A Periphery of Pogamut: From Bots to Agents and Back Again

Jakub Gemrot, Cyril Brom, and Tomáš Plch

Charles University in Prague, Faculty of Mathematics and Physics
Malostranské nám. 2/25, Prague, Czech Republic

Abstract. Despite virtual characters from 3D videogames – also called bots – seem to be close relatives of intelligent software agents, the mechanisms of agent reasoning are only rarely applied in videogames. Why is this? One possible reason is the incompatibility between representations used by agent decision making systems (DMS) and videogame worlds, as well as different handling of these representations. In recent years, we developed Pogamut, which is a toolkit for coupling videogame worlds with DMSs originating within the agent oriented research as well as other disciplines, allowing for controlling in-game characters by these DMSs. To this end, Pogamut features an interface bi-directionally bridging the "representational gap" between a game world and an external DMS. This paper conceptualises functionality of this interface based on our experience with connecting Pogamut to various game worlds, most notably Unreal Tournament 2004. We present a general abstract framework, which verbalises requirements an agent researcher must fulfil in order to employ his/her reasoning mechanism for controlling in-game virtual characters. This paper also reviews Pogamut, which the researcher can utilise.

Keywords: videogames, agents, action selection, Unreal Tournament.

1 Introduction

After years of stagnation, the field of artificial intelligence (AI) for videogames seems to have caught second wind [1]. Results achieved in disciplines such as planning [2], evolutionary computation [3], or stochastic modelling [4] inspire new solutions and approaches to known problems. Can knowledge accumulated by the multi-agent systems (MAS) community during the last decade be, to some extent, also employed in the context of 3D videogames? This idea stems from the fact that it may seem, superficially, that fields of gaming AI and MAS study entities of the same kind.

Entities studied by the MAS community are often called *intelligent software agents* [5]. Creatures employed in the gaming AI field are typically called *bots*, *non-player characters* (NPCs), *virtual characters* or *virtual agents*. To avoid ambiguity, we will strictly use the term *agents* for the former entities while *bots* for the latter.

The broad field of MAS studies includes many subfields, such as multi-agent communication, cooperation and negotiation, learning, and agent reasoning. It can be contemplated about how the knowledge gained in each of these subfields can be employed in the context of videogames. This paper is concerned with reasoning of a single or few agents, leaving the other subfields for future work.

The field of videogames is as diverse as the MAS field. In this paper, when speaking about videogames, we restrict ourselves only to 3D games modelling individual persons such as first-person shooters (FPS) or role-playing games (RPG). For the purposes of this paper, we also include 3D virtual reality simulations, such as crowd simulations or virtual storytelling systems, into the definition of a “3D game”. That is, virtual characters inhabiting these simulations and carrying out multiple goals, often called intelligent virtual agents, will be considered as bots here. However we will not use the term bots for conversational characters, unless they are fully embodied within a virtual world and have multiple goals besides chatting (e.g., a sport game commentator is not considered as a bot here unless he can also shoot, run, etc.). Similarly, we exclude from the definition of a “3D game” games without individual persons, e.g., statistical strategies and logical games, even though decision processes in these games can be conceived as agents by some (e.g., [6]). We also exclude games simulating only or predominantly vehicles, such as flight simulators and racing games, even though when these vehicles are controlled by a computer, they may be called bots by some.

Now, we can rephrase and refine the question this paper aims at analysing: *Can knowledge accumulated in the MAS field concerned with agent reasoning be used for reasoning of individual bots or a couple of bots?* That means, can developers of bots employ MAS knowledge representations, reasoning algorithms, or goal-oriented software architectures such as Jason [7], Jack [8] or Jadex [9]? How to achieve this? What are the obstacles, what are the drawbacks and benefits?

This paper does not provide conclusive answers. Instead, it proposes the way towards practical utilisation of MAS knowledge in the videogame domain, which leads through connecting these architectures to game engines, enabling empirical experiments. The paper explains what exactly it means to establish such connection and offers a robust toolkit we have developed to help in this work.

An agent decision making system (DMS) can be connected to a game engine either internally or externally (see also [10] on this point). *Internal* connection means integrating a DMS into a game engine when the engine’s source code is available. However, the source code of games is available only rarely and typically for ten years old games (with some exceptions). Fortunately, several modern games allow for information exchange between their worlds and an external system, enabling a researcher to couple his/her architecture with a game *externally*, in most cases on a client-server basis. However, the researcher should be aware of two stumbling blocks: a) the way how bots are controlled internally (i.e., from the game within) may differ from the way how the bots can be controlled externally, b) bot DMSs use different input/output information than typical agent DMSs, no matter whether an agent DMS is connected internally or externally (mind the distinction between a bot DMS and an agent DMS[1]). While Point (a) is merely technical, Point (b) presents a deep conceptual difference. To understand this difference, Section 2 of this paper will explain the main distinctions between agents and bots in detail.

[1] Whereas a bot DMS is a native controlling mechanism hardwired within the game engine, an agent DMS is a stand-alone application its creator is claiming it is MAS principles compliant.

When aiming at connecting an agent DMS to a game, it is an advantage to know about what can be expected from game engines; which information is available and which it is not, and how to acquire the desired information. To enable the reader to understand these points, Section 3 formally conceptualises game engines, providing a model of a bot DMS, game engine and information it holds. Based on this formalisation, Section 4 discusses the issue of connecting an agent DMS.

Sections 3 and 4 did not appear from thin air. On the contrary, we have been working for several years on Pogamut, which is a robust toolkit for coupling videogame worlds with external DMSs [11]. This toolkit has been widely used, both by us [e.g., 12, 13] as well as others for the purpose of prototyping gaming algorithms [14] or for prototyping BDI-based agents acting in real-time, dynamic and complex environments [15]. The formalisations are derived based on our experience we gained during our work on Pogamut. Section 5 reviews Pogamut and presents a particular instantiation of the formal model, a multi-layered AgentSpeak(L)-based DMS [16] coupled to the Unreal Tournament 2004 game (UT 2004) [17]. The most important point is that Pogamut can be utilised for connecting other agent DMSs to videogames in an "out of the box" fashion (see also [15] on this point). Section 6 documents that Pogamut can be used not only in the context of the UT 2004 game: it reviews our work in progress on extending Pogamut to operate with Virtual Battle Space 2 [18], which is a multi-agent military simulator, as well as connecting Pogamut to StarCraft [19] and Defcon [20], strategy games.

2 Bots Are Not Agents

The border between bots (as defined above) and agents (as understood by MAS community) is not clear-cut, but there are several traits that help understand the difference. Arguably, the two most important traits are believability and embodiment. Firstly, bots should be *believable*, which is the ability to convey the *illusion* of reality [21] by whatever means necessary. In this aspect, bots and agents differ; while the ideal of agents is strong autonomy, which does not necessarily imply believability in terms of [21], believable bots need not be strongly autonomous. Secondly, bots are embodied; they have virtual bodies subject to constraints of their 3D virtual worlds[2].

Because of their embodiment, and this is crucial, bots require different information inputs about their surroundings than typical agent-oriented DMSs work with. Because of real-time constraints, bots have to cope with ever-changing virtual world rapidly, which requires acquiring and processing of external information in a timely fashion.[3] We will now elaborate on these two points.

One useful way of categorising incoming information is based on the *level of abstraction*. It is useful when the information about some aspects of a bot's surrounding

[2] Recall that we call intelligent virtual agents bots here.

[3] Note that the meaning of the word "rapid" differs from the MAS community's use: in gaming industry, constant algorithms tend to be considered as fast while polynomial are considered to be slow in general, let alone exponential (of course, this depends on a particular situation). For instance, traversing expression trees of dynamic lengths built out of domain specific language checking for event triggers, which is linear, is found to be slow [22]. That work suggests using only expression trees that fit into a rather small pre-allocated array.

is provided in an abstract manner while other aspects are presented in a low level way, i.e., more akin to inputs from robotic sensors. Thus, at the same time, a bot can get information like “a projectile is coming from 170°”, “there is a wall or something 2.34 meters at 54°”, “this is a position from which you can shoot”. The important point is that the level of abstraction is not determined by an ideal of psychological plausibility or the logical coherence of the DMS, but by technical rationale, including the real-time constraints and the believability requirement.

Another useful categorisation divides information into *static* and *dynamic* [10]. Static information is bound to properties of the virtual environment that do not change in the course of the simulation. For instance, whether a place is suitable as a cover in respect to a sentry gun fixed inside a bunker depends on the outlook of the sentry gun and the landscape. In an environment that does not change its topology, this information is known during the design time, i.e., the information has a static context, and can be precompiled. On the other hand, the information where defenders of a bunker have the weakest spot depends on their current positions, their patrol behaviour etc., i.e., it has a dynamic context implying the necessity to compute or acquire it during runtime. This distinction is crucial both technically and conceptually.

Now, to be able to contemplate on how an agent DMS coupled with a game engine can access information, we need to know how data are provided by engines. In our experience, game engines tend to export only information that is available via regular bots’ access mechanisms (because providing different access mechanisms would present only additional unnecessary development). Thus, we have to take a look on how bots sense their worlds.

Due to real-time constraints, game developers must balance the necessity of bots having to perform *active* sensor querying against bots’ automatic, i.e., *passive*, event notifications whenever a respective event occurs in the game engine. Note that the active vs. passive distinction may not mirror the static vs. dynamic dichotomy. The rationale behind the active vs. passive mechanisms is to automatically notify bots about important events that the bots would check anyway, such as “bot has been killed” or “bot has hit a wall” – this is the passive sensing. On the other hand, there are many events that bots need to know only from time to time. It is more efficient when bots actively request information about these events only when the information is really needed.

What does it imply for an external agent DMS? Generally, there are three ways how information can be obtained. 1) The information that is passively sensed by bots is also automatically exported by the game engine (*push* strategy). 2) The information that can be actively requested by bots can also be requested from the engine (*pull* strategy). 3) The information that is not provided by the game engine itself but it may be inferred from existing information (*inference* strategy).

Here is a distinction between agents and bots concerning these information access strategies: While bot DMSs tend to access information by several special purpose mechanisms, capitalising on both push and pull strategies, agent DMSs tend to use one generic mechanism only. This fact alone poses technical troubles for some agent DMSs. For instance, many goal-oriented agent architectures, such as AgentSpeak(L) [23] derivates, require for underlying agent worlds to keep them automatically informed about events, which corresponds to the push strategy. For these architectures, the lack of ability to cope with pull and inference strategies must be compensated.

Having the notion of key differences between bots and agents, the paper continues with discussing game engines in general. This will help the reader to better understand the prerequisites of connecting an agent DMS to a game engine.

3 Game Engines

This section presents a generic framework for understanding game engines (GE). The section starts in an informal tone and continues with a formal definition of a GE, functioning of a bot DMS, and relations between various data kept by the engine.

3.1 GEs Informally

"A game engine is a software system designed for the creation and development of video games. Game engines provide a suite of visual development tools in addition to reusable software components. These tools are generally provided in an integrated development environment to enable simplified, rapid development of games in a data-driven manner." [24, see also 25] A GE itself is not a game, rather a middleware used by the game developers, who may arbitrarily extend it to suit the game's needs. As a middleware, it usually empowers the developers with the ability to script game rules using interpreted languages such as Lua [26], Python [27] or a proprietary language such as UnrealScript [28].

The high level task of a GE is to simulate a game's virtual world in (nearly) real-time providing smooth visualisation to players while reflecting their actions. One of the challenges GEs are facing is to arbitrate available CPU and GPU power between:

1) game visualization, e.g., performing animations, managing polygons, textures and shader programs in the graphic card's memory, etc.;
2) (simplified) physics simulation, e.g., computing the trajectories of moving objects, performing collisions and deformations, etc.;
3) game mechanics, e.g., triggering game events at correct time, executing scripted situations, etc.;
4) artificial intelligence, e.g., providing believable behaviour for interactive game objects such as bots.

Game engines tend to prioritize these issues in the given order. Smooth visualization is preferred over AI computations, which do not affect the simulated world every frame. Thus AI computations are interleaved with scene rendering, physics engine etc., and in most cases, are not given extensive computational facilities.

In the previous section, we discussed how information can traverse from a GE to a bot. Now, we are going to discuss how these data are stored within the GE and how they are managed.

GEs as managers of facts. GEs can be conceptualised as managers of game facts (grounded formulae of first order logic), which are true in the certain point of time. GEs represent them as data structures of a native programming language. GEs can also be seen as rule engines that transform given facts as the game proceeds. A GE maintains a virtual clock, which measures time in ticks. Each tick, the GE transforms game facts according to game rules that are encoded inside the GE or written outside by game developers in a language the GE can interpret. We will refer to this transformation as *TICK* function.

The following (non-exhaustive) list presents examples of such game rules:

1) Physics. Change locations of objects, the speed of which is greater than 0, as if 0.05 seconds has passed, and compute collisions along the way.

2) Game mechanics. If a rocket explodes, compute damage and process the damage (lower health) to all bots and players in the radius of 500 m.

3) Trigger. When a player steps on a jump pad, apply force of 1000 N to his avatar in the direction perpendicular to the jump pad.

4) Bot API: If a bot issues a move command, change the bot's speed to 20 km/h.

Section 2 has divided a bot's information into *dynamic* and *static*. This dichotomy comes from the GE itself as it is the GE which defines, which information can change in the course of the simulation and which can not. Static information (facts) contain the GE's configuration, together with the underlying geometry of the land, game triggers, events or programs in a language that GE can interpret, etc. Examples of dynamic information (facts) includes the current number of bots and items in the game along with their state, i.e. position, current speed, animation in progress, etc.

The distinction between static and dynamic facts is technical as well as conceptual. Technically, a DMS must handle dynamic facts differently, i.e., it must be ready to handle their change and offer the developer a way to act upon such changes. Conceptually, the more the game world is dynamic, the more sophisticated algorithms must be employed. For example, there are games where objects forming the borders of the virtual world are destructible by bots and players, e.g., it is allowed to blast a hole into a building's wall, which makes the navigation mesh a dynamic fact. Thus a dynamic path finding algorithm, e.g., D* [29], must be used instead of a classical one.

Now, let us proceed to the question how these facts are stored within a GE.

Game facts division. Game facts that are true about the game in a given time can be categorised into three groups. We will refer to these groups as *game facts classes*.

1) A GE usually provides an API to access all game facts it stores within its internal data structures, such as level geometry or a bot's level of health. Accessing these facts is computationally efficient. We will refer to these facts as *Class 1 facts*.

2) Additionally, there are facts that can be computed on request by invoking a method of an API that uses *Class 1 facts* to infer new facts. Such facts are, e.g., ray cast results or a path to a distant location. We will refer to these facts as *Class 2 facts*.

3) Finally, there are facts that can not be obtained through the GE's API but that are algorithmically inferable, such as the shortest path a bot should follow to collect all items in the world (leads to the well-known Travelling Salesman Problem). We will call these facts *Class 3 facts*.

Sending requests to GEs. So far, only the information flow from a GE to a bot has been discussed. Fortunately, the other way is easier. A GE usually offers a set of methods that can be invoked by bots, e.g., `GoTo(Location)`, `Shoot(Actor)`, or `GetPath(Location)`. These methods are of two types: 1) actions the bot's body should carry out in the game world (`GoTo`, `Shoot`, etc.), and 2) computation requests that induce facts not present implicitly in GE's data structures, i.e. Class 2 facts (e.g. `GetPath`). Both commands and computation requests may last (and usually last) several *TICK*s before they are finished. The number of *TICK*s depends on the request's complexity, e.g., say "Hello!" command will take less time to perform than `GoTo(Hospital)`. Similarly, a ray cast request will finish well before the A* algorithm finds out that a requested path does not exist.

Two major issues linked to the requests are to be handled by an agent DMS. Firstly, the DMS should be able to handle the lag between a computation request and its returned result. Especially the path requests should be made in advance. Secondly, GEs may not report success or failure of an action. In these cases, a DMS must actively watch over a bot's related facts to infer whether the action is being carried out as expected (e.g., has the bot just hit an obstacle?). If a deviation from an expected result is detected, actions to compensate should take place. Such success/failure reporting is required, e.g., by BDI systems. Whenever the bot has an intention to go to the hospital, it needs to know whether the `GoTo(Hospital)` command has been executed successfully or failed in order to maintain, delete or re-plan the intention.

3.2 GE (More) Formally

This part of the paper summarises formally description of GEs in order to facilitate thinking about connecting agent DMSs to GEs. Therefore, the definition intentionally emphasises management of game facts and their accessibility, but not the issue of bots' action selection and carrying out action commands by GEs. Note also that GEs may differ in important implementation details; what we present here is only our view of GEs. We are not aware of any widely accepted formal definition of a GE.

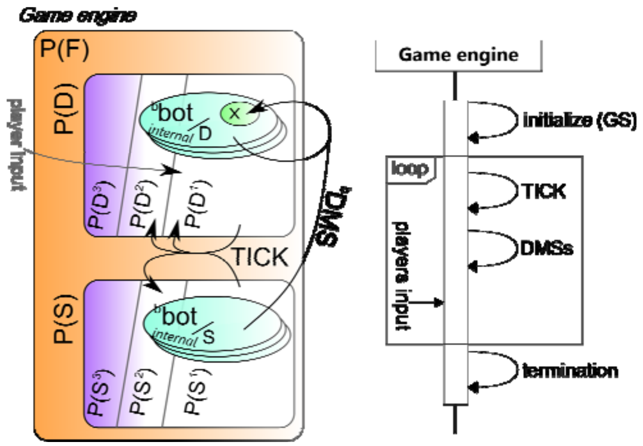

Fig. 1. Visualization of TICK and DMS functions together with the sequence diagram of subsequent calls. The mark "X" (on the left figure) denotes a bot's mental states and requests, i.e., $P(^bVM) \times P(^bR)$. The bots in the figure are marked as "internal", i.e., they are native to the GE and controlled by bot DMSs. This labelling will become important later on in Figure 2.

Definition 1. ***(Game engine, managed facts)*** *Game engine $GE = (F, I, P, B, TICK, GS, T)$ is characterized by:*

- ***Facts*** *F. This is the set of all ground formulae that can be possibly true about the game during some moments.*
  - *The set F consists of three classes: $F = F^1 \cup F^2 \cup F^3$.*
    - ***Class 1*** *game facts $F^1 \subseteq F$ is a set of all ground formulae that (a) can be possibly true about the game and (b) they are stored in appropriate data structures* iff *they are true.*
    - ***Class 2*** *game facts $F^2 \subseteq F$ is a set of all ground formulae that can be inferred by any inference function $i \in I$ (see below), i.e., $F^2 = \bigcup_{i \in I} range(i)$.*[4]
    - ***Class 3*** *game facts $F^3 \subseteq F$ is a set of all ground formulae that can be possibly true about the game, but they are never stored within a GE's data structure and they are not obtainable through any inference function $i \in I$.*
  - *Additionally, F can be divided amongst* **dynamic** *facts $D = D^1 \cup D^2 \cup D^3 \subseteq F$ and* **static** *facts $S = S^1 \cup S^2 \cup S^3 \subseteq F$, where $F^i = D^i \cup S^i$, $D^i \cap S^i = \emptyset$. Dynamic facts can be changed between TICKs (see below), whereas static facts remain the same throughout the whole simulation.*
- ***Inference functions*** *I. This is a set of all functions that the GE can use to infer facts of Class 2: $i \in I : P(F^1) \times P(F^2) \to P(F^2)$.*
- ***Embedded bots*** *B. This is a set of all possible bots in the game. See Def. 2.*
- ***Embedded players' abstractions*** *P. This is a set of all possible manifestations of players in the game. See below.*
- *Function* ***TICK****. This function $TICK : P(S^1) \times P(D^1) \to P(D^1) \times P(D^2) \times P(S^2)$ is used to advance the game situation.*
  - *Concrete facts that can be changed by TICK function, i.e., $(def(TICK) \cup range(TICK)) \setminus S^1$ are called* ***managed facts****.*
- ***Game settings*** *GS. $GS = (\omega, \beta, \pi)$ is characterized by*
  - *initial game situation $\omega \in P(F^1)$,*
  - *list of bots $\beta \subseteq B$ and players $\pi \subseteq P$ connected to the game.*
- ***Terminal states*** *T. This is a list of game situations that terminates the GE, $T \subseteq P(F^1 \cup F^2)$.*

[4] In fact, a GE may cache results of inference functions making these facts Class 1 facts. However, these functions often compute dynamic facts, which are being quickly invalidated as the simulation advance forward, therefore we will omit caching.

Definition 2. ***(Bot, bot's facts)*** *Bot* $b \in B = (VB, P, VM, R, DMS)$ *is characterized by:*

- *the set of* ***virtual body facts*** $^{b}VB \subseteq F^{1}$,
- *the set of facts that can be possibly* ***perceived*** $^{b}P \subseteq F^{1} \cup F^{2}$,
- *the set of* ***mental facts*** $^{b}VM \subseteq F^{1}$,
- *the set of* ***requests*** *the bot can possible make* $^{b}R \subseteq D^{1}$,
- ***decision making system*** *function* $^{b}DMS: P(^{b}VB) \times P(^{b}P) \times P(^{b}VM) \times P(^{b}R) \to P(^{b}VM) \times P(^{b}R)$,
- *the set of all* ***bots' facts*** $^{b}bot = P(^{b}VB) \cup P(^{b}P) \cup P(^{b}VM) \cup P(^{b}R)$.

Abstractions of players from Def. 1 can be defined similarly to the bot's definition (Def. 2) except the *DMS* function is unknown and brings uncertainty to the game.

Def. 1 and 2 result from the informal discussion from Section 2 and 3.1. The last thing to explain is how the function *TICK* and *DMS* work.

$TICK: P(S^{1}) \times P(D^{1}) \to P(D^{1}) \times P(D^{2}) \times P(S^{2})$ is being used to compute a next game state ($P(D^{1})$ from range(*TICK*)) as well as requests made by bots ($P(S^{1}) \times P(D^{1})$). Concerning dom(*TICK*), inside $P(S^{1}), P(D^{1})$, there lies currently true facts about all bots' virtual bodies, true facts bots are currently informed about, as well as their mental states and their active requests. The *TICK* function applies game rules to advance the game's progress a small fraction of time forward, i.e., it replaces the current $t \in P(D^{1})$ of true facts with a new set. Additionally, it handles the bots' requests by removing them and providing DMSs with $P(S^{2}), P(D^{2})$ facts that are accessible for a short period of time (they are invalidated in next few *TICKs*).

$DMS: P(^{b}VB) \times P(^{b}P) \times P(^{b}VM) \times P(^{b}R) \to P(^{b}VM) \times P(^{b}R)$ assesses the current state of the virtual body $P(^{b}VB)$, facts that are known to the bot $P(^{b}P)$, the current DMS state $P(^{b}VM)$ and unfinished bot's requests $P(^{b}R)$, and produces zero or more requests while altering its own state. Note that facts $P(^{b}P)$ and $P(^{b}VB)$ are computed by the *TICK* function. Conceptually, the *TICK* function also computes the *DMS* functions; however, for intelligibility, it is better to conceive these two functions as separate.

The *DMS* function model suggests what needs to be done in order to connect an external agent DMS to a GE. This issue will be further elaborated in the next section.

4 Connecting an Agent DMS to a GE

Arguably, the final goal of the whole endeavour is that game industry starts using some ideas stemming from the subfield of agent reasoning in videogames or even employs a whole agent DMS for the purpose of controlling in-game bots. As already said, an agent DMS can be coupled with a game either internally or externally. In a final application, it is likely that bots will be controlled internally – this is more efficient than external connection both in terms of memory and processor requirements (see [10] for a different view). However, in our opinion, before this can happen, it is necessary to connect several agent DMSs to a single GE and evaluate them. These DMSs should be compared with existing AI techniques currently used by the game industry, such as finite state machines [30], behavioural trees [31] and simple planning [22]. They should be compared along several lines, most notably in terms of computationally efficiency, improvement of bots' cognitive abilities, and design time. For instance, before these DMSs can start to compete with industry solutions, they should accommodate multiple bots at the same time and they should be intelligible for game designers who may not have strong AI knowledge.

All of this means that for the time being, the goal is not to employ an agent DMS within a videogame to be marketed, but to implement several prototypes. Should they employ the external or internal coupling? This section evaluates these two approaches and argues that the external one is better for prototyping purposes. It also makes it explicit what the external coupling means in terms of the formalism from Sec. 3.

Be it internally or externally, both approaches require researchers to write additional code binding a DMS and a GE together. It would be an advantage if the researchers can use a middleware facilitating this infrastructure work. It would be even better, for the purposes of evaluation, if all of these researchers use a common middleware. Section 5 proposes that Pogamut can be used for this purpose.

4.1 Internal or External Coupling?

Integrating a DMS into a game engine. Integration of an existing DMS into a GE means to re-implement the existing solution inside the framework (code base) of the engine.

The advantages are as follows:

- The integration may utilise all features of the GE.
- The implementation may blend with the original code of GE resulting in optimal performance, which suits the needs of the game industry.

The disadvantages are as follows:

- The DMS must be re-implemented in the native programming language of the GE.
- The code of the engine must be opened.
- The solution will be GE dependent and may not be reusable with different engines.
- The implementation could not be developed over a common architecture that adapts the GE to the DMS, which will make empirical comparison of different DMSs troublesome.

Note that in [10], this is called a server-side approach. That work also claims that the DMS must be completely synchronised with the GE, being integrated in the default game loop. However, this is not the case. Even internally coupled DMS can perform decisions in several time steps if it is able to interrupt and resume its computations.

Connecting DMS to GE externally. The other way is to utilise existing DMS implementations and connect them to a GE externally via, for example, TCP/IP.

The advantages are as follows:

- The translation of game facts and requests on both the GE's and the DMS's sides can be reused.
- The creation of a general layer between the GE and the DMS can result in a common platform for empirical comparison of different DMSs.
- The general layer could be extended to comfort more GEs.
- The DMS and the GE may run on different computers, allowing for distributed simulations.
- The DMS can be implemented in a language favoured by the developer.

The disadvantages are as follows:

- Game facts and requests have to be exported and translated between the GE and the DMS, which results in worsened performance.
- The bot's reactions to events take longer because there is a round trip time between the DMS and GE.

Our experience gained during the implementation of Pogamut has shown that the disadvantages of the external approach are not so sever or troublesome. The translation of facts and requests between the GE and the DMS does not take such extensive time to harm the bot's reactive capabilities (i.e., Pogamut can easily communicate synchronously with several bots on 4Hz, while asynchronous messages are handled in milliseconds). Thus, we will continue only by assessing the external approach. We will briefly return to the internal approach in Sec. 6.

4.2 DMS as an External GE's Component

According to Def. 1, an internal bot DMS is a function $DMS : P({}^bVB) \times P({}^bP) \times P({}^bVM) \times P({}^bR) \rightarrow P({}^bVM) \times P({}^bR)$. To provide the same mechanism externally, we need to export facts about the virtual body and facts perceived by the bot from the GE to an external DMS and to provide the DMS with a way to pass requests to the GE. The bot's mental states need not be present inside the GE as the mental states are utilised only by a native DMS that will not be active. Instead, agent DMS may represent mental states derived by itself. Figure 2 depicts the desired model of a GE bound together with an external DMS.

When comparing Figure 2 with Figure 1, Figure 2 divides bots between internal and external. The internal bots are controlled by their bot DMS functions (native to the GE) whereas external bots are controlled by agent DMSs, which are modelled separately of GE. The sequence diagram on the right reflects this distinction. Figure 2 also contains many additional arrows between the GE and the agent: 1) load, 2) update, 3) requested, 4) requests, commands. These arrows denote the translation of facts and requests between the GE and the engine. The *load*, *update* and *requested* arrows are functions exporting facts from the GE and translating them to the representation used by the agent DMS. *Load* is invoked only at the beginning of the simulation, and it exports some (or all) S^1 facts. Some S^1 facts may stay managed by the GE and become later requested. Thus, the *requested* arrows realise the pull strategy (see Sec. 2) returning facts from D^1, S^1 and more importantly from D^2, S^2. The *update* arrow realises the push strategy returning some facts from D^1 regularly.

Note that technically, even external bots may use some AI algorithms implemented by the GE, algorithms operating with unexported facts, such as path-finding algorithms.

The *requests, commands* arrow depicts the way the agent DMS passes requests back to the bot inside the GE. Requests are managed inside D^1 class, i.e., as dynamic facts. Additionally, the agent DMS may implement supplementary inference algorithms to infer Class 3 facts as *inferred* arrow suggests.

Importantly, by introducing push and pull strategy, the GEs conceptually differ from simple worlds such as grid worlds, in which many AI algorithms are tested, where all information is usually present as F^1 facts but not F^2 facts.

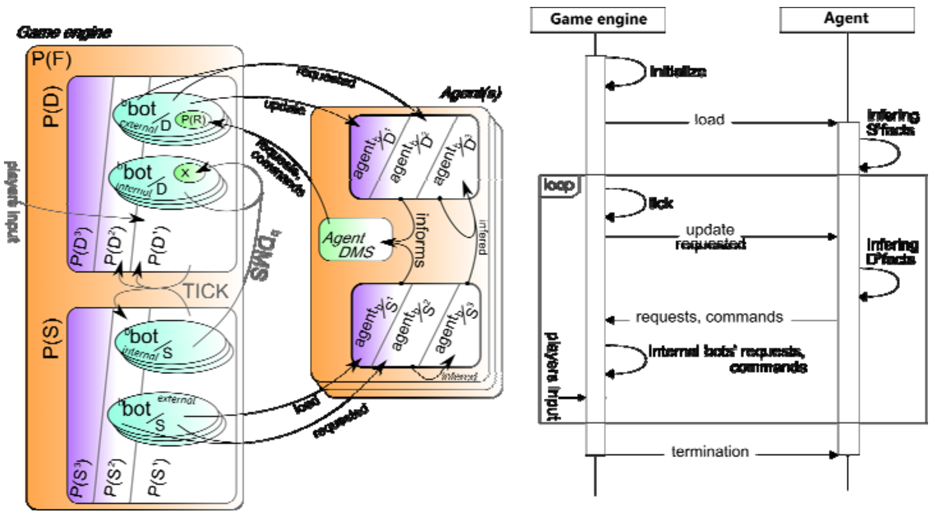

Fig. 2. The figure pictures the information flow between GE–DMS together with sequence diagram of GE–DMS interaction. The mark X has the same meaning as on Fig. 1.

4.3 Summary

This section has presented a model of a GE bound with an external DMS, capitalising on the conceptual framework from Section 3. As every abstraction, the model presented here may not fit an actual GE entirely, but we believe it is robust enough to embrace most of them and to provide useful guidance in coupling agent DMSs with GEs. The next section will look at this topic from more technical standpoint, introducing a particular implementation of the abstract model.

5 Pogamut 3 Platform, GE–DMS Mediation Layer

The purpose of this section is to present the architecture of Pogamut 3 as a mediation layer between a GE and an agent DMS. The development of such layer is technically hard by itself. First, developers need to understand the complex, often undocumented code of the GE before they can even start thinking about the translation of facts into an agent DMS. Then, the developers' have to address many tedious technical issues, mostly out of the main scope of their work, such as development of a debugger. Many of such issues are addressed repeatedly, which is in most cases a waste of time.

For past four years, we have been developing the Pogamut platform, which provides general solutions for many of these issues, allowing developers to focus on their main goals – experimenting with various DMSs inside simulated 3D worlds provided by the game industry. We remark that Pogamut is already widely used [e.g., 12, 13, 14, 15]. A new major version, Pogamut 3, has been already released [11]. Importantly, Pogamut 3 is also suitable for education [32].

Pogamut 3 currently utilises the well-known Unreal Tournament 2004 action videogame [17]. The game features a lot of pre-built objects, maps, and a map editor, allowing for custom modifications of the original game content, including creating simple maps for experiments. Section 6 reviews our work in progress concerning bindings Pogamut 3 with more game engines.

Pogamut 3 is programmed in the Java language and is currently most suited for utilisation of DMSs implemented in Java, Python or Groovy language. The platform already allows to experiment with a few agent DMSs, namely POSH [33], ACT-R [34] and an AgentSpeak(L)-like system [16].

Pogamut 3 has been already discussed in depth in [11, 32, 35] where comparison with related work is given as well. This paper focuses more on the layers standing between DMSs and GEs. In the rest of this section, Pogamut's generic agent architecture implementing the ideas from Sec. 3 and 4 will be introduced and exemplified on an implemented AgentSpeak(L)–based agent [16], called here *AS agent*, demonstrating Pogamut's technical flexibility.

5.1 Architecture of Pogamut Agent

The generic architecture of Pogamut 3 agent is depicted on Figure 3. The architecture introduces a set of layers that shields an agent DMS from the low-level communication with a GE, taking care of the *load, update, requested* and *requests* arrows from Figure 2. These layers are: *WorldView*, *Working memory*, *Inference engine*, and *Reactive layer*. The layers were implemented by the AS agent using Java, tuProlog [36], RETE-engine Hammurapi rules [37], and again Java, respectively (tuProlog and Hammurapi where chosen due to their available Java implementation that fits nicely with Pogamut). Additionally, the AS agent employed our proprietary AgentSpeak(L)-like system [16] as the DMS.

The DMS selects actions to execute based on facts represented in Working memory. It is more comfortable to work with facts like "bot Tom is chasing bot Clara" at the level of DMS (and thus at the level of Working memory) rather than with facts like "bot Tom is at position <*x,y,z*>" and "bot Clara's speed is *V*". It is the job of Inference engine to infer the former kind of facts, i.e., Class 3 facts.[5]

The main component of the architecture, and the only mandatory, is WorldView. This component arose from the need to provide an abstraction of game facts for the

[5] We also tried to infer these facts inside the DMS directly, but, at least for our AS agent, it turned out that that was inefficient and specification of the inference rules was cumbersome. A separate RETE engine ([37]) turned out to be more suitable for this task. These practical aspects motivated the separate component for inferring Class 3 facts in our architecture.

other components of the architecture. WorldView has two major functions: 1) synchronising facts incoming from a GE as well as the other components, 2) providing these facts to multiple components at once. The component can be seen as a blackboard [38].

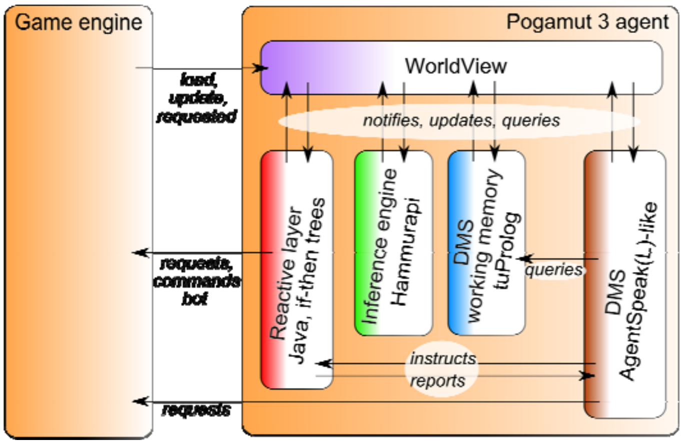

Fig. 3. High-level architecture of the Pogamut 3 agent implemented by the AS agent

It implements both *push* and *pull* strategy information retrieval, i.e., it is possible to query WorldView directly, or attach listeners that are informed whenever facts are updated. Importantly, this *update event model* of WorldView is flexible, in the sense that WorldView understands an ontology of Java objects that comes out of the Java class hierarchy. This works as follows. Let us assume that the virtual world simulated by the GE can be populated by items such as fruits and vehicles, but currently only apples and cars are supported. Concerning agents populating this world, this means that designers must provide a way for the agents to recognise items of these two kinds. When WorldView is used, designers have to create the following classes representing the objects' types and their categories: "item", "fruit", "vehicle", "apple", and "car". Utilising Java class inheritance, designers will further define that classes "vehicle" extends "item", "fruit" extends "item", "apple" extends "fruit", and "car" extends "vehicle". WorldView then propagates events according to this ontology; for instance, when an event happens on the object of class "apple", it is propagated also to "fruit" and "item". This makes the object model of WorldView flexible as any other agent's component may listen on all "fruit" events by attaching a listener on the class "fruit" instead of attaching the listener on every instance of fruit. This feature is not present in most GEs today.

The application of WorldView for various knowledge representations is straightforward. Whenever a different knowledge representation is required, it suffices to create a translator from Java objects into a desired representation and vice versa. The WorldView event model will then take care of propagation of fact updates to such a translator. For the AS agent, we have implemented this mechanism for all the four components WorldView communicates with (Fig. 3). For instance, the RETE inference engine has its listeners attached to desired Java classes inside WorldView and when it infers a new fact, this fact is propagated via WorldView to other components when they have their listeners up.

The last component of the architecture is Reactive layer, which is useful for coping with situations like “projectile is coming” in a swift way. While it is possible to model this kind of reflexive behaviour within a high-level goal-oriented DMS, our experience is that it is better to have a separate module handling these reflexes (for instance due to time efficiency and due to the fact that reflexive behaviour is easier to manage in Java than in such DMS). This reflexive—deliberative decomposition is a reminiscence of layered control architectures [39].

The key point of the blackboard architecture is that individual components are, to a large extent, oblivious to existence of the other components. This is important for at least two reasons. First, a developer can use arbitrary Java libraries for different modules. Second, a new module can be added during development. In our case, we have added the inference engine in this way, but in the context of videogames, other modules come into mind, such as an emotion model or episodic memory.

6 Pogamut 3 beyond UT 2004

So far, this paper discussed predominantly coupling between a GE and an external agent DMS for controlling a single bot. Additionally, only a UT 2004 implementation has been presented. It is natural to ask whether our approach can be extended. This section discusses three possible directions of such extension.

1) Different game engine. Does our approach work well for a different GE?

2) Internal coupling. How the framework from Fig. 3 should be refined when an agent DMS is coupled internally?

3) Multiple agent generalisation. How our framework can be generalised when multiple agents connect to a same GE?

In fact, we already have prototypes of binding to Unreal Tournament 3 [40], Unreal Development Kit [41], Defcon [20], StarCraft [19] and Virtual Battle Space 2 (VBS) [18] which provides us a broad range of game engines and game types for the validation of our approach (Point 1).

Internal coupling (Point 2) is being tested mainly on binding to Defcon [20]. Defcon is a completely different game than UT 2004. It is a simulation of a global thermonuclear conflict played in real-time, where a player takes a role of a commander in charge of military forces under the flag of one nuclear nation. Secondly, this work demonstrates that Pogamut can be used also beyond the domain of 3D bots. Thirdly, Pogamut is connected to Defcon via the game’s internal API, demonstrating that internal binding is possible. In future, we plan to investigate the relation between the framework presented on Fig. 2 and the Defcon binding in detail.

Finally, connecting Pogamut to Virtual Battle Space 2 (VBS) [18], a military simulator, is important due to Point 3. Almost every 3D game features multiple bots. When these bots are represented as individuals, that is, each follows the architecture from Fig. 3, following observations can be made:

a) Repetitive queries. Facts, both static and dynamic, are often queried repetitively.

b) Concurrent access. Bots may access the game engine concurrently, creating synchronisation issues.

c) Locality of facts. Nearby bots tend to acquire similar fact sets.

d) Shared facts. Bots may share some static and dynamic information, e.g. discovered map and known topology, communication channels, etc.

e) Bots communication. If it is believable, bots may be allowed to communicate directly each other with, which opens the possibility to bypass the GE concerning the communication.

Based on these observations, we designed the *MultiBotProxy* (MBP) architecture extending the architecture from Fig. 3. The MBP architecture covers the possibility of multiple externally connected DMSs to a single GE. This architecture is now being implemented using the VBS simulator.

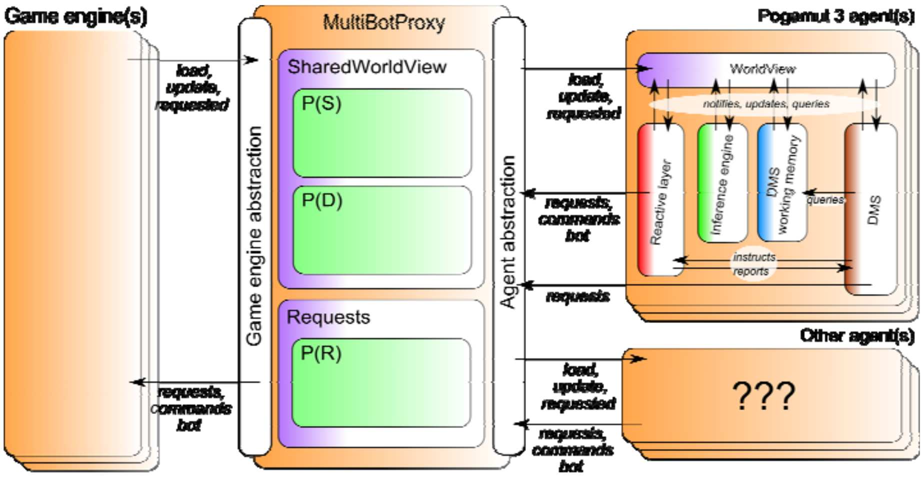

Fig. 4. MultiBotProxy schema. The MBP may allow connection of agent DMSs developed by different means (i.e., not only by Pogamut) to the same GE.

The MBP is a 3-tier architecture (see Figure 4), where the MBP node is the place between the GE and the externally connected agents. The MBP provides, besides its obvious proxy functionality, an interface for communication and data flow management – information caching, request optimisation, and data processing. *SharedWorldView*, the main component of the architecture, can be conceived as a blackboard shared by the GE and all the agents.

7 Conclusion

This paper started with the question *Can knowledge accumulated in the MAS field concerned with agent reasoning be used for reasoning of individual bots or a couple of bots?* We argued that to answer this question, it is necessary to connect several agent DMS to a single GE and compare them against existing AI techniques currently used by the game industry. To facilitate this process, we have presented a theoretical framework (Fig. 1, 2) for thinking about coupling external agent DMSs to GEs and the software toolkit Pogamut for practical development of such couplings. We have also presented an AgentSpeak(L)-based system fully connected to the UT 2004 game, demonstrating applicability of Pogamut. Finally, we reviewed our work in progress aiming at applying Pogamut beyond the domain of UT 2004: most notably for two strategy games, for new UT versions, and for a 3D game featuring teams of bots.

We conclude this paper with an interesting observation that our architecture from Fig. 3 implementing the theoretical framework from Fig. 2 could be, *per se*, conceived as a multi-agent system. In other words, our suggestion is that a "mind" of a *single* agent connected to a game engine can be perceived as a system of interacting, fully or partly autonomous agents. It would be interesting to elaborate on this idea in future, since this may bring the possibility to use, in games, knowledge the MAS field gained on negotiation; not for negotiation between different bots, but between different components of their "minds".

Acknowledgement. This work was partially supported by grant P103/10/1287 (GA ČR), research project MSM0021620838 (MŠMT ČR) and by students grants GA UK No. 44910 and GA UK No. 21809.

References

1. Champandard, A.J.: Key Trends in Game AI – Are You Ready for These? In: AiGameDev.com online, `http://aigamedev.com/open/editorial/key-trends/` (April 17, 2009)
2. Orkin, J.: Three States & a Plan: The AI of F.E.A.R. In: Proceedings of Game Developer's Conference (2006)
3. Stanley, K.O., Bryant, B.D., Miikkulainen, R.: Real-time Neuroevolution in the NERO Video Game. IEEE Transactions on Evolutionary Computation 9(6), 653–668 (2005)
4. Zubek, R.: Introduction to Hidden Markov Models. In: AI Game Programming Wisdom 3, pp. 633–646. Charles River Media, Hingham (2006)
5. Wooldridge, M.: An Introduction to MultiAgent Systems. John Wiley & Sons, Chichester (2002)
6. Briot, J.-P., Sordoni, A., Vasconcelos, E., de Azevedo Irving, M., Melo, G., Sebba-Patto, V., Alvarez, I.: Design of a Decision Maker Agent for a Distributed Role Playing Game – Experience of the SimParc Project. In: Dignum, F., Bradshaw, J., Silverman, B., van Doesburg, W. (eds.) Agents for Games and Simulations. LNCS, vol. 5920, pp. 119–134. Springer, Heidelberg (2009)
7. Bordini, R.H., Hübner, J.F.: BDI Agent Programming in AgentSpeak Using *Jason* (Tutorial Paper). In: Toni, F., Torroni, P. (eds.) CLIMA 2005. LNCS (LNAI), vol. 3900, pp. 143–164. Springer, Heidelberg (2006)
8. Busetta, P., Ronnquist, R., Hodgson, A., Lucas, A.: JACK Intelligent Agents - Components for Intelligent Agents in Java, AgentLink News (2) (1999)
9. Braubach, L., Pokahr, A.: Jadex: BDI Agent System, `http://jadex.informatik.uni-hamburg.de` (27.9.2009)

10. Dignum, F., Westra, J., van Doesburg, W.A., Harbers, M.: Games and Agents: Designing Intelligent Gameplay. International Journal of Computer Games Technology, vol. 2009 (2009)
11. Gemrot, J., Kadlec, R., Bída, M., Burkert, O., Píbil, R., Havlíček, J., Zemčák, L., Šimlovič, J., Vansa, R., Štolba, M., Plch, T., Brom, C.: Pogamut 3 Can Assist Developers in Building AI (Not Only) for Their Videogame Agents. In: Dignum, F., Bradshaw, J., Silverman, B., van Doesburg, W. (eds.) Agents for Games and Simulations. LNCS, vol. 5920, pp. 1–15. Springer, Heidelberg (2009)
12. Brom, C., Bída, M., Gemrot, J., Kadlec, R., Plch, T.: Emohawk: Searching for a "Good" Emergent Narrative. In: Iurgel, I.A., Zagalo, N., Petta, P. (eds.) ICIDS 2009. LNCS, vol. 5915, pp. 86–91. Springer, Heidelberg (2009)
13. Burkert, O., Brom, C., Kadlec, R., Lukavský, J.: Timing in Episodic Memory: Virtual Characters in Action. In: Proceedings of AISB workshop Remembering Who We Are – Human Memory For Artificial Agents, Leicester, UK (to appear)
14. Arrabales, R., Ledezma, A., Sanchis, A.: Towards Conscious-like Behavior in Computer Game Characters. In: Proceedings of the IEEE Symposium on Computational Intelligence and Games, pp. 217–224 (2009)
15. Hindriks, K. V., van Riemsdijk, M. B., Behrens, T., Korstanje, R., Kraaijenbrink, N., Pasman, W., de Rijk, L.: Unreal GOAL Bots: Conceptual Design of a Reusable Interface (AGS 2010) (May 2010)
16. Gemrot. J.: Joint behaviour for virtual humans. Master's thesis. Charles University, Prague (2009)
17. Epic Games: Unreal Tournament 2004 (2004), `http://www.unrealtournament2003.com/` (7.2.2010)
18. Virtual Battlespace 2. Bohemia Interactive, `http://www.bistudio.com/bohemia-interactive-simulations/virtual-battlespace-2_czech.html` (7.2.2010)
19. StarCraft. Blizzard Entertainment, `http://eu.blizzard.com/en-gb/games/sc/` (21.8.2010)
20. Defcon: Everybody dies. Introversion software, `http://www.introversion.co.uk/defcon/` (7.2.2010)
21. Loyall, B.A.: Believable Agents: Building Interactive Personalities. Ph.D. diss. Carnegie Mellon University (1997)
22. Orkin, J.: 3 States & a Plan: The AI of F.E.A.R. In: Game Developer's Conference Proceedings (2006)
23. Rao, A.S.: AgentSpeak(L): BDI agents speak out in a logical computable language. In: Perram, J., Van de Velde, W. (eds.) MAAMAW 1996. LNCS, vol. 1038. Springer, Heidelberg (1996)
24. Game Engine. Wikipedia, `http://en.wikipedia.org/wiki/Game_engine` (7.2.2010)
25. Zerbst, S., Duvel, O.: 3D game engine programming. In: Course Technology PTR (2004)
26. Lua programming language, `http://www.lua.org/` (7.2.2010)
27. Python programming language, `http://www.python.org/` (7.2.2010)
28. UnrealScript programming language, `http://unreal.epicgames.com/UnrealScript.htm` (7.2.2010)
29. Stentz, A.: Optimal and Efficient Path Planning for Partially-Known Environments. In: Proceedings of the International Conference on Robotics and Automation (1994)
30. Houlette, R., Fu, D.: The Ultimate Guide to FSMs in Games. In: AI Game Programming Wisdom 2. Charles River Media, Hingham (2003)

31. Isla, D.: Managing Complexity in the Halo 2 AI System. In: Proceedings of the Game Developers Conference (2005)
32. Brom, C., Gemrot, J., Burkert, O., Kadlec, R., Bída, M.: 3D Immersion in Virtual Agents Education. In: Spierling, U., Szilas, N. (eds.) ICIDS 2008. LNCS, vol. 5334, pp. 59–70. Springer, Heidelberg (2008)
33. Bryson, J.J.: Inteligence by design: Principles of Modularity and Coordination for Engineering Complex Adaptive Agent. PhD Thesis. MIT, Department of EECS, Cambridge, MA (2001)
34. Anderson, J.R.: How can the human mind occur in the physical universe? Oxford University Press, Oxford (2007)
35. Gemrot, J., Brom, C., Kadlec, R., Bída, M., Burkert, O., Zemčák, M., Píbil, R., Plch, T.: Pogamut 3 – Virtual Humans Made Simple. In: Gray, J., Nefti-Meziani, S. (eds.) Advances in Cognitive Systems. IET Publisher (in press)
36. TuProlog, http://alice.unibo.it/xwiki/bin/view/Tuprolog/ (7.2.2010)
37. Hammurapi rules, http://www.hammurapi.biz/hammurapi-biz/ef/xmenu/hammurapi-group/products/hammurapi-rules/index.html (7.2.2010)
38. Hayes-Roth, B.: A blackboard architecture for control. Artificial Intelligence 26(3), 251–321 (1985)
39. Wooldridge, M.: An Introduction to MultiAgent Systems, p. 99. John Wiley & Sons, Chichester (2002)
40. Epic Games: Unreal Tournament 3, http://www.unrealtournament.com/uk/index.html (1.9.2010)
41. Epic Games: Unreal Development Kit (UDK), http://www.udk.com/ (1.9.2010)