. (*Notice that the server **sends** the information. There is no way for the client to get the information from the server other than waiting for the server to send it.*) These client programs process the information received, decide the actions for these bots to take, and send them back to the UT server. The goal of the Gamebot project is to allow AI researchers to be able to use other clients written in languages other than UnrealScripts and the ability to incorporate other AI tool with their client programs to control the bots. Figure 2 shows a nice feature of the Gamebot. This feature "allows human players to play with the agents, thus providing an opportunity to study human team behavior, and to construct agents that play collaboratively with humans." (Adobbati et al., 2001) Currently, there are many clients for UT that use the Gamebot API such as JavaBot (Javabot, 2005), Soarbot (Soarbot, 2005), Tclbot (Tclbot, 2005) and Tielt (Tielt, 2005).

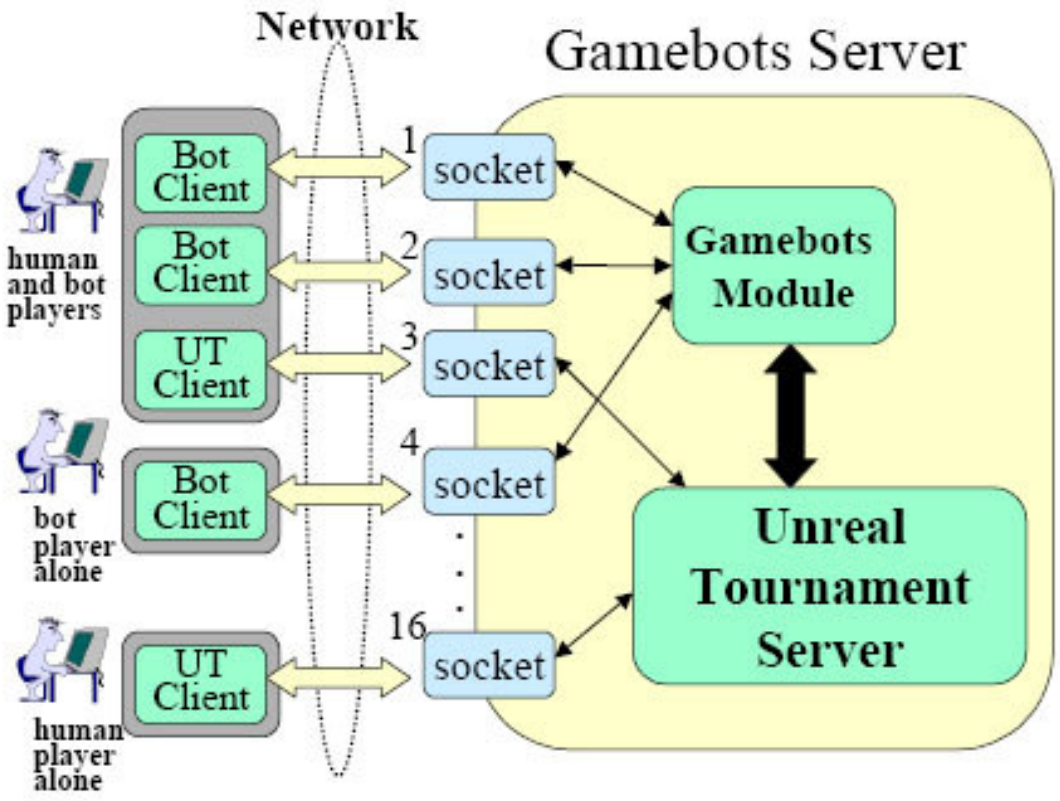

Figure 2. Gamebot Architecture- Multiple human and Gamebot clients are connected to the UT server. (Adobbati et al., 2001)

For this research project, the Javabot was chosen because of the easy of use, and since it was in Java, integrating it with the SHOP planner, which was also written in Java, will be easier. The Javabot API was developed by Andrew Marshall at USC-ISI. "The purpose of the Javabot architecture and API is to provide a higher level interface to the Gamebot modification protocol and make it easier for research groups to develop agents/bots for this rich domain. Javabot can be created by extending java classes and working with objects without having to worry about network protocols, message parsing, etc." (Javabot, 2005)

Figure 3 shows the Javabot architecture. It acts as an interface for other tools and applications to talk to the UT server. The "BotRunnerApp" uses the Javabot API to

connect bots to the UT server. Once the bots are connected, the "Java-based AI engine" uses the Javabot API to control the bots in the game by sending commands to the UT server. The "Java-based visualization tools" use the information received from the UT server through the Javabot API to provide a visualization of the current game. The "Other Java-based Unreal Tournament Application" could also use the Javabot API to receive information about the current state of the game and the bots and send commands to the UT server. In our implementation, the "Other Java-based Unreal Tournament Application" is the Unreal Tournament Shared Object and/or SHOP (more details on this later).

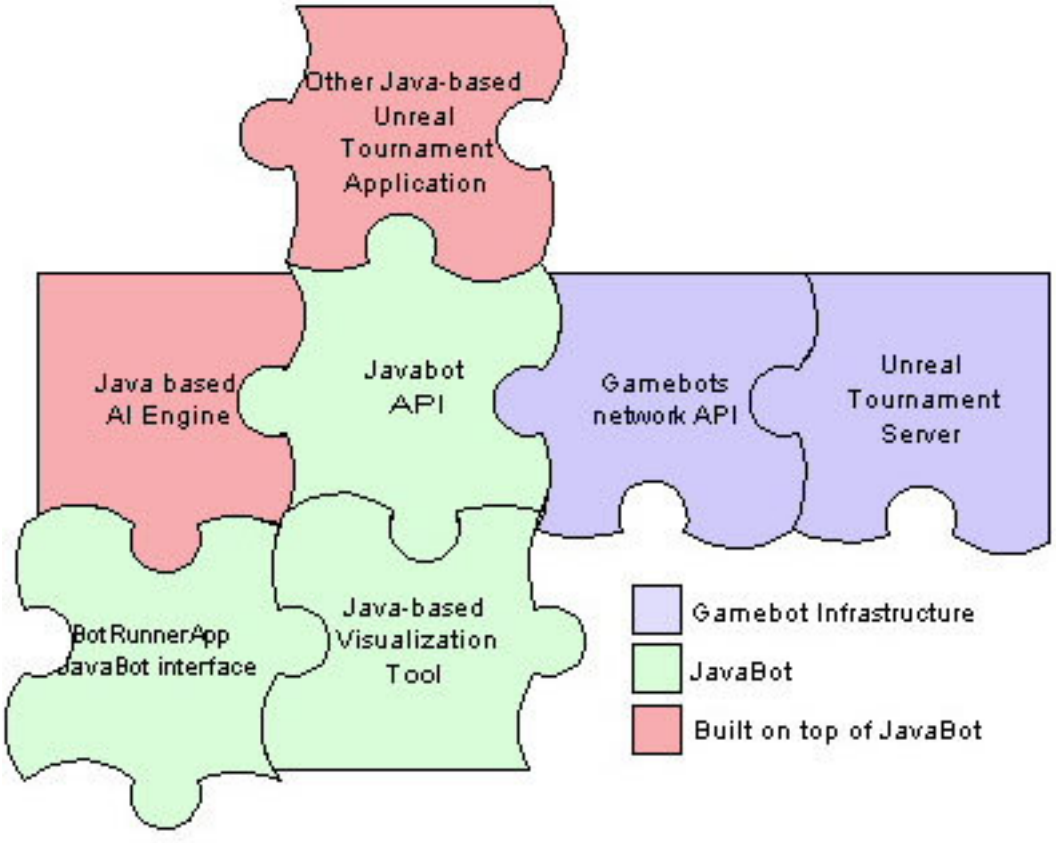

Figure 3. Javabot Architecture – different components working together using the Javabot API (Javabot).

Creating a new bot is very easy; all that is needed for the new class is to extend the edu.isi.gamebots.Bot and override the two functions:

```
public void handleSyncMessage(MessageBlock msgBlock);
public void handleASyncMessage(Message msg);
```

Synchronous messages are the ones that the server sends at predefined time intervals such as game states information, team scores, etc. Asynchronous messages are only sent when necessary such as when a player got shot, weapon found, etc.

All of the information about the current states of the game and the bot are sent to the bot via these two functions. The bot then uses this information to decide which action to take and sends it back to the server.

There are other optional functions that can be overridden to handle events in the game, such as connected(), disconnected(), and destroyed(). They provide a facility for the programmers to define what the bot should do after it is connected, disconnected, or destroyed (disconnected upon the user's request).

The Javabot also comes with a BotRunnerApp that is used to manage the bots. It can start multiple bots, connect them to the UT server, disconnect them from the server, and show the log of each bots. Once started, it allows the user to select which bot to use and assign it to a team before connecting them to the server. Figure 4 shows a screenshot of the BotRunnerApp.

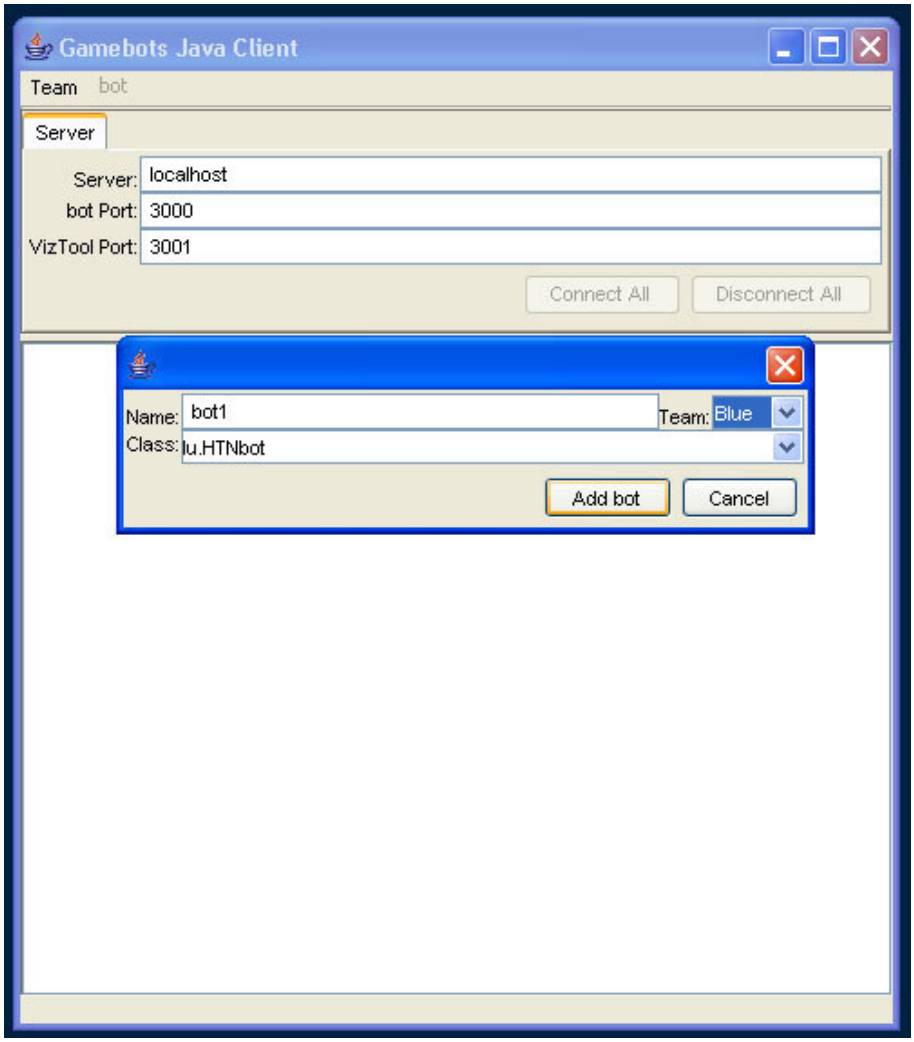

Figure 4. A screenshot of the BotRunnerApp's User Interface

3. Coordinating Bots with HTN Planning

A ***planning problem*** is defined as a collection of goals to achieve, an initial situation or state of the world, and a collection of operators. A well known difficulty of using planning algorithms in real-world problems is solving planning problems efficiently. However, heuristics can be used to improve the efficiency of planning. One of the well known facts in planning is that there is no universal heuristic; therefore, heuristics are usually domain-dependent which means that it is tailored to work with only a specific

domain. A ***solution*** to a planning problem is a sequence of actions, called a ***plan***, that fulfill the goals of the problem relative to the state of the world. **Hierarchical Task-Network (HTN) planning** is a form of planning that advocates reasoning on the level of high-level tasks rather than on the level of the actions (Erol *et al.*, 1994). HTN planning decomposes high-level tasks into simpler ones, until eventually all tasks have been decomposed into actions. There are two kinds of tasks: compound and primitive. ***Compound tasks*** can be further decomposed into subtasks, whereas, ***primitive tasks*** cannot. The primitive tasks indicate concrete actions. Each level in an HTN brings more details on how to achieve the high-level tasks. The sequencing of the leaves in a fully expanded HTN indicates the plan for achieving the high-level tasks. In the context of game AI, the decompositions can be used to encode game strategies and the leaves to be actual in-game actions, such as patrol, attack, etc.

3.1 Components of HTN

The main knowledge artifacts in HTN planning are called **methods**. A method encodes how to achieve a compound task. Methods consists of 3 elements: (1) the task being achieved, called the head of the method, (2) the set of preconditions indicating the conditions that must be fulfilled for the method to be applicable, and (3) the subtasks needed to achieve the head. The second knowledge artifacts are the **operators**. Operators in HTN planning represent action schemes. They consist of the primitive tasks to achieve and the effects, indicating how the world changes when the operator is applied. They have no preconditions because the applicability conditions are determined in the methods. A HTN plan, therefore, consists of methods and operators.

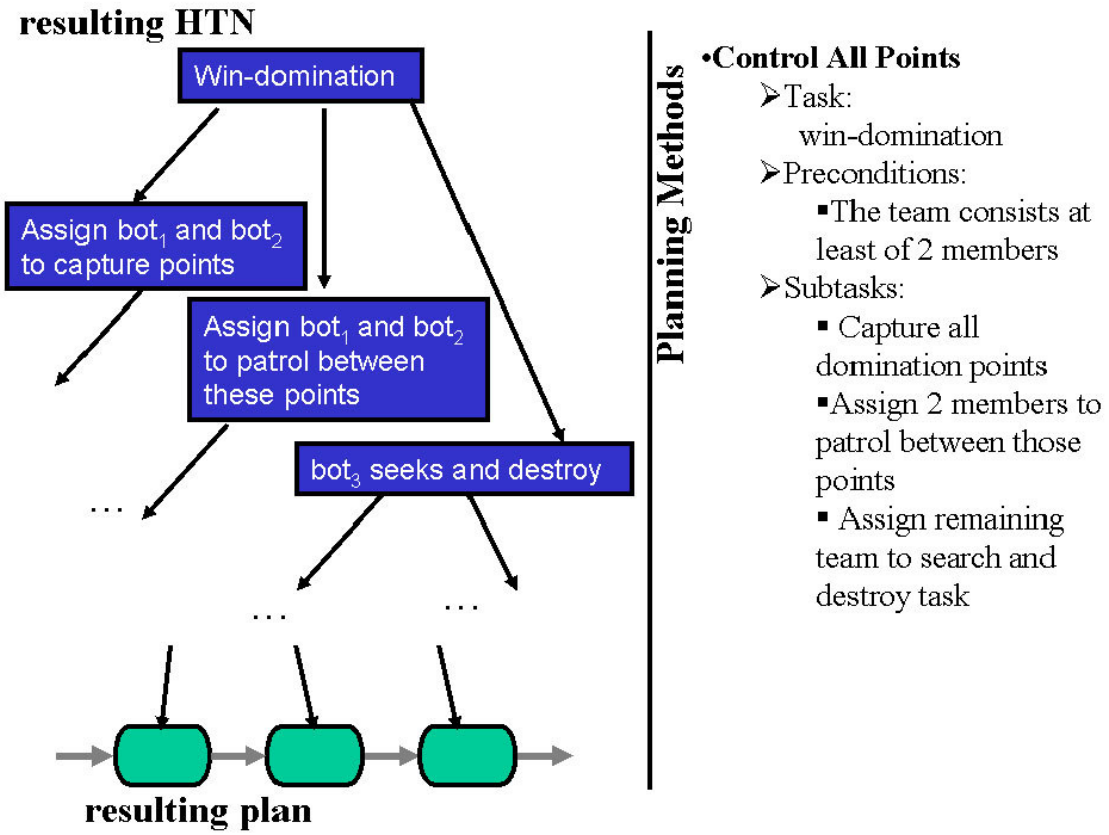

Figure 5. A method in HTN for UT bots. The method "Control All Points" consists of 1 precondition and 3 subtasks.

3.2 HTN and UT Bots

Figure 5 shows an example of the method *Control All Points* encoding a strategy for the task *win-domination*, to win a domination game. This strategy requires that the team consists of at least 2 members. The strategy calls for two members to capture all domination points, and patrol between them. The remaining team members are assigned to the search and destroy tasks (provided that there are more than 2 team members). Achieving these tasks require sub-strategies defined by other methods. For example, when needed (e.g., for search and destroy tasks), we group bots together that move from waypoint to waypoint. A waypoint is a predefined location and is used by the bots for

navigation purposes. This strategy increases the chances of killing enemy bots due to numeric superiority.

Figure 5 also sketches a resulting HTN when *Control All Points* is used in a game. In this situation there are 3 domination points and 3 team members. The first 2 team members are assigned to the domination points and patrol between them, and the third is assigned to search and destroy tasks. The resulting plan is the sequence of all leaves (i.e., primitive tasks) in the HTN.

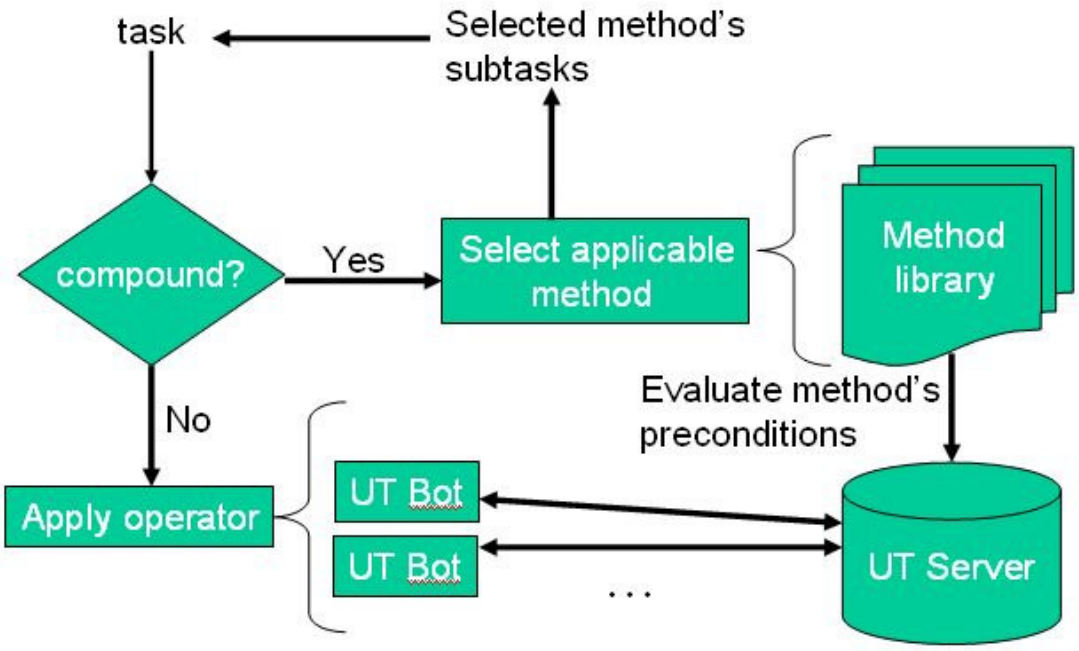

Figure 6. Dataflow of HTN planning for coordinating UT bots

Figure 6 shows the dataflow of the system. Given a task to achieve (e.g., *win-domination*), there are two possible cases:

- If the task is compound, applicable methods are found by processing the updates from the UT server. That is, the method's preconditions are evaluated based on the information provided by the UT server. When an applicable method is found, it is decomposed into its subtasks and the process is repeated.

- If the task is primitive, a UT bot performs this task. Which UT bot gets activated is decided as part of the HTN decomposition process. For example, the subtask *assigns* bot_2 *to* dom_2 in Figure 6 will eventually be decomposed into a concrete action, whereby bot_2 will move to dom_2.

3.3 HTN and SHOP

For plan generation, the HTN planner SHOP was used. SHOP (Simple Hierarchical Ordered Planner) is a domain-independent automated planning system based on ordered task decomposition, which is a type of Hierarchical Task Network (HTN) planning (Nau et al., 1999). However, to be able to use a planning system like SHOP to generate HTNs controlling teams of UT bots in actual domination games, we needed to address two technical challenges: (1) information about the world is maintained by the UT server, and (2) the world is dynamic; e.g., when a bot is accomplishing a task, it might get attacked.

The first challenge affects how the method's preconditions are evaluated and how actions are executed. Planning systems like SHOP assume that the situation in the world is maintained in an internal data structure and actions are executed by modifying this structure directly. The second challenge affects how actions are executed. The assumption in SHOP is that the state of the world only changes by executing actions. This does not hold in games like UT, where other factors (like the opposing team) also change the state of the world.

To address these problems we updated the internal structure of SHOP as the UT server messages were received indicating changes in the game world. The most important extension, however, refers to the actions. We use standard event-driven UT bots coded in Java to execute the actions, but we extend them so they can also perform

the primitive tasks assigned by the HTN, such as going to a certain waypoint. As a result, a grand strategy is laid out by the HTNs and event-driven programming allows the bots to react in this highly dynamic environment while contributing to the grand task. The event-driven program encoded in the Javabot FSM allows the bot to react if, for example, an enemy bot is shooting at it.

3.4 Strategies Encoded with HTNs

For this project, three strategies are encoded with HTNs. The first strategy is called *Control Half Plus One Points*. This strategy sets bots to capture at least half plus one of the domination points. In the *Dom-Stalwart* map that is used for this experiment, there are three Dompoints so half, plus one is two. After capturing two Dompoints, the bots will patrol between these places to defend them. The strategy here is that moving between two points is more efficient than around all three points. This gives the bots a chance to recapture these two Dompoints quicker; therefore, giving the team more points. Another reason is that you are controlling two Dompoints, the enemy has only one left to control. You are most likely going to accrue points quicker than your enemy.

The second strategy is the "*Control All Points*", which has been described in great detail in section 3.2. For our implementation, we modified the theoretical strategy a little bit. This strategy sets two bots to capture all domination points and patrol between them. Since only two bots are used for this experiment, the part where setting the rest of the team members to the search and destroy tasks has been removed. The reason for this strategy is to show that since the planner has the information about all of the Dompoints, it does not send two bots to the same location; thus, the bots are more efficient.

A paper describing the two strategies mentioned above and the results of the experiments has been accepted by the First Annual Artificial Intelligence for Interactive Digital Entertainment Conference (AIIDE-05). Since then, an additional feature has been added. The planner now has information about who is the owner of the Dompoints. This feature is used in the third strategy described below.

The third strategy is a combination of the two strategies mentioned above, but it also takes into account who is the owner of the Dompoint. We called this strategy the *Dynamic Domination Strategy*. This strategy sets all bots out to search and capture Dompoints. Once it has two Dompoints, it will only send bot to recapture a Dompoint if it has been recaptured by the enemy; otherwise, it will continue searching for the third Dompoint. Once it has all three Dompoints, it will concentrate on recapturing Dompoints that has been captured. If it already owned all three Dompoints, it will send two bots to two randomly selected Dompoints.

4. Implementation

In order to get our implementation to work correctly, we have to address the two technical challenges mentioned in section 3.3. The first challenge is that the UT server maintains the state information about the game, while SHOP assumes that it has the state information in its internal data structure and it can modify the state of the world by modifying the internal data structure. The second challenge is that SHOP assumes that the state of the world only changes by executing actions.

Ideally, SHOP should be able to get information about the game states and the bots from the UT server directly, and SHOP should be able to send commands directly to the

UT server to tell what action the bots should execute. But this is not the case because there is no way for SHOP to get state information from the UT server; the server only *sends* the information to the clients at fixed time intervals or when there is an event happening in the game. At the same time, SHOP *cannot* tell the bots what to do by directly talking to the UT server. For our implementation, we have to create a shared object (from now on UTSO for Unreal Tournament Shared Object) where SHOP can get the information for its internal data structure. It is where the Javabot saves the state information that it has received from the UT server. Since the state information the bot has is basically the current state of the game, SHOP can assume that it has gotten those information directly from the UT server. The UTSO works as a middleware that allows SHOP and the UT server to work together.

When a plan is created by SHOP, a series of actions is the resulting plan. These actions are what SHOP assumes to change the state of the world. Since SHOP cannot modify the state of the game in the UT server directly, it has to store these actions as commands back into the UTSO so that the particular bot can carry out the action. At each iteration of the bot's FSM, the bot first checks to see if it has any command to carry out, which was stored for it in the UTSO. If there is a command, the bot will carry out the command before entering its FSM. Once the command is executed, the bot is technically changing the state of the world as SHOP has assumed. The UTSO is a novel solution for both of the challenges mentioned in Section 3.3.

4.1 The UT Shared Object

The UTSO is an object that could be accessed both by the Javabot and SHOP. It has a static function which could be used to get the reference to the object and other functions to store and get information about the current state of the game. With the reference to this object, the Javabot can store and update the state information about the game and the bots. SHOP can also get the reference to this object, get information about the game, and place commands that tell the bots what to do. All of the functions in the UTSO are synchronized to make sure that there is no race condition and the data is consistent.

4.2. The UT Server

There are two UT servers that could be used for our experiment. One is a stand-alone server that is freely available for download. The stand-alone server can only be used to host online games. In order to play the game, we need the actual game, which also comes with the stand-alone server that is essentially the same as the one available for download.

For this experiment, we used the stand-alone server with the latest update, patch version 436. In order to see how the bot behave in the game, we installed the retail version of the game to join in the game hosted by the stand-alone server. This is a very nice feature because it allows us to see how the bots behaves in the game. It is also one of the reasons why UT was chosen by people at USC-ISI and CMU. The reason for running the stand-alone server instead of the server that came with the retail version of the game is that the stand-alone server loads up a lot faster. The reason for this is that the

stand-alone version does not have a lot of the packages that comes with the retail version to load at startup.

With the UT server, the only modification was the installation of the Gamebot files along with a few changes to the ini files that come with the Gamebot. A new map, the Dom-Stalwart, was also added. Other than that, no other modification was made to the server.

4.3 The Javabot

The Javabot version 0.1 was used as the stating point. It comes with an example bot and the CMU_Jbot. The example bot is just a bot to illustrate the simple functionalities of how to create a bot. It is very primitive and not good at all. The students at CMU extended the sample bot and made it better. It is a lot better but still has some bugs.

We first fixed the CMU_Jbot that came with the Javabot before modifying it for use in this experiment. First, the CMU_Jbot was modified to fix the numerous bugs that come with it. Next, the way the bot dominates and navigates around in the game has been greatly enhanced. Lastly, the bot was modified to update the game state information to the UTSO and to carry out the strategy encoded in HTN, which was created by SHOP. The CMU_Jbot that contains the first two fixes are then called the CMU_Improved bot. The CMU_Jbot that contains the first two fixes plus the third one is called the HTNbot.

4.3.1 CMU_Jbot's Bugs and Solutions

There are four bugs that really affect the performance of the bot. The first bug is the "stateChangeTime" variable was never initialized; therefore, the bot always start out

trying to dominate instead of exploring for domination points. Without domination points, there is no way for it to dominate so it just stands still for 40 seconds before it times out and start exploring.

The second bug is the way the threads are being used. The threads are being put to sleep for various amount of time and woken up with the interrupt() function when a synchronous message was received from the server. When first looking at the code, there seems to be nothing wrong with the use of the interrupt() function. The problem is that the bot does not seem to behave as intended. It turns out the server sends out the synchronous message very often and the bot tends to get woken up by the interrupt() function too frequently. The problem is that it gets woken up when it should be sleeping. The solution was to not use the interrupt() function at all. The result is that the bots behaves as expected. This works because the bot are usually not put to sleep for long.

The third bug has to do with the way the bot requests for a path to a location. When a bot asks for a path, it has to attach an "id" to identify the query it sends to the server. With the CMU_Jbot, the same id is used for all queries. Problems can arise when the bot sends a query but then send another query before the result for first query arrives. While waiting for the result of the second query, the result for the first query arrives. The bot mistakenly uses the result for the first query as if it was the result for the second query. This could have negative effects on the way the bot behave in the game. The solution for this is to randomly generate an integer and use it as the id for the query. The bot only uses the result of the query if the id of the result matches the id of the query.

The fourth bug is the bot sometimes would just disappear from the game. This phenomenon usually happens when the bot is at this one location in the game where it has just jumped off the edge of a platform. Even more interesting is the fact that it only happens when there are other bots in the game. If the CMU_Jbot is the only bot playing, there seems to be no problem at all. Another interesting fact is that the bot still has a TCP/IP connection with the UT server, meaning that it has not been disconnected by the UT server or there is still an active connection going on but just no message is being sent. Currently, we still do not know why that happens, but we have come up with a solution to fix the problem. The solution works for more than 90% of the time. The solution works as follow. We know that the UT server sends out the synchronous messages every 200 milliseconds. Every time the bot receives a synchronous message from the UT server, it would save the time of the event. Also, every time it enters its FSM, it first checks to make sure that the time of last message received is less than 2 seconds. Why 2 seconds instead of some small number such as 300 milliseconds? The answer is just to be sure since there could be some network lag time. If the bot detects that it has been more than 2 seconds since the last message was received, the bot would request to be disconnected from the server. Whether the server disconnects the bot or not, after 1 second, the bot would just kill the connection and request to connect back to the server and the bot should be back in the game.

The reason why this solution does not work 100% of the time is because the bot could fail to connect back to the server. We are not sure if this has to do with Windows XP SP2 feature or function that limits the number of concurrent connection attempts per second.

4.3.2 CMU_Jbot's Dominating, Navigating, and Exploring Algorithms

The idea and format of the original algorithms for dominating and navigating has been kept the same. The algorithms have only been enhanced to work better. In the process of fixing the bug with the id for the path requests, the algorithm was modified but not changed.

The exploring algorithm seemed to work alright without any modification. Even though it was not that great, it was left alone with only some minor tweaks.

4.3.3 HTNbot: Combining CMU_Jbot and HTN Planning

In order for the bot to carry out the strategies encoded in HTNs, a few modifications have to be made to the bot. As explained in the previous sections, SHOP cannot get information from the UT server directly, and the only way to get the information is to get it through the bots.

For the HTNbot, the CMU_Jbot has to be modified to update the information about the current state of the game that it has received from the UT server to the UTSO. Information that the bot puts in the UTSO includes:

- Information about itself such as its name, team, current location, etc.
- Information about a domination point when it finds one.
- The name of the team that owns the domination point when the UT server sends it that information.

These are the only information needed by SHOP right now to create the simple plans needed for the domination game. More information can be added, if necessary, when implementing new strategies for the Domination game or strategies for other game modes.

In order for the HTNbot to carry out the strategies layout by the SHOP planner, it has to check the UTSO every time before it enters its FSM to see it has a command stored there for it to execute. Once it has executed the command, it has to remove the command from the UTSO. If the HTNbot finds no command for it to execute, it would just follow its FSM and behave just as the CMU_Improved bot.

Currently, the only command for the HTNbot to execute is the command telling it to go to a certain location. This command is the only one needed to implement the strategy for the Domination game. As more strategies for the Domination game or other game modes are created, new commands can be added to extend the functionality of the HTNbot.

4.4 SHOP

SHOP is a domain-independent HTN planning system. It comes in two versions, one is the original SHOP, which is written in LISP, and the other is written in Java. For this project, the Java version of SHOP is used. Since the original version of SHOP was written in LISP, SHOP uses the notations similar to LISP for plan representation. These notations are still being used to represent a plan in the Java version.

Two modifications were made for SHOP. The first modification was when an instance of SHOP is created, it gets the reference to the UTSO to get the state information about the game and the bots and to update its internal data structure. The second modification was after the plan is the plan is generated, it would place the commands back into the UTSO telling what the bots should do.

4.5 An Overview of the Process

This section is devoted to describing how the different pieces that have been described earlier in Section 4 work together and the order in which these programs run and communicate with each others. This is described from the perspective of a user running one of the experiments.

The first thing to run is the UT server, and after a few seconds the server should be started and ready for clients to connect to the game. The game mode and the map to use are specified as the parameters to the UT server exe file. If a bat file is used, this information is all contained in the bat file. Once the server is started, the user can now connect the TclViz tool to the server. If the TclViz tool is run from the same computer as the one running the UT server, the user can just connect without having to change the parameters.

Next, the user can start the Javabot application, which is the BotRunnerApp. The BotRunnerApp's User Interface allows the user to add the bot they want and assign it to a team. Once all of the bots are added, the user clicks connect.

When the user clicks connect, a few things happen:

1) The BotRunnerApp connects these bots to the UT server. It also adds the bot to the UTSO.

2) It also starts another thread that wakes up and run every 8 seconds. Each time it runs, it creates an instance of SHOP to run then go back to sleep. SHOP then accesses the UTSO to update its internal data structure about the current state of the game and the bots. It then generates a plan. If a plan is

generated successfully, it places commands telling the bots what to do back into the UTSO.

For the bot, once connected, it updates information about itself and the current state of the game when it receives information from the UT server. For each iteration of its FSM, the bot first checks to see if it has commands telling it what to do in the UTSO before entering its FSM. If there is a command, the bot executes it. Upon completion, it updates the UTSO showing that it has completed the command. If there is no command, the bot enters the FSM. This cycle continues until the bot gets disconnected or the game is over.

5. Installation

The installation process is fairly simple or could be very simple. This includes installing the stand-alone UT server, the Gamebot files, the Javabot, and other tools needed to carry out the experiments. The user is assumed to be familiar with Java. With a little bit of background knowledge in creating a bat file should help. The website for this project is at the HTNbots Project's Homepage (HTNbots, 2005), which has files related to this project for download.

5.1 Installing the UT server

Installing the UT server is the simplest process. The UT server package available for download comes in a zip file. All that has to be done is to extract the content of the zip file to a folder. Also run the patch to update the server to the latest version. Just double clicking the download file should start the installer for the patch.

Next is to create a bat file to start the server. This is necessary if you do not want to type in the long line of text at the command line to start it. A sample file is available on this project's website. This sample bat file has the pathmarkers enabled which shows where the waypoints are. Bots use these waypoints for navigation in the game. Just disable the pathmarker to prevent them from being displayed in the game.

Figure 7. A bot is navigating in the game using the waypoints shown in the game as pathmarkers (represented as the heads of a camel).

5.2 Installing the Gamebot

There are two types of Gamebot packages available for download. One is an installer file with the .umod extension. If UT is installed, double clicking on this file should start the installer prompting you to install the Gamebot. The other package is a

zip file. Just put the contents of the “system” folder into the “system” folder in the directory where the UT server was installed.

After the Gamebot is installed, copy the map used for this experiment, the “DOM-Stalwart.unr” map, to the map directory. There are a few parameters that needed to be changed in the BotAPI.ini file in the system directory. Instead of changing them, you can get the modified version from this project’s website.

5.3 Installing the TclViz

TclViz is a visualization tool that shows a top-down view of the actions in the game. It can also be considered a client for the Gamebot because it uses the Gamebot API to connect to the server so the server can send it information about the game to display on the screen.

TclViz is a script file. Therefore, no installation is necessary, but in order to run TclViz, you must have Tcl/Tk 8.0 or higher. Once Tcl/TK is installed, just double click on TclViz.tcl and the tool should be started.

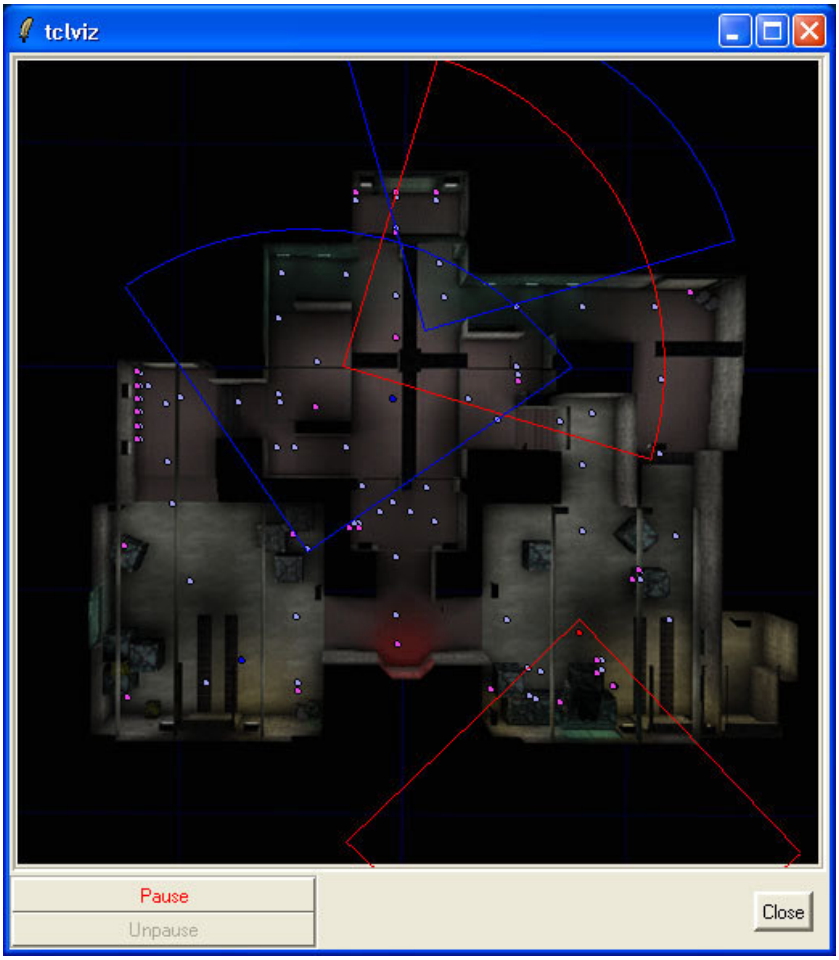

Figure 8. A screenshot of the TclViz visualization tool for UT. It provides a top-down view of the actions in the game. The base of the cone represents a bot and the area of the cone is the bot's line of sight. The dots are the objects in the game (e.g., Dompoints, health packs, weapons, etc).

5.4 Installing the Source Code for the Javabot

The Javabot source code package can be downloaded from this project website. It is a JBuilder project. Just extract the content of the zip file to the JBuilder's project directory and open the project up in JBuilder. This package was created with JBuilder Foundation version X. Borland has a free version of their commercial JBuilder tool available for download on their website.

The source files in the "src" folder could be compiled with the Java SDK. It has been tested to work with version 1.42 of the Java SDK. Other versions might work, but they have not been tested.

6. Experiments

There are two phases for the experiments. The first phase consists of the experiments for the *Control All Points* and *Control Half Plus One* Points strategies. These experiments are for the AIIDE-05 submission but are also included here. The second phase of the experiments is for this thesis. The details of the experiments are described below.

6.1 Experiments for the AIIDE-05 Paper

For the experiments, we had two opponent teams, with two bots each. The first team consisted of the standard UT bots that came with Javabot, in particular the CMU_Jbot. We refer to the first team as *standard* team. For the second team we did several improvements to the code of the standard UT bots. In particular, we improved navigation issues and domination tactics. We refer to this team as the *improved* team. The team that uses the HTNs uses the same improved code. The only difference is that the domination strategies are dictated by the HTNs. We refer to this team as the *HTN* team.

We ran the experiments on the domination map, Dom-Stalwart, that came with Gamebot. We counted results only when a match terminated where no bot from either team got disconnected from the server. Since the positions of the bots are determined by the UT server randomly and these positions improve the chances that either team will

win, we ran the experiments 5 times and averaged their results. These results are shown in *Figures 9 and 10* for the *Control Half Plus One Points* strategy versus the standard and the improved teams, respectively. *Figures 11 and 12* show the results for the *Control All Points* strategy versus the standard and the improved teams, respectively. The number of points to win a match was set to 50. Usually in a game, the number of points to win a match is set at 100, but since the standard bot that came with the Javabot has bugs it in. Sometimes, the bot would just disappear from the game. In order to complete the game successfully, we decided to change the total point to 50 for a win. Even after setting it to 50, it took more than three hours just to get five successful runs. This just shows how serious the one bug in the original CMU_Jbot was.

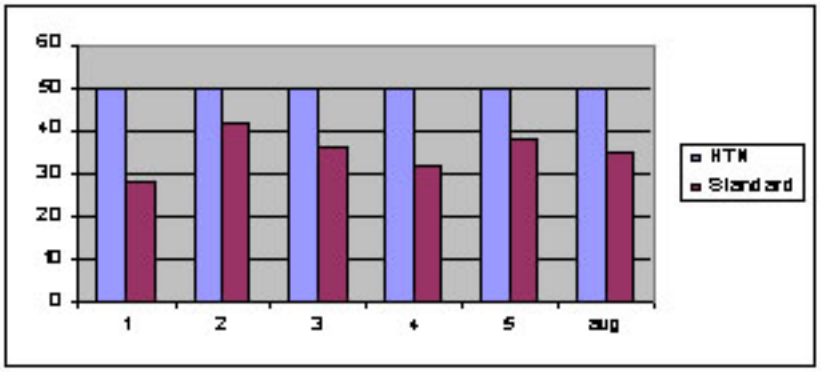

Figure 9. Control Half Plus One vs. Standard

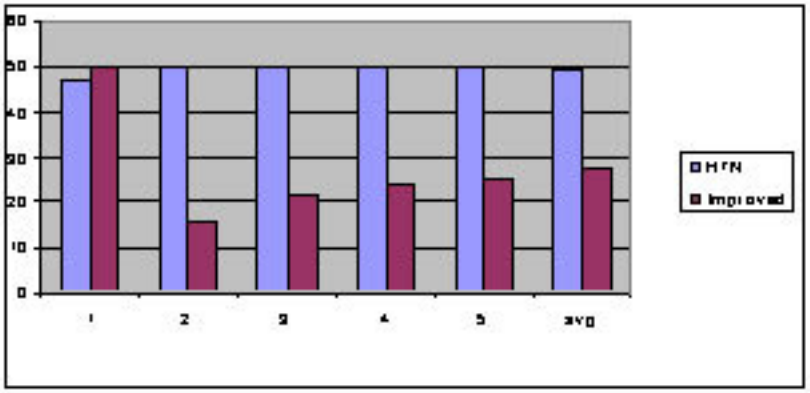

Figure 10. Control Half Plus One vs. Improved

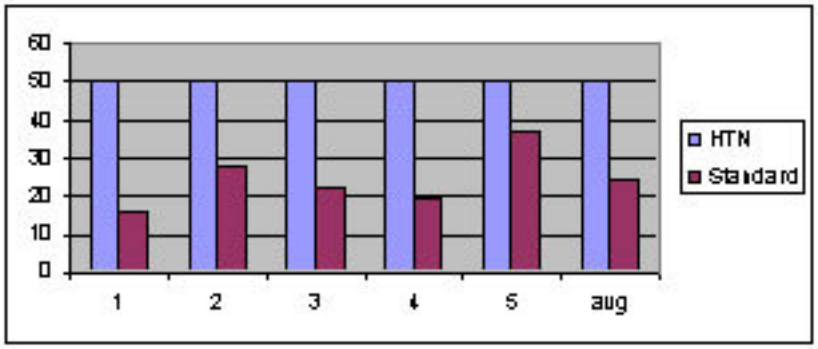

Figure 11. Control All Points vs. Standard

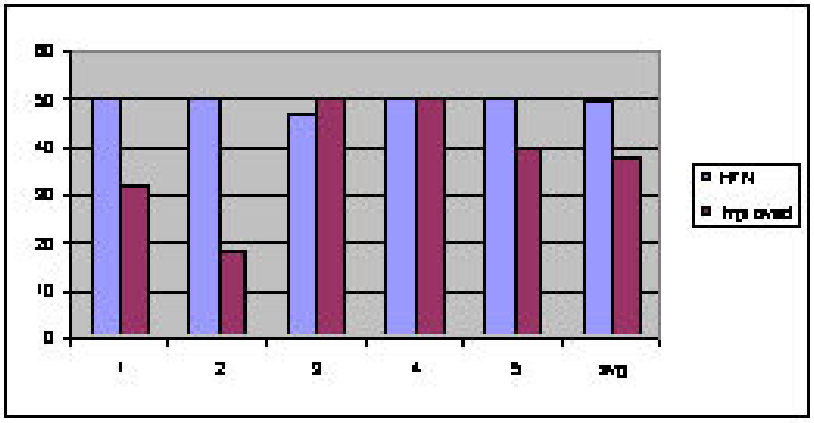

Figure 12. Control All Points vs. Improved

6.2 The Second Set of Experiments

The second set of experiments is for the *Dynamic Domination Strategy.* We had two opponent teams, with two bots each. The first team consists of the CMU_Improved bots which is an improved and bug fixed version of the CMU_Jbot. The second team consists of HTNbots using the same code as the CMU_Improved bots but the difference is that the domination strategies are dictated by the HTNs.

The experiment is divided into two set of runs, each consisting of 15 games each. The average score for each set of runs is then computed. The results for the two sets of runs are provided in the tables below.

Table 1:

	HTNbot	CMU Improved
1	100	70
2	100	65
3	100	48
4	98	100
5	100	62
6	98	100
7	100	54
8	100	98
9	98	100
10	100	79
11	100	54
12	100	75
13	100	47
14	100	74
15	100	75
Average	99.6	73.4

Table 2:

HTNbot	CMU Improved
100	86
100	48
100	73
100	52
100	67
99	100
100	70
100	81
100	69
100	77
100	80
100	79
100	82
100	66
100	89
99.93333333	74.6

Figure 13. Results of the Dynamic Domination Strategy

These results show a clear dominance by the HTN team over the other teams of bots. The HTNbots won most of the games. The HTNbots lost a few games but not by many points. This is due to the fact that sometimes the bot got spawned in a bad location and it got struck trying to get out of the bad location.

7. Limitations

The goal of this project is to demonstrate that HTN planning techniques can be used to effectively coordinate team-based strategies in a FPS game, not to make the best or an unbeatable bot. With this goal in mind, a lot of the implementations have been kept as simple as possible. Time is spent mostly on making the bot works with SHOP and the UTSO. Little time is concentrated on making the bot better. Making the CMU_Jbot better is not intentional but as the result of bugs fixing.

Every time we describe this project to someone new. The one question they usually ask is "*can your HTNbot beat the UT bots that came with Unreal Tournament?*" The answer is no. That is not the objective of this research project. In order to make bots that good, more work needs to be done. There are several reasons why the HTNbot cannot beat the UT bots, but we will only list some of the most important ones below:

- Currently the bots are not very intelligent with navigating in maps that have obstacles that require the bots to jump. It works well with the Dom-Stalwart map because it does not require the bots to jump up any platform. This is the reason why the Gamebot recommends using the Dom-Stalwart map for initial testings.
- No attempt has been made on making the bot shoot intelligently. It just shoots at the enemy when it sees one.
- The bot does not even go look for health packs or weapons. It also does not even choose the weapon to use depending on the situation. Using the right

weapon is very important in a fragging game. Since the experiment is for a domination game, no consideration was given to improving this.

If HTN planning can be integrated with the UT bots that came with UT, we conjecture that the HTNbots could possibly beat the Unreal Tournament bots. We have not done any experiments yet to support this claim since the UT bots use UnrealScript and integrating it with SHOP is not easy. However, our conjecture is based on the fact that the original CMU-bots and improved CMU-bots were beaten by the HTN variants of the original and
the improved one, respectively.

8. Future Work

The goal of this project is to create a prototype that demonstrates that HTN planning techniques can be used to effectively coordinate team-based strategies in a FPS game. This project is in the initial phase, and it is still evolving. There are still a lot of rooms for improvements. Below is a list of suggestions about things that could be improved in the future.

Exploration Algorithm. The exploration algorithm for the CMU_Jbot does work but not very well. It was left alone without modification for this project since it was adequate for the purpose of the experiments. While running the experiments, it can be seen that the bot sometimes keeps exploring the same area over and over again. It should somehow know that it has already been there and try to get out of that particular area.

Bot Should Be Able to Handle New Maps. Currently these bots only work well with the Dom-Stalwart map. The reason is not because the algorithm for the bots is hard-coded, but because these bots currently cannot handle jumping on to platforms and other complicated obstacles. If these bots are good enough to handle such obstacles, it should technically work on all Domination maps.

Fragging Skills. These bots only know to shoot at the enemy when they see one. They are not intelligent at all when it comes to shooting. No time has been spent on improving its fragging skills because fragging does not give the team any point in a Domination game. The fragging skills will be very important if these bots are to be used in other games modes.

Using HTN Planning with Other Game Modes. Currently, only the Domination game mode is supported. This could be extended to the other type of game modes where more strategies could be defined.

Collecting Health and Weapons. The bots should collect health when it is low. It should also be intelligent enough to choose the weapons to collect and use, depending on the current situation.

Learning Coordination Strategies. The strategies are currently hand-coded in HTNs and stored in a file that SHOP loads on startup. A possible extension could be to learn new coordination strategies and use those strategies if applicable. For example, the system could observe human or AI-controlled bots in a team, record their actions and the game conditions under which these actions were made and then perform these same actions when the same game conditions occur.

One Instance of SHOP. As described in Section 4.5, each time the thread runs, it creates an instance of SHOP. With this happening every 8 seconds, the performance of the system is not optimal. For future work, *only one* instance of SHOP should be created when the BotRunnerApp starts and each time the thread runs, it should call a function to update the internal data structure of SHOP, generate a plan, and update the UTSO with commands.

9. Conclusion

This research project provides an actual implementation of using HTN techniques to encode strategies to coordinate UT bots. HTNs were used to encode strategies that coordinate a team of bots in an UT Domination game and run them effectively using standard FSMs to encode individual bots behavior. As a result, a grand strategy is laid out by the HTNs and event-driven programming allows the bots to react in this highly dynamic environment while contributing to the grand strategy. A few sets of experiments were carried out to see if this implementation meets the performance metric, which was to win the domination games against the bots with the same capabilities except without HTN planning. The results show that the bots with HTN coordination meet the performance metric by beating the bots without HTN planning. We conclude that HTN techniques can be used to encode strategies for game AI.

10. Reference

Adobbati, R., Marshall, A., Scholer, A., Tejada, S., Kaminka, G., Schaffer, S., & Sollitto, C. Gamebots: A 3D Virtual World Test-Bed For Multi-Agent Research. (2001)

Erol, K., Nau, D., & Hendler, J. HTN planning: Complexity and expressivity. *AAAI-94 Proceedings.* AAAI Press, 1994.

GameArea. URL: http://www.gamearena.com.au/games/title/pc_unrealtournament/index.php (last viewed: April 27, 2005)

Gamebot. URL: http://www.planetunreal.com/gamebots/ (last viewed: April 27, 2005)

Hoang, H., Lee-Urban, S., & Muñoz-Avila, H. Hierarchical Plan Representations for Encoding Strategic Game AI, 2005.

HTNbots. URL: http://www.cse.lehigh.edu/~munoz/projects/HTNbots (last viewed: April 27, 2005)

Javabot. URL: http://utbot.sourceforge.net/ (last viewed: April 27, 2005)

Muñoz-Avila, H. & Fisher, T. Strategic Planning for Unreal Tournament Bots. *Challenges of Game AI: AAAI'04 Workshop Proceedings* (Technical Report WS-04-04). San Jose, CA: AAAI Press, 2004.

Nau, D., Cao, Y., Lotem, A., & Muñoz-Avila, H. SHOP: Simple Hierarchical Ordered Planner. *Technical Report CS-TR-3981*, UMIACS-TR-9904, 1999.

Soarbot. URL: http://www.planetunreal.com/gamebots/downloads.html (last viewed: April 27, 2005)

Tclbot. URL: http://www.planetunreal.com/gamebots/downloads.html (last viewed: April 27, 2005)

Tielt. URL: http://www.nrl.navy.mil/aic/iss/ida/projects/tielt/TIELTResources.php (last viewed: April 27, 2005)

UnrealScript for Dummies. URL: http://www.planetunreal.com/gamebots/uscript1.html (last viewed: April 27, 2005)

UnrealScript Language Reference. URL: http://unreal.epicgames.com/UnrealScript.htm (last viewed: April 27, 2005)

11. Vita

Hai Hoang was born on October 9, 1982 in Can Tho, Vietnam. He graduated from Bethlehem Catholic High School in 2000 as the Salutatorian. He graduated from Lehigh University, Magna Cum Laude, with a Bachelor degree in Computer Science in May of 2004. As a President's Scholar, he continues his graduate studies at Lehigh University. He will receive his Master of Science degree in Computer Science in May of 2005.