Implementing Agent Roles in Massivley Multi-Player On-Line Role-Playing Games

Igor Tomičić, Bogdan Okreša Đurić, Markus Schatten
Artificial Intelligence Laboratory
Faculty of Organization and Informatics, University of Zagreb
Pavlinska 2, 42000 Varaždin, Croatia
`{dokresa, igor.tomicic, markus.schatten}@foi.hr`

Abstract. *Organizational roles in multiagent systems (MASs) are an important concept in modelling and implementation of complex interactive systems like massively multi-player on-line role-playing games (MMORPGs). The paper presents a novel approach to implementing such roles as sets of agent behaviours. An initial implementation in Smart Python Agent Development Environment (SPADE) is presented and applied to the implementation of artificial players that are able to enact various roles in The Mana World (TMW) an open source MMORPG. The expressivity and applicability of the presented approach is discussed and examples of usage are provided.*

Keywords. organizational roles, multi-agent systems, massively multi-player on-line role-playing games, artificial player implementation, agent behaviour

1 Introduction

Agent roles are a novel implementation technique in the development of artificial agents as well as MASs and more recently large-scale multiagent systems (LSMASs). Herein we will introduce the usage of this implementation technique in MMORPGs since they provide an important application domain for LSMASs (Schatten, Tomičić, et al., 2017).

The most often used tool for modelling artificial game characters like non-player character (NPC) include finite state machines (FSMs), where each state relates to the state of the character and defines its choice from available actions. For a more complex behaviour, FSMs is inherently insufficient, and thus more more flexible planning capabilities are required. There are other techniques that are used for modelling NPC and player behaviour like behavioural trees. Behavioural trees, as opposed to FSMs that model states, model actual behaviours and include sequences, probability and priority selectors as well as decorators (Yannakakis and Togelius, 2017). Another ad-hoc method is using a utility function similarly to fuzzy logic in which the decision about current action depends on a vector of variables which is evaluated against a fuzzy set function (Yannakakis and Togelius, 2017).

Stanford Research Institute Problem Solver (STRIPS) and goal oriented action planning – based for example on belief-desire-intention (BDI) – are used for planning techniques in games such as F.E.A.R. (Orkin, 2006), and lately in research of MMORPGs such as (Schatten, Đurić, et al., 2017).

When modelling more complex behaviours, all these methods can become quite bulky and cumbersome, since a complex character might encompass many different behaviours and even more states to act in a game environments such as MMORPGs. Herein we would like to outline agent roles as a possible solution for the implementation of such complex sets of behaviours as well as provide examples by using SPADE (Gregori et al., 2006).

The rest of this paper is organized as follows: firstly in section 2 we provide an overview of related research. Then in section 3 we show how agent roles can be implemented as sets of behaviours in SPADE and give an example role implementation in 4. In the end in section 5 we draw our conclusions and give guidelines for future research.

2 Related Work

Agent roles are a concept introduced from organization theory (see (Schatten, Ševa, et al., 2016) for an in-depth review) that allow for the implementation of agents that are able to enact a given role based on the role specification. There have been a few propositions to use such agent roles in games in related literature.

For example, in (Westra, F. Dignum, et al., 2008) the authors propose to view games explicitly as organizations designed and developed for achieving certain goals and requirements. Within the organization, individual agents use appropriate behaviours in order to reach external goals. The agent organization defines constraints and capabilities of organizational concepts: roles, tasks, interaction protocols, and parties. Within their paper, authors also propose a system for agents that are adapting to the user during gameplay, but considering that the game does not reach unwanted states, e.g. breaking the game's story line, which might happen should the game contain randomly adapting agents. The proposed agents are adapting individually, but the adaptation is guided using an agent organization. The OperA (M. Dignum, 2004) framework which is used within the paper, distinguishes organizational aspects from the individual ones, enabling the specification of organizational requirements and objectives, but at the same time allowing individuals to act according to their own demands and capabilities.

In (Westra, Van Hasselt, et al., 2008) the authors contemplate about adaptability within on-line adapting games. They highlight the limitations of using centralized control in dynamic adjustability, and to meet the rising complexity and the number of adaptable elements, authors suggest the use of an multi-agent approach, specifically for *"adapting serious games to the skill level of the trainee"*. Considering the research on "reconciling" the two main aspects in games – the flexibility of adaptability, and the control of a game's story line, authors use the idea of agent organizations as a means for mediation. The authors also use the OperA model for agent organizations, which *"enables the specification of organizational requirements and objectives, and at the same time allows participants to have the freedom to act according to their own capabilities and demands."* Figure 1 shows a simple example of an OperA interaction structure through scenes (depicted with squares), which can progress in parallel.

Agent organizations are also considered in (Huber and Hadley, 1997), where authors describe the architecture and performance of autonomous agents that are able to play Netrek, a complex, multi-player, multi-team, real-time internet game. Authors recognize the challenge in creating agents which are able to play the game, not only autonomously as an individual, but also by cooperating and coordinating with other members of the same team, and coordinating against members of the opposing team. There are several agent roles that authors are suggesting, each tied to specific actions they are required to perform. The roles are named "engage", "assault", "escort", "ogg", "protect", "get armies", and vary from the relatively simple behaviours (like attacking the closest opponents) to the more complex ones (bombing a planet, dropping armies on an opponents planet, etc.). In the conclusion, author argue that their agents were able to pursue complex goals "within a very complex, dynamic environment", with roles as the key enablers for task solving processes.

Merging agent technology with game technology (game engines) is not a trivial task, as argued by (F. Dignum et al., 2009); for this consolidation to work, agents should run in their separate threads and only loosely be coupled with the game engine. The synchronization between the two proved to be an important aspect within this context; the communication between agents, and between agents and the game world should be enabled, but also a means of translation between the agent and the gaming world, as the agents operate on a more abstract level. Authors argue that these challenges can be faced by using *"agent technology to its full extent"*. Also, authors argue that *"using a conceptual stance allows for connecting the agent concepts to the game concepts such that agent actions can be connected to actions that can be executed through the game engine and that agents can reason intelligently on the information available from the game engine."* These connections could be implemented through agent roles.

The method based on MAS architecture which would be used for defining a game is described in (Aranda et al., 2012), and the first phase authors are remarking is the role definition phase. The roles are herein specified with names and sets of attributes, with attributes defining properties bound to game-playing agents. Also, agents have some predefined set of roles in order to provide basic features that any massively multi-player on-line game (MMOG) should have by default, and these basic features can be extended by the game designers in the form of new features within the game. Hierarchy is used to relate roles, and agent organizations are used to represent agent behaviour related to forms of collaboration, such as player clans.

3 Implementing Agent Roles

We will use agent roles as sets of agent behaviours. According to (Marian et al., 2004) agent behaviours can be:

- **role factory** (a role added/deleted at runtime to be enacted/stopped by the agent);
- **itinerary** (allows mobile agents to travel across various locations and perform tasks);
- **periodic** (looped behavior possibly with a given period of time intervals between iterations);
- **observer** (an agents awaits an event in order to perform its actions);
- **listener** (a special type of observer in which an agent awaits a special message of some other agent);
- **client/server** (resembles the client-server model);
- **one-shot behavior or task** (represents a simple task or activity);
- **finite state machine** (resembles a finite state machine in which every node is an activity to be performed);
- **sequential behavior** (a sequence of other behaviors);
- **parallel** (various behaviors are run in parallel).

In addition to these types of behaviour, we would like to add an additional one, which from our practice, has shown to be very useful for various practical implementations where an agent is using a resource concurrently: **exclusive behaviour** – allows an agent to run the behaviour exclusively, by stopping all other

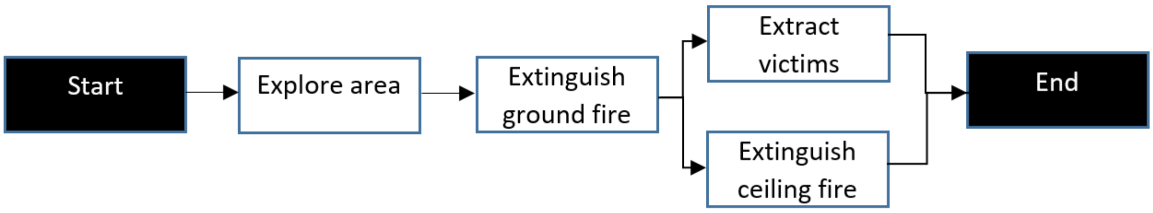

Figure 1: A simple example of an OperA interaction structure (Westra, Van Hasselt, et al., 2008)

exclusive behaviours. Such a behaviour, usually implemented with locking or semaphores, allows the agent to use some kind of resource (like a database, a file, a network socket etc.) that isn't thread-safe, by putting all other concurrent behaviours, that might use the same resource, at hold.

As per definition, we can define an agent's role as a set of behaviours $\mathbb{R} = \{b_1, b_2, \ldots, b_n\}$ which are added to the agents behaviours at run-time at the exact moment the agents acquires the role and starts enacting the defined behaviours. In the same manner, these behaviours are removed in the moment the agent stops enacting the role.

Translated into Python and more precisely SPADE the implementation of an agents role is straightforward as shown in listing 1.

Listing 1: An agent role class in Python/SPADE

```
class Role:
  ''' An (organizational) role is
      basically a set of behaviours.
    The behaviours should be a list of
        elements having the form: (
       behaviour instance, template
       instance ).
    The second item only applies to
        EventBehaviours (template),
        else it is None'''
  def __init__( self, behaviours=[] )
      :
    self.behaviours = behaviours
```

Thus, an instance of a role is independent of actual agents. It is just a container to hold various behaviour instances possibly with templates used for event behaviours (which are the implementation of observer behaviours for messages in SPADE).

In addition to a role class, we define two behaviours which implement the role factory behaviour which allows agents to acquire or stop enacting a role. The first behaviour allows an agent to acquire a role and is shown in listing 2.

Listing 2: An agent behavior class for adding a role implemented in Python/SPADE

```
class AddRole( spade.Behaviour.
    OneShotBehaviour ):
  """Behaviour to add a Role to the
      Agent. The Agent will acquire
      behaviours of the given Role."""
  def __init__( self, role, *args, **
      kwargs ):
    spade.Behaviour.OneShotBehaviour.
        __init__( self, *args, **kwargs
        )
    self.role = role

  def _process( self ):
    if not hasattr( self.myAgent, "
        roles" ):
      self.myAgent.roles = []
    self.myAgent.roles.append( self.
        role )
    for behaviour, template in self.
        role.behaviours:
      self.myAgent.addBehaviour(
          behaviour, template )
```

Basically, the behaviour appends the role to the list of roles the agent already has (or initializes the list of roles to an empty list first if the agent has no roles), and then adds all behaviours from the role to the agent to start using them.

The second behaviour removes a role from the agent, and is shown in listing 3.

Listing 3: An agent behavior class for deleting a role implemented in Python/SPADE

```
class DeleteRole( spade.Behaviour.
    OneShotBehaviour ):
  """Delete a role of the Agent. The
      Agent will lose all behaviours
      of the given Role."""
  def __init__( self, role, *args, **
      kwargs ):
    spade.Behaviour.OneShotBehaviour.
        __init__( self, *args, **kwargs
        )
    self.role = role

  def _process( self ):
    if not self.role in self.myAgent.
        roles:
      raise ValueError, "The␣agent␣isn'
          t␣playing␣the␣role␣to␣be␣
        deleted!"
    for behaviour, _t in self.role.
        behaviours:
      self.myAgent.removeBehaviour(
        behaviour )
    self.myAgent.roles.remove( self.
        role )
```

The behaviour basically checks if the agent has the role to be removed, and if yes, removes all behaviours of the role, and then removes the actual role.

4 Example

In order to provide additional clarification we provide an example of how such an implementation of agent roles could be used and has been used in the Large-Scale Multi-Agent Modelling of Massively Multi-Player On-Line Role-Playing Games (ModelMMORPG) project. In listing 4 a *Leader* role is implemented.[1]

Listing 4: An example Leader role implemented in Python/SPADE

```
class Leader( Role ):
  class LeaderBehaviour( spade.
      Behaviour.OneShotBehaviour ):
    def _process( self ):
      (...)

  class PartyStats( spade.Behaviour.
      OneShotBehaviour ):
    def _process( self ):
      (...)

  class InvitePlayers( spade.
      Behaviour.PeriodicBehaviour ):
    def _onTick( self ):
      (...)

  def __init__( self ):
    lb = self.LeaderBehaviour()
    ipb = self.InvitePlayers( 30 )
    psb = self.PartyStats()
    self.behaviours = [ ( psb, None ),
        ( lb, None ), ( ipb, None ) ]
```

The implemented role has been used for the implementation of artificial intelligence (AI) players inside the open source MMORPG TMW to create players which are able to create parties of players, invite new players and provide party statistics to party members.

The behaviours of the role have been implemented as nested classes inside the role class to be grouped together in one logical sequence of code, but could have been used separately, for example if the same behaviour is part of multiple roles. During the initialization of the role, instances of the behaviours are created and added to the list of behaviours.

To add the role to some agent (for example a TMW-player agent) one could use the code in listing 5.

Listing 5: Adding a role to an agent in Python/SPADE

```
role = Leader()
a = ManaWorldPlayer( SERVER, PORT,
    USERNAME, PASSWORD, CHARACTER, '%
    s@127.0.0.1' % USERNAME,
    SERVER_PASSWORD )
a.addBehaviour( AddRole( role ) )
a.start()
```

With these simple few lines of code, we were able to implement a very complex agent that acquires a role and starts enacting the defined behaviours.

5 Conclusion

Through the course of this paper we have shown a way of implementing agent roles in the context of the MMORPG domain, using the implementation possibilities provided by SPADE multi-agent development platform.

Roles that are defined as a part of this research represent normative concepts that denote sets of behaviors that can be played or performed by agents when enacting a specific role. Role enactment makes it possible for in-game characters to change their available actions dynamically, based on the role they enact. Having behaviour and actions defined in such a way makes the system's agents capable of adapting to various states of the system.

In this way very complex sets of behaviours can be implemented in a clear way with a higher level of abstraction then using common FSMs or behavioural tree methods. By using only a few lines of Python/SPADE code, we were able to define agent roles as sets of behaviours that can then be enacted by particular agents.

Our future research is aimed towards implementing more complex organizational features into MMORPGs by using agent technologies especially in regard of structural features.

Acknowledgments

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

[1]For the sake of readability we have removed the details of each behaviour implementation, but the interested reader can refer to https://github.com/tomicic/ModelMMORPG/blob/master/TMWhlinterface.py for the whole implementation.

References

Aranda, G., Trescak, T., Esteva, M., Rodriguez, I., & Carrascosa, C. (2012). Massively multiplayer online games developed with agents. In *Transactions on edutainment vii* (pp. 129–138). Springer.

Dignum, F., Westra, J., van Doesburg, W. A., & Harbers, M. (2009). Games and agents: Designing intelligent gameplay. *International Journal of Computer Games Technology*, *2009*.

Dignum, M. (2004). *A model for organizational interaction: Based on agents, founded in logic*. SIKS.

Gregori, M. E., Cámara, J. P., & Bada, G. A. (2006). A jabber-based multi-agent system platform. In *Proceedings of the fifth international joint conference on autonomous agents and multiagent systems* (pp. 1282–1284). ACM.

Huber, M. J. & Hadley, T. (1997). Multiple roles, multiple teams, dynamic environment: Autonomous netrek agents. In *Proceedings of the first international conference on autonomous agents* (pp. 332–339). ACM.

Marian, T., Dumitriu, B., Dinsoreanu, M., & Salomie, I. (2004). A framework of reusable structures for mobile agent development. In *Proceedings of ieee international conference on intelligent engineering systems (ines2004)* (pp. 279–284). IEEE.

Orkin, J. (2006). Three states and a plan: The ai of fear. In *Game developers conference* (Vol. 2006, p. 4).

Schatten, M., Đurić, B. O., Tomičić, I., & Ivković, N. (2017). Agents as bots–an initial attempt towards model-driven mmorpg gameplay. In *International conference on practical applications of agents and multi-agent systems* (pp. 246–258). Springer.

Schatten, M., Ševa, J., & Tomičić, I. (2016). A roadmap for scalable agent organizations in the internet of everything. *Journal of Systems and Software*, *115*, 31–41.

Schatten, M., Tomičić, I., & Đurić, B. O. (2017). A review on application domains of large-scale multiagent systems. In *Central european conference on information and intelligent systems*.

Westra, J., Dignum, F., & Dignum, V. (2008). Modeling agent adaptation in games. In *Bnaic 2008 belgian-dutch conference on artificial intelligence* (p. 381).

Westra, J., Van Hasselt, H., Dignum, V., & Dignum, F. (2008). On-line adapting games using agent organizations. In *Computational intelligence and games, 2008. cig'08. ieee symposium on* (pp. 243–250). IEEE.

Yannakakis, G. N. & Togelius, J. (2017). *Artificial intelligence and games*. Springer.