Automated Planning in Computer Games

René Bjørn Hansen

Kongens Lyngby 2008
IMM-B.Sc.-2008-02

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

Summary

This paper introduces Hierarchical Task Networks (HTN) as an alternative to Finite State Machines for controlling bots in Unreal Tournament. The challenges within the domain are identified and corresponding improvements to the HTN are proposed. Finally a solution is implemented in Java by using the GameBots mod for Unreal Tournament.

Acknowledgements

First of all I would like to thank my supervisor Thomas Bolander for agreeing to supervise this project and always finding the time to answer my questions.

Secondly I would like to thank Jakob Udsholt for his great feedback in the last phase of the project.

Contents

Chapter 1

Introduction

1.1 Introduction

This paper will study the use of artificial intelligence to control the non-player characters (NPC) in a first person shooter (FPS). A first person shooter is a fast pace-, highly reactive- game. The player controls his character in a 3d rendered world where his visual perspective is that of the character. The game tests the players skills in aiming his weapon and killing his enemies before they kill him.

The game of Unreal Tournament (UT) will mainly be used as domain. UT is an FPS game that takes place in an arena like setting. In the classic game type, also known as Deathmatch, every player is by himself and the main objective is to shoot and kill as many opponents as possible. Each player is aware of his own health, weapons picked up from around the map and corresponding ammunition. Every character can either be controlled by a player or a computer (NPC), which in this context is referred to as a bot.

UT has several different game types besides the classical, and this paper will mainly focus on the type called Domination. A Domination map has two or more domination locations and the players/bots are divided into two teams. Whenever a team member walks over a domination location, his team will then ”dominate” that location. Every domination location awards a certain amount of points every second to the team that controls it. The goal of the game is then to control as many domination locations as possible, and get a predefined amount of points before the opposing team [5].

Since the game concept itself is highly reactive[1] it would therefore be reasonable to make a reactive artificial intelligence. In a vast majority of FPS' this has been done by letting finite state machines (FSM) control the behavior of the bots[2]. This seems to be the most effective approach in simple game-types such as the classical UT, however as more and more complex game-types appear, it becomes increasingly difficult to make an FSM that also deals with team coordination, point domination, etc.

This paper will outline the use of automated planning as an alternative to FSMs, more precisely the use of Hierarchical Task Networks (HTN). The purpose of the HTN is to make a framework to specify strategies on a higher level of abstraction then with an FSM. This should give the designer of the bots a more intuitive way of defining the overall strategy.

The first part of this paper will take a theoretical approach to the problem and the second part will show how a solution to the game of Unreal Tournament is implemented.

Chapter 1 contains an introduction, section 1.1, and a domain analysis in section 1.2.

Chapter 2 introduce and discuss Finite State Machines in section 2.1, in section 2.2 planning is introduced and in section 2.3 Hierarchical Task Networks are given as an alternative to Finite State Machines.

Chapter 3 deals with the potential challenges and solutions when applying a Hierarchical Task Network to a first person shooter. Section 3.1 discusses these challenges one by one and section 3.2 rounds up and describes the theoretical solution for a planner within the domain.

Chapter 4 deals with an implementation of the planner. Section 4.1 gives an overview of the implementation, section 4.2 describes the UT server, section 4.3 outlines the implementation of the Hierarchical Task Network, sections 4.4 and 4.5 details the implementation of the bots and their cooperation, section 4.6 describes the client/server communication and section 4.7 is dedicated to a discussion with regard to further work on the implementation.

[1] The concept of reacting to input, e.g. the bot moves around randomly, when it spots an enemy it reacts by shooting at the enemy, when it is low on health it reacts by running away, etc.

[2] See chapter 8.1 First-Person Shooter AI Architecture in [12]

Chapter 5 concludes the paper by outlining the results in section 5.1 and a conclusion in section 5.2.

1.2 Domain Analysis

Unreal Tournament can be extended with a mod[3] called Gamebots [8]. This modification opens up a programming environment for creating AI controlled bots. This programming environment follows a client/server architecture, meaning that the developer can create a client that logs on to the UT server and controls a bot. The server provides sensory information about the events in the game and sends the information to the client in the form of asynchronous messages.

Furthermore UT has different game types where most of them are team based, meaning that any single bot both has friends and foes.

Time constraints: Due to the fast pace nature of an FPS game, the client has very little time to react to sensory updates. From the point in time where the client receives information about being attacked until it sends its counter actions to the server, the attacker would already have gained some form of advantage. It is therefore crucial that the client does not waste time on unnecessary computations, and has to react almost instantly.

This leaves the AI developer with the task of having to create a bot that reacts intelligently to events, with almost no computation time.

Uncertainty: The sensory data send by the server contains only information about the bots local environment. This means that the bot does not have any knowledge about what is going on elsewhere. Furthermore the bots opponents act non-deterministically, rendering the world unpredictable and very hard to reason about. An action being executed might get interrupted, and even become impossible to finish. E.g. while a bot is running from one location to another, it is pushed over an edge and its current path is no longer valid.

[3] A mod is a way to modify the rules of the game such as, goals, weapons, textures etc.

Temporal actions: Most of the actions done by the bot takes an unknown amount of time, e.g. running form one location to another, turning around, discharging a weapon, etc. The duration is assumed to be finite, however the bot needs to wait for the action to finish before it can start another one.

Concurrent actions: Given the domain, it is an obvious choice to make a team of bots that coordinates their efforts in order to win. The actions of the different bots runs concurrently. A stronger AI would take advantage of the concurrency, making the bots act at the same time.

These four elements are the key challenges when making an AI within this domain.

Chapter 2

Finite State Machines and Planners

2.1 Finite State Machines

This section will first outline the use of FSMs with regard to bot AI and then point out potential problems when implementing more complex strategies.

2.1.1 Finite State Machines in Artificial Intelligence

In a game of Deathmatch, the only objective is to kill as many opponents as possible. In order to do so the bot will need a weapon, health and the ability to search for opponents and kill them. A simplified version where the bot has a weapon with unlimited ammunition and is not concerned with its health, is modeled in figure 2.1 on the following page.

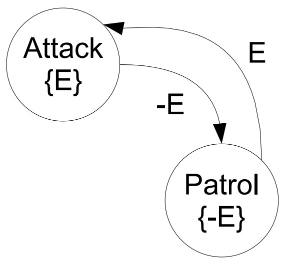

Figure 2.1: Simple FSM

The bot will primarily be in the `Patrol` state, which will make it roam the map looking for enemies `E`. When the bot spots an enemy the FSM will go to the `Attack` states, which will make the bot engage in combat until the enemy is either dead or in another way out of the bots vicinity.

One of the main issues about using an FSM is when it needs to be extended. If the FSM given, in figure 2.1, is extended with another state variable describing the bots current amount of ammunition `A`, the FSM would have to be updated with two additional states.

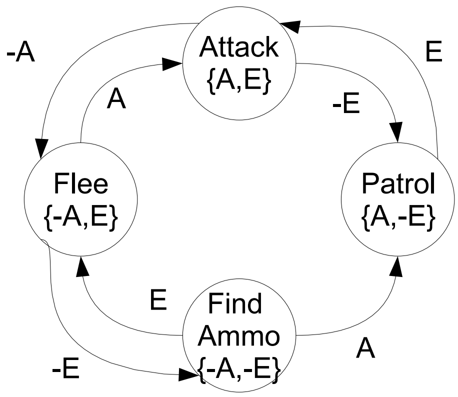

Figure 2.2: Updated FSM

In figure 2.2 the states `Flee` and `Find Ammo` are added. As visualized in figure 2.2 the bot might run out of ammunition while in the `Attack` state. In this case the bot will flee until it either finds some more ammunition, in which case it will resume the combat, or it actually escapes, which will make it look for more ammunition before it can resume the patrol.

The last thing that needs to be added is the state variable describing the bots health, where `H` describes the health being above a predefined threshold and `-H` being below it. The result of this is adding four new states to the FSM (see figure 2.3).

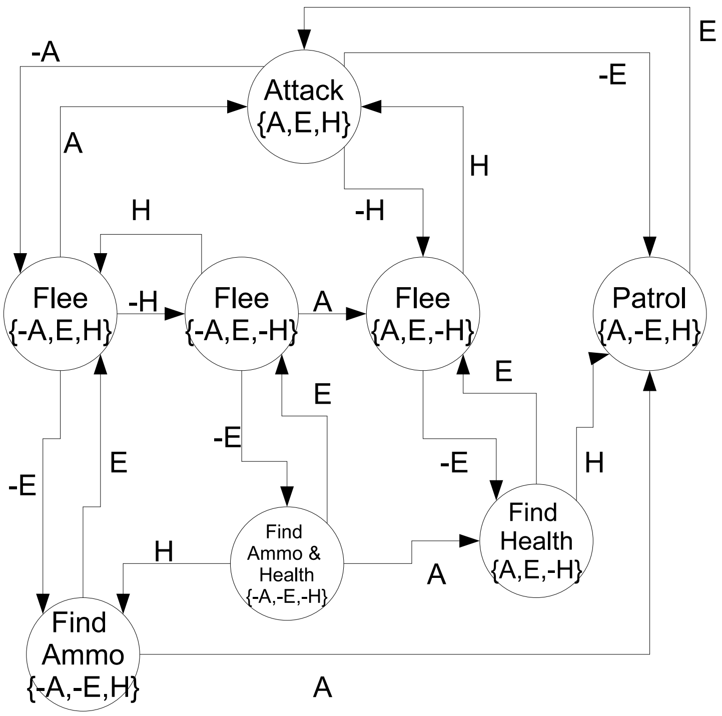

Figure 2.3: Final FSM

It is obvious that for each state variable added the complexity of the FSM is increased. In worst case the number of states in the FSM is $2^{\#state\ variables}$ and the worst case number of transitions would be $(\#state\ variables) \cdot 2^{\#state\ variables}$.

In a game of domination the bots need not only to look out for themselves, they have to cooperate and act in their teams best interest. In order to do this, it is needed to implement some sort of grand strategy for the bots to follow. The bots would have different goals and thereby also perform different tasks. Furthermore they would need to take into account the current state of the other bots on their team. All of this would imply implementing several FSMs to control the tasks of the different bots.

These complexity issues are further discussed in [4], which treats the advantages of using a planner over FSMs. Furthermore it is beyond the scope of this paper to create such FSMs, however the complexity of the task is one of the main motivations behind the paper.

2.2 Automated Planning

As an alternative for using FSMs (see section 2.1 on page 5) in first person shooters, planning is proposed [4] [3] [2]. This section will first outline the general concepts of planning, and then give a comparison between the use of FSMs and the F.E.A.R. project [4].

2.2.1 Planning

A classical planner takes a description of the initial state (s_0 or set of initial states S_0), the goal state (s_g or set of goal states S_g) and a set of all actions (A) as input. It then searches through the state space by applying actions until a goal state is reached. From that search, a sequence of actions makes up a plan (π) that will lead from an initial state to a goal state.

In classical planning, any planning domain can be described as a state-trinsition system:

$$\Sigma = (S, A, \gamma)$$

Where as above, S is the set of states, A is the set of possible actions and γ is a function that produces a state s' by adding an action to s.

An action a is usually made up by a set of preconditions `precond(a)` and a set of effects `effects(a)`. An action could be described as followes:

```
move1(b,l,m)
    /* bot b moves from location l to m */
    precond:   at(b,l)
    effects:   -at(b,l),at(b,m)
```

In this case, it is required that the bot is at the location it has to move from (`at(b,l)`) before this action can be applied. The result of the action removes the bot from locaion `l` and places it in location `m`.

Due to the large search space, several runtime problems arises with this type of planning. Only by applying severe restrictions to the representation it is possible to minimize to a polynomial worst case running time[1].

Furthermore the classical planning approaches does not account for uncertainties. If the state of the world were to change while planning or while the plan was being executed, the planner would be forced to re-plan. With the amount of time it takes to create a plan and given the challenges with regard to the runtime constraints identified in the domain analysis (section 1.2 on page 3), there is a risk that the plan would keep getting invalidated.

2.2.2 Planning in F.E.A.R

The planner used in F.E.A.R [4] is based on [6] Goal Oriented Action Planner (GOAP). Here the actions compete for activation, and as soon as an actions preconditions are met, it is activated. The main difference between the GOAP planner and the F.E.A.R. project is that F.E.A.R. has added a cost per action, meaning that if action a_1 with a cost of 8 and action a_2 with a cost of 2 both have their preconditions fulfilled, a_2 will be used. To search for the lowest cost action to activate, an $A*$ algorithm is used [7].

The article describes, that by using a planner they only had to add actions and goals (see figure 2.4).

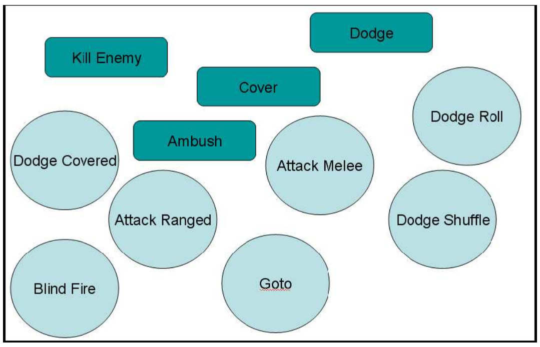

Figure 2.4: Actions in the F.E.A.R engine

The planner would be in charge of connecting the actions with one another,

[1]See chapter 2: Representation for classical planning and chapter 3: complexity of classical planning in [1] for further details.

while in an FSM this would be up to the designer of the AI (see figure 2.5).

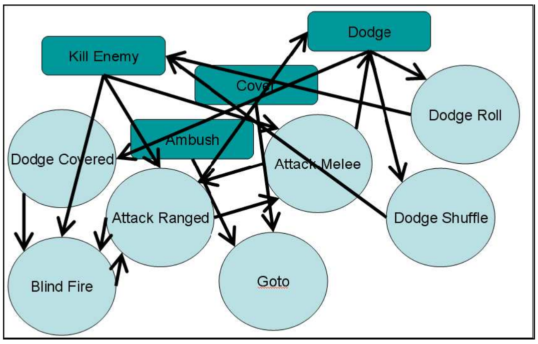

Figure 2.5: Actions connected into an FSM

This is presented as a huge advantage, if the designer were to connect all the actions into an FSM, adding new actions later on would be very complicated. An example of this is given in the F.E.A.R article [4]: they wanted to get their NPCs to turn on the light whenever entering a room. In the FSM version they would have to modify every state that could make the given NPC enter a room. However when using the planner, they simply had to add a `LightsOn` precondition to the `Goto` action, which would effect every goal that was satisfied by using the `Goto` action.

2.2.3 Domain Specific Planning

As shown in the domain analysis, in section 1.2 on page 3, Unreal Tournament has four main challenges to keep in mind when creating a planner:

- **(Time constraints)** It is a fast paced game, so the planner has very limited computation time when it comes to plan creation.
- **(Uncertainty)** The game is real-time, the opponents are unpredictable and able to change the state at any time. This adds a high amount of uncertainty for the planner.
- **(Temporal actions)** Actions take time, meaning that the planner would have to wait for one action to complete before the next can be executed.
- **(Concurrent actions)** The actions of different bots happen concurrently, so the planner has to be able to coordinate the actions of several bots.

In [1] chapter 1.5. a restricted model is presented and eight assumptions are given. These assumptions are a way to measure the complexity of the domain when it comes to planning. When all of the assumptions hold, the problem can easily be solved using classical planning. However for each assumption that does not hold, the complexity of the problem is increased which poses as a further challenge for the planner. These assumptions are going to be used as a guideline to identify which problems the planner should be able to handle. Only two of the eight assumptions hold when using this type of FPS as the domain.

The domain is simplified to only controlling one bot, team coordination will be handled in chapter 3.

A0 (Finite Σ) The state transistion system Σ has a finite set of states. It can seem that it is infinite, however no new objects are brought into the world, and all possible events can be accounted for.

A1 (Fully Observable Σ) The bot is provided with sensory information, which makes the system partially observable, hence the bot does not know what is happening if it can't see it.

A2 (Deterministic Σ) The bot plays against opponents that can change its state, this results in a system that appears non-deterministic.

A3 (Static Σ) For the same reason as A2, other bots might change the system and it is thereby dynamic and not static.

A4 (Restricted Goals) A goal is usually a specific state s_g, or a set of goal states S_g, which the system desires to reach. The goals are extended by adding e.g. subgoals which could express intermediate states either to avoid or to preferably reach as sub-goals. A subgoal can also be some sort of requirement such as patrolling two locations exactly twice.

A5 (Sequential Plan) The plans can be kept sequential. Even though they might be invalidated, a new sequential plan can be planned for. This only holds for a planner which controls one bot, since ceveral bots would execute their actions concurrently.

A6 Implicit Time Actions take time, running from one place to another is not done instantaneous. This means that time is not represented implicitly.

A7 Offline Planning For the same reason as A2, offline planning could result in invalid plans, and re-planning would constantly be necessary.

As shown above, only assumptions A0 and A5 holds. A domain that does not support assumption A1, A2 and A3, can't possibly support A7, and a solution for these three assumption would also be a solution that would deal with A7. Therefore will A7 no longer be discussed.

An alternative, to classical planning, is by describing planning problems using Hierarchical Task Networks (HTN). The main difference between classical planning techniques and planning with HTNs are that HTNs does not search through a state transition system trying to find a path to achieve a set of goals. An HTN is made up of simple- and compound-tasks. Each compound task is then decomposed into either other compound- or simple-tasks. This is continued until there is only simple tasks left, which can then be executed (see section 2.3 for further details).

The objective in an HTN planner is therefore not to achieve a set of goals, but to perform some set of tasks. It plans not for a complete solution and the information it relies on depends on the domain. So even though the system is only partially observable, an HTN planner might still be able to overcome A1.

In [1] there is not mentioned anything about uncertainty for HTN planners, however in [2] it is mentioned that by continuously monitoring the conditions of the topmost task. When the applicability of it falls below a certain threshold, another task should either be selected or the current one should be re-planned. This will be handled in section 3.1 on page 15 and deals with A2 and A3.

As explained in [1] chapter 11.8 Extended Goals, some of the extended goals are easily overcome due to the domain specific ways HTNs are implemented, while others need the HTN syntax to be extended. See section 3.1 on page 15

Finally the notion of time and thereby assumption A6 will also be handled in section 3.1.

2.3 Hierarchical Task Networks (HTN)

A Hierarchical Task Network is a tree structured network of tasks. A task can either be a compound task or a simple task. A compound task can be recursively decomposed into other compound- or simple tasks. Simple tasks are domain specifically implemented and corresponds to an action that changes the world state. However decomposing compound tasks does not change the state. The compound tasks represents higher level goals and encapsulates the strategies to achieve them [2].

A task in an HTN is also referred to as a method. The definition of a method is in [1] given as a 4-tuple:

$$m = \langle name(m), task(m), subtasks(m), constraints(m) \rangle \tag{2.1}$$

- $name(m)$ is an expression of the form $n(x_1, ..., x_k)$, where n is a unique method name and $x_1, ..., x_k$ are all of the variable symbols that occur anywhere in m.
- $task(m)$ is a compound task.
- $subtasks(m)$ is a set of compound and simple tasks.
- $constraints(m)$ is a set of constraints, either on the use of the methods subtasks or constraints in regards to the current state of the world.

A simple example of an HTN:

```
move2(b,l1,l2,l3) /*method to move bot b from location l1
                    to l2, and then from l2 to l3*/
task:           move-double(b,l1,l2,l3)
subtasks:       t1 = move-single(b,l1,l2)
                t2 = move-single(b,l2,l3)
constraints:    at(b,l1), t1 < t2

move1(b,l1,l2) /*method to move bot b from location l1
                 to location l2*/
task:           move-single(b,l1,l2)
subtasks:       t = move(b,l1,l2)
constraints:    at(b,l1)

move0(b,l1,l2) /*method to do nothing if b is already at l2*/
task:           move-single(b,l1,l2)
subtasks:       none
constraints:    at(b,l2)
```

`move1` is a compound task wrapped around the simple task `move`. `move2` uses `move1` two times, its constraints makes sure that the two `move1` calls are done in the right order. Furthermore both methods ensures that the bot is in the correct location, so that it is actually able to make the correct movement. `move0` is an alternative to `move1` in the case where the bot is already at its destination.

The `move2` call can be translated into an and/or graph (see figure 2.6 on the next page).

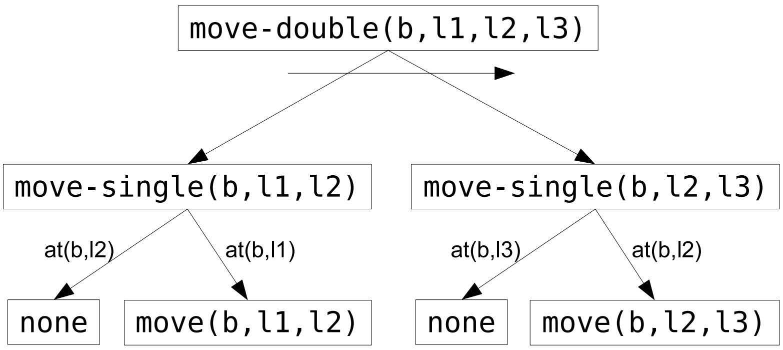

Figure 2.6: And-or graph corresponding to the `move2` and `move1` methods

The `move2` method is decomposed into its subtasks and the subtasks that are non-simple tasks are recursively decomposed and so on. Finally `move2` is fully decomposed into simple tasks, which then formulates the plan π. In figure 2.6 the leaves of the tree represents the plan $\pi = \{move(b, l1, l2), move(b, l2, l3)\}$ in the case where the bot starts at location $l1$. The horizontal arrow indicates an 'and' node, which means that all the nodes children must be executed in the order denoted by the arrow. An 'or' node is the case where there is no horizontal arrow, meaning that only one of the children is executed, which one depends on the constraints.

Chapter 3

Planning in Unreal Tournament

3.1 Applying HTNs to a First Person Shooter

This section will take the challenges outlined in section 2.2.3 and discuss possible solutions, which can then be used to modify the standard HTN description given in section 2.3 in order to create a planner that works in the domain described in section 1.2.

3.1.1 Domain Specific Challenges

In section 2.2.3 on page 10 eight assumption from [1] are given in order to identify where possible difficulties might arise when planning in an FPS game. These assumptions were analyzed and it was concluded that assumption A0 and A5 remains, while the rest are violated by the domain. Before a planner can be implemented the violated assumptions will be discussed, and a possible strategy to solve them will be given.

3.1.2 A1 (Fully Observable Σ)

The bot receives sensory information, and has very limited knowledge about places it can not observe. This means that it can not rely on items that other bots can interfere with or that the opponents are located where they were last seen. However the bot does have information about what it observes, and the status of the domination locations. This means that the bot can only react to what it can see and the general status of the game. It is therefore these two elements that make up the state of the world. Since the bot only has partial information about the world, observations returns sets of states. Two different states in a fully observable system, might be perceived as the same state with partial knowledge. In classical planning this would increase the size of the search space from the set of states in the domain, to its power set. When dealing with HTNs there are no searches done over the state space, the HTN methods are domain specifically designed and can therefore only use what is observable in the given domain. It is therefore left to the design of the specific HTN methods to deal with the problems of acting in a partially observable world.

3.1.3 A2 (Deterministic Σ) and A3 (Static Σ)

As clarified in section 2.2.3 on page 10 the world appears to be non-deterministic and highly dynamic. The bot has no knowledge about how its opponents act/react and these opponents can possibly change the state of the bot. This means that the bot can not plan for the actions done by its opponents and its current plan might be invalidated at any time.

This is not handled by HTNs in [1], however in [2] the applicability of the HTN methods used are continuously evaluated. This means that if the applicability of the current method falls below a predefined threshold, either a new method is selected, or the current method is re-planned[1].

Following this approach a method such as `patrol-dom-location(d1,d2)` (see figure 3.1 on the next page) will evaluate to the plan $\pi = \langle a_1, a_2, ..., a_k \rangle$ and then be executed. If `patrol- dom-location(d1,d2)` is an HTN method that patrols two domination locations `d1` and `d2` and then stocks up on ammunition, the plan π could be described as the plan where the bot first moves to location `d1` and attacks possible bots there, then moves to `d2` and attacks possible bots there and finally stocks up on more ammunition. The actions that count on uncertain conditions, such as whether or not an enemy is at either of the

[1] Meaning that the method is re-evaluated, possibly creating a new HTN tree if some of the conditionals have changed.

domination locations or whether or not the ammunition, which is planned to be collected, will still be there, are not known during the planning phase and has to be either simple actions, which can then act as small FSMs, or re-planned once the knowledge is acquired.

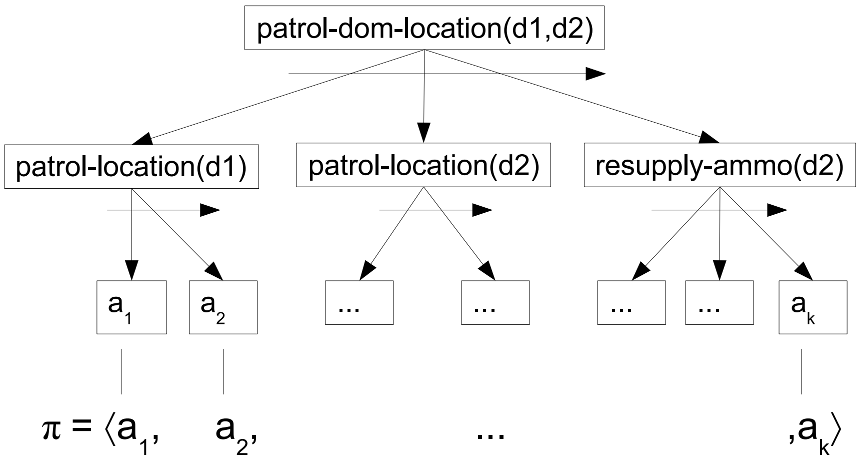

Figure 3.1: And/or graph of `patrol-dom-location(d1,d2)`

Following the method described in [2], the bot would have to assume some facts about these uncertain details, e.g. it might assume that it would encounter one enemy at each domination location and that a certain ammunition item will be available near location `d2`. However when the bot would arrive at the different locations, its assumption would most likely be invalid and it would have to re-plan.

Instead of following the traditional approach where the complete plan is extracted from the HTN and then executed, I propose an approach where the actions are executed right after they are selected. Since the plan is already represented in the HTN choosing action a_i has no influence on the choice of action a_j, where $i \neq j$ and $\{a_i, a_j\} \in \pi$. This means that right after a methods conditionals are tested its actions are executed, which gives the action less time to be invalidated due to the interference of other bots.

With this approach applied to the example given in figure 3.1, the bot will first plan how to move to `d1` and then execute it, meaning that it will move to location `d1`. Then it will plan how to attack possible bots there. With the approach described earlier it would have made some assumptions about the conditions at `d1`. However now that the bot is at `d1` there are no longer any uncertain conditions and the bot can plan while knowing all the facts. This means that if there is a hostile opponent there the bot can plan how to kill it, otherwise it will just move on and plan how to get to location `d2`. At location `d2` the bot again plans to kill the opponents present and then plans how to get some more ammunition.

With the original approach the planner would most likely, in this example, have had to re-plan three times. However by dynamically executing the actions while traversing the HTN, less assumptions would have to be made, and the number of re-plannings are significantly lowered.

However this approach has a downside as well. If the preconditions of action a_k can not be satisfied, actions $a_1 - a_{k-1}$ will still be executed since the preconditions of a_k are only checked right before the method is supposed to be used. This means that the bot would have executed all the former actions, but is not able to complete the HTN. There are two ways this could be handled:

1. All the preconditions in the HTN are verified before the HTN method is selected. However this leaves us back at the original problem, since the system is not fully observable, assumptions would have to be made in regards to the preconditions that deals with uncertain elements (e.g. whether or not a bot is at a future location).
2. Leave it to the design of the HTN to place complete and relevant preconditions at the topmost HTN methods. Meaning that if it can be decided whether or not a method can be fully executed when it is chosen, the preconditions for this should be placed as high as possible in the HTN tree.

Furthermore the idea of adding goals to the HTN methods are introduced. The goals are suppose to be evaluated in the same way as the preconditions, however if a methods goals are already fulfilled, the method will succeed without calling any of its subtasks. The purpose of this is that if the subtask `findAmmunition` is applied to a method. This subtask should have the goal `hasAmmunition`, meaning that if it is called and the bot already has enough ammunition, the subtask will simply terminate as if it is successfully completed. Otherwise it will make the bot find ammunition.

This changes the structure on how to write HTNs. Instead of having a precondition on the `attack` method stating that it should have ammunition, the subtask `findAmmunition` should be added to the `findEnemy` method, causing the bot to find ammunition before it starts looking for enemies, and in the case it has ammunition, the `findAmmunition` method would have no effect.

3.1.4 A4 (Restricted Goals)

As described in [1] chapter 11.8, the expressiveness of the HTN itself contains the use of extended goals. Which is shown in the following examples:

Consider the HTN example from section 2.3 on page 12. In the case where we would want to add a subgoal preventing the bot from moving to location `bad-loc`. This can be achieved by adding a precondition to the `move2` method. Thereby preventing the `move2` method from being executed in the case where the result would move the bot to location `bad-loc`.

```
move2(b,l1,l2,l3) /*method to move bot b from location l1
                    to l2, and then from l2 to l3*/
task:          move-double(b,l1,l2,l3)
subtasks:      t1 = move-single(b,l1,l2)
               t2 = move-single(b,l2,l3)
constraints:   at(b,l1), t1 < t2, l2 != bad-loc,
               l3 != bad-loc

move1(b,l1,l2) /*method to move bot b from location l1
                 to location l2*/
task:          move-single(b,l1,l2)
subtasks:      t = move(b,l1,l2)
constraints:   at(b,l1)

move0(b,l1,l2) /*method to do nothing if b is already at l2*/
task:          move-single(b,l1,l2)
subtasks:      none
constraints:   at(b,l2)
```

This could of course also have been added in the `move1` method, however we might want to add a method later on called `emergency-move` that ignores this extended goal. Furthermore it is desirable to put the preconditions at the highest level possible of the HTN.

Consider the case where we would like an extended goal stating that the bot should move from location `l1` to location `l2` and back again exactly two times. This can be done by adding the method `move3` and the auxiliary method `round-trip` as follows:

```
move3(b,l1,l2) /*moves bot b from location l1 to l2 and back
                 back again exactly two time*/
task:          two-times-round-trip(b,l1,l2)
subtasks:      t1 = round-trip(b,l1,l2)
               t2 = round-trip(b,l1,l2)
constraints:   at(b,l1), t1 < t2

roundTrip(b,l1,l2) /*moves bot b from location l1 to l2 and
                     back back again exactly two time*/
task:          round-trip(b,l1,l2)
subtasks:      t1 = move-single(b,l1,l2)
               t2 = move-single(b,l2,l1)
constraints:   at(b,l1), t1 < t2
```

This second example can not be expressed as a classical planning problem, and shows the expressiveness of HTNs (see [1] Chapter 11.8 Extended Goals).

3.1.5 A6 (Implicit Time)

As described in section 2.2.3 on page 10 every action takes time, which means that time is not represented implicitly. However neither time nor actions have an absolute value. Even though the bots speed and movement distance could be used to calculate the duration of an action, it would be highly inaccurate due to network latency[2] and possible interfering events. As described in the domain analysis, in section 1.2 on page 3, the domain is event based so instead of using time to represent the duration of the actions, server events will indicate that an action has completed. E.g. when the bot is going to move from one location to another, the planner executes the move command and then monitors the bots sensory information. When the bot no longer has a velocity, it would indicate that it has stopped moving, and if interfering events has not occurred during the movement, it can then be assumed that the bot has safely reached its destination.

By solely using sensory events to represent the duration of actions the complexities of dealing with a real time system is avoided as much as possible.

[2] The time it takes from when the bot sends an action, to the game server receives and executes it.

3.2 The Complete Planner

This section will describe the theoretical solution for an HTN inspired planner for the domain of Unreal Tournament described in section 1.2 on page 3.

3.2.1 HTN syntax

The HTN syntax used up until now is the example syntax used in [1]. However in order to accommodate the dynamic HTN approach suggested in section 3.1.3 on page 16, a few changes are made to the syntax.

An HTN method is made up of a 4-tuple:

$$m = \langle Head(m), Goals(m), Preconditions(m), Subtasks(m) \rangle$$

where each method returns success or fail.

- $Head(m)$ is an expression of the form $n(x_1, ..., x_k)$, where n is a unique method name and $x_1, ..., x_k$ are all of the arguments used by m. This is a collapsed version of $name(m)$ and $task(m)$ given in section 2.3 on page 12.
- $Goals(m)$ is a set of boolean evaluations which will instantly make m return successfully if they all evaluate to true. These goals are described in the same way as the preconditions, however the purpose is to make a method return as if it has completed its task if the task were already achieved beforehand. E.g. if a bot is to move from location `l1` to location `l2`, but is already located at `l2` it would return successfully. This is instead of writing a second move method with the precondition `at(b,l2)` that does nothing. This goal is added to the first move method, and thereby making the method behave as if the bot had successfully moved.
- $Preconditions(m)$ is a set of constraints which needs to evaluate to true for m to be applicable. Otherwise m will fail.
- $Subtasks(m)$ is a set of compound and simple tasks. Each line in the syntax represents an 'and' node in the corresponding HTN tree. Meaning that each of the lines must return successfully in order for the method to return successfully. Each line is made up of one or more subtasks. These subtasks act as the 'or' nodes in the HTN tree, meaning that only one of them needs to succeed in order for the line to succeed. If a subtask in a line fails the next subtask is executed, if there are no more subtasks left in the line the line fails and so does the method.

Below is a small example of two HTN methods:

```
/*method to move bot b from location l1 to l2,
  and then from l2 to l3*/
Head move-double(b,l1,l2,l3)
Goals
at(b,l3)
Preconditions
none
Subtasks
move-single(b,l1,l2)
move-single(b,l2,l3)


/*method to move bot b from location l1 to location l2*/
Head   move-single(b,l1,l2)
Goals
at(b,l2)
Preconditions
at(b,l1)
Subtasks
move(b,l1,l2)
```

In the case where the bot is initially located at `l1` both `move-single` methods would be invoked and their corresponding simple tasks `move` would be executed. In the case where the bot is initially located at `l2` the first `move-single` would instantly succeed and the second one would result in the bot executing the corresponding `move` action. And finally in the case where the bot is initially at `l3`, the `move-double` method would instantly succeed, appearing as if the bot has moved.

Recursion: This syntax allows both direct- and indirect recursion[3]. However given the undefined underlaying implementation and the complexity of trying to solve the halting problem[4]. It is not possible to prevent never ending loops. Furthermore depending on the implementation details (see chapter 4), using recursion to keep the HTN methods from terminating could result in memory problems.

Total Ordered Planning: Given by the syntax it is clear that the HTN methods only supports total ordered planning (as described in [1] Chapter 11),

[3] Direct recursion, meaning a method that calls itself, while indirect recursion is when two or more methods call eachother in a cyclic manner

[4] See http://en.wikipedia.org/wiki/Halting_problem

meaning that two methods subtasks can not interleave with each other. The preconditions of an HTN method only decides the methods applicability, while the ordering of the subtasks are given by the order in which they are written. A partially ordered approach would have left more decisions to the planner and given a more random bot behavior, however since the approach of executing the tasks as soon as they are found is pursued, it is not possible.

Neither [3] nor [2] describes any of these choices, however given the syntax presented in the two papers, it seems that neither of them considers partially ordered HTNs.

3.2.2 Time

Since there is no notion of concurrency within an HTN, the actions of a bot has to be executed sequential. This means that when a bot is ordered to run from one location to another, the planner has to wait for the bot to arrive at its destination before it is given a new task. Because all simple tasks, and thereby the bots actions, are domain specifically implemented, it is up to the individual task to know when it has completed and then return[5]. However it could occur that the bot got interrupted while performing a task, e.g. an unfriendly bot started to shoot at it or someone was standing in the way and thereby blocking its path.

This problem can be handled in two ways:

1. Every kind of interference is handled within the implementation of the simple task. This would however make the simple tasks somehow complicated to implement, and it would be difficult to prove that all possible situations were accounted for.

2. The task fails if it is interrupted, and it is left to the designer of the HTN to create the methods so that they are able to recover from every possible situation.

A third possibility is pursued, which is the mixture of the two. Interference that has no direct influence with the completion of the task or that are easily recovered from, should not make the task fail. However such a solution requires that the implementation of the simple tasks are well documented so that their behavior are known to the designer of the HTN.

[5]See chapter 4 for implementation details on how to identify when a task has completed.

Other work: The notion of time is not mentioned in [1]. [2] uses the idea of grouping bots up based on the belief that they would have a greater chance of killing an opponent if they engage in numbers. However the paper does not mention the issue of time or how the bots synchronize these actions.

Furthermore in [3] an HTN is used to coordinate the strategy of the bots, while FSMs control the bots themselves and implements the tasks given by the HTN. However it is not described how the HTN handles concurrency. If their planner controls more than one bot and the actions of the bots takes time, it is impossible to avoid concurrency issues. E.g. when one bot has completed its task, should it then wait for all the other bots to complete theirs, or is it possible to asses the status of the other bots and then assign a new task.

3.2.3 Team Coordination

When dealing with the coordination of a team of bots, the bots act concurrently. Since time is handled as described in section 3.2.2 on the previous page and that there are no simple way of converting an HTN to handle concurrency, each bot needs an HTN of their own. However having a team of bots going about their own business, controlled by each their HTN, does not make them cooperate to achieve a grander goal. A coordinating program is needed.

The coordinator does not have any notion of time, and its choices are solely based on the preconditions available.

The job of the coordinator is to assign strategies to each of the bots. This is done by giving the individual bot a set of HTN methods it is allowed to use. The bot repeatedly finds an applicable method, in this set, and executes it. The coordinator can, at any time, update the set contained in any of the bots and thereby change the behavior of the bot. The coordinator itself is controlled by an HTN that describes the grand strategy of the team.

The coordinator can then make the bots cooperate tightly together by only allowing them to use one method, or it can make the bots more independent by giving them a set of possible methods. This way the bots keep their sequential planning, while acting concurrently with one another.

This set contains only top level methods and the bots are allowed to use any subtask used by these methods.

Other work: [3] also uses a bot/coordinator principle, however the bots are controlled by FSMs and their available actions are not easily extendable. In [2] an HTN also gives out tasks to different bots, however it is not clarified how the coordination between the bots work.

In [10] it is proposed that the plans of different bots are summarized and then compared, though this is mostly to prevent colliding plans. Given the computational time constraints it does not seem feasible in the domain described in section 1.2.

3.2.4 HTN Algorithm

The HTNs are executed dynamically as previously proposed. This means that the algorithm will traverse the corresponding HTN tree and execute the leftmost simple task first, then the second leftmost and so on.

When an HTN method is selected it is executed by algorithm 1 on page 27. Line 1-3 checks whether all the goals evaluate `true`, if they do the algorithm will return `true` disregarding all preconditions. Line 4-6 checks the preconditions and if one of them evaluates to `false` the algorithm will return false.

Given an HTN method with the following subtasks:

Subtasks
a_1, a_2, a_3
b_1
c_1, c_2

These subtasks can be represented as a list of lists $S = \langle l_1, l_2, l_3 \rangle$ (see figure 3.2 on the next page).

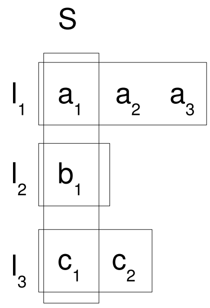

Figure 3.2: Subtasks represented as a list S of lists l_1, l_2, l_3

Considering algorithm 1, the outer `foreach` on line 8 iterates through S and the inner `foreach` on line 10 iterates through l_i, where i is the current index of S. Each element in l_i are called recursively, if it returns true the inner loop breaks and the outer loop continues with its next element. If it does not return true the inner loop continues with its next element. If none of the elements in l_i returns true the `result` variable would not have been set to `true`, on line 12, which will make the method return `false` on line 17.
In other words, if one of the tasks in each l_i returns true the method call will return true.

In the case where the task executed is a simple task it will still check both the goals and the preconditions. Whether or not the execution of the task returns `true` or `false`, in lines 21-27, depends on the implementation of the specific task and whether or not it gets interrupted.

Algorithm 1: HTN-Execute(Task t)

```
Data: G set of goals, P set of preconditions, S list of lists of subtasks
Input: HTN task
Output: true if the task succeeds, otherwise false
if All the goals in G evaluate to true then
    return true
end
if There exists a precondition in P that evaluates to false then
    return false
end
if t is a compound task then
    foreach Sublist l in S do
        result ← false;
        foreach Subtask u in l do
            if HTN-Execute(u) = true then
                result ← true;
                break;
            end
        end
        if result = false then
            return false
        end
    end
    return true
else t is a simple task
    Make the bot execute the task;
    if the bot is interrupted then
        return false
    else
        return true
    end
end
```

3.2.5 Further Work

Many extensions could be added to the syntax given in section 3.2.1 on page 21.

- The use of boolean operators when defining the goals and the preconditions. As it is now the different evaluations are separated by conjunctions.
- Return values from subtasks, which could be used in other subtasks.
- Arithmetic operators both to be used on return values and when defining goals and preconditions.
- List manipulation, so e.g. a list of possible targets could be handled in some cyclic manner.

One of the major difficulties is to keep track of concurrent actions within an HTN. An interesting topic for further research would be to expand the HTN syntax to include a concurrent operator ($\oplus$). This operator should make it possible to execute tasks concurrently and then rendezvous when both tasks have terminated. E.g. making two bots attack the same enemy at the same time by calling:

`attack(b1,t1)` $\oplus$ `attack(b2,t1)`

Or making two bots meet up at a specific location:

`moveTo(b1,l1)` $\oplus$ `moveTo(b2,l1)`

Chapter 4

Implementing the Planner

This chapter will describe the relevant implementation details of an HTN inspired artificial intelligence for controlling a team of bots in the game of Unreal Tournament.

4.1 Overview

The implementation consists of four major parts:

- **Coordinator:** The coordinator tells the bots which of the available HTN methods they should pursue.
- **Bot:** The bots get input from the coordinator and sensory information from the server, which they use to chose and execute HTN methods.
- **HTN:** The HTN which the bots processes in order to plan their actions.
- **ComHandler:** The ComHandler deals with all communication to and from the UT server.

See figure 4.1 on the next page.

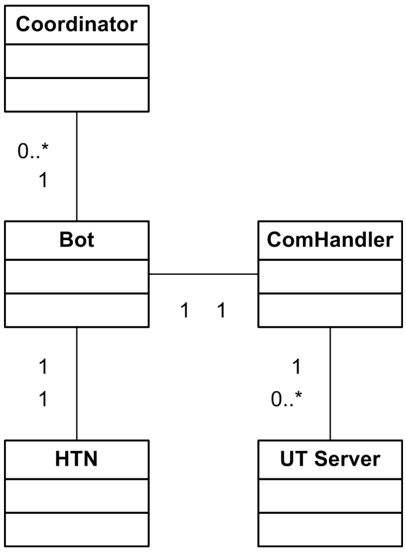

Figure 4.1: Implementation overview

As shown in figure 4.1 the coordinator has any number of bots, which makes up the team. Every bot has its own HTN and server communication.

4.2 UT Server

The UT server runs a map which consist of domination locations, navigational nodes and inventory nodes.

Figure 4.2: Map DomStalwart

Figure 4.2 is a screenshot from the program tclviz that visualizes the activities on the UT server without having to load the game. The domination locations are the white marks outlined with a white circle and when a bot runs over a domination location it changes color to match the bots team. The blue marks represents navigational nodes that the bot can use for orientation. The inventory nodes are represented by the pink marks and indicate that an inventory item will be at that location.

The sensory information sent by the server contains all nodes and domination locations visible to the bot.

4.3 HTN

4.3.1 Syntax

The HTN methods are described in almost the same syntax as given in section 3.2.1 on page 21. However a few simple changes have been made to ease the parsing.

First of all the simple tasks and the compound tasks are written in two different files. This does not effect the workings of the HTNs, however when the syntax is parsed it is not needed to identify the simple tasks among the compound ones.

Secondly the parenthesis around the arguments are removed and the arguments are separated by a space instead of a comma. Furthermore if a subtask uses one of the arguments, given to the method, it is identified by its position and not by its name. E.g. if a subtask is to use the first argument it should be identified by `$0`, the second by `$1` and so on. It is also possible, for the designer of the HTN, to use predefined values instead of the arguments.

```
Head RunTo destination
Goals
At $0
Preconditions
neighborTo $0
```

The `RunTo` method is a simple task, which means that its action is implemented. The goal `At` uses the first argument given in the argument list and so does the precondition `neighborTo`.

In this second example the `RunAround` method does not use any arguments, however it uses two simple tasks: `RunToRandom`, which makes the bot run to a random visible node, if there are no visible nodes in sight the task fails and `Rotate`, which makes the bot rotate a given number of UT units[1]. `RunAround` shows how a compound task can have a predefined value as an argument for the `Rotate` method.

[1] $2\pi = 65535$ UT units

```
Head RunAround
Goals
Preconditions
Subtasks
RunToRandom,Rotate 2000
RunAround
```

4.3.2 Data structure

The abstract class `Task` contains the common elements from the two subclasses `SimpleTask` and `CompoundTask`. The common elements include the name, goals, preconditions and methods to evaluate the preconditions and the goals. Furthermore the `Task` class has an abstract method `doTask`, which must be implemented in both subclasses.

`SimpleTask` implements `doTask` by implementing lines 1-6 and lines 21-27 as described by algorithm 1 on page 27.

`CompoundTask` implements `doTask` by implementing lines 1-6 and lines 7-20 as described by algorithm 1 on page 27. This means that when `doTask` is called, the `if` statement on line 7 is handled by the polymorphic structure of `SimpleTask`, `CompoundTask` and their common superclass `Task`. See figure 4.3.

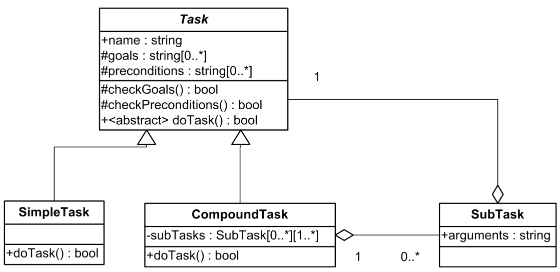

Figure 4.3: UML diagram of Task

`CompoundTask` has a list of lists of `SubTasks`. A `SubTask` contains a `Task`, meaning either a `SimpleTask` or a `CompoundTask`.

4.3.3 Parsing

The file containing the `SimpleTasks` are parsed first. A `SimpleTask` has no subtasks, meaning there are no recursions to keep in mind when parsing these. The `SimpleTasks` are parsed line by line and the different goals and preconditions are added as they are identified. If a `SimpleTask` can not be matched with its corresponding implementation, an exception is thrown.

When parsing the `CompoundTasks` the parser runs through the file creating an 'empty' `CompoundTask` for each method. Then the parser runs through the file again adding all the elements such as the goals, preconditions and subtasks. The file has to be parsed in two iterations since the first `compoundTask` could potentially have the last `CompoundTask` as a subtask.

4.3.4 Execution

The HTN is executed as described in algorithm 1 on page 27. However a few details need to be clarified in regards to time and concurrency.

When a `SimpleTask` is executed it sends a message to the server giving the bot a corresponding command. Then the `SimpleTask` has to wait for the outcome, of the given action, before it can return. This is controlled by a monitor. When a movement action is initiated the bot first calls a method on the monitor, which makes it record the bots position. Then the movement command is sent to the server and a method `waitForMovement` is called on the monitor. The monitor will then put the executing thread to sleep, making it wait for the state where its current position is different from the one recorded and the bot no longer has a velocity. If the bot has not been interrupted during the action, it is assumed that the task has been executed correctly and therefore returns `true`.

Conditionals: The goals and preconditions used in the HTN has to have corresponding implemented methods. The syntax does not support any form of arithmetics or boolean expressions in regards to the conditionals. The `Conditionals` class has a public method `checkConditional` which takes the name of the conditional, given in the syntax, and its list of arguments. The `checkConditional` then identifies and executes the corresponding implementation. If the conditional does not exist the method will throw an exception.

Following is a complete list of all the implemented conditionals:

- `At loc` - returns `true` if the bot is at location `loc`.
- `neighborTo loc` - returns `true` if the bot has a direct passage to location `loc`.
- `hasWeapon` - returns `true` if the bot has a weapon equipped.
- `hasMoreAmmoThan amount` - returns `true` if the bot has more ammo than specified by `amount`.
- `hasMoreHealthThan amount` - returns `true` if the bot has more health than specified by `amount`.

4.3.5 Simple Tasks

This is a complete list of all simple tasks implemented:

- `RunTo loc` - Makes the bot turn and run to location `loc`. Fails if the bot does not have direct passage to `loc`.
- `RunToRandom` - Makes the bot run to a random visible node. Fails if there is no nodes in line of sight of the bot.
- `Rotate yaw` - Makes the bot rotate `yaw` amount of UT units[2].
- `TravelTo loc` - Makes the bot travel to the location defined by `loc`. Fails if the bot is killed.
- `ShootAtEnemy` - If there is an enemy in line of sight the bot will open fire. If the enemy dies or runs out of line of sight the bot will shoot at the next enemy in sight. This will continue until there are no more enemies in sight of the bot. If there were no enemies visible to begin with, the method will retun `false` and otherwise `true`.
- `RunToRandomINV` - Makes the bot travel to a random inventory node.
- `AddBot name team` - Adds a bot to the game with the given `name` and on the specified `team`.
- `StartBot name` - starts the bot identified by `name`.
- `AssignTask botName taskName` - Assigns the task specified by `taskName` to bot `botName`.

[2] $2\pi = 65535$ UT units

- `ClearTasks botName` - Removes all tasks from the bot specified by `botName`.

The last three methods are mainly there for the coordinator, however it is possible for the bots to assign tasks to each other through the coordinator.

4.4 Bot

The `Bot` class extends the `Thread` class so it can be run concurrently. Furthermore it has a `State` which maintains the actual state of the world such as visible enemies, current ammo, weapons, health, etc. The `ComHandler` takes care of the communication with the UT server and it updates the `State` whenever messages are received. The `State` is also used to access the monitors needed when the thread has to wait while an action is taking place.

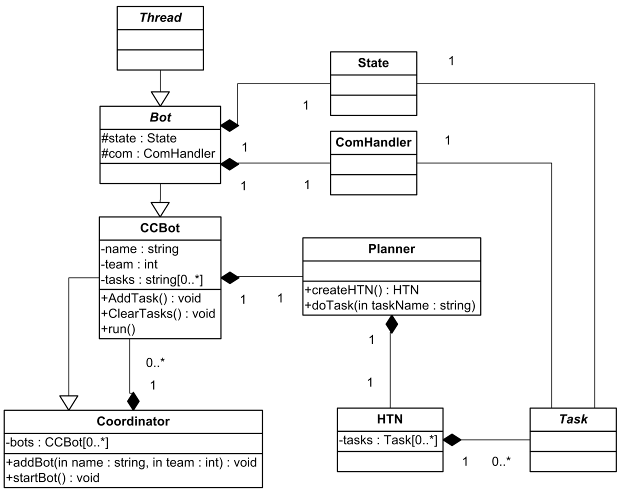

Figure 4.4: UML diagram of Bot

`CCBot` is the bot version that works with the `Coordinator`. The `State` and the `ComHandler` are placed in its superclass so other bot implementations would work on the same framework. The `CCBot` uses the `Planner` in order to build the `HTN` and execute the different tasks. The `Task` knows the `State` so that it has access to its monitors and is able to evaluate the conditionals. Furthermore the `Task` has access to the `ComHandler` so that the actions can be send directly to the server.

The `CCBot` has a list of task names which it iterates through repeatedly and executes one by one. This list can be concurrently manipulated through methods provided by the `CCBot`.

4.5 Coordinator

There `Coordinator` has been designed using the singleton pattern, meaning that only one instance of the class can exist at any time. The instance is obtained through a static method making it accessible to all `CCBots`. The `Coordinator` has a `HashMap` of all the `CCBots` and methods to manipulate their list of task names. The `Coordinator` is a subclass of `CCbot` which makes it inherit the list of task names and it too iterates through and executes this list. Furthermore the `Coordinator` is an entry in its own `CCBot HashMap`, which means that methods can also be assigned to the coordinator. Since the `Coordinator` is known to all bots it is possible for the bots to assign each other tasks through the `Coordinator`.

When the program is started the HTN method named `Main` is assigned to the `Coordinators` task list. Through the HTN definition of this method it is hereby possible to create bots and assign tasks without having to recompile the program. This makes the `Main` method the default entry point.

4.6 Client/Server communication

All communication between the client and the server is done over a TCP socket. This means that when the server is running the client will open a socket and connect to the server. The network API uses a `String` protocol where every message is on the form:

```
\texttt{MSGTYPE {arg1 arg1value} {arg2 arg2value} ... {argn argnvalue}
```

E.g. to initialize a bot on the server, after a connection has been established, the following command is sent:

```
INIT {Team 1} {Name Bob}
```

which will create a bot on team 1, with the name `Bob`.

The client will then start receiving sensory updates from the server. These update messages are either synchronous or asynchronous. The synchronous messages are periodic blocks of updates containing the state of the bot and its surrounding environment. While these blocks are being transmitted no asynchronous messages will be sent by the server. The synchronous messages include:

- Status of the bot: current rotation, location, velocity, name, team, health, weapons, armor, etc.
- Information about the game such as score and domination status.
- Everything visible to the bot such as: Objects on the ground, navigational nodes, other bots, etc.

Whenever an event occurs the bot receives an asynchronous message with the update. The asynchronous messages include:

- Inventory items picked up.
- Chat messages.
- Collisions with walls and other bots.
- Sound information: footsteps, bots picking up inventory items and shooting.
- Damage done/taken.

For full network API see [9].

4.7 Further work

As described under further work in section 3.2.5 on page 28 a lot of extensions could be made to the syntax. Furthermore it might be worth considering making the implementation in a functional language due to the functional definition of the HTN syntax. A functional language might provide an easier implementation of possible return values and list manipulation. Furthermore it might be possible in such a language to make the underlaying implementation tail recursive and thereby avoid memory problems with non terminating recursive calls.

The implementation is created as a proof of concept for this paper and has not gone through any extensive testing. Any bugs that might occur, related related to incorrect syntax, might not produce insightful output. It was never the intention to make a robust system, the purpose was to make a testbed for the use of HTNs in Unreal Tournament. I leave it as future task to shape these ideas into a robust framework.

Chapter 5

Conclusion

Chapter 5 will identify and discuss possible findings discovered by using the HTN inspired planner outlined in chapter 3 to control a team of bots in the game of Unreal Tournament as implemented in chapter 4. These findings will then be used to draw up the final conclusion.

5.1 Findings

The map shown in figure 4.2 on page 31 is used as a test map to run the implementation on. It contains three domination locations and a large number of navigational- and inventory-nodes.

When it comes to formalizing a strategy the HTN really shows its worth. With only a handful of HTN methods it quickly becomes possible to describe quite complex strategies. A good example of this is the top level methods `DefendDomPoint` and `Defend`.

```
Head DefendDomPoint domPoint
Goals
Preconditions
Subtasks
GetWeapon
GetAmmo
Defend $0

Head Defend domPoint
Goals
Preconditions
hasWeapon
hasMoreAmmoThan 1
Subtasks
TravelTo $0
ShootAtEnemy,Rotate 7000
```

The `DefendDomPoint` makes the bot find a weapon, then ammunition and then execute the `Defend` method. Both `GetWeapon` and `GetAmmo` has goal predicates describing that if the bot already has a weapon and ammunition, they will instantly return successfully. This means that once the bot has acquired a weapon and some ammunition it will constantly call the `Defend` method. If the bot should run out of ammunition the `Defend` method will no longer be applicable and the `GetAmmo` will once again make the bot find ammunition. The `Defend` method makes the bot travel to the specified location and then continuously rotate until an enemy is spotted, which will then be attacked.

The path finding implemented in the UT server works great and there is no reason to believe that any other AI would be able to do this more effectively. By running the program several times and letting two teams of bots fight against each other, it has become obvious that a lot more simple tasks are needed. E.g. the current `TravelTo` method only gets interrupted if the bot is killed, this means that even though the bot is attacked it continues to its destination. It would be convenient to have a version where the bot would stop if it got attacked, so that it would be possible to fight back, and even to have a version where the bot would get interrupted if it spotted an enemy so that this could be used as a "patrol" task.

Shooting at the opponents, trying to kill them, is a big part of the domain and the `ShootAtEnemy` is a very simple way to try and implement this. The bot is stationary while performing this action so the outcome of the battle is really a matter of which bot spotted the other bot first. It seems reasonable to conclude that the abilities needed to be a dangerous adversary would be to avoid incoming fire and have a perfect aim. These two abilities are purely reactive and are not something that needs a great deal of planning. It would require a lot of simple tasks to reach this level of reactiveness, however it might still be more effective to have small FSMs controlling the combat. The difference between a simple task and a small FSM is that the FSM maintains several actions and the simple task only performs one. Furthermore a small FSM would integrate well into the framework since it would appear and behave as any other simple task.

5.2 Conclusion

Unreal Tournament is a domain that is non-deterministic, highly dynamic and the AI has very limited computation time to calculate intelligent behavior. It is understandable that Finite State Machines has been the dominating solution, however as the game types evolve the AI strategies needs to be more and more comprehensive which makes the state machines explode in complexity.

A total ordered Hierarchical Task Network is presented as a possible solution. Plan creation is well within the required computation time and by extending the original approach, by dynamically executing its tasks, it seems that the challenges in connection with the dynamic environment are reduced to a minimum. However when the bot engages in combat and its reactiveness gets put to the test, the use of small state machines as actions might be needed for the bot to prevail.

The real power of the HTN is expressed when complex strategies needs to be implemented. The designer of a strategy starts out by making simple methods, then these simple methods are used to create more complex methods and so on. With an FSM it is required for the designer of the AI to have a profound knowledge of the underlaying implementation, however with a solid HTN framework it becomes possible to apply complex bot strategies without any such insight.

Since there are neither any non-deterministic choices or machine learning integrated into the HTN, it might be argued that this type of AI is just another way of representing an FSM. However the purpose of this paper was to shape a framework containing a more intuitive way of describing bot behavior and where the creation of more complex strategies would be lifted to an abstraction level impossible with FSMs. By using this approach, and with the expressiveness of the HTN syntax, complex and coordinating strategies can be defined in a matter of minutes.

Furthermore the ability for a bot/coordinator to drastically change another bots strategy, while the game is running, is well beyond the capabilities of any normal FSM.

Appendix A

CD Contents

A CD is attached to the report which contains the following:

- **Executables** - All programs needed to run the implementation (see appendix B).
- **Source** - Contains all the source code.
- **UTBot** - The executable Jar file and example HTN definitions.
- **IMM-B.Sc.-2008-02** - This report.

Appendix B

Running the Program

The "Executables" folder on the CD, handed in with the report, contains all the installations needed in order to run the program.

1. Extract and install the "UTServer428-2.zip" to install the Unreal Tournament server.
2. Run the "Gamebots.umod" to modify the server.
3. Extract and install the "TclViz.zip" to install the visualizer.

Everything should now be installed correctly.

Run the server with the command:

```
ucc server dom-stalwart?game=FriendlyBotAPI.FriendlyBotDomination
```

Finally run the visualizer and then the UTBot.jar program:

```
java -jar UTBot.jar simpleTasks.htn compoundTask.htn
```

The `simpleTasks.htn` is the name of the file with the HTN simple tasks and `compoundTask.htn` is the file with the compound tasks. Notice that the `Stalwart` file needs to be in the same library as the UTBot.jar. This file contains map information needed by the program.

Appendix C

Source Code

Following is all the source code used in the implementation. The source code is also included on the CD attached to this paper.

C.1 ai

C.1.1 ai.Planner.java

```
package ai;

import com.ComHandler;
import exceptions.DuplicateTaskException;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.HashMap;
import map.Map;
import state.State;
import util.Log;

/**
 * The planner is a container and access point to the HTN
 * @author Rene B. Hansen
 */
public class Planner {

    private State state;
    private HashMap<String,Task> HTN;
```

```
    private Conditionals conditionals;
    private ComHandler com;


    /**
     * Creates a new instance of Planner
     * @param state state.State
     * @param com com.ComHandler
     * @throws java.io.FileNotFoundException
     * @throws java.io.IOException
     * @throws exceptions.DuplicateTaskException
     */
    public Planner(State state,ComHandler com) throws FileNotFoundException,
         IOException, DuplicateTaskException {
        this.state = state;
        conditionals = new Conditionals(state);
        this.com = com;
        this.createHTN();
    }

    /**
     * Creates the HTN by using the HTNBuilder
     * @throws java.io.FileNotFoundException
     * @throws java.io.IOException
     * @throws exceptions.DuplicateTaskException
     */
    private void createHTN() throws FileNotFoundException, IOException,
         DuplicateTaskException{
        HTNBuilder b = new HTNBuilder(conditionals,com,state);
        HTN = b.createHTN();
    }

    /**
     * Executes the HTN task given by taskName
     * @param taskName Name of a Task
     * @param arg Arguments that the task should be executed with
     */
    public void doTask(String taskName,String[] arg){
        if (HTN.containsKey(taskName)){
            if (HTN.get(taskName).doTask(arg)){
                //System.out.println("Bot:"+state.getSLF().getName()+" Task:
                     "+taskName+" completed");
            }else /*System.err.println("Bot:"+state.getSLF().getName()+" Task
                 : "+taskName+" failed")*/;
        } else /*System.err.println("Bot:"+state.getSLF().getName()+" No such
             task: "+taskName)*/;
    }

}
```

C.1.2 ai.HTNBuilder.java

```
package ai;

import com.ComHandler;
import exceptions.DuplicateTaskException;
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.HashMap;
import state.Rotation;
```

```
import state.State;
import test.Main;

/**
 * Class to parse and build the HTNs
 * @author Rene B. Hansen
 */
public class HTNBuilder {

    private String HTNCompundFile;
    private String HTNSimpleFile;
    private FileReader fstream;
    private BufferedReader in;
    private ComHandler com;

    private HashMap<String,Task> tasks;

    private Conditionals conditionals;
    private State state;

    /**
     * Creates a new instance of HTNBuilder
     * @param conditionals used to evaluate conditionals
     * @param com com.Comhandler
     * @param state state.State
     * @throws java.io.FileNotFoundException
     */
    public HTNBuilder(Conditionals conditionals,ComHandler com, State state)
         throws FileNotFoundException {
        this.HTNSimpleFile = Main.simpleTasks;
        this.HTNCompundFile = Main.compoundTasks;
        this.conditionals = conditionals;
        this.state = state;
        this.com = com;
        tasks = new HashMap<String, Task>();
    }

    /**
     * Creates a comple HTN from the two files
     * @return A hashmap with all HTN methods
     * @throws java.io.IOException
     * @throws exceptions.DuplicateTaskException
     */
    public HashMap<String,Task> createHTN() throws IOException,
         DuplicateTaskException{

        this.createAllSimpleTasks();

        fstream = new FileReader(HTNCompundFile);
        in = new BufferedReader(fstream);

        String temp = in.readLine();
        String[] elem;
        CompoundTask t = null;
        while (temp != null){
            elem = temp.split(" ");
            if (elem.length > 0){
                if (elem[0].equals("Head")){
                    if (tasks.containsKey(elem[1])) throw new
                         DuplicateTaskException("Method head: "+elem[1]+" not
                          unique");
                    t = new CompoundTask(elem[1],conditionals,state);
                    tasks.put(t.name,t);
                }
            }
```

```
            temp = in.readLine();
        }
        in.close();
        fstream = new FileReader(HTNCompundFile);
        in = new BufferedReader(fstream);

        temp = in.readLine();

        boolean inPreconditionals = false;
        boolean inGoals = false;
        boolean inSubtasks = false;

        while (temp != null){

            elem = temp.split(" ");
            if (temp.trim().length() > 0){
                if (elem[0].equals("Head")){
                    t = (CompoundTask)tasks.get(elem[1]);
                    inGoals = false;
                    inPreconditionals = false;
                    inSubtasks = false;
                }  else if (elem[0].equals("Goals")){
                    inGoals = true;
                    inPreconditionals = false;
                    inSubtasks = false;
                } else if (elem[0].equals("Preconditions")){
                    inGoals = false;
                    inPreconditionals = true;
                    inSubtasks = false;
                } else if (elem[0].equals("Subtasks")){
                    inGoals = false;
                    inPreconditionals = false;
                    inSubtasks = true;
                } else if (inGoals){
                    t.addGoal(temp.trim());
                } else if (inPreconditionals){
                    t.addPrecondition(temp.trim());
                } else if (inSubtasks){
                    String [] commaDel = temp.split(",");
                    ArrayList<SubTask> result = new ArrayList<SubTask>();
                    for (String c : commaDel){
                        elem = c.split(" ");
                        SubTask s = new SubTask(tasks.get(elem[0]),c.substring(elem[0].length()).trim());
                        result.add(s);
                    }
                    t.addSubTask(result);
                } else { System.err.println(elem[0] + " did not equal anything");}

            }
            temp = in.readLine();
        }

        in.close();
        return tasks;
    }

    /**
     * Parses and creates all the simple tasks
     * @throws java.io.FileNotFoundException
     * @throws java.io.IOException
     */
    private void createAllSimpleTasks() throws FileNotFoundException, IOException{
```

```
        fstream = new FileReader(HTNSimpleFile);
        in = new BufferedReader(fstream);

        SimpleTask t = null;

        String temp = in.readLine();
        String[] elem;

        boolean inPreconditionals = false;
        boolean inGoals = false;

        while (temp != null){
            elem = temp.split(" ");
            if (temp.trim().length() > 0){
                if (elem[0].equals("Head")){
                    t = createSimpleTask(elem[1]);
                    tasks.put(t.name,t);
                    inGoals = false;
                    inPreconditionals = false;
                }  else if (elem[0].equals("Goals")){
                    inGoals = true;
                    inPreconditionals = false;
                } else if (elem[0].equals("Preconditions")){
                    inGoals = false;
                    inPreconditionals = true;
                } else if (elem[0].equals("Subtasks")){
                    inGoals = false;
                    inPreconditionals = false;
                } else if (inGoals){
                    t.addGoal(temp.trim());
                } else if (inPreconditionals){
                    t.addPrecondition(temp.trim());
                } else { System.err.println(elem[0] + " did not equal
                     anything");}
            }
            temp = in.readLine();
        }
        in.close();
    }

    /**
     * Identifies and instantiates the corresponding simpletask identified by
          the name
     * @param name task name
     * @return SimpleTask
     */
    private SimpleTask createSimpleTask(String name){
        SimpleTask t = null;
        if (name.equals("RunTo")){
            t = new RunTo(conditionals,com,state);
        } else if (name.equals("RunToRandom")){
            t = new RunToRandom(conditionals,com,state);
        } else if (name.equals("Rotate")){
            t = new Rotate(conditionals,com,state);
        } else if (name.equals("AddBot")){
            t = new AddBot(conditionals,com,state);
        } else if (name.equals("StartBot")){
            t = new StartBot(conditionals,com,state);
        } else if (name.equals("AssignTask")){
            t = new AssignTask(conditionals,com,state);
        } else if (name.equals("ClearTasks")){
            t = new ClearTasks(conditionals,com,state);
        } else if (name.equals("TravelTo")){
            t = new TravelTo(conditionals,com,state);
        } else if (name.equals("ShootAtEnemy")){
```

```java
            t = new ShootAtEnemy(conditionals,com,state);
        } else if (name.equals("RunToRandomINV")){
            t = new RunToRandomINV(conditionals,com,state);
        }else {
            System.err.println(" !!! ERROR : Simple task >"+name+"< does not exist !!!");
        }
        return t;
    }
}
```

C.1.3 ai.Coordinator.java

```java
package ai;

import bot.CCBot;
import com.ComHandler;
import exceptions.DuplicateTaskException;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.ArrayList;
import java.util.HashMap;
import map.Stalwart;
import state.State;

/**
 * The coordinator is used to control the task lists of all the bots. When the
 * coordinator is started, it adds the Main htn method to its task list.
 * @author Rene B. Hansen
 */
public class Coordinator extends CCBot {

    private HashMap<String,CCBot> bots;
    private static Coordinator coordinator;

    /**
     * Creates a new instance of Coordinator
     * @throws java.io.IOException
     * @throws exceptions.DuplicateTaskException
     */
    private Coordinator() throws IOException, DuplicateTaskException {
        super(/*new Stalwart(),*/"Coordinator",-1);
        bots = new HashMap<String,CCBot>();
        bots.put(this.name,this);
        Coordinator.coordinator = this;
        this.run();
    }

    /**
     * Starts the coordinator
     */
    public void run() {
        this.addTask("Main");
        while(true){
            try {
                this.doTasks(this.getNextTask());
            } catch (InterruptedException ex) {
                ex.printStackTrace();
            }
        }
    }

```

```

    /**
     * Adds a bot to the game
     * @param name bot name
     * @param team team number
     * @throws java.io.IOException
     * @throws exceptions.DuplicateTaskException
     */
    public void addBot(String name, int team) throws IOException,
         DuplicateTaskException{
        bots.put(name,new CCBot(/*new Stalwart(),*/name,team));
    }

    /**
     * Initialzes and starts a previously added bot on the server
     * @param name bot name
     */
    public void startBot(String name){
        if (bots.containsKey(name)){
            Thread b = bots.get(name);
            if (!b.isAlive()){
                b.start();
            }
        } else System.err.println("No such bot - "+name);
    }

    /**
     * Removes all tasks from a bot
     * @param bot name
     */
    public void clearAllTasks(String bot){
        if (bots.containsKey(bot)){
            bots.get(bot).clearAllTasks();
        }
    }

    /**
     * Adds a task to a bot
     * @param bot name
     * @param task taskName
     */
    public void addTask(String bot, String task){
        if (bots.containsKey(bot)){
            bots.get(bot).addTask(task);
        }
    }

    /**
     * Return this coordinator, if the coordinator has not been instanciated
          a new one
     * is
     * @return this coordinator
     */
    public static Coordinator getInstance()  {
        if (Coordinator.coordinator == null){
            try {
                return Coordinator.coordinator = new Coordinator();
            } catch (DuplicateTaskException ex) {
                ex.printStackTrace();
            } catch (IOException ex) {
                ex.printStackTrace();
            }
        }
        return Coordinator.coordinator;
    }
```

```
}
```

C.1.4 ai.Conditionals.java

```
package ai;

import state.State;
import map.Map;
import util.Log;

/**
 * Class used to evaluate all conditionals
 * @author Rene B. Hansen
 */
public class Conditionals {

    private State state;
    private Map map;

    /**
     * Creates a new instance of Conditionals
     * @param state state.State
     */
    public Conditionals(State state) {
        this.state = state;
        this.map = state.getMap();
    }

    /**
     * Evaluates the given conditional
     * @param condition String name of the conditional to be evaluated
     * @param arg Argument list that the conditional might use
     * @return true if the conditional evaluates to true, otherwhise false.
     */
    public boolean checkConditional(String condition, String[] arg){

        String[] con = condition.split(" ");

        if (con[0].equals("At")) return At(con[1],arg);
        else if (con[0].equals("neighborTo")) return neighborTo(con[1],arg);
        else if (con[0].equals("hasWeapon")) return hasWeapon();
        else if (con[0].equals("hasMoreAmmoThan")) return hasMoreAmmoThan(con
            [1],arg);
        else if (con[0].equals("hasMoreHealthThan")) return hasMoreHealthThan
            (con[1],arg);
        else return false;
    }

    /**
     * Evaluate the At conditional
     * @param id value or argument identifier
     * @param arg list of arguments
     * @return true if it evaluates to true
     */
    private boolean At(String id, String[] arg){
        boolean result;
        String output = "";
        if (id.startsWith("$")) {
            result = state.getSLF().getTarget().equals(arg[Integer.parseInt(
                id.substring(1))]);
            output = arg[Integer.parseInt(id.substring(1))];
        }
        else {
```

```
            result = state.getSLF().getTarget().equals(id);
            output = id;
        }
        return result;
    }

    /**
     * Evaluate the neighborTo conditional
     * @param id value or argument identifier
     * @param arg list of arguments
     * @return true if it evaluates to true
     */
    private boolean neighborTo(String id, String[] arg){
        boolean result;
        String output;
        if (id.startsWith("$")){
            result = map.neighbours(arg[Integer.parseInt(id.substring(1))],
                 state.getSLF().getTarget());
            output = arg[Integer.parseInt(id.substring(1))];
        }
        else{
            result = map.neighbours(id,state.getSLF().getTarget());
            output = id;
        }
        return result;
    }

    /**
     * Evaluate the hasMoreAmmoThan conditional
     *
     * @return true if it evaluates to true
     */
    private boolean hasWeapon(){
        boolean result = !state.getSLF().getWeapon().equals("None");
        return result;
    }

    /**
     * Evaluate the neighborTo conditional
     * @param value value or argument identifier
     * @param arg list of arguments
     * @return true if it evaluates to true
     */
    private boolean hasMoreAmmoThan(String value,String[] arg){
        boolean result;
        int temp;
        if (value.startsWith("$")) {
           temp = Integer.parseInt(arg[Integer.parseInt(value.substring(1))])
                ;
        } else {
            temp = Integer.parseInt(value);
        }
        result = state.getSLF().getCurrentAmmo() > temp;
        return result;
    }

    /**
     * Evaluate the neighborTo conditional
     * @param value value or argument identifier
     * @param arg list of arguments
     * @return true if it evaluates to true
     */
    private boolean hasMoreHealthThan(String value, String[] arg){
        boolean result;
        int temp;
```

```
        if (value.startsWith("$")){
            temp = Integer.parseInt(arg[Integer.parseInt(value.substring(1))
                ]);
        }else {
            temp = Integer.parseInt(value);
        }
        result = state.getSLF().getHealth() > temp;
        return result;
    }
}
```

C.1.5 ai.Task.java

```
package ai;

import java.util.ArrayList;
import state.State;

/**
 * SuperClass of all SimpleTasks and CompoundTasks
 * @author Rene B. Hansen
 */
public abstract class Task {

    public final String name;

    protected ArrayList<String> preconditions;
    protected ArrayList<String> goals;
    protected Conditionals conditionals;
    protected State state;

    /**
     * Creates a new instance of Task
     * @param name Task name
     * @param conditionals Conditional
     * @param state state.State
     */
    public Task(String name, Conditionals conditionals, State state) {
        this.name = name;
        this.conditionals = conditionals;
        this.state = state;

        this.goals = new ArrayList<String>();
        this.preconditions = new ArrayList<String>();
    }

    /**
     * Used by the parser to add goals to this Task
     * @param goal String conditional
     */
    public void addGoal(String goal){
        goals.add(goal);
    }

    /**
     * Used by the parser to add preconditions to this Task
     * @param constraint string conditional
     */
    public void addPrecondition(String constraint){
        preconditions.add(constraint);
    }

    /**
```

```
     * Evaluates the goals of the task
     * @param arg Arguments that the goals are to be evaluated with
     * @return true all the goals evaluate to true
     */
    protected boolean checkGoals(String[] arg){
        //Checking that the goal of the task is not already fullfilled
        for (String s : goals){
            if (conditionals.checkConditional(s,arg)){
                return true;
            }
        }
        return false;
    }

    /**
     * Evaluates the preconditions of the task
     * @param arg Arguments that the preconditions are to be evaluated with
     * @return false if one of the preconditions returns false
     */
    protected boolean checkConstraints(String[] arg){
        for (String s : preconditions){
            if (!conditionals.checkConditional(s,arg)){
                return false;
            }
        }
        return true;
    }

    /**
     * Abstract method which is to be implemented in the instantiating
         subclass
     */
    public abstract boolean doTask(String[] arg);

}
```

C.1.6 ai.CompoundTask.java

```
package ai;

import java.util.ArrayList;
import java.util.List;
import state.State;

/**
 * Task that consist of goals, conditionals and subtasks. Executes/decomposes
      its subtasks.
 * @author Rene B. Hansen
 */
public class CompoundTask extends Task {

    /**
     * list of lists of subtasks
     */
    private ArrayList<ArrayList<SubTask>> subTasks;

    /**
     * Creates a new instance of CompoundTask
     * @param name Task name
     * @param conditionals AI.Conditionals to evaluate conditionals
     * @param state state state.State
     */
    public CompoundTask(String name,Conditionals conditionals, State state) {
```

```
        super(name,conditionals,state);
        this.subTasks = new ArrayList<ArrayList<SubTask>>();
    }

    /**
     * Method used by the parser to add subtasks
     * @param t A subtask list
     */
    public void addSubTask(ArrayList<SubTask> t){
        subTasks.add(t);
    }

    /**
     * Decomposes/executes the subtasks
     * @param arg Argument list this compound task is executed with
     * @return returns true if one task in each of the subTasks list returns
           true.
     */
    public boolean doTask(String[] arg) {
        if (this.checkGoals(arg)) return true;

        if (!this.checkConstraints(arg)) return false;

        for (ArrayList<SubTask> a : subTasks){
            boolean success = false;
            for (SubTask t : a){
                //Building up the arguments for the subtask
                String[] temp = t.arguments.split(" ");
                for (int i = 0; i < temp.length; i++){
                    if (temp[i].startsWith("$")){
                        temp[i] = arg[Integer.parseInt(temp[i].substring(1))
                            ];
                    }
                }

                //Calling the subtasks
                if (t.subTask.doTask(temp)) {
                    success = true;
                    break;
                }
            }
            if (!success) return false;
        }
        return true;
    }

    /**
     * toString method
     * @return String representation of the context in this compound task
     */
    public String toString(){
        String result = "name: " + this.name +"\n" +
                "Goals: \n";
        for (String s : this.goals){
            result += "    " + s + "\n";
        }

        result += "Preconditions: \n";
        for (String s : this.preconditions){
            result += "    " + s + "\n";
        }
        result += "SubTasks: \n";

        for (ArrayList<SubTask> a : subTasks){
            result += "    ";
```

```
            for (SubTask t : a){
                result += t.toString() + ",";
            }
            result += "\n";
        }
        return result;
    }
}
```

C.1.7 ai.SimpleTask.java

```
package ai;

import com.ComHandler;
import state.State;

/**
 * SuperClass of all the simple tasks
 * @author Rene B. Hansen
 */
public abstract class SimpleTask extends Task {

    protected ComHandler com;

    /**
     * Creates a new instance of SimpleTask
     * @param name Task name
     * @param con Conditionals
     * @param com com.ComHandler
     * @param state state.State
     */
    public SimpleTask(String name, Conditionals con, ComHandler com, State state) {
        super(name,con,state);
        this.com = com;
    }

    /**
     * toString method that returns a String representation of the SimpleTask values
     * @return String representation of the SimpleTask values
     */
    public String toString(){
        String result = "";
        result = "name: " + this.name +"\n" +
                        "Goals: \n";
        for (String s : this.goals){
            result += "    " + s + "\n";
        }

        result += "Preconditions: \n";
        for (String s : this.preconditions){
            result += "    " + s + "\n";
        }
        return result;
    }
}
```

C.1.8 ai.AddBot.java

```
package ai;
```

```java

import com.ComHandler;
import exceptions.DuplicateTaskException;
import java.io.IOException;
import state.State;

/**
 * Task to add a bot on the server.
 * @author Rene B. Hansen
 */
public class AddBot extends SimpleTask {

    /**
     * Creates a new instance of the AddBot Task
     * @param con AI.Conditionals to evaluate conditionals
     * @param com com.ComHandler to handle server communication
     * @param state state.State
     */
    public AddBot(Conditionals con, ComHandler com, State state) {
        super("AddBot",con,com,state);
    }

    /**
     * Execute the AddBot task
     * @param arg Arguments - botName and teamNumber
     * @return Returns true if the task succeeds and false if it fails
     */
    public boolean doTask(String[] arg) {

        //Checking that the goal of the task is not already fullfilled
        if (super.checkGoals(arg)) return true;


        //Checking that any possible constraints are not violated
        if (!super.checkConstraints(arg)) return false;

        try {
            Coordinator.getInstance().addBot(arg[0],Integer.parseInt(arg[1]))
                ;
        } catch (NumberFormatException ex) {
            ex.printStackTrace();
        } catch (DuplicateTaskException ex) {
            ex.printStackTrace();
        } catch (IOException ex) {
            ex.printStackTrace();
        }
        return true;
    }
}
```

C.1.9 ai.StartBot.java

```java
package ai;

import com.ComHandler;
import state.State;

/**
 * SimpleTask that starts the given ot on the server
 * @author Rene B. Hansen
 */
public class StartBot extends SimpleTask {

```

```java
    /**
     * Creates a new instance of the StartBot Task
     * @param con AI.Conditionals to evaluate conditionals
     * @param com com.ComHandler to handle server communication
     * @param state state.State
     */
    public StartBot(Conditionals con, ComHandler com, State state) {
        super("StartBot",con,com,state);
    }

    /**
     * Execute the StartBot task
     * @param arg Arguments - botName
     * @return Returns true if the task succeeds and false if it fails
     */
    public boolean doTask(String[] arg) {
        //Checking that the goal of the task is not already fullfilled
        if (super.checkGoals(arg)) return true;


        //Checking that any possible constraints are not violated
        if (!super.checkConstraints(arg)) return false;

        Coordinator.getInstance().startBot(arg[0]);

        return true;
    }
}
```

C.1.10 ai.AssignTask.java

```java
package ai;

import com.ComHandler;
import state.State;

/**
 * Task to assign tasks to the bots/coordinator task list
 * @author Rene B. Hansen
 */
public class AssignTask extends SimpleTask {

    /**
     * Creates a new instance of the AssignTask Task
     * @param con AI.Conditionals to evaluate conditionals
     * @param com com.ComHandler to handle server communication
     * @param state state.State
     */
    public AssignTask(Conditionals con, ComHandler com, State state) {
        super("AssignTask",con,com,state);
    }

    /**
     * Execute the AssignTask task
     * @param arg Arguments - botName and taskName with its arguments
     * @return Returns true if the task succeeds and false if it fails
     */
    public boolean doTask(String[] arg) {
        //Checking that the goal of the task is not already fullfilled
        if (super.checkGoals(arg)) return true;


        //Checking that any possible constraints are not violated
```

```
        if (!super.checkConstraints(arg)) return false;

        String temp = arg[1];
        if (arg.length > 2){
            for (int i = 2; i < arg.length; i++){
                temp += " "+arg[i];
            }
        }
        Coordinator.getInstance().addTask(arg[0],temp);
        return true;
    }
}
```

C.1.11 ai.ClearTasks.java

```
package ai;

import com.ComHandler;
import state.State;

/**
 * Task to clear the taskList of a bot/coordinator
 * @author Rene B. Hansen
 */
public class ClearTasks extends SimpleTask {

    /**
     * Creates a new instance of the ClearTasks Task
     * @param con AI.Conditionals to evaluate conditionals
     * @param com com.ComHandler to handle server communication
     * @param state state.State
     */
    public ClearTasks(Conditionals con, ComHandler com, State state) {
        super("ClearTasks",con,com,state);
    }

    /**
     * Execute the ClearTasks task
     * @param arg Arguments - botName
     * @return Returns true if the task succeeds and false if it fails
     */
    public boolean doTask(String[] arg) {
        //Checking that the goal of the task is not already fullfilled
        if (super.checkGoals(arg)) return true;


        //Checking that any possible constraints are not violated
        if (!super.checkConstraints(arg)) return false;

        Coordinator.getInstance().clearAllTasks(arg[0]);
        return true;
    }
}
```

C.1.12 ai.Rotate.java

```
package ai;

import com.ComHandler;
import state.State;
```

```

/**
 * SimpleTask that makes the bot rotate a predefined amount
 * @author Rene B. Hansen
 */
public class Rotate extends SimpleTask {

    /**
     * Creates a new instance of the Rotate Task
     * @param con AI.Conditionals to evaluate conditionals
     * @param com com.ComHandler to handle server communication
     * @param state state.State
     */
    public Rotate(Conditionals con, ComHandler com, State state) {
        super("Rotate",con,com,state);
    }

    /**
     * Execute the Rotate task
     * @param arg Arguments - amount to rotate
     * @return Returns true if the task succeeds and false if it fails
     */
    public boolean doTask(String[] arg) {
        //Checking that the goal of the task is not already fullfilled
        if (super.checkGoals(arg)) return true;


        //Checking that any possible constraints are not violated
        if (!super.checkConstraints(arg)) return false;


        state.getSLF().startRotation(Integer.parseInt(arg[0]));
        com.rotateAmount(Integer.parseInt(arg[0]));
        try {
            //Waits until the movement has been completesd
            state.getSLF().waitForRotation();
        } catch (InterruptedException ex) {
            ex.printStackTrace();
            //If the movement is for some reason interrupted, it is assumed
                that
            //the destination has no been reached.
            return false;
        }
        return true;
    }
}
```

C.1.13 ai.RunTo.java

```
package ai;

import com.ComHandler;
import state.State;

/**
 * SimpleTask that makes the bot runto the given location
 * @author Rene B. Hansen
 */
public class RunTo extends SimpleTask{

    /**
     * Creates a new instance of the RunTo Task
     * @param con AI.Conditionals to evaluate conditionals
```

```
     * @param com com.ComHandler to handle server communication
     * @param state state.State
     */
    public RunTo(Conditionals con, ComHandler com, State state) {
        super("RunTo",con,com,state);
    }

    /**
     * Execute the RunTo task
     * @param arg Arguments - visible destination to run to
     * @return Returns true if the task succeeds and false if it fails
     */
    public boolean doTask(String[] arg) {
        //Checking that the goal of the task is not already fullfilled
        if (super.checkGoals(arg)) return true;


        //Checking that any possible constraints are not violated
        if (!super.checkConstraints(arg)) return false;


        state.getSLF().startMovement();
        com.runToLocation(state.getNodeLocation(arg[0]));
        state.getSLF().setTarget(arg[0]);
        try {
            //Waits until the movement has been completesd
            state.getSLF().waitForMovement();
        } catch (InterruptedException ex) {
            ex.printStackTrace();
            //If the movement is for some reason interrupted, it is assumed
                that
            //the destination has no been reached.
            state.getSLF().setTarget("none");
            return false;
        }
        return true;
    }
}
```

C.1.14 ai.RunToRandom.java

```
package ai;

import com.ComHandler;
import state.State;

/**
 * SimpleTask that makes the bot run to a random visible location
 * @author Rene B. Hansen
 */
public class RunToRandom extends SimpleTask {

    /**
     * Creates a new instance of the RunToRandom Task
     * @param con AI.Conditionals to evaluate conditionals
     * @param com com.ComHandler to handle server communication
     * @param state state.State
     */
    public RunToRandom(Conditionals con, ComHandler com, State state) {
        super("RunToRandom",con,com,state);
    }

    /**
```

```
     * Execute the RunToRandom task
     * @param arg Arguments - none
     * @return Returns true if the task succeeds and false if it fails
     */
    public boolean doTask(String[] arg) {
        //Checking that the goal of the task is not already fullfilled
        if (super.checkGoals(arg)) return true;


        //Checking that any possible constraints are not violated
        if (!super.checkConstraints(arg)) return false;


        String id = state.getAReachableNodeId();
        if (id == null)return false;
        state.getSLF().startMovement();
        com.runToLocation(state.getNodeLocation(id));
        state.getSLF().setTarget(id);
        try {
            //Waits until the movement has been completesd
            state.getSLF().waitForMovement();
        } catch (InterruptedException ex) {
            ex.printStackTrace();
            //If the movement is for some reason interrupted, it is assumed
                that
            //the destination has no been reached.
            state.getSLF().setTarget("none");
            return false;
        }
        return true;
    }
}
```

C.1.15 ai.RunToRandomINV.java

```
package ai;

import com.ComHandler;
import map.Node;
import state.State;

/**
 * SimpleTask that makes the bot run to a random inventory node
 * @author Rene B. Hansen
 */
public class RunToRandomINV extends SimpleTask{

    /**
     * Creates a new instance of the RunToRandomINV Task
     * @param con AI.Conditionals to evaluate conditionals
     * @param com com.ComHandler to handle server communication
     * @param state state.State
     */
    public RunToRandomINV(Conditionals con, ComHandler com, State state) {
        super("RunToRandomINV",con,com,state);
    }

    /**
     * Execute the RunToRandomINV task
     * @param arg Arguments - none
     * @return Returns true if the task succeeds and false if it fails
     */
    public boolean doTask(String[] arg) {
```

```
        //Checking that the goal of the task is not already fullfilled
        if (super.checkGoals(arg)) return true;


        //Checking that any possible constraints are not violated
        if (!super.checkConstraints(arg)) return false;

        Node n = state.getMap().getRandomInventoryLocation();
        if (n != null){
            com.getPath(n.getLocation());
            try {
                String[] path = state.getSLF().getPath();
                if (path == null) return false;
                if (path.length > 0 && !path[0].equals("NOPATH")){
                    for (int i = 0; i < path.length; i++){
                        state.getSLF().startMovement();
                        com.runToLocation(path[i]);
                        if (state.getSLF().waitForMovement() == false) {
                            state.getSLF().setTarget("none");
                            return false;
                        }
                    }
                    state.getSLF().setTarget(n.getId());
                } else {
                    state.getSLF().startMovement();
                    com.runToLocation(n.getLocation());
                    if (state.getSLF().waitForMovement() == false) {
                        state.getSLF().setTarget("none");
                        return false;
                    }
                    state.getSLF().setTarget(n.getId());
                }


            } catch (InterruptedException ex) {
                ex.printStackTrace();
                state.getSLF().setTarget("none");
                return false;
            }
            return true;
        } else return false;
    }
}
```

C.1.16 ai.ShootAtEnemy.java

```
package ai;

import com.ComHandler;
import state.PLR;
import state.State;

/**
 * SimpleTask that makes the bot shoot at any visible enemies
 * @author Rene B. Hansen
 */
public class ShootAtEnemy extends SimpleTask {

    /**
     * Creates a new instance of the ShootAtEnemy Task
     * @param con AI.Conditionals to evaluate conditionals
     * @param com com.ComHandler to handle server communication
     * @param state state.State
```

```
     */
    public ShootAtEnemy(Conditionals con, ComHandler com, State state) {
        super("ShootAtEnemy",con,com,state);
    }

    /**
     * Execute the ShootAtEnemy task
     * @param arg Arguments - none
     * @return Returns true if the task succeeds and false if it fails
     */
    public boolean doTask(String[] arg) {
        //Checking that the goal of the task is not already fullfilled
        if (super.checkGoals(arg)) return true;


        //Checking that any possible constraints are not violated
        if (!super.checkConstraints(arg)) return false;

        PLR plr = state.getAVisibleEnemyPLR();
        if (plr != null){
            while (plr != null){
                //com.turnToLocation(plr.getLocation().toString());
                com.shootAt(plr.getLocation().toString(),plr.getId());
                try {
                    state.waitForEnemyPLRNonVisible(plr);
                    //com.turnToLocation(plr.getLocation().toString());
                } catch (InterruptedException ex) {
                    ex.printStackTrace();
                    return false;
                }
                if (state.getSLF().getCurrentAmmo() == 0)return false;
                plr = state.getAVisibleEnemyPLR();
            }
            com.stopShoot();
        } else return false;
        return true;
    }
}
```

C.1.17 ai.SubTask.java

```
package ai;

import java.util.ArrayList;

/**
 * Container for a subTask
 * @author Rene B. Hansen
 */
public class SubTask {

    public final Task subTask;
    public final String arguments;

    /**
     * Creates a new instance of SubTask
     * @param subTask Task contained by this subTask
     * @param arguments Arguments for the task
     */
    public SubTask(Task subTask,String arguments) {
        this.subTask = subTask;
        this.arguments = arguments;
    }
```

```
    /**
     * Returns a String representation of this subtask
     * @return String of name and arguements
     */
    public String toString(){
        String result = "";
        result += this.subTask.name + " " + this.arguments;
        return result;
    }
}
```

C.1.18 ai.TravelTo.java

```
package ai;

import com.ComHandler;
import state.State;

/**
 * Task to make the bot travel to the given location
 * @author Rene B. Hansen
 */
public class TravelTo extends SimpleTask{

    /**
     * Creates a new instance of the TravelTo Task
     * @param con AI.Conditionals to evaluate conditionals
     * @param com com.ComHandler to handle server communication
     * @param state state.State
     */
    public TravelTo(Conditionals con, ComHandler com, State state) {
        super("TravelTo",con,com,state);
    }

    /**
     * Execute the TravelTo task
     * @param arg Arguments - destination
     * @return Returns true if the task succeeds and false if it fails
     */
    public boolean doTask(String[] arg) {
        //Checking that the goal of the task is not already fullfilled
        if (super.checkGoals(arg)) return true;


        //Checking that any possible constraints are not violated
        if (!super.checkConstraints(arg)) return false;

        com.getPath(state.getNodeLocation(arg[0]));
        try {
            String[] path = state.getSLF().getPath();
            if (path == null)return false;
            if (path.length > 0 && !path[0].equals("NOPATH")){
                for (int i = 0; i < path.length; i++){
                    state.getSLF().startMovement();
                    com.runToLocation(path[i]);
                    if (state.getSLF().waitForMovement() == false) return
                         false;
                }
                state.getSLF().setTarget(arg[0]);
            } else {
                state.getSLF().startMovement();
                com.runToLocation(state.getMap().getLocation(arg[0]));
```

```
                if (state.getSLF().waitForMovement() == false) return false;
                state.getSLF().setTarget(arg[0]);
            }

        } catch (InterruptedException ex) {
            ex.printStackTrace();
            state.getSLF().setTarget("none");
            return false;
        }
        return true;
    }
}
```

C.2 bot

C.2.1 bot.Bot.java

```
package bot;

import com.ComHandler;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import state.State;

/**
 * SuperClass of the bots
 * @author Rene B. Hansen
 */
public abstract class Bot extends Thread{

    public state.State state;
    public ComHandler com;

    /**
     * Creates a new instance of Bot
     * @throws java.io.IOException
     */
    public Bot() throws IOException {
        state = new state.State();
        com = new ComHandler(state);
    }
}
```

C.2.2 bot.CCBot.java

```
package bot;

import ai.Planner;
import com.ComHandler;
import exceptions.DuplicateTaskException;
import java.io.IOException;
import java.util.ArrayList;
import java.util.HashMap;
import map.Map;

/**
```

```
 * Coordinator Controlled Bot, the bot version which is to be used with the
 * coordinator
 * @author Rene B. Hansen
 */
public class CCBot extends Bot  {

    private Planner planner;
    private ArrayList<String> tasks;
    private int count;

    public final String name;
    public final int team;


    /**
     * Creates a new instance of CCBot
     * @param name bot name
     * @param team bot team
     * @throws java.io.IOException
     * @throws exceptions.DuplicateTaskException
     */
    public CCBot(String name, int team) throws IOException,
         DuplicateTaskException {
        this.planner = new Planner(state,com);
        this.tasks = new ArrayList<String>();
        this.name = name;
        this.team = team;
        count = 0;
    }

    /**
     * Starts the bot
     */
    public void run() {
        com.init(name,team);
        try {
            Thread.sleep(2000);
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }
        try {
            while(true){
                doTasks(getNextTask());
            }

        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }
    }

    /**
     * Executes the selected task via the planner
     * @param s String representation of the taskName and arguments in
     * @throws java.lang.InterruptedException
     */
    protected void doTasks(String s) throws InterruptedException{
        String[] temp = s.split(" ");
        if (temp.length > 1){
            planner.doTask(temp[0],s.substring(temp[0].length()).trim().split
                (" "));
        }else {
            planner.doTask(temp[0], null);
        }
        this.sleep(100);
    }
```

```


    /**
     * Adds a task to the bots taskList
     * @param task String representation of the task to add
     */
    public synchronized void addTask(String task){
        this.tasks.add(task);
    }

    /**
     * Removes the specified task from the bots task list
     * @param task String representation of the task to be removed
     */
    public synchronized void removeTask(String task){
        this.tasks.remove(task);
    }

    /**
     * Removes all tasks from the bots tasklist
     */
    public synchronized void clearAllTasks(){
        this.tasks.clear();
    }

    /**
     * Returns the next task to be executed
     * @throws java.lang.InterruptedException
     * @return String representation of the next task
     */
    protected synchronized String getNextTask() throws InterruptedException{
        while(tasks.size() == 0) wait();
        if (count > tasks.size()-1){
            count = 1;
            return tasks.get(0);
        } else {
            return tasks.get(count++);
        }
    }
}
```

C.3 state

C.3.1 state.State.java

```
/*
 * State.java
 *
 * Created on 3. september 2007, 12:25
 *
 * To change this template, choose Tools | Template Manager
 * and open the template in the editor.
 */

package state;

import java.util.ArrayList;
import java.util.HashMap;
import map.Map;
import map.Stalwart;
```

```
/**
 * The state contains the SLF along with all relevant information visible to
     the bot
 * @author Rene B. Hansen
 */
public class State {

    private String NFO;
    private SLF slf;
    private Map map;

    private HashMap<String, PathNode> nodes;
    private HashMap<String, PathNode> visibleNodes;

    private HashMap<String, PLR> players;
    private HashMap<String, PLR> visiblePlayers;


    private boolean inSyncMSG = false;



    /** Creates a new instance of State */
    public State() {
        this.nodes = new HashMap<String, PathNode>();
        this.visibleNodes = new HashMap<String, PathNode>();
        this.players = new HashMap<String, PLR>();
        this.visiblePlayers = new HashMap<String, PLR>();

        slf = new SLF();
        map = new Stalwart();
    }

    public void setNFO(String NFO){
        this.NFO = NFO;
    }

    public SLF getSLF(){
        return this.slf;
    }

    /**
     * called as a synchronous block is started
     */
    public synchronized void startSync(){
        this.inSyncMSG = true;
        this.clearNodes();
    }

    /**
     * called as a synchronous block is ended
     */
    public synchronized void endSync(){
        this.inSyncMSG = false;
        this.notifyAll();
    }

    /**
     * Updates the NAVnodes
     */
    public synchronized void addNAVNode(String id, String location, boolean
         reachable){
        PathNode n;
        if (nodes.containsKey(id)){
```

```
            n = nodes.get(id);
            n.setReachable(reachable);
            visibleNodes.put(n.getId(),n);
        } else {
            n = new NAVNode(id,location,reachable);
            nodes.put(id,n);
            visibleNodes.put(id,n);
        }
    }

    /**
     * Updates the INV nodes
     */
    public synchronized void addINVNode(String id, String location, boolean
         reachable, String type){
        PathNode n;
        if (nodes.containsKey(id)){
            n = nodes.get(id);
            n.setReachable(reachable);
            visibleNodes.put(n.getId(),n);
        } else {
            n = new INVNode(id,location,reachable,type);
            nodes.put(id,n);
            visibleNodes.put(id,n);
        }
    }

    /**
     * updates the DOM nodes
     */
    public synchronized void addDOMNode(String id, String location, boolean
         reachable, int controller){
        PathNode n;
        if (nodes.containsKey(id)){
            n = nodes.get(id);
            n.setReachable(reachable);
            visibleNodes.put(n.getId(),n);
        } else {
            n = new DOMNode(id,location,reachable,controller);
            nodes.put(id,n);
            visibleNodes.put(id,n);
        }
    }

    /**
     * updates the MOV nodes
     */
    public synchronized void addMOVNode(String id, String location, boolean
         reachable, boolean damageTrig, String type){
        PathNode n;
        if (nodes.containsKey(id)){
            n = nodes.get(id);
            n.setReachable(reachable);
            visibleNodes.put(n.getId(),n);
        } else {
            n = new MOVNode(id,location,reachable,damageTrig, type);
            nodes.put(id,n);
            visibleNodes.put(id,n);
        }
    }

    /**
     * updates the visible players
     */
```

```
    public synchronized void updatePlayers(String id, String rotation, String
         location, String velocity, String name, int team, boolean reachable
        , String weapon){
        if (players.containsKey(id)){
            PLR plr = players.get(id);
            plr.updatePLR(rotation, location, velocity, reachable, weapon
                );
            visiblePlayers.put(plr.getId(),plr);
        }else {
            PLR plr = new PLR( id, rotation, location, velocity, name,
                team, reachable, weapon);
            players.put(plr.getId(),plr);
            visiblePlayers.put(plr.getId(),plr);
        }
    }

    /**
     * returns a visible player. Returns null if there is no visible player
         in sight
     * @return player id or null
     */
    public synchronized PLR getAVisibleEnemyPLR(){
        if (!visiblePlayers.isEmpty()){
            for (PLR p : visiblePlayers.values()){
                if (p.getTeam() != this.getSLF().getTeam()){
                    return p;
                }
            }
        }
        return null;
    }

    /**
     * monitor to wait until no enemy players are in sight
     */
    public synchronized void waitForEnemyPLRNonVisible(PLR plr) throws
        InterruptedException{
        while (visiblePlayers.containsKey(plr.getId()) && this.getSLF().
            getCurrentAmmo() > 0) {
            wait();
        }
    }

    /**
     * Clears all visible nodes and players right before an update
     */
    public synchronized void clearNodes(){
        visibleNodes.clear();
        visiblePlayers.clear();
    }


    /**
     * returns a reachable node id
     * @return node id
     */
    public synchronized String getAReachableNodeId(){
        for (PathNode n : visibleNodes.values()){
            if (n.getReachable() && !n.getId().equals(slf.getTarget())){
                return n.getId();
            }
        }
        return null;
    }

```

```
    /**
     * returns the map
     */
    public Map getMap(){
        return this.map;
    }

    /**
     * returns a node location given its id
     * @param id String id of a node
     * @return The given node corresponding coordinates
     */
    public synchronized String getNodeLocation(String id){
        return map.getLocation(id);
    }
}
```

C.3.2 state.SLF.java

```
/*
 * SLF.java
 *
 * Created on 5. november 2007, 13:05
 *
 * To change this template, choose Tools | Template Manager
 * and open the template in the editor.
 */

package state;

import util.Log;

/**
 * Datastructure used to mainain all relevant information about the bot
 * @author Rene B. Hansen
 */
public class SLF {

    public static final String ZERO_VELOCITY = "0.000000,0.000000,0.000000";

    /* Auto-updating self variables*/
    private String id;
    private Rotation rotation;
    private Location location;
    private String velocity;
    private String name;
    private int team;
    private int health;
    private String weapon;
    private int currentAmmo;
    private int armor;
    private int altFiring;

    /* Return value for GETPATH method*/
    private String[] path = null;


    /*Manual-updating self variables*/
    private String target = "none";
    private boolean killed = false;

    //For moving
    //private String storedLocation = "none";
```

```
    //private String storedRotation = "none";


    private Rotation targetRotation;
    private Location formerLocation;

    /** Creates a new instance of SLF */
    public SLF() {
        rotation = new Rotation();
        targetRotation = new Rotation();
        location = new Location();
        formerLocation = new Location();
    }

    /**
     * Method used by the parser to update all values.
     */
    public synchronized void updateSLF(String id, String rotation, String location,
            String velocity, String name, int team, int health, String weapon,
            int currentAmmo, int armor, int altFiring){
        this.id = id;
        this.rotation.parseFromString(rotation);
        this.location.parseFromString(location);
        this.velocity = velocity;
        this.name = name;
        this.team = team;
        this.health = health;
        this.weapon = weapon;
        this.currentAmmo = currentAmmo;
        this.armor = armor;
        this.altFiring = altFiring;
        this.notifyAll();
    }

    /**
     * Updates the bots current location id
     * @param target node id
     */
    public synchronized void setTarget(String target){
        this.target = target;
    }

    /**
     * returns the bots id location
     */
    public synchronized String getTarget(){
        return new String(this.target);
    }

    public synchronized String getVelocity(){
        return new String(this.velocity);
    }

    public synchronized Location getLocation(){
        return new Location(this.location);
    }

    /*public synchronized void waitForVelocity(String velocity) throws InterruptedException{
        while (!this.velocity.equals(velocity)){
            wait();
        }
    }*/
```

```
    /**
     * Used to record movement relevant information before a movement command is sent
     * to the server
     */
    public synchronized void startMovement(){
        this.formerLocation.parseFromFloats(location.getX(),location.getY(),location.getZ());
    }



    /**
     * Used to record rotational relevant information before a rotation command is sent
     * to the server
     */
    public synchronized void startRotation(int yaw){
        this.targetRotation.parseFromInts(rotation.getPitch(),rotation.getYaw(),rotation.getRoll());
        this.targetRotation.addToYaw(yaw);
    }


    /**
     * monitor to let the thread sleep while a rotation is taking place
     */
    public synchronized void waitForRotation() throws InterruptedException{
        int timeout = 10;
        while ( !this.rotation.equalTo(this.targetRotation) && timeout != 0 ){
           wait();
           timeout--;
        }
    }


    /**
     * monitor to let the thread sleep while a movement is taking place
     */
    public synchronized boolean waitForMovement() throws InterruptedException{
        int timeout = 10;
        while ( (formerLocation.equalTo(this.location) || !(this.velocity.equals(SLF.ZERO_VELOCITY))) && timeout != 0){
            wait();
            if (this.velocity.equals(SLF.ZERO_VELOCITY)) timeout--;
        }
        if (timeout == 0) return false;
        else return true;

    }


    /**
     * used by the parser to update with a requested path
     */
    public synchronized void updatePath(String[] path){
        this.path = path;
    }
```

```
    /**
     * Block until server has responded with a path to last requested postion
     * @return String[] containing the locations of the path that needs to be
     * traversed in order to reach the requested destination
     */
    public synchronized String[] getPath() throws InterruptedException{
        int timeout = 5;
        while(this.path == null && timeout > 0) {
            wait();
            timeout--;
        }
        if (this.path == null) return null;
        String[] result = new String[this.path.length];
        System.arraycopy(this.path,0,result,0,this.path.length);
        this.path = null;
        return result;
    }

    public synchronized int getTeam(){
        return this.team;
    }

    public synchronized String getWeapon(){
        return this.weapon;
    }

    public synchronized int getCurrentAmmo(){
        return this.currentAmmo;
    }

    public synchronized int getHealth(){
        return this.health;
    }

    /**
     * variable to tell relevant monitors that the bot has been killed
     */
    public synchronized void setKilled(boolean killed){
        this.killed = killed;
    }




    public synchronized String toString(){
        return "{Id " + id + "} {Rotation " + rotation + "} {Location " +
             location +
                "} {Velocity " + velocity + "} {Name " + name + "} {Team " +
                team + "} {Health " + health + "} {Weapon " + this.weapon +
                "} {CurrentAmmo " + currentAmmo + "} {Armor " + armor + "} {
                    AltFiring " + altFiring;
    }

    public synchronized void gotKilled() {
        this.setTarget("none");
    }

    public synchronized String getId(){
        return this.id;
    }

    public synchronized String getName(){
        return this.name;
    }

```

```
}
```

C.3.3 state.PLR.java

```
/*
 * PLR.java
 *
 * Created on 27. januar 2008, 14:48
 *
 * To change this template, choose Tools | Template Manager
 * and open the template in the editor.
 */

package state;

/**
 * The representation of another player
 * @author Rene B. Hansen
 */
public class PLR {

    private final String id;
    private Rotation rotation;
    private Location location;
    private String velocity;
    private final String name;
    private final int team;
    private boolean reachable;
    private String weapon;



    /**
     * Creates a new instance of PLR
     */
    public PLR(String id, String rotation, String location, String velocity,
         String name, int team, boolean reachable, String weapon) {
        this.id = id;
        this.rotation = new Rotation(rotation);
        this.location = new Location(location);
        this.velocity = velocity;
        this.name = name;
        this.team = team;
        this.reachable = reachable;
        this.weapon = weapon;
    }

    public synchronized void updatePLR(String rotation, String location,
         String velocity, boolean reachable, String weapon){
        this.rotation = new Rotation(rotation);
        this.location = new Location(location);
        this.velocity = velocity;
        this.reachable = reachable;
        this.weapon = weapon;
    }

    public String getId(){
        return this.id;
    }

    public Rotation getRotation(){
```

```
        return this.rotation;
    }

    public synchronized Location getLocation(){
        return this.location;
    }

    public String getVelocity(){
        return this.velocity;
    }
    public String getName(){
        return this.name;
    }

    public int getTeam(){
        return this.team;
    }

    public boolean getReachable(){
        return this.reachable;
    }

    public String getWeapon(){
        return this.weapon;
    }

    public String toString(){
        String result = "";
        result += "id:"+this.id+" ";
        result += "rotation:"+this.rotation+" ";
        result += "location:"+this.location+" ";
        result += "velocity:"+this.velocity+" ";
        result += "name:"+this.name+" ";
        result += "team:"+this.team+" ";
        result += "reachable:"+this.reachable+" ";
        result += "weapon:"+this.weapon;
        return result;
    }

}
```

C.3.4 state.PathNode.java

```
/*
 * PathNode.java
 *
 * Created on 22. oktober 2007, 14:14
 *
 * To change this template, choose Tools | Template Manager
 * and open the template in the editor.
 */

package state;

/**
 * Used to represent a path from one node to another
 * @author Rene B. Hansen
 */
public class PathNode {

    private String id;

```

```
    /*Until it is needed to do geometric calculations, the location might aswell
      be represented as a string*/
    private String location;

    private boolean reachable;

    /** Creates a new instance of PathNode */
    public PathNode() {
    }

    /**
     * Creates a new instance of PathNode
     */
    public PathNode(String id, String location, boolean reachable){
        this.id = id;
        this.location = location;
        this.reachable = reachable;
    }

    public void setId(String id){
        this.id = id;
    }

    public String getId(){
        return this.id;
    }

    public void setLocation(String location){
        this.location = location;
    }

    public String getLocation(){
        return this.location;
    }

    public void setReachable(boolean reachable){
        this.reachable = reachable;
    }

    public boolean getReachable(){
        return this.reachable;
    }

    public String toString(){
        return "ID: " + id + " : " + location + " : " + reachable;
    }
}
```

C.3.5 state.NAVNode.java

```
/*
 * NAVNode.java
 *
 * Created on 22. oktober 2007, 16:13
 *
 * To change this template, choose Tools | Template Manager
 * and open the template in the editor.
 */

package state;

/**
```

```
 * Navigational node, denotes the location of a navigation point
 * @author Rene B. Hansen
 */
public class NAVNode extends PathNode {

    /** Creates a new instance of NAVNode */
    public NAVNode() {
    }

    /**
     * Creates a new instance of NAVNode
     */
    public NAVNode(String id, String location, boolean reachable){
        super(id,location,reachable);
    }

}
```

C.3.6 state.DOMNode.java

```
package state;

/**
 * Domination node, denotes a domination location
 * @author Rene B. Hansen
 */
public class DOMNode extends PathNode {

    private int controller;

    /** Creates a new instance of DOMNode */
    public DOMNode() {
    }

    /**
     * Creates a new instance of DOMNode
     */
    public DOMNode(String id, String location, boolean reachable, int
         controller){
        super(id,location,reachable);
        this.controller = controller;
    }

    public void setController(int controller){
        this.controller = controller;
    }

    public int getController(){
        return this.controller;
    }

    public String toString(){
        return super.toString() + " : " + controller;
    }
}
```

C.3.7 state.INVNode.java

```
package state;

```

```
/**
 * Inventory node, denotes the location of an inventory item
 * @author Rene B. Hansen
 */
public class INVNode extends PathNode {

    private String type;

    /** Creates a new instance of INVNode */
    public INVNode() {
    }

    /**
     * Creates a new instance of INVNode
     */
    public INVNode(String id, String location, boolean reachable, String type
        ){
        super(id,location,reachable);
        this.type = type;
    }

    public void setType(String type){
        this.type = type;
    }

    public String getType(){
        return this.type;
    }

    public String toString(){
        return super.toString() + " : " + type;
    }

}
```

C.3.8 state.MOVNode.java

```
/*
 * MOVNode.java
 *
 * Created on 23. oktober 2007, 02:44
 *
 * To change this template, choose Tools | Template Manager
 * and open the template in the editor.
 */

package state;

/**
 * Mover node, denotes the location of a mover, such as a lift
 * @author Rene B. Hansen
 */
public class MOVNode extends PathNode {

    private boolean damageTrig;
    private String type;

    /** Creates a new instance of MOVNode */
    public MOVNode() {
    }

    public MOVNode(String id, String location, boolean reachable, boolean
        damageTrig, String type){
```

```
        super(id,location,reachable);
        this.damageTrig = damageTrig;
        this.type = type;
    }

    public void setDamageTrig(boolean damageTrig){
        this.damageTrig = damageTrig;
    }

    public boolean getDamageTrig(){
        return damageTrig;
    }

    public void setType(String type){
        this.type = type;
    }

    public String getType(){
        return this.type;
    }

    public String toString(){
        return super.toString() + " : " + damageTrig + " : " + type;
    }

}
```

C.3.9 state.Location.java

```
package state;

/**
 * Internal representation of a location
 * @author Rene B. Hansen
 */
public class Location {

    private float x;
    private float y;
    private float z;

    /** Creates a new instance of Location */
    public Location(){

    }

    /**
     * Creates a new instance of Location
     */
    public Location(float x, float y, float z) {
        parseFromFloats(x,y,z);
    }

    /**
     * Creates a new instance of Location
     */
    public Location(String location){
        parseFromString(location);
    }

    /**
     * Creates a new instance of Location
     */
```

```
    public Location(Location location){
        this.x = location.getX();
        this.y = location.getY();
        this.z = location.getZ();
    }

    public void parseFromString(String location){
        String[] elements = location.split(",");
        this.x = Float.parseFloat(elements[0]);
        this.y = Float.parseFloat(elements[1]);
        this.z = Float.parseFloat(elements[2]);
    }

    public void parseFromFloats(float x, float y, float z){
        this.x = x;
        this.y = y;
        this.z = z;
    }

    public float getX(){
        return this.x;
    }

    public float getY(){
        return this.y;
    }

    public float getZ(){
        return this.z;
    }

    public boolean equalTo(Location l){
        return (this.x == l.getX()) && (this.y == l.getY()) && (this.z == l.
            getZ());
    }

    public String toString(){
        return "" +this.x+","+this.y+","+this.z;
    }

}
```

C.3.10 state.Rotation.java

```
/*
 * Rotation.java
 *
 * Created on 7. november 2007, 14:14
 *
 * To change this template, choose Tools | Template Manager
 * and open the template in the editor.
 */

package state;

/**
 * Internal representation of a bots current rotation
 * @author Rene B. Hansen
 */
public class Rotation {

    private int pitch;
    private int yaw;
```

```
    private int roll;

    /** Creates a new instance of Rotation */
    public Rotation(){

    }

    /**
     * Creates a new instance of Rotation
     */
    public Rotation(int pitch, int yaw, int roll) {
        parseFromInts(pitch,yaw,roll);
    }

    /**
     * Creates a new instance of Rotation
     */
    public Rotation(String rotation){
        parseFromString(rotation);
    }

    /**
     * Creates a new instance of Rotation
     */
    public Rotation(Rotation rotation){
        this.pitch = rotation.getPitch();
        this.yaw = rotation.getYaw();
        this.roll = rotation.getRoll();
    }

    public void parseFromString(String rotation){
        String[] elements = rotation.split(",");
        this.pitch = Integer.parseInt(elements[0]);
        this.yaw = Integer.parseInt(elements[1]);
        this.roll = Integer.parseInt(elements[2]);
    }

    public void parseFromInts(int pitch, int yaw, int roll){
        this.pitch = pitch ;
        this.yaw = yaw;
        this.roll = roll;
    }

    public void addToYaw(int rotation){
        this.yaw = (yaw + rotation) % 65535;
    }

    public int getPitch(){
        return this.pitch;
    }

    public int getYaw(){
        return this.yaw;
    }

    public int getRoll(){
        return this.roll;
    }

    public boolean equalTo(Rotation r){
       return (Math.abs(this.pitch - r.getPitch()) < 500 ) && (Math.abs(this.
            yaw - r.getYaw()) < 500) && (/*Math.abs(this.roll - r.getRoll())
            < 100*/ true);

    }
```

```

    public String toString(){
        return "" +this.pitch+","+this.yaw+","+this.roll;
    }

}
```

C.4 com

C.4.1 com.ComHandler.java

```
package com;

import state.State;

/**
 * Class that handles all communication to and from the UT server
 * @author Rene B. Hansen
 */
public class ComHandler {

    private Parser parser;
    public Connection connection;


    /**
     * Creates a new instance of ComHandler
     * @param state state to update uppon incoming msg
     */
    public ComHandler(State state) {
        parser = new Parser(state);
        connection = new Connection(parser);
    }


    /**
     * Initializes the bot on the server, with a random name and team
     */
    public void init(){
        connection.write("INIT");
    }


    /**
     * Initializes the bot on the server, with the given name and on a random
         team
     * @param name The name of the bot
     */
    public void init(String name){
        connection.write("INIT {Name "+name);
    }


    /**
     * Initialzes the bot on the server, with the given name and team
     * @param name The name of the bot
     * @param team The team of the bot
     */
    public void init(String name, int team){
        connection.write("INIT {Name "+name+"} {Team "+team+"}");
```

```
    }


    /**
     * Runs to the given target. The target most be in visual range
     * @param id The Id of the target, most be visual to the bot
     */
    public void runToTarget(String id){
        connection.write("RUNTO {Target "+id+"}");
    }


    /**
     * Turns and runs in a straight line to the raget.
     *
     * @param location The x,y,z coordinates of the location, most be on the
     * form 'x,y,z' or 'x y z'
     */
    public void runToLocation(String location){
        connection.write("RUNTO {Location "+location+"}");
    }

    /**
     * rotation should be ("0 50000 0" or "0,50000,0") and 2Pi = 65535 units
     * @param pitch pitch value
     * @param yaw yaw value
     * @param roll roll value
     */
    public void turnToRotation(int pitch, int yaw, int roll){
        connection.write("TURNTO {Rotation "+pitch+","+yaw+","+roll+"}");
    }

    /**
     * Turns the bot towards the given location
     * @param location location to turn to
     */
    public void turnToLocation(String location){
        connection.write("TURNTO {Location "+location+"}");
    }

    /**
     * Makes the bot rotate the given amount
     * @param amount value to rotate
     */
    public void rotateAmount(int amount){
        connection.write("ROTATE {Amount "+amount+"}");
    }

    /**
     * Requests the server for a path from the bots current location, to the location
     * specified by the x,y,z values
     * @param x value
     * @param y value
     * @param z value
     */
    public void getPath(int x, int y, int z){
        connection.write("GETPATH {Location "+x+" "+y+" "+z+"}");
    }

    /**
     * Requests the server for a path from the bots current location, to the location
     * specified by the string
     * @param location x,y,z string location
```

```
     */
    public void getPath(String location){
        connection.write("GETPATH {Location "+location+"}");
    }

    /**
     * Makes the bot shoot at the given target and/or location
     * @param location location on the form x,y,z
     * @param target target id
     */
    public void shootAt(String location,String target){
        connection.write("SHOOT {Location "+location+"} {Target "+target+"}")
            ;
    }

    /**
     * Makes the bot stop shooting
     */
    public void stopShoot(){
        connection.write("STOPSHOOT");
    }

}
```

C.4.2 com.Connection.java

```
package com;

import java.io.BufferedReader;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;
import java.net.UnknownHostException;

/**
 * Connection to the UT server
 * @author Rene B. Hansen
 */
public class Connection extends Thread{

    private Socket socket;
    private BufferedReader in;
    private DataOutputStream out;

    private Parser parser;


    /**
     * The connection to the UT server
     * @param parser The parser for incomming messages
     */
    public Connection(Parser parser) {
        this.parser = parser;
        try {
            socket = new Socket("localhost",3000);
            in = new BufferedReader(new InputStreamReader(socket.
                getInputStream()));
            out = new DataOutputStream(socket.getOutputStream());
            this.start();
        } catch (UnknownHostException ex) {
```

```
                ex.printStackTrace();
            } catch (IOException ex) {
                ex.printStackTrace();
            }
        }

        /**
         * run method for the connection
         */
        public void run(){
            try{
                String responseLine;
                while ((responseLine = in.readLine()) != null) {
                    parser.parseMsg(responseLine);
                }
                System.err.println("COM ERROR - response line was null");

            }catch (IOException ioe){ioe.printStackTrace();}

        }

        /**
         * Sends the given String to the output of the connection
         * @param msg output to the server
         */
        public void write(String msg){
            try {
                out.writeBytes(msg+"\r");
                out.flush();
            } catch (IOException ex) {
                ex.printStackTrace();
            }
        }
}
```

C.4.3 com.Parser.java

```
package com;

import state.DOMNode;
import state.INVNode;
import state.MOVNode;
import state.NAVNode;
import state.State;
import java.util.StringTokenizer;
import util.Log;

/**
 * Parser for msg's from the server
 * @author Rene B. Hansen
 */
public class Parser {

    private State state;
    private StringTokenizer tokenizer;
    private String command;

    private boolean inSyncMSG;

    /**
     * Creates a new instance of Parser
     * @param state The state of the world
     */
```

```
    public Parser(State state) {
        this.state = state;
        this.inSyncMSG = false;
    }

    /**
     * Method called to parse an incoming message
     * @param msg message to be parsed
     */
    public void parseMsg(String msg){
        if (msg.length() >= 3) command = msg.substring(0,3);
        else System.err.println("ERROR - invalid msg: "+msg);


        if (command.equals("NFO")){
            NFO(msg.substring(4));
        } else if (command.equals("NAV")){
            NAV(msg.substring(4));
        } else if (command.equals("BEG")){
            BEG();
        } else if (command.equals("END")){
            END();
        } else if (command.equals("INV")){
            INV(msg.substring(4));
        } else if (command.equals("DOM")){
            DOM(msg.substring(4));
        } else if (command.equals("MOV")){
            MOV(msg.substring(4));
        } else if (command.equals("FIN")){
            System.out.println("The Game has finished...");
        } else if (command.equals("SLF")){
            SLF(msg.substring(4));
        } else if (command.equals("PTH")){
            PTH(msg.substring(4));
        }else if (command.equals("PLR")){
            PLR(msg.substring(4));
        }else if (command.equals("BMP")){
            BMP();
        }else if (command.equals("DIE")){
             DIE();
        }else if (command.equals("KIL")){
        }else{
            //System.err.println("THIS WAS NOT COUGHT IN THE PARSER: " + msg)
                ;
        }
    }

    private void NFO(String NFO){
        state.setNFO(NFO);
    }

    private void NAV(String NAV){
        String[] elements = NAV.split(" ");
        state.addNAVNode(elements[1].substring(0,elements[1].length() - 1),
                elements[3].substring(0,elements[3].length() - 1),
                Boolean.valueOf(elements[5].substring(0,elements[5].length()
                    - 1)));
    }

    private void INV(String INV){
        String[] elements = INV.split(" ");
        state.addINVNode(elements[1].substring(0,elements[1].length() - 1),
                elements[3].substring(0,elements[3].length() - 1),
                Boolean.valueOf(elements[5].substring(0,elements[5].length()
                    - 1)),
```

```
                elements[7].substring(0,elements[7].length() - 1));
    }

    private void DOM(String DOM){
        String[] elements = DOM.split(" ");
        state.addDOMNode(elements[1].substring(0,elements[1].length() - 1),
                elements[3].substring(0,elements[3].length() - 1),
                Boolean.valueOf(elements[5].substring(0,elements[5].length()
                    - 1)),
                Integer.parseInt(elements[7].substring(0,elements[7].length()
                     - 1)));
    }

    private void MOV(String MOV){
        String[] elements = MOV.split(" ");
        state.addMOVNode(elements[1].substring(0,elements[1].length() - 1),
                elements[3].substring(0,elements[3].length() - 1),
                Boolean.valueOf(elements[5].substring(0,elements[5].length()
                    - 1)),
                Boolean.valueOf(elements[7].substring(0,elements[7].length()
                    - 1)),
                elements[9].substring(0,elements[9].length() - 1));
    }

    private void SLF(String SLF){
        String[] elements = SLF.split(" ");
        state.getSLF().updateSLF(
                elements[1].substring(0,elements[1].length() - 1),//Id
                elements[3].substring(0,elements[3].length() - 1),//Rotation
                elements[5].substring(0,elements[5].length() - 1),//Location
                elements[7].substring(0,elements[7].length() - 1),//Velocity
                elements[9].substring(0,elements[9].length() - 1),//Name
                Integer.parseInt(elements[11].substring(0,elements[11].length
                    () - 1)),//Team
                Integer.parseInt(elements[13].substring(0,elements[13].length
                    () - 1)),//Health
                elements[15].substring(0,elements[15].length() - 1),//Weapon
                Integer.parseInt(elements[17].substring(0,elements[17].length
                    () - 1)),//CurrentAmmo
                Integer.parseInt(elements[19].substring(0,elements[19].length
                    () - 1)),//Armor
                Integer.parseInt(elements[21].substring(0,elements[21].length
                    () - 1)) );//AltFiring
        //state.printSLF();
    }

    private void PTH(String PTH){
        String[] elements = PTH.split(" ");
        String[] result = {"NOPATH"};
        if(elements.length > 2){
            result = new String[(elements.length-2) / 3];
            for (int i = 0; i < result.length; i++){
                result[i] = elements[((i+1)*3) - 1 + 2].substring(0, elements
                    [((i+1)*3) - 1 + 2].length() - 1 );
            }
        } else {
        }
        state.getSLF().updatePath(result);
    }

    private void PLR(String PLR){
        String[] elements = PLR.split(" ");
        state.updatePlayers(
                elements[1].substring(0,elements[1].length() - 1),//Id
                elements[3].substring(0,elements[3].length() - 1),//Rotation
```

```
                elements[5].substring(0,elements[5].length() - 1),//Location
                elements[7].substring(0,elements[7].length() - 1),//Velocity
                elements[9].substring(0,elements[9].length() - 1),//Name
                Integer.parseInt(elements[11].substring(0,elements[11].length
                    () - 1)),//Team
                Boolean.parseBoolean(elements[13].substring(0,elements[13].
                    length() - 1)),//Reachable
                elements[15].substring(0,elements[15].length() - 1)//Weapon
                );
    }


    private void BMP(){
    }

    private void DIE(){
        state.getSLF().gotKilled();
    }

    private void BEG(){
        state.startSync();
    }

    private void END(){
        state.endSync();
        //state.printNodes();
    }

}
```

C.5 map

C.5.1 map.Map.java

```
package map;

import java.io.BufferedReader;
import java.io.IOException;
import java.util.HashMap;
import util.FileHandler;

/**
 * An internal representation of the map. Is used to get information about
     map
 * specific details, e.g. to convert an map id to a map location.
 * @author Rene B. Hansen
 */
public class Map {

    private HashMap<String,Node> map;
    private boolean mapUpdated = false;
    private String mapName = "Stalwart";
    private int counter = 0;


    /**
     * Creates a new instance of Map
     * @param mapName name of the map
     */
    public Map(String mapName){
```

```
        this.mapName = mapName;
        map = new HashMap<String,Node>();
        loadMap(mapName);
    }

    /**
     * Adds a node to the map
     * @param id node id
     * @param location node location on the form x,y,z
     * @return the node just added
     */
    public synchronized Node addNode(String id, String location){
        if (!map.containsKey(id)){
            Node n = new Node(id,location);
            map.put(id,n);
            mapUpdated = true;
            return n;
        }else return map.get(id);
    }

    /**
     * Load a map from a file specified by the name
     * @param name file name to be loaded
     */
    public synchronized void loadMap(String name){
        BufferedReader in = FileHandler.In(name);
        if (in != null){
            try {
                String temp = in.readLine();
                String[] elem;
                Node current = null;
                while (temp != null){
                    elem = temp.split(" ");
                    if (elem.length == 2){
                        current = this.addNode(elem[0],elem[1]);
                    }else if (elem.length == 3 && current != null){
                        current.addNode(this.addNode(elem[1],elem[2]));
                    }else {
                        System.err.println("Error... : invalid line parsed from map" +
                                " '" + temp+"'");
                    }
                    temp = in.readLine();
                }
            } catch (IOException ex) {
                ex.printStackTrace();
            }
        }else System.err.println("Reader Failed");
    }

    /**
     * Writes the map to a file
     */
    public synchronized void updateMap(){
        if (true){
            mapUpdated = false;
            FileHandler.Out(mapName+counter,this.toFile());
            counter++;
        }
    }

    /**
     * Returns true if the two ndoes are neighbours
     * @param id1 node1
     * @param id2 node2
```

```
     * @return true if node1 is neighbour to node2
     */
    public synchronized boolean neighbours(String id1,String id2){
        if (map.containsKey(id1) && map.containsKey(id2)){
            Node n = map.get(id1);
            return n.getReachableNodes().contains(map.get(id2));
        }else return false;
    }

    /**
     * Aux method used to convert the map to a proper string form, so that it can be
     * written to a file
     * @return String representation of the map
     */
    public synchronized String toFile(){
        String result = "";
        for (Node n : map.values()){
            result += n.getId() + " "+n.getLocation()+"\n";
            for (Node neighbor : n.getReachableNodes()){
                result += "@ "+neighbor.getId()+" "+neighbor.getLocation()+"\n";
            }
        }
        return result;
    }

    /**
     * Returns the location of the specified node
     * @param id id of a node
     * @return corresponding location on the form x,y,z
     */
    public synchronized String getLocation(String id){
        if (map.containsKey(id)){
            return map.get(id).getLocation();
        } else return null;
    }

    /**
     * Returns a random inventory node
     * @return inventory node id
     */
    public synchronized Node getRandomInventoryLocation(){
        double temp = Math.random();
        temp = temp * (double)map.size();
        int i = (int) temp;
        Node result = null;
        for (Node n : map.values()){
            if (n.getId().startsWith("dom-stalwart.InventorySpot")){
                result = n;
            }
            i--;
            if (i <= 0 && result != null) return result;
        }
        return result;
    }




}
```

C.5.2 map.Node.java

```
package map;

import java.util.ArrayList;

/**
 * Datastructure of a map node
 * @author Rene B. Hansen
 */
public class Node {

    private String id;
    private String location;
    private ArrayList<Node> reachableNodes;

    boolean discovered;
    boolean finalized;

    /** Creates a new instance of Node */
    public Node(String id, String location) {
        this.id = id;
        this.location = location;
        this.discovered = false;
        this.finalized = false;
        reachableNodes = new ArrayList<Node>();
    }

    public void addNode(Node n){
        if (!reachableNodes.contains(n)){
            reachableNodes.add(n);
        }
    }

    public String getId(){
        return this.id;
    }

    public void setDiscovered(boolean discovered){
        this.discovered = discovered;
    }

    public boolean getDiscovered(){
        return this.discovered;
    }

    public void setFinalized(boolean finalized){
        this.finalized = finalized;
    }

    public boolean getFinalized(){
        return this.finalized;
    }

    public String getLocation(){
        return this.location;
    }

    public ArrayList<Node> getReachableNodes(){
        return this.reachableNodes;
    }

    public String toString(){
        return "id:"+id+" - location:"+location+" - discovered:"+discovered;
    }
```

```

    public String toFile(){
        String result = "";

        result += id + "," + location + "\n\n";
        for (Node n : reachableNodes){
            result += n.getId()+"\n";
        }

        return result;
    }

}
```

C.5.3 map.Stalwart.java

```
package map;

/**
 *
 * @author Rene B. Hansen
 */
public class Stalwart extends Map{


    /** Creates a new instance of Stalwart */
    public Stalwart() {
        super("Stalwart");
    }
}
```

C.6 test

C.6.1 test.Main.java

```
package test;

import ai.Coordinator;
import bot.Bot;
import exceptions.DuplicateTaskException;
import java.io.IOException;

/**
 * Main method
 * @author Rene B. Hansen
 */
public class Main {

    /**
     * Path and name for simpleTasks HTN file
     */
    public static String simpleTasks = "simpletasks.htn";
    /**
     * Path and name for compoundTasks HTN file
     */
    public static String compoundTasks = "compoundtasks.htn";

```

```

    /** Creates a new instance of Main */
    public Main() {
        Coordinator c = Coordinator.getInstance();
    }

    /**
     *
     * @param args the command line arguments denotes the names of the
         simpletask and compound task
     * file.
     */
    public static void main(String[] args)  {
            if (args.length > 1){
                Main.simpleTasks = args[0];
                Main.compoundTasks = args[1];
            }else if (args.length == 1){
                Main.simpleTasks = args[0];
            }

            Main m = new Main();

    }

}
```

C.7 exceptions

C.7.1 exceptions.DuplicateTaskException.java

```
package exceptions;

/**
 * Exception to be thrown by the parser if a method appear more than once in
     the syntax
 * @author Rene B. Hansen
 */
public class DuplicateTaskException extends Exception {

    /**
     * Creates a new instance of DuplicateTaskException
     */
    public DuplicateTaskException() {
        super();
    }

    /**
     *
     * @param message String message
     */
    public DuplicateTaskException(String message){
        super(message);
    }
}
```

C.8 util

C.8.1 util.FileHandler.java

```
package util;

import java.io.*;

/**
 * Class used to handle the IO files
 * @author Rene B. Hansen
 */
public class FileHandler {

    /**
     * Creates a new instance of FileHandler
     */
    public FileHandler() {
    }

    public static void Out(String name, String contents){
        try{
            // Create file
            FileWriter fstream = new FileWriter(name);
            BufferedWriter out = new BufferedWriter(fstream);
            out.write(contents);
            //Close the output stream
            out.close();
        }catch (Exception e){//Catch exception if any
            System.err.println("Error: " + e.getMessage());
        }
    }

    public static BufferedReader In(String name){
        try{
            FileReader fstream = new FileReader(name);
            BufferedReader in = new BufferedReader(fstream);
            return in;
        }catch (Exception e){
            System.err.println("Error: "+ e.getMessage());
        }
        return null;
    }

}
```

C.8.2 util.Log.java

```
package util;

/**
 * Utility to print pretty print Object[]
 * @author Rene B. Hansen
 */
public class Log {

    public static String print(Object[] in){
        String result = "";
        if (in == null) result = "null";
        else {
```

```
            for (Object o : in){
                result += o.toString() + " ";
            }
        }
        return result.substring(0,result.length()-1);
    }
}
```

Bibliography

[1] Ghallab, Malik; Nau, Dana and Traverso, Paolo (2004). Automated Planning Theory and Practice.

[2] Muñoz-Avila, Héctor and Fisher, Todd (2002). Strategic Planning for Unreal Tournement© Bots. Article, Department of Computer Science and Engineering, Lehigh University.

[3] Hoang, Hai; Lee-Urban, Stephen; Muñoz-Avila, Héctor. Hierarchical Plan Representations for Encoding Strategic Game AI. Department of Computer Science & Engineering, Lehigh University, Bethlehem, PA 18015-3084 USA.

[4] Orkin, Jeff (2006). Three States and a Plan: The A.I. of F.E.A.R. Monolith Productions / M.I.T Media Lab, Cognitive Machines Group. Game Developers Conference 2006.

[5] RaptoR. Unreal Tournament "You are inferior, human!". Rewiev by RaptoR:
http://www.planetdreamcast.com/games/reviews/unrealtournament/

[6] AIISC, IGDA's AI Special interest group. http://www.igda.org/ai/

[7] Wikipedia. A* search algorithm.
http://en.wikipedia.org/wiki/A*_search_algorithm

[8] Gamebots. http://gamebots.planetunreal.gamespy.com/index.html

[9] Gamebots Network API. http://gamebots.planetunreal.gamespy.com/docapi.html

[10] Clement, Bradley J. Durfee, Edmund H. (1999). Theory for Coordinating Concurrent Hierarchical Planning Agents Using Summary Information. Article, University of Michigan

[11] Priestley, Mark (2003). Practical Object-Oriented Design With UML - Second Edition. McGraw-Hill Education.

[12] Rabin, Steve (2002). AI Game Programming Wisdom. Charles River Media.

[13] Raim, Jarret. Finite State Machine in Games. Slides by: Jarret Raim.