Artificial Intelligent Player's Planning in Massively Multi-Player On-Line Role-Playing Games

Marko Maliković
Faculty of Humanities and Social Sciences
University of Rijeka
Sveučilišna avenija 4, 51000 Rijeka, Croatia
marko.malikovic@ffri.hr

Markus Schatten
Faculty of Organization and Informatics
University of Zagreb
Pavlinska 2, 42000 Varaždin, Croatia
markus.schatten@foi.hr

Abstract. *Massively multi-player on-line role-playing games (MMORPGs) are relatively new phenomena in the gaming industry in which a potentially large number of players play simultaneously. An interesting problem in such games is to develop artificial players (bots) that are able to play such games and interact with other players. In this paper we provide an initial implementation of such an artificial player in Prolog based on an automated planning system and a belief-desire-intention (BDI) agent model. The artificial player is able to receive tasks (quests) from non-playing characters (NPCs) and solve them by taking one action after another according to its internal planning mechanism.*

Keywords. MMORPG, AI player, Planning, Formalization, Prolog

1 Introduction

Massively multi-player on-line role-playing games (MMORPGs) allow human players to control the actions of their protagonist (avatar) in Internet-based games, in which a very large number of players interact with one another in a virtual world [10]. A usual setting is that a protagonist is placed into a world in which he interacts with various NPCs (non-player characters) which give out tasks (quests) that he has to solve to be able to buy better equipment, learn new skills, or proceed to higher levels. MMORPGs provide a good foundation for investigating the design and implementation of large-scale distributed artificial intelligence (in the form of NPCs on the one hand, as well as AI players on the other). Such research can be carried out using agent-based methods and multi-agent systems (MAS). In this paper we use Belief-Desire-Intention (BDI) agent theory [2] for the definition of artificial players. The BDI model is a software development abstraction that allows the implementation of intelligent agents that are able to practically reason about a given domain (their view of the environment they are situated in), and autonomously generate plans in order to achieve their design objectives (usually given to them by their owners). In essence, BDI consists of:

- *Beliefs* which represent the knowledge base of the agent (e.g. its representation of the world);
- *Desires* which represent the motivations of an agent (e.g. the goals and objectives an agent wants to achieve);
- *Intentions* which represent the deliberative conclusions of practical reasoning of how to achieve a desire the agent has committed themselves to (e.g. plans which represent sequences of actions).

Usually, a BDI based agent features two important processes: (1) deliberation about objectives (e.g. *what* to try and achieve next), and (2) deliberation about the means (e.g. *how* to achieve the goals the agent has committed to). The second process is deeply connected to the field of automated planning. It usually features (1) a (formal) description of objectives that have to be achieved, (2) a description of actions that can be done, and (3) a description of the state of the environment. By using these descriptions automated planning software tries to generate a plan that will achieve the stated objectives by using the available actions on the environment [11]. For the sake of this paper, in the following paragraphs we will call artificial participants in the game *AI players*.

In this paper, we will present our artificial player's planning system formalized in SWI Prolog [9]. This planning system is developed as one of the components within the ModelMMORPG (Large-Scale Multi-Agent Modeling of Massively On-Line Role-Playing Games) project[1], and our artificial players will play a MMORPG game alongside real (human controlled) players. When we experimented with various programming options in the early stage of the project, we concluded that logic programming with Prolog [1] is the most suitable for the planning system that we require. As we have already said, our AI players receive tasks (quests) of various NPC's, so

[1] See http://ai.foi.hr/modelmmorpg for details.

our system, based on our formal description of quests and on AI player's knowledge and/or communication with other AI players, generates an action plan that will lead him to achieve the goal (if that is possible given the current state of knowledge and state of the AI player, as well as the state of the environment). In the ModelMMORPG project we have chosen *The Mana World* MMORPG [8] to conduct our research. The reasons for selection were: (a) it is open source (GPL licensed) allowing us to modify code and add additional functionality, (b) it has a supportive community, (c) it supports a number of interaction techniques which can be studied (e.g. trade among players, IRC based chat, organizing teams called parties, social network functions e.g. friends, enemies, parties etc.), (d) it is a (more or less) finished game featuring lots of quests that can be analyzed.

Our Prolog-based artificial players will be connected with The Mana World game using Python as an intermediate scripting language. Especially the SPADE (Smart Python Agent Development Environment) [7] which is a FIPA-ACL [6] compliant agent development platform that allows for the implementation of BDI agents and has interfaces to a number of deductive engines including XSB Prolog, SWI Prolog, Flora-2, ECLiPSe Prolog, as well as a SPARQL querying engine.

Currently there are not many studies that deal with the implementation of artificial players in MMORPG games. Most studies try to develop countermeasures to prevent artificial players, e.g. to detect bots based on their behavior and consequently ban them from the game. Only few studies use an agent based approach in MMORPGs, but an artificial planning system is only implemented in [3]. This study deals with the area of computer-mediated storytelling, and describes the "development of an expert case-based character director system which dynamically generates and controls a story, which is played out in a multiplayer networked game world". The system is based on work described in [4], in which the original implementation was limited to one player taking control of a hero in a simple scripted hero/villain story structure, with no AI story-generation capability. The proposed system includes a story director system which utilizes the case based planning paradigm, and facilitates multiplayer stories. Stories are modeled as cases, and their planning and scheduling is the primary activity of the system. This work is story-centered, and the notion of artificial intelligence for the most part refers to the story director agent. Modeling of the NPCs is performed in a layered structure, from low-level behaviors to higher level targeted goals (low level such as collision detection, followed by social simulation, idle behaviors, targeted behaviors, attitudes). Thus, the cited study does not try to use an automated planning system to implement an artificial player, but to make the storyline of the game more interesting.

The rest of this paper is organized as follows: in Section 2 we describe our formalization of the state of the world in The Mana World game, in Section 3 we describe our formalization of quests in The Mana World game, in Section 4 we give our plan of dealing with changes in the environment, and in Section 5 we draw our conclusions and give an outline of future research steps.

2 Formalization of the state of the world

In using the label “the formal state of the world” (in our case a game, The Mana World system), we mean the formal description of all the elements (static or dynamic) that appear in that world. The state of the world includes various aspects. One aspect determines what the world looks like, and at the game level that aspect is realized through graphic elements. This includes maps that give a visual representation of the world, as well as various sprites and locations of beings and other visual things. In Fig. 1 we see a screenshot of a moment when playing The Mana World using the ManaPlus client.[2]

Figure 1. The Mana World

In our Prolog-based system the above mentioned state of the world is described through a number of predicate instances.

Very important points on the maps represent the so-called *blocked points*. Blocked points are points where the AI player cannot stand or cross (e.g. walls and other immovable objects). Using open source code from The Mana World, we generated a predicate *blocked* in our Prolog system where instance *blocked(m,x,y)* states that a particular map *m* has a blocked point at coordinates *(x,y)*. Another generated predicate *warp*(m_1,x_1,y_1,m_2,x_2,y_2) states that a particular map m_1 has a gateway at coordinates (x_1,y_1)

[2] ManaPlus is a free OpenSource 2D MMORPG client for The Mana World and similar servers.

that leads to map m_2 at coordinates (x_2,y_2). Of course, the possibilities for the movement of the AI player are mainly determined by blocked points and by gateways from the maps to the maps. Fig. 2 represents one simplified map in which we generate blocked points and gateways.

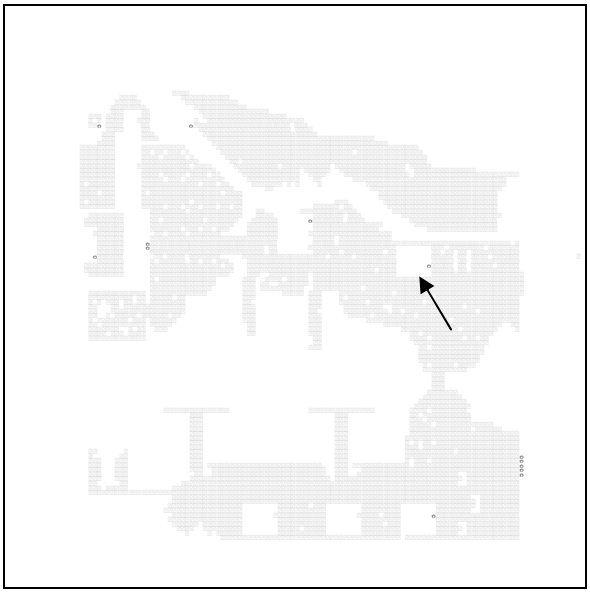

Figure 2. A simplified map with blocked points and gateways

Blocked points are shown in gray shading whilst the gateways from and to the map are represented as black dots. Among some locations the AI player knows to move "independently" (i.e. at the game level). In some other cases, we have to introduce heuristic solutions obtained by observation. For example, in Fig. 2 in the area marked by an arrow, the AI can come only from some other map but he does not know how to do it independently.

Another important aspect of the state of the world includes the attributes of beings and movable things that exist in the world. So, in addition to predicates that we have already described, there are groups of predicates that describe:

1. Positions of movable objects on maps (human players, AI players, a variety of items, NPCs, monsters/mobs, etc.)
   - Positions of such objects are stored in the predicate *location(t,o,m,x,y)* whose five parameters are: *t* (type of the object), *o* (name of the object), *m* (map where the object is located) and *x* and *y* (coordinates of the object on map *m*);
   - Instances of predicates *location* and *blocked* contain all the movable and motionless objects on the maps which provides us with a description of fields that are free and that the AI players can move upon;
2. Attributes of AI players (game level, weapons, clothes, shield, various collected items, money, etc.)
   - Attributes of AI players are stored in the predicate *ownership(a,att,m)* whose three parameters are: *a* (name of AI player), *att* (attribute) and *m* (value of attribute's measure);
3. Attributes of monsters or mobs (in most cases equivalent to the attributes of AI players);
4. Attributes of various game-related items including, but not limited to, weapons, shield, clothing, etc.

3 Planning system implementation

In order to allow our AI players to reason about and plan the execution of quests, we need to provide a formalization of these quests. Here, we give a description of this formalization that will lead us to the implementation of the planning system.

3.1 Description of quests

Quests given out by NPCs can include simple missions like collecting or buying certain items, searching for and attacking various beings, walking to certain locations or talking to several NPCs, but also more complicated things like solving puzzles or winning a boss fight. Some quests have requirements like previously completing another quest, or being at a certain level or above it (the level can be either required or just recommended). Also, some quests have costs like items or money.

Each quest will reward the player with something. The rewards can be experience points (EXP), money, items, equipment, daily points, boss points, skills, magic spells or something else. EXP rewards will grant the total amount, meaning they don't get cut off when reaching a new level, so an AI player might also raise more than a single level while completing a quest. Asides from those characteristics listed earlier, each quest has a starting location and redoables (e.g. a player can repeat that quest multiple times).

As we will see, the above mentioned quest's characteristics will be of special importance in our formalization of quests. In this paper, the quest *Vincent* from the set of the so-called *Newbie Quests* will serve as an example of our formalization. All quests from the set *Newbie Quests* must be performed in a precisely defined order, and the quest *Vincent* is just one of them.

The starting location of quest *Vincent* is in Candor Island (see Fig. 2) and given out by NPC Vincent. The required level for performing this quest is 1 but the recommended level is 20. This quest is not redoable, thus an AI player can perform it only once. Prerequisites of this quest are several other *Newbie Quests* (concretely quests *Bernard*, *Mikhail* and

Sarah). Once it is completed, the AI player receives a reward of 1000 gold points (GPs), but performing this quest will cost the AI player 10 items (in this case Bug Legs which have already been, or have to be, collected).

Figure 2. Candor Island

To perform this quest, the AI player has to follow these instructions: *From the Candor Island port, go east from Valon until you reach a beach and you can see Vincent. Talk to Vincent, and he will ask you to collect 10 Bug Legs for his action figure. Go and collect 10 Bug Legs. Talk to Vincent one last time and give him the 10 Bug Legs you collected to complete the quest.*

3.2 Formalization of quests

In this Section, we show a detailed formalization of quests using SWI Prolog based on the example quest *Vincent*. All quests can be deconstructed into basic actions. These basic actions can be of various kinds: walking towards a certain location, attacking a certain monster, taking an item, talking to a player or NPC, buying something (weapons, clothing), etc. Thus, a possible formalization of quest *Vincent* which we have chosen to use is mainly a set of basic actions lined up in the required order and looks like the following:

```
do_quest(NPC,A,vincent) :-
   location(npc,vincent,M,XVT,YVT),
   ownership(A,level,La), La>=1,
   done_quest(A,bernard),
   done_quest(A,mikhail),
   done_quest(A,sarah),
   waiting_quest(NPC,A,vincent),
   \+ done_quest(A,vincent),
   ownership(A,money,Ma),
   retract(ownership(A,money,Ma)),
   NewMa is Ma+1000,
   assert(ownership(A,money,NewMa)),
   walk_to_location(A,M,XVT,YVT),
   assert(plan(talk(A,vincent,'Hi Vincent!'))),
   assert(plan(talk(vincent,A,'Give me 10 Bug
Legs for my action figure!'))),
   ownership(A,bug_legs,BL),
  (BL>=10
   ->
   NewBL is BL-10,
   assert(ownership(A,bug_legs,NewBL)),
   retract(ownership(A,bug_legs,BL));
   collect_items(A,bug_legs,10),
   assert(ownership(A,bug_legs,0)),
   retract(ownership(A,bug_legs,BL))),
   walk_to_location(A,M,XVT,YVT),
   assert(plan(talk(A,vincent,'I collected 10 Bug
Legs. I give it to you!'))),
   retract(waiting_quest(NPC,A,vincent))
   ->
   assert(done_quest(A,vincent));
   assert(failed_quest(NPC,A,vincent)).
```

The parameters of the rule *do_quest(NPC,A,Q)* state that AI player *A* got instructions for the quest *Q* from a non-player character named *NPC*.

Also, we describe and show the formalization of certain basic actions which are used in quest *Vincent*. The Prolog specification of action *walk_to_location(A,M,X,Y)* which means "AI player *A* on map *M* walk to location *(X,Y)*" called by *do_quest(NPC,A,vincent)* is implemented in our system and tells the AI player which way to go. However, we note that the rule *walk_to_location* is used only in some cases, because (as we have already said) AI players are (at the game level) usually able to move independently. Therefore, the specification of this rule will not be shown here.

As we can see, apart from the recursive action *walk_to_location*, from quest *Vincent* the recursive action *collect_items(A,Item,N)* by which the AI player *A* collects *N* items of some kind *Item* is called as well. If the AI player already has some items of the kind *Item* then he will collect just as much as he needs to *N*. In general, it can happen that the AI player (for other reasons) collected some things earlier that he needs to solve the current quest. Additional important information for this (and some other) quests is that some items are dropped in certain percentages by some mobs. In the case of the Bug Legs which need to be collected in order to solve the quest Vincent, they have been dropped by: Maggot (4%), Cave Maggot (4%), Scorpion (7%), Bat (4%), Angry Scorpion (7%), Spiky Mushroom (0,5%), Fire Goblin (8%), Ice Goblin (8%), Sea Slime (5%), Red Scorpion (5%), Giant Maggot (7,5%), and Black Scorpion (8%). This tells us which mobs the AI player should catch (kill) in order to collect enough items of a certain kind (in this case Bug Legs). Therefore, in the formalization of the state of the world we also have information about which items are (in a certain percentage) dropped by which mobs. For example, the following instances of predicates say which mobs dropped bug legs in what percentage:

```
dropped(bug_legs,maggot,4).
dropped(bug_legs,cave_maggot,4).
dropped(bug_legs,scorpion,7).
dropped(bug_legs,bat,4).
dropped(bug_legs,angry_scorpion,7).
dropped(bug_legs,spiky_mushroom,0.5).
dropped(bug_legs,fire_goblin,8).
dropped(bug_legs,ice_goblin,8).
dropped(bug_legs,sea_slime,5).
dropped(bug_legs,red_scorpion,5).
dropped(bug_legs,giant_maggot,7.5).
dropped(bug_legs,black_scorpion,8).
```

Based on these predicates, we can formalize a generic action *collect_items* in such a way that it allows the AI player to collect the concrete item he needs, depending on the mobs that drop them:

```
collect_items(A,Item,N) :-
  (ownership(A,Item,I),
   I<N,
   dropped(Item,Mob,P),
   location(agent,A,M,XA,YA),
   location(mob,Mob,M,XMob,YMob),
   line_of_sight(A,Mob)
   ->
   walk_to_location(A,M,XMob,YMob),
   assert(plan(attack(A,Mob))),
   retract(location(mob,Mob,M,XMob,YMob)),
   retract(ownership(A,Item,I)),
   NewI is I+1, assert(ownership(A,Item,NewI)),
   collect_items(A,Item,N)).
```

As we can see in the above rule, we have limited the attacks on adequate mobs to only those that are located in the line of sight of the AI player. In the case when the monster *Mob* is close enough to the AI player *A*, predicate *attack(A,Mob)* is activated. Again, the action *collect_items* calls the *walk_to_location* action.

So, once we start an instance of command *do_quest(npc,ai_player,quest)* its result is a sequence of Prolog instances of basic actions. Specifically, after running the command *do_quest(npc,a,vincent)* which states that AI player *a* has got quest instructions of an NPC, a sequence of actions which AI player *a* needs to solve this quest is generated.

When the AI player needs a strategic plan, that is when the AI player needs to decide between two or more possible options, then it is necessary to introduce branching in the rule that solves the quest. A very simple example is the quest *Letter Quest* whose instructions, among other things, state that: *You have two options to get to the Graveyard. First, you can go south of Hurnscald and talk to Dyrin and pay 750 GP to be transported there. If you're feeling more adventurous, you can try to find your way there by foot.*

The part of the code in Prolog which allows such branching is as follows:

```
do_quest(NPC,A,letter_quest) :-
   ...
   location(place,graveyard,M,Xgyard,Ygyard),
  (ownership(A,money,Ma),Ma>=750,
   % Additional reasons to pay the fare
   location(npc,dyrin,M,XDyrin,YDyrin)
   ->
   walk_to_location(A,M,XDyrin,YDyrin),
   assert(plan(talk(A,dyrin,'Hi Dyrin! I pay you
750 GP for transport to Graveyard.'))),
   assert(plan(talk(dyrin,A,'OK. Let`s go!'))),
   retract(ownership(A,money,Ma)),
   NewMa is Ma-750,
   assert(ownership(A,money,NewMa));
   walk_to_location(A,M,Xgyard,Ygyard)),
   ...
```

First, the player must make a decision based on whether he even has the money to pay the fare. However, it is necessary to bring a deeper strategic decision about which way of traveling is better for him. Such a decision may depend on many factors. One such factor may be that the player can solve some waiting quest along the way as it travels to the destination (the travel route can in this case be important). It may be wise to go on foot because on the road he can collect some items that he lacks to start or finish some other quest.

The AI player can make more complex conditional decisions in the case of communication and interaction with other players or NPC's. In the case when an AI player is physically close enough to another player or NPC, the simplest form of communication may be formalized in the form of a predicate whose name and parameters describe the type of interaction. For example, if an AI player a_1 has a need to exchange or purchase some item, and an AI player a_2 is close enough, then such a need can be expressed by the predicate *trade(a1,a2,item)*. If the AI player a_2 is interacting with AI player a_1 then his response can be formalized in the form of predicate *cost* by which AI player a_2 informs AI player a_1 about the price of the requested item: *cost(a2,a1,item,price)*. Then follows a decision about the exchange, and depending on that decision, the exchange will be carried out or not.

Such decisions can become even more complex if multiple items have to be purchased or traded. In such cases, some form of automated negotiation algorithm has to be implemented similar to [5] which is out of the scope of this paper.

3.3 Solving quests

The AI player is going to solve a quest if he has met all the preconditions for that quest (for example the required level). If he gets a quest for which there are no preconditions to resolve, then it remains in his mind, and he is going to go and solve something else. Therefore, quests that AI players get, but have not started to resolve, are recorded in our system as instances of predicate *waiting_quest(npc,ai_player,quest)*.

In addition, there is a possibility of failure during the solving of the quest. An instance of the *failed_quest(npc,ai_player,quest)* predicate records a failure to resolve the quest.

We have implemented an algorithm to select a quest which the AI player will try to solve first. This algorithm is based on the so-called significance of the quest for the AI player. The significance of the quest for an AI player is given by the instance of predicate *quest_sign(ai_player,quest,sign)*. In order that our system knows which quest it should firstly begin to solve, we have introduced a *quest_no(ai_player,quest,no)* predicate, where the number *no* indicates the ordinal number (priority) of the quest. Priority is calculated based on the above properties of quests and is recalculated when necessary. Thus, quests belonging to an AI player *A* are sorted by priority, and sorting is conducted according to the following recursive Prolog rule:

```
sort_quests(A) :-
   waiting_quest(NPC1,A,Q1),
   waiting_quest(NPC2,A,Q2),
   quest_sign(A,Q1,QS1),
   quest_sign(A,Q2,QS2),
   quest_no(NPC1,A,Q1,QN1),
   quest_no(NPC2,A,Q2,QN2),
   QS1>QS2,
   QN1>QN2
   ->
   retract(quest_no(NPC1,A,Q1,QN1)),
   retract(quest_no(NPC2,A,Q2,QN2)),
   assert(quest_no(NPC1,A,Q1,QN2)),
   assert(quest_no(NPC2,A,Q2,QN1)),
   sort_quests(A);
   true.
```

Depending on the recalculation of priorities of quests, it is possible that the current quest is temporarily interrupted, another quest is conducted and afterwards the interrupted quest continues.

If the AI player has no waiting quests or cannot solve any waiting quest for some reason, then its goal is to explore the map hoping to find a new NPC that will give him a new quest. For this purpose, we have implemented a Prolog rule by which the AI player walks to a random location on the map it is currently located on:

```
random_walk(A) :-
   location(agent,A,M,Xa,Ya),
   randomX(M,RLX),
   randomY(M,RLY),
   location(npc,NPC,M,XNPC,YNPC),
   line_of_sight(A,NPC)
   ->
   walk_to_location(A,M,XNPC,YNPC);
   walk_to_location(A,M,RLX,RLY).
```

whereby the actions *randomX(M,RLX)* and *randomY(M,RLY)* yield random coordinates *(RLX,RLY)* on map *M*. If during a random walk, the AI player passes close to an NPC, then it will turn to him.

Using the above mentioned Prolog rules we reach the point that we must not "tell" the AI player which quest should be solved or what to do at all. He knows it according to the following Prolog rule:

```
do(A) :-
   sort_quests(A),
   quest_no(NPC,A,Q,1)
   ->
   do_quest(NPC,A,Q);
   random_walk(A).
```

The rule *do(A)* says that AI player *A* will first sort the quests based on importance. Then, if there is a quest on waiting with the highest priority (priority 1), the AI player should go to solve it. Otherwise, the AI player should perform a random walk in order to approach some NPC if it passes next to him close enough.

4 Changes in the environment

Since the environment of MMORPGs can be highly dynamic due to the large number of various interacting elements, it is necessary to deal with possible changes in the environment of the AI player. Therefore, AI players should be intelligent which means that they should be able to adapt to changes in the environment. We approach this by using a modified BDI control loop approach described below.

4.1 Formalization of the overall control loop of AI players

The overall control loop of our AI player is formalized mostly in accordance with the overall control structure of a practical reasoning agent given in [11]. Namely, in our system we are focused on reasoning directed towards actions and not only on theoretical reasoning which affects only the AI player's beliefs about the world. In our formalization the AI player continuously:

1. Observes the environment and changes in that environment. Changes can occur in:
   - the location of certain players, NPCs, monsters, items, etc.;
   - attributes of players;
   - attributes of quests (a quest is waiting to be performed, priority, done or failed);
   - etc.
2. Deliberates to decide what intention to achieve. Deliberation is done by firstly determining the available options and then by filtering. Options of an AI player are quests, or eventually random walks around the map, until he finds a new NPC that will give him a new quest. Thus, deliberation is actually deciding which quest the AI player will try to perform next. In order to select the most appealing between the competing options, an AI player uses a filtering mechanism. In our case, the filtering is actually the sorting of quests in order of priority.

3. Establishing a plan to achieve the selected intention, that is to solve the selected quest. Namely, when the AI player in step 2 chooses a quest to perform, it is necessary to determine the way in which the quest can be performed.
4. Executes the plan.

As in other similar planning systems, an important question occurs: How often (and when) should an AI player re-observe the environment and reconsider his intentions? Given that in The Mana World game these processes are not computationally cheap, it is necessary to reconsider as rarely as possible. On the other hand, during this time the environment changes, possibly rendering his newly formed intentions irrelevant. It is obvious that we have to find the optimum. Of course, the optimum is contained in this kind of system behavior: the AI player has to reconsider his intentions if, and only if, afterwards the intentions will have to be changed. Our plan is to reach this optimum experimentally by observing the behavior of our AI players in The Mana World test scenario.

4.2 Implementation of the overall control loop of AI players

Our planning system is separated from the physical realization of the AI players at the game level. Namely, at the game level, the AI player engine is implemented as a knowledge base into SPADE (Smart Python multi-Agent Development Environment) [7] and connected as a client to the The Mana World server, in order to provide an interface between the planning system and actual avatars in the game that shall play alongside human players. By starting the planning system, we get a sequence of actions that the AI player should take to achieve his objective.

We need to connect the physical implementation at the game level with a generated plan, and with a periodic control of the state of the environment. If there are changes in the environment (in a way that affects the existing plan) it is necessary to generate a new plan. The above is realized by the following general algorithm:

1. Load the physical state at the game level in the Prolog planning system;
2. In the Prolog system generate a (possible) AI player's plan to achieve the objective based on the current state of the environment (for example, to solve a certain quest or do a random walk);
3. Carry out only *N* first steps of the plan at the game level (we discussed what should be *N* in section 4.1);
4. Upgrade the state in the Prolog system to the newly arisen state on the game level;
5. Go back to step 2.

The above loop (algorithm) should proceed until the AI player completes his task, that is until he achieves his objective.

5 Conclusion & Future Work

In this paper, we have shown a possible implementation of artificial players in MMORPGs. MMORPG environments provide an interesting environment to test large-scale multi-agent system (LSMAS) design techniques, since for most of the methods used in such systems, adequate scenarios (quests) can be implemented to test them.

In our endeavors, we have chosen to use an open source MMORPG platform and implement AI players using Prolog. An automated planning system has been implemented based on a formalization of quests, as well as a practical reasoning control loop. We have discussed some of the practical issues which might arise in the testing of such AI players including the optimal time when to reconsider the environment, since this is a resource consuming action.

The developed approach provides an initial step towards using automated planning agents in complex social environments, since MMORPGs feature a complex world in which a multitude of humans interact through their avatars with the world and with one or more instances of artificial agents. MMORPGs's imagined worlds are in a way an approximation of the real world, whilst the complex interaction of social entities (both human and artificial) provide us with a glimpse into the processes an artificial agent might have to deal with in real world scenarios.

In our future studies we plan to conduct a large-scale experiment in a specially designed test scenario in which human players will play alongside AI players similar to the one described herein. We will gather statistical data about both human and artificial players with the intention of identifying the most valuable organizational methods used by players in order to optimize artificial players in the future.

6 Acknowledgments

This work has been supported in full by the Croatian Science Foundation under the project number 8537.

References

[1] Bramer, M. *Logic Programming with Prolog.* Springer-Verlag, London, UK, 2013.

[2] Bratman, M. E. *Intention, Plans, and Practical Reason*. Harvard University Press, Cambridge, UK, 1987.

[3] Fairclough, C; Cunningham, P. A multiplayer case based story engine. In *4th International Conference on Intelligent Games and Simulation*, pages 41-46, EUROSIS, 2003.

[4] Fairclough, C; Cunningham, P. An interactive story engine. In O'Neill, Sutcliffe, Ryan, Eaton and Griffith (Editors), *Proceedings of the 13th Irish International Conference on Artificial Intelligence and Cognitive Science, Lecture Notes in Artificial Intelligence*, 2464: 171-176, 2002.

[5] Faratin, P. *Negotiation among groups of autonomous computational agents*, Ph.D. thesis. Department of Electronic Engineering, Queen Mary and Westfield College, University of London, UK, 1998.

[6] Foundation for Intelligent Physical Agents. FIPA ACL Message Structure Specification, http://www.fipa.org/specs/fipa00061/SC00061G.html, accessed: June 30th 2015.

[7] Gregori, M. E; Cámara, J. P; Bada, G. A. A jabber-based multi-agent system platform. In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 1282-1284, ACM, New York, USA, 2006.

[8] The Mana World Contributors. The Mana World - A free open source 2D MMORPG in development, https://www.themanaworld.org, accessed: November 15th 2014.

[9] Wielemaker, J; Fruehwirth, T; De Koninck, L; Triska, M; Uneson, M. *SWI Prolog Reference Manual 7.1*. Books on Demand, 2014.

[10] Wikipedia Contributors. Massively multiplayer online role-playing game, http://en.wikipedia.org/wiki/Massively_multiplayer_online_role-playing_game, accessed: March 12th 2015.

[11] Wooldridge, M. *An Introduction to Multiagent Systems*. John Wiley & Sons, Chichester, UK, 2002.