UNREAL GOAL Bots
Conceptual Design of a Reusable Interface

Koen V. Hindriks[1], Birna van Riemsdijk[1], Tristan Behrens[2], Rien Korstanje[1], Nick Kraayenbrink[1], Wouter Pasman[1], and Lennard de Rijk[1]

[1] Delft University of Technology, Mekelweg 4, 2628 CD, Delft, The Netherlands
{k.v.hindriks,m.b.vanriemsdijk}@tudelft.nl
[2] Clausthal University of Technology, Julius-Albert-Straße 4, 38678 Clausthal, Germany
behrens@in.tu-clausthal.de

Abstract. It remains a challenge with current state of the art technology to use BDI agents to control real-time, dynamic and complex environments. We report on our effort to connect the GOAL agent programming language to the real-time game UNREAL TOURNAMENT 2004. BDI agents provide an interesting alternative to control bots in a game such as UNREAL TOURNAMENT to more reactive styles of controlling such bots. Establishing an interface between a language such as GOAL and UNREAL TOURNAMENT, however, poses many challenges. We focus in particular on the design of a suitable and reusable interface to manage agent-bot interaction and argue that the use of a recent toolkit for developing an agent-environment interface provides many advantages. We discuss various issues related to the abstraction level that fits an interface that connects high-level, logic-based BDI agents to a real-time environment, taking into account some of the performance issues.

Categories and subject descriptors: I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence—*Intelligent Agents*; I.6.7 [**Simulation Support Systems**]: Environments

General terms: Design, Standardization, Languages.

Keywords: agent-environment interaction, agent-oriented programming.

1 Introduction

Connecting cognitive or rational agents to an interactive, real-time computer game is a far from trivial exercise. This is especially true for logic-based agents that use logic to represent and reason about the environment they act in. There are several issues that need to be addressed ranging from the technical to more conceptual issues. The focus of this paper is on the design of an interface that is suitable for connecting logic-based BDI agents to a real-time game, but we will also touch on some related, more technical issues and discuss some of the challenges and potential applications that have motivated our effort.

The design of an interface for connecting logic-based BDI agents to a real-time game is complicated for at least two reasons. First, such an interface needs to be

F. Dignum (Ed.): Agents for Games and Simulations II, LNAI 6525, pp. 1–18, 2011.
© Springer-Verlag Berlin Heidelberg 2011

designed at the right *abstraction level*. The reasoning typically employed by logic-based BDI agents does not make them suitable for controlling low-level details of a bot. Intuitively, it does not make sense, for example, to require such agents to deliberate about the degrees of rotation a bot should make when it has to make a turn. This type of control is better delegated to a behavioral control layer. At the same time, however, the BDI agent should be able to remain in control and the interface should support sufficiently fine grained control. Second, for reasons related to the required responsiveness in a real-time environment and efficiency of reasoning, the interface should not flood an agent with percepts. Providing a logic-based BDI agent with huge amounts of percepts would overload the agents' processing capabilities. The *cognitive overload* thus produced would slow down the agent and reduce its responsiveness. At the same time, however, the agent needs to have sufficient information to make reasonable choices of action while taking into account that the information to start with is at best incomplete and possibly also uncertain.

We have used and applied a recently introduced toolkit called the Environment Interface Standard to implement an interface for connecting agents to a gaming environment, and we evaluate this interface for designing a high-level interface that supports relatively easy development of agent-controlled bots. We believe that making environments easily accessible will facilitate the evaluation and assessment of performance and the usefulness of features of agent platforms.

Several additional concerns have motivated us to investigate and design an interface to connect logic-based BDI agents to a real-time game. First, we believe more extensive evaluation of the application of logic-based BDI agents to challenging, dynamic, and potentially real-time environments is needed to assess the current state of the art in programming such agents. Such an interface will facilitate *putting agent (programming) platforms to the test*. Although real-life applications have been developed using agent technology including BDI agent technology, the technology developed to support the construction of such agents may be put to more serious tests. As a first step, we then need to facilitate the connection of such agents to a real-time environment, which is the focus of this paper. This may then stimulate progress and development of such platforms into more mature and effectively applicable tools. Second, the development of a high-level agent-game bot-interface may *make the control of game bots more accessible* to a broader range of researchers and students. We believe such an interface will make it possible for programmers with relatively little experience with a particular gaming environment to develop agents that can control game bots reasonably well. This type of interface may be particularly useful to prototype gaming characters which would be ideal for the gaming industry [1]. We believe it will also *facilitate the application of BDI agent technology by students* to challenging environments and thus serve educational purposes. The development of such an interface has been motivated by a project to design and create a new student project to teach students about agent technology and multi-agent systems. Computer games have been recognized to provide a fitting subject [2]. As noted in [1],

> Building agents situated in dynamic, potentially antagonistic environments that are capable of pursuing multiple, possibly conflicting goals not only teaches students about the fundamental nature and problems of agency but also encourage them to develop or enhance programming skills.

Finally, an interesting possibility argued for in e.g. [2,3] is that the *use of BDI agents to control bots* instead of using scripting or finite-state machines *may result in more human-like behavior*. As a result, it may be easier to develop characters that are believable and to provide a more realistic feel to a game. Some work in this direction has been reported in [4], which uses a technique called Applied Cognitive Task Analysis to elicit players' strategies, on incorporating human strategies in BDI agents. [3] also discuss the possibility to use data obtained by observing actual game players to specify the Beliefs, Desires, and Intentions of agents. It seems indeed more feasible to somehow "import" such data expressed in terms of BDI notions into sophisticated BDI agents, rather than translate it to finite-state machines. The development of an interface that supports logic-based BDI agent-control of bots thus may offer a very interesting opportunity for research into human-like characters (see also [1,5,6,7]).

As a case study we have chosen to connect the agent programming language Goal to the game Unreal Tournament 2004 (UT2004). UT2004 is a first-person shooter game that poses many challenges for human players as well as computer-controlled players because of the fast pace of the game and because players only have incomplete information about the state of the game and other players. It provides a real-time, continuous, dynamic multi-agent environment and offers many challenges for developing agent-controlled bots. It thus is a suitable choice for putting an agent platform to the test. [8] argue that Unreal Tournament provides a useful testbed for the evaluation of agent technology and multi-agent research. These challenges also make UT2004 a suitable choice for defining a student project as students will be challenged as well to solve these problems using agent technology. Multi-agent team tasks such as coordination of plans and behavior in a competitive environment thus naturally become available. In addition, the 3D engine, graphics and the experience most students have with the game will motivate students to actively take up these challenges. Moreover, as a competition has been setup around UT2004 for programming human-like bots [5], UT2004 also provides a clear starting point for programming human-like virtual characters. Finally, the Unreal engine has enjoyed wide interest and has been used by many others to extend and modify the game. As a result, many modifications and additional maps are freely available. It has, for example, also been used in competitions such as the RobocupRescue competition [9] which provides a high fidelity simulation of urban search and rescue robots using the Unreal engine. Using the Unreal Tournament game as a starting point to connect an agent platform to thus does not limit possibilities to one particular game but rather is a first step towards connecting an agent platform to a broad range of real-time environments. Moreover, a behavioral control layer called *Pogamut* extending *Gamebots* is available for UT2004 [10,8] which facilitates bridging the gap that exists when trying to implement an interface

oriented towards high-level cognitive control of a game such as UT2004. Throughout the paper the reader should keep in mind that we use these frameworks. Technically, UT2004 is state of the art technology that runs on Linux, Windows, and Macintosh OS.

Summarizing, the paper's focus is on the design of a high-level interface for controlling bots in a real-time game and is motivated by various opportunities that are offered by such an interface. Section 2 discusses some related work. Section 3 briefly introduces the GOAL agent programming language. Section 4 discusses the design of an agent-interface to UT2004, including interface requirements, the design of actions and percepts to illustrate our choices, and the technology that has been reused. This section also introduces and discusses a recently introduced technology for constructing agent-environment interfaces, called the Environment Interface Standard [11,12]. Section 5 concludes the paper.

2 Related Work

Various projects have connected agents to UT2004. We discuss some of these projects and the differences with our approach.

Most projects that connect agents to UT2004 are built on top of Gamebots [8] or Pogamut [10], an extension of Gamebots: See e.g. [13,14] which use Gamebots and [7] which use Pogamut.[1] Gamebots is a platform that acts as a UT2004 server and thus facilitates the transfer of information from UT2004 to the client (agent platform). The GameBots platform comes with a variety of predefined tasks and environments. It provides an architecture for connecting agents to bots in the UT2004 game while also allowing human players to connect to the UT2004 server to participate in a game. Pogamut is a framework that extends GameBots in various ways, and provides a.o. an IDE for developing agents and a parser that maps Gamebots string output to Java objects. We have built on top of Pogamut because it provides additional functionality related to, for example, obtaining information about navigation points, ray tracing, and commands that allow controlling the UT2004 gaming environment, e.g. to replay recordings.

A behavior-based framework called pyPOSH has been connected to UT2004 using Gamebots [14]. The motivation has been to perform a case study of a methodology called Behavior Oriented Design [1]. The framework provides support for reactive planning and the means to construct agents using Behavior Oriented Design (BOD) as a means for constructing agents. BOD is strongly inspired by Behavior-based AI and is based on "the principle that intelligence is decomposed around expressed capabilities such as walking or eating, rather than around theoretical mental entities such as knowledge and thought." [14] These agents thus are behavior-based and not BDI-based.

Although we recognize the strengths and advantages of a behavior-based approach to agent-controlled virtual characters, our aim has been to facilitate the use of *cognitive* agents to control such characters. In fact, our approach has been to design and create an interface to a behavior-based layer that provides access

[1] [15] is an exception, directly connecting READYLOG agents via TCP/IP to UT2004.

to the actions of a virtual character; the cognitive agent thus has ready access to a set of natural behaviors at the right abstraction level. Moreover, different from [1] the actions and behaviors that can be performed through the interface are clearly separated from the percepts that may be obtained from sensors provided by the virtual environment (although the behaviors have access to low-level details in the environment that is not all made available via the interface). The main difference with [1] thus is the fact that cognitive agent technology provides the means for action selection and this is not all handled by the behavior-layer itself (though e.g. navigation skills have been "automated", i.e. we reuse the navigation module of Pogamut).

An interface called *UtJackInterface* is briefly discussed in [16]. This interface allows JACK agents [17] to connect to UT2004. The effort has been motivated by the "potential for teaming applications of intelligent agent technologies based on cognitive principles". The interface itself reuses components developed in the Gamebots and Javabots project to connect to UT2004. As JACK is an agent-oriented extension of Java it is relatively straightforward to connect JACK via the components made available by the Gamebots and Javabots projects. Some game-specific JACK code has been developed to "explore, achieve, and win" [16]. The interface provides a way to interface JACK agents to UT2004 but does not provide a design of an interface for logic-based BDI agents nor facilitates reuse.

The cognitive architecture Soar [18] has also been used to control computer characters. Soar provides so-called *operators* for decision-making. Similar to Goal - which provides reserved and user-defined actions - these operators allow to perform actions in the bots environment as well as internal actions for e.g. memorizing. The action selection mechanism of Soar is also somewhat similar to that of Goal in that it continually applies operators by evaluating if-then rules that match against the current state of a Soar agent. Soar has been connected to UT2004 via an interface called the *Soar General Input/Output* which is a domain independent interface [19]. Soar, however, does not provide the flexibility of agent technology as it is based on a fixed cognitive architecture that implements various human psychological functions which, for example, limit flexible access to memory. An additional difference is that Soar is knowledge-based and does not incorporate declarative goals as Goal does.

Similarly, the cognitive architecture ACT-R has been connected to Unreal Tournament [20]. Interestingly, [20] motivate their work by the need for cognitively plausible agents that may be used for training. Gamebots is used to develop an interface from Unreal Tournament to ACT-R.

Arguably the work most closely related to ours that connects high-level agents to Unreal Tournament is the work reported on connecting the high-level logic-based language ReadyLog (a variant of Golog) to UT2004 [15]. Agents in ReadyLog also extensively use logic (ECLiPSe Prolog) to reason about the environment an agent acts in. Similar issues are faced to provide an interface at the right abstraction level to ensure adequate performance, both in terms of responsiveness as well as in terms of being effective in achieving good game performance. A balance needs to be struck in applying the agent technology

provided by ReadyLog and the requirements that the real-time environment poses in which these agents act. The main differences between our approach and that of [15] are that our interface is more detailed and provides a richer action repertoire, and, that, although ReadyLog agents are logic-based, ReadyLog agents are not BDI agents as they are not modelled as having beliefs and goals.

Summarizing, our approach differs in various ways from that of others. Importantly, the design of the agent interface reported here has quite explicitly taken into account what would provide the right abstraction level for connecting logic-based BDI agents such as Goal agents to UT2004. As the discussion below will highlight (see in particular Figure 1), a three-tier architecture has been used consisting of the low-level Gamebots server extension of UT2004, a behavioral layer provided by a particular bot run on top of Pogamut, and, finally, a logic-based BDI layer provided by the Goal agent platform. Maybe just as important is the fact that we have used a generic toolkit [11,12] to build the interface that is supported by other agent platforms as well. This provides a principled approach to reuse of our effort to facilitate control of Unreal bots by logic-based BDI agents. It also facilitates comparison with other agent platforms that support the toolkit and thus contributes to evaluation of agent platforms.

3 Agent Programming in Goal

Goal is a high-level agent programming language for programming rational or cognitive agents. Goal agents are logic-based agents in the sense that they use a knowledge representation language to reason about the environment in which they act. The technology used here is SWI Prolog [21]. Due to space limitations, the presentation of Goal itself is very limited and we cannot illustrate all features present in the language. For more information, we refer to [22,23], which provides a proper introduction to the constructs introduced below and discusses other features such as modules, communication, macros, composed actions, and more.

The language is part of the family of agent programming languages that includes e.g. 2APL, Jadex, and Jason [24]. One of its distinguishing features is that Goal agents have a mental state consisting of *knowledge*, *beliefs* and *goals* and Goal agents are able to use so-called *mental state conditions* to inspect their mental state. Mental state conditions allow to inspect both the beliefs and goals of an agent's mental state which provide Goal agents with quite expressive reasoning capabilities.

A Goal agent program consists of various sections. The *knowledge base* is a set of concept definitions or domain rules, which is optional and represents the conceptual or domain knowledge the agent has about its environment. For the purposes of this paper, the **knowledge** section is not important and we do not explain the relation to beliefs and goals here (see for a detailed discussion [23]). The **beliefs** section defines the initial *belief base* of the agent. At runtime a belief base, which is a set of beliefs coded in a knowledge representation language (i.e. Prolog in our case), is used to represent the current state of affairs. The **goals** section defines the initial *goal base*, which is a set of goals also coded in the same

knowledge representation language, used to represent in what state the agent wants to be. The **program** section consists of a set of action rules which together define a strategy or policy for action selection. The **actionspec** section consists of action specifications for each action made available by the environment; an action specification consists of a *precondition* that specifies when an action can be performed and a *postcondition* that specifies the effects of performing an action. Although Goal provides the means to write pre- and post-conditions it does not force a programmer to specify such conditions, and actions may be introduced with empty pre- and/or postconditions; we will discuss the usefulness of empty conditions later in the paper again. Finally, a set of *the percept rules* specify how percepts received from the environment modify the agent's mental state.

Actions are selected in Goal by so-called *action rules* of the form

if `<cond>` **then** `<action>`

where `<cond>` is a mental state condition and `<action>` is either a built-in or an action made available by the environment. These rules provide Goal agents with the capability to react flexibly and reactively to environment changes but also allow a programmer to define more complicated strategies. Modules in Goal provide a means to structure action rules into clusters of such rules to define different strategies for different situations [25]. Percept rules are special action rules used to process percepts received from the environment. These rules allow (pre)processing of percepts and allow a programmer to flexibly decide what to do with percepts received (updating by inserting or deleting beliefs, adopting or dropping goals, or send messages to other agents). Additional features of Goal include a.o. a macro definition construct to associate intuitive labels with mental state conditions which increases the readability of the agent code, options to apply rules in various ways, and communication.

4 Agent Interface for Controlling Unreal Bots

One of the challenges of connecting BDI agents such as Goal agents to a real-time environment is to provide a well-defined interface that is able to handle events produced by the environment, and that is able to provide sensory information to the agent and provides an interface to send action commands to the environment. Although Gamebots or Pogamut do provide such interfaces they do so at a very low-level. The challenge here is to design an interface at the right abstraction level while providing the agent with enough detail to be able to "do the right thing". In other words, the "cognitive load" on the agent should not be too big for the agent to be able to efficiently handle sensory information and generate timely responses; it should, however, also be plausible and provide the agent with more or less the same information as a human player. Similarly, actions need to be designed such that the agent is able to control the bot by sending action commands that are not too finegrained but still allow the agent to control the bot in sufficient detail. Finally, the design of such an interface should also pay attention to technical desiderata such as that it provides support for

debugging agent programs and facilitates easy connection of agents to bots. This involves providing additional graphical tools that provide global overviews of the current state of the map and bots on the map as well as event-based mechanisms for launching, killing and responding to UT server events. In the remainder of this section, we describe in more detail some of the design choices made and the advantages of using the Environment Interface toolkit introduced in [11,12]. We begin with briefly discussing Unreal Tournament 2004 and then continue with discussing the interface design.

4.1 Unreal Tournament

UT2004 is an interactive, multi-player computer game where bots can compete with each other in various arenas. The game provides ten different game types including, for example, *DeathMatch* in which each bot is on its own and competes with all other bots present to win the game where points are scored by disabling bots, and *Team DeathMatch* which is similar to DeathMatch but is different in that two teams have to compete with each other. One of the key differences between DeathMatch and Team DeathMatch is that in the latter bots have to act as a team and cooperate and coordinate. The game type that we have focused on is called *Capture The Flag* (CTF). In this type of game, two teams compete with each other and have as their main goal to conquer the flag located in the home base of the other team. Points are scored by bringing the flag of the opponent's team to one's own home base while making sure the team's own flag remains in its home base.

The CTF game type requires more complicated strategic game play [15] which makes CTF very interesting for using BDI agents that are able to perform high-level reasoning and coordinate their actions to control bots. An interface "at the knowledge level" [26] facilitates the design of strategic agent behavior for controlling bots as the agent designer is not distracted by the many low-level details concerning, for example, movement. That is, the interface discussed below allows an agent to construct a high-level environment representation that can be used to decide on actions and focus more on strategic action selection. Similarly, by facilitating the exchange of high-level representations between agents that are part of the same team, a programmer can focus more on strategic coordination. As one of our motivations for building an agent interface to UT2004 has been to teach students to apply agent technology in a challenging environment, we have chosen to focus on the CTF game type and provide an interface that supports all required actions and percepts related to this scenario (e.g. this game type also requires that agents are provided with status information regarding the flag, and percepts to observe a bot carrying a flag).[2]

[2] Our experience with student projects that require students to develop soccer agents using basically Java is that students spent most of their time programming more abstract behaviors instead of focussing on the (team) strategy. Similar observations related to UT2004 are reported in [13], and have motivated e.g. [10]. We hope that providing students with a BDI programming language such as Goal will focus their design efforts more towards strategic game play.

4.2 Requirements

As has been argued elsewhere [1], in order to make AI accessible to a broad range of people as a tool for research, entertainment and education various requirements must be met. Here, we discuss some of the choices we made related to our objective of making existing agent technology available for programming challenging environments.

The tools that must be made available to achieve such broad goals as making AI, or, more specifically, agent technology accessible need to provide quite different functionality. One of the requirements here is to make it possible to use an (existing) agent platform to connect to various environments. We argue that agent programming languages are very suitable as they provide the basic building blocks for programming cognitive agents. Agent programming languages, moreover, facilitate incremental design of agents, starting with quite simple features (novices) to more advanced features (more experienced programmers).

Additional tools typically need to be available to provide a user-friendly development environment, such as tools to inspect the *global* state of the environment either visually or by means of summary reports. Auxiliary tools that support debugging are also very important. Goal provides an Integrated Development Environment with various features for editing (e.g. syntax highligting) and debugging (e.g. break points). Similar requirements are listed in [19], which adds that it is important that the setup is flexible and allows for low-cost development such that easy modifications to scenarios etc are feasible. For example, in the student project, we plan to use at least two maps to avoid student teams to bias their agents too strongly with respect to one map. This presumes easy editing of maps, which is facilitated by the many available UT2004 editors.

4.3 Interface Design

The *Environment Interface Standard* (EIS) [11,12] is a proposed standard for interfaces between (agent-)platforms and environments. It has been implemented in Java but its principles are portable. We have chosen to use EIS because it offers several benefits. First of all, it increases the reusability of environments. Although there are a lot of sophisticated platforms, the exchange of environments between them is very rare, and if so it takes some time to adapt the environment. EIS on the other hand makes complex multi-agent environments, for example gaming environments, more accessible. It provides support for event and notification handling and for launching agents and connecting to bots.

EIS is based on several principles. The first one is *portability* which means in this context that the easy exchange of environments is facilitated. Environments are distributed via jar-files that can easily be plugged in by platforms that adhere to EIS. Secondly, it imposes only *minimal restrictions on the platform or environment*. For example, there are no assumptions about scheduling, agent communication and agent control. Also there are no restrictions on the use of different technical options for establishing a connection to the environment, as TCP/IP, RMI, JNI, wraping of existing Java-code et cetera can be used. Another principle is *the separation of concerns*. Implementation issues related to

the agent platform are separated from those related to the environment. Agents are assumed to be percept-processors and action-generators. Environment entities are only assumed to be *controllable*, i.e. they can be controlled by agents and provide sensory and effectoric capabilities. Otherwise EIS does not assume anything about agents and entities and only stores identifiers for these objects, and as such assures the interface is agnostic about agent and bot specifics.

EIS provides various types of implementation support for connecting an agent platform to an environment. It facilitates acting, active sensing (actions that yield percepts), passive sensing (retrieving all percepts), and percepts-as-notifications (percepts sent by the environment). Another principle is a *standard for actions and percepts*. EIS provides a so called *interface intermediate language* that is based on an abstract-syntax-tree-definition. The final principle is the *support for heterogeneity*, that is that EIS provides means for connecting several platforms to a single instance of an environment. EIS is supported by and has been tested with 2APL, Jadex, Jason, and by GOAL.

The connection established using EIS between GOAL-agents, which are executed by the GOAL-interpreter, and UT2004 bots in the environment consists of several distinct components (see Fig. 1). The first component is GOAL's support for EIS. Basically this boils down to a sophisticated MAS-loading-mechanism that instantiates agents and creates the connection between them and entities, together with a mapping between GOAL-percepts/actions and EIS ones. Connecting to EIS is facilitated by Java-reflection. Entities, from the environment-interface-perspective, are instances of *UnrealGOALBot*, which is a heavy extension of the *LoqueBot* developed by Juraj Simlovic. LoqueBot on the other hand is built on top of Pogamut[10]. Pogamut itself is connected to *GameBots*, which is a plugin that opens UT2004 for connecting external controllers via TCP/IP.

Entities consist of three components: (1) an instance of UnrealGOALBot that allows access to UT, (2) a so called *action performer* which evaluates EIS-actions and executes them through the UnrealGOALBot, and (3) a *percept processor* that queries the memory of the UnrealGOALBot and yields EIS-percepts.

The instantiation of EIS for connecting GOAL to UT2004 distinguishes three classes of percepts. *Map-percepts* are sent only once to the agent and contain static information about the current map. That is navigation-points (there is a graph overlaying the map topology), positions of all items (weapons, health, armor, power-ups et cetera), and information about the flags (the own and the one of the enemy). *See-percepts* on the other hand consist of what the bot currently sees. That is visible items, flags, and other bots. *Self-percepts* consist of information about the bot itself. That is physical data (position, orientation and speed), status (health, armor, ammo and adrenaline), all carried weapons and the current weapon. Although these types of percepts are implemented specifically for UT2004, the general concepts of percepts that are provided only once, those provided whenever something changes in the visual field of the bot, and percepts that relate to status and can only have a single value at any time (e.g. current weapon) can be reapplied in other EIS instantiations. Here are some examples: `bot(bot1,red)` indicates the bot's name and

its team, `currentWeapon(redeemer)` denotes that the current weapon is the Redeemer, `weapon(redeemer,1)`, indicates that the Redeemer has one piece of ammo left, and `pickup(inventoryspot56,weapon,redeemer)` denotes that a Redeemer can be picked up at the navigation-point `inventoryspot56`.

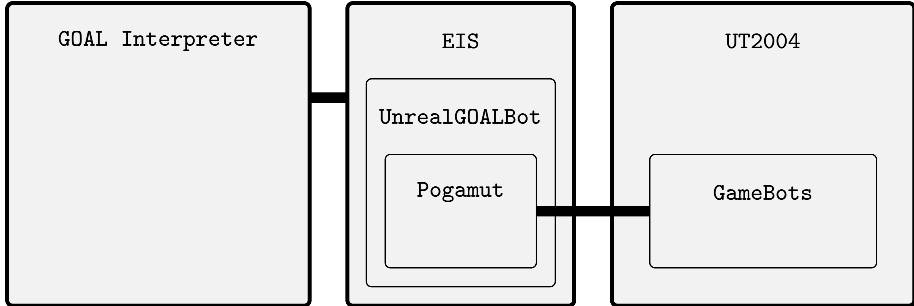

Fig. 1. A schematic overview of the implementation. The Goal-interpeter connects to the EIS via Java-reflection. EIS wraps UnrealGOALBot, a heavy extension of Loquebot. UnrealGOALBot wraps Pogamut, which connects to GameBots via TCP/IP. GameBots is an Unreal-plugin.

Actions are high-level to fit the BDI abstraction. The primitive behaviors that are used to implement these actions are based on primitive methods provided by the LoqueBot. Design-choices however were not that easy. We have identified several layers of abstraction, ranging from (1) really low level interaction with the environment, that is that the bot sees only neighboring waypoints and can use raytracing to find out details of the environment, over (2) making all waypoints available and allowing the bot to follow paths and avoid for example dodging attacks on its way, to (3) very high-level actions like *win the game*. The low level makes a very small reaction-time a requirement and is very easy to implement, whereas the high level allows for longer reaction times but requires more implementation effort. We have identified the appropriate balance between reaction-time implementation effort to be an abstraction layer in which we provide these actions: `goto` navigates the bot to a specific navigation-point or item, `pursue` pursues a target, `halt` halts the bot, `setTarget` sets the target, `setWeapon` sets the current weapon, `setLookat` makes the bot look at a specific object, `dropweapon` drops the current weapon, `respawn` respawns the bot, `usepowerup` uses a power-up, `getgameinfo` gets the current score, the game-type and the identifier of the bot's team. Due to space limitations we do not provide all the parameters associated with these actions in detail. Note that several but in particular the first two actions take time to complete and are only initiated by sending the action command to UT2004. Durative actions such as goto and pursue may be interrupted. The agent needs to monitor the actions through percepts received to verify actions were succesful. EIS does support providing percepts as "return values" of actions but this requires blocking of the thread

executing the action and we have chosen not to use this feature except if there is some useful “immediate" information to provide which does not require blocking. Special percepts were implemented to monitor the status of the goto action, including e.g. whether the bot is stuck or has reached the target destination. Moreover, the agent can control the route towards a target destination but may also delegate this to the behavioral control layer.

4.4 An Example: The Unreal-Pill-Collector

Figure 2 shows the agent-code of a simple Goal-agent that performs two tasks: (1) collecting pills and (2) setting a target for attack. The agent relies on the reception of percepts that are provided by the environment to update its beliefs during runtime. The beliefs present in the **beliefs** section in the agent program code are used to initialize the belief state of the agent. The first fact listed states that initially the agent has no target. The second fact represents the initial parameters associated with the bot's position, its rotation, velocity and moving state, together called the *physical-state* of the bot (the moving state of a bot can be `stuck`, `moving`, and `reached`). Similarly, the **goals** section is used to initialize the agent's goal base and initially will contain the goal of collecting special items, represented simply by the abstract predicate `collect`, and the goal to target all bots (implicitly only bots part of another team will be targeted as it is not possible in UT2004 to shoot your own team mates). The first rule in the **program** section makes the bot go to the specific location of a special-item (a so-called `pickup` location) if the agent knows about such a location and has the goal of collecting special items. The second rule sets the targets from none to all bots.

In the example only two out the total number of actions that were briefly introduced above have been used. We discuss these action more extensively here because they help to clarify how the interface with UT2004 works. Actions defined in the **actionspec** section need to be made available by the environment, in our case UT2004. They need to be specified in Goal because the *name* and *parameters* of the action need to be specified to be able to use it in action rules, and because preconditions and postconditions of actions may be specified (but need not be; they can be left empty). The `goto` action in the **actionspec** section allows the bot to move in the environment. The `setTarget` action sets the enemy bots that will be targeted if visible. These actions are quite different. The `goto` action takes more or less time to complete depending on the distance to be traveled. The `setTarget` action in contrast is executed instantaneously as it only changes a mode of operation (a parameter). This difference has important consequences related to specifying the pre- and postcondition of these actions. Whereas it is quite easy to specify the pre- and postcondition of the `setTarget` action, this is not the case for the `goto` action. As `goto` is a durative action that may fail (if only because an enemy bot may kill the bot) it is not possible to specify the postcondition uniquely. Moreover, some of the “details” of going somewhere as, for example, the exact route taken may (but need not be) delegated to the behavioral layer; this means that most of the time only

through percepts the exact route can be traced. Therefore, it makes more sense in a dynamic environment that an agent relies on percepts that are made available by the environment to inform it about its state than on the specification of a postcondition. For this reason, when an action is selected, Goal does not "block" on this action until it completes. Instead, upon selection of an action, Goal sends the action command to the environment and then simply continues executing its reasoning cycle; this design explicitly allows for *monitoring* the results of executing the action command *while it is being performed* by the bot in the environment. For some actions, among which the `goto` action, the interface has been designed such that specific monitoring percepts are provided related to events that are relevant at the cognitive level. The moving state percepts `stuck`, `moving`, and `reached` are examples that illustrate how an agent may conclude the `goto` action has failed, is ongoing, or has been successful. The setup of sending an action command to the environment while continuing the agent's reasoning cycle also allows for *interrupting* the action if somehow that seems more opportune to the agent; it can simply select a `goto` action with another target to do so.

This discussion also clarifies that providing an action specification in an agent programming language like Goal in dynamic environments is more of a (pragmatic) design issue than a task to provide a purely logical analysis and specification of a domain. It would require unreasonably complex specifications to handle all possible effects whereas perception allows for much more effective solutions.[3] The action specification for `goto` has been setup in such a way in the example program, however, that another `goto` action is only selected if the agent believes a position has been successfully reached in order to make sure that the agent does not change its mind continually (something which obviously will need to be changed in a truly multi-agent setting where the bot can get killed; the example program is mainly used here for illustrative purposes).

The previous discussion will have made clear the importance that perception has for controlling bots in a real-time strategy game such as UT2004. Rules to process percepts (as well as possibly messages sent by agents) are part of the **perceptrules** section of a Goal agent. In our example, the first percept rule stores all `pickup` positions in the belief base whereas the second one stores the movement state.

Though this agent is simple it does show that it is relatively simple to write an agent program using the interface that does something useful like collecting pills. Information needed to control the bot at the knowledge level is provided at start-up such as where pickup locations are on the map. The code also illustrates that some of the "tasks" may be delegated to the behavioral layer. For example, the agent does not compute a route itself but delegates determining a route to

[3] To be sure, we do not want to suggest that these remarks provide a satisfactory or definite solution for these issues; on the contrary, there remain many issues for future work. It does make clear, however, that in simulated environments such as games some of these issues can be resolved by the design of a specific perceptual interface, as we have done.

pickup navigation point. One last example to illustrate the coordination between the agent and the bot routines at lower levels concerns the precondition of the goto action. By defining the precondition as in Figure 2 (which is a design choice not enforced by the interface), this action will only be selected if a previously initiated goto behavior has been completed, indicated by the `reached` constant.

```
main: unrealCollector { % simple bot, collecting special items, and setting shooting mode
  beliefs{
    targets([]). % remember which targets bot is pursuing
    moving(triple(0,0,0), triple(0,0,0), triple(0,0,0), stuck). % initial physical state
  }
  goals{
    collect. targets([all]).
  }
  program{
    % main activity: collect special items
    if goal(collect), bel(pickup(UnrealLocID,special,Type)) then goto([UnrealLocID]).
    % but make sure to shoot all enemy bots if possible.
    if bel(targets([])) then setTarget([all]).
  }
  actionspec{
    goto(Args) {
      % The goto action moves to given location and is enabled only if
      % a previous instruction to go somewhere has been achieved.
      pre { moving(Pos, Rot, Vel, reached) }
      post { not(moving(Pos, Rot, Vel, reached)) }
    }
    setTarget(Targets) {
      pre { targets(OldTargets) }
      post { not(targets(OldTargets)), targets(Targets) }
    }
  }
  perceptrules{
    % initialize beliefs with pickup locations when these are received from environment.
    if bel( percept(pickup(X,Y,Z)) ) then insert(pickup(X,Y,Z)).
    % update the state of movement.
    if bel(percept(moving(Pos, Rot, Vel, State)), moving(P, R, V, S))
      then insert(moving(Pos, Rot, Vel, State)) + delete(moving(P, R, V, S)).
  }
}
```

Fig. 2. A very simple UNREAL-GOAL-agent collecting pills and setting targets

4.5 Implementation Issues

It is realized more and more that one of the tests we need to put agent programming languages to concerns performance. With the current state of the art it is not possible to control hundreds or even tens of bots in a game such as UT2004.[4] The challenge is to make agent programs run in real-time and to reduce the CPU load they induce. The issue is not particular for agent programming, [2] reports, for example, that Soar executes its cycle 30-50 times per second (on a 400MHz machine), which provides some indication of the responsiveness that can be maximally achieved at the cognitive level. Although we recognize

[4] Part of the reason is UT2004: increasing the number of bots also increases the CPU load induced by UT2004 itself.

this is a real issue, our experience has been that using the Goal platform it is possible to run teams that consist of less than 10 agents including UT2004 on a single laptop. Of course, a question is how to support a larger number of bots in the game without sacrificing performance. Part of our efforts therefore have been directed at gaining insight in which parts of a BDI agent induce the CPU load. The issues we identified range from the very practical to more interesting issues that require additional research; we thus identify some topics we believe should be given higher priority on the research agenda. Some of the more mundane issues concern the fact that even GUI design for an integrated development environment for an agent programming language may already consume quite some CPU. The reason is quite simple: most APLs continuously print huge amounts of information to output windows for the user to inspect, ranging from updates on the mental states to actions performed by an agent. More interesting issues concern the use and integration of third-party software. For example, various APLs have been built on top of JADE [27]. In various initial experiments, confirmed by some of our colleagues, it turned out that performance may be impacted by the JADE infrastructure and performance improves when agents are run without JADE (although this comes at the price of running a MAS on a single machine the performance seems to justify such choices). Moreover, as is to be expected, CPU is consumed by the internal reasoning performed by BDI agents. Again, careful selection of third-party software makes a difference. Generally speaking, when Prolog is used as reasoning engine, the choice of implementation may have significant impact. Finally, we have built on top of *Pogamut* to create a behavioral controller for UT bots. Measuring the performance impact of this layer in the architecture illustrated in Figure 1 that connects Goal through various layers to UT2004 is complicated, however; to obtain reasonable results for this layer using e.g. profilers therefore remains for future work.

In retrospective, we have faced several implementation challenges when connecting to UT2004 using EIS. EIS though facilitated design of a clean and well-defined separation of the agent (programmed in Goal) and the behavioral layer (the UnrealGOALBot) to the Unreal-AI-engine. The strict separation of EIS between agents as percept-processors and action-generators and entities as sensor- and effector-providers facilitated the design. We also had in mind right from the beginning that we wanted to use the UT2004-interface in order to provide the means for comparing APL platforms in general. Since support for EIS is easily established on other platforms we have solved this problem as well, by making the interface EIS-compliant.

4.6 Applications

The developed framework will be used in a student project for first year BSc. students in computer science. Before the start of the project, students will have had a course in agent technology where Prolog and Goal programming skills are taught. The students are divided into groups of five students each. Every group will have to develop a team of Goal agents that control UT bots in a CTF scenario where two teams attempt to steal each other's flag. In this

scenario, students have to think about how to implement basic agent skills regarding walking around in the environment and collecting weapons and other relevant materials, communication between agents, fighting against bots of the other team, and the strategy and team work for capturing the flag. The time available for developing the agent team is approximately two months, in which each student has to spend about 1 to 1,5 days a week working on the project. At the end of the project, there will be a tournament in which the developed agent teams compete against one another. The grade is determined based on the students' report and their final presentation. The purpose of the project is to familiarize students with basic aspects of agent technology in general and cognitive agent programming in particular, from a practical perspective.

Designing the interface at the appropriate level of abstraction as discussed above, is critical for making the platform suitable for teaching students agent programming. If the abstraction level is too low, students have to spend most of their time figuring out how to deal with low-level details of controlling UT bots. On the other hand, if the abstraction level is too high (offering actions such as *win the game*), students hardly have to put any effort into programming the GOAL agents. In both cases they will not learn about the aspects of agent technology that were discussed above.

5 Conclusion and Future Work

As is well-known, the UNREAL engine is used in many games and various well-known research platforms such as the USARSIM environment for crisis management that is used in a yearly competition [9]. We believe that the high-level Environment Interface that we have made available to connect agent platforms with UT2004 will facilitate the connection to other environments such as USARSIM as well. We believe the availability of this interface makes it possible to connect arbitrary agent platforms with relatively little effort to such environments which opens up many possibilities for agent-based simulated or gaming research. This is beneficial to put agent technology to the test. It will also make it possible to research human-agent mixed systems that control bots in UT2004.

The interface and architecture for connecting GOAL to UT2004 have been used successfully in a large student project at Delft University of Technology with 65 first-year BSc students that were trained to program GOAL agents first in a course on multi-agent systems. The multi-agent systems that were developed by the students competed against each other in a competition at the end of the project. Some lessons learned and an analysis of the agent programs that were written are reported in [28]. The project has resulted in many insights on how to design the agents controlling bots themselves, as well as on how to improve some of the associated tools and methodologies for authoring agent behavior. At the moment of writing, we are in the process of migrating the code of the behavioral layer based on Pogamut 2 to the new, redesigned Pogamut 3 [29].

The connection of an agent programming language for rational or BDI agents to UT2004 poses quite a few challenging research questions. A very interesting

research question is whether we can develop agent-controlled bots that are able to compete with experienced human players using the same information the human players possess. The work reported here provides a starting point for this goal. Even more challenging is the question whether we can develop agent-controlled bots that cannot be distinguished by experienced human players from human game players. At this stage, we have only developed relatively simple bots but we believe that the interface design enables the development of more cognitively plausible bots.

As noted in [30] and discussed in this paper, efficient execution is an issue for BDI agents. By increasing the number of bots and the number of agents needed to control these bots performance degrades. A similar observation is reported in [31]. Although it is possible to run teams of Goal agents to control multiple bots, our findings at this moment confirm those of [30]. We believe that efficiency and scalability are issues that need to be put higher on the research agenda.

References

1. Brom, C., Gemrot, J., Bida, M., Burkert, O., Partington, S.J., Bryson, J.: POSH Tools for Game Agent Development by Students and Non-Programmers. In: Proc. of the 9th Computer Games Conference (CGAMES 2006), pp. 126–133 (2006)
2. Laird, J.E.: Using a computer game to develop advanced ai. Computer 34(7), 70–75 (2001)
3. Patel, P., Hexmoor, H.: Designing Bots with BDI Agents. In: Proc. of the Symposium on Collaborative Technologies and Systems (CTS 2009), pp. 180–186 (2009)
4. Norling, E., Sonenberg, L.: Creating Interactive Characters with BDI Agents. In: Proc. of the Australian Workshop on Interactive Entertainment (IE 2004) (2004)
5. Botprize competition, `http://www.botprize.org/` (Accessed 30, January 2010)
6. Davies, N., Mehdi, Q.H., Gough, N.E.: Towards Interfacing BDI With 3D Graphics Engines. In: Proceedings of CGAIMS 2005. Sixth International Conference on Computer Games: Artificial Intelligence and Mobile Systems (2005)
7. Wang, D., Subagdja, B., Tan, A.H., Ng, G.W.: Creating Human-like Autonomous Players in Real-time First Person Shooter Computer Games. In: Proc. of the 21st Conference on Innovative Applications of Artificial Intelligence (IAAI 2009) (2009)
8. Kaminka, G., Veloso, M., Schaffer, S., Sollitto, C., Adobbati, R., Marshall, A., Scholer, A., Tejada, S.: Gamebots: A flexible test bed for multiagent team research. Communications of the ACM 45(1), 43–45 (2002)
9. RobocupRescue, `http://www.robocuprescue.org` (Accessed 30, January 2010)
10. Burkert, O., Kadlec, R., Gemrot, J., Bída, M., Havlíček, J., Dörfler, M., Brom, C.: Towards fast prototyping of iVAs behavior: Pogamut 2. In: Pelachaud, C., Martin, J.-C., André, E., Chollet, G., Karpouzis, K., Pelé, D. (eds.) IVA 2007. LNCS (LNAI), vol. 4722, pp. 362–363. Springer, Heidelberg (2007)
11. Behrens, T.M., Dix, J., Hindriks, K.V.: Towards an Environment Interface Standard for Agent-Oriented Programming. Technical report, Clausthal University of Technology, IfI-09-09 (September 2009)
12. Behrens, T., Hindriks, K., Dix, J., Dastani, M., Bordini, R., Hübner, J., Braubach, L., Pokahr, A.: An interface for agent-environment interaction. In: Proceedings of the The Eighth International Workshop on Programming Multi-Agent Systems (2010)

13. Kim, I.C.: UTBot: A Virtual Agent Platform for Teaching Agent System Design. Journal of Multimedia 2(1), 48–53 (2007)
14. Partington, S.J., Bryson, J.J.: The behavior oriented design of an unreal tournament character. In: Panayiotopoulos, T., Gratch, J., Aylett, R.S., Ballin, D., Olivier, P., Rist, T. (eds.) IVA 2005. LNCS (LNAI), vol. 3661, pp. 466–477. Springer, Heidelberg (2005)
15. Jacobs, S., Ferrein, A., Ferrein Lakemeyer, G.: Unreal GOLOG Bots. In: Proceedings of the 2005 IJCAI Workshop on Reasoning, Representation, and Learning in Computer Games, pp. 31–36 (2005)
16. Tweedale, J., Ichalkaranje, N., Sioutis, C., Jarvis, B., Consoli, A., Phillips-Wren, G.: Innovations in multi-agent systems. Journal of Network and Computer Applications 30(3), 1089–1115 (2007)
17. JACK: Agent Oriented Software Group, `http://www.aosgrp.com/products/jack` (Accessed 30, January 2010)
18. Laird, J.E., Newell, A., Rosenbloom, P.: Soar: An architecture for general intelligence. Artificial Intelligence 33(1), 1–64 (1987)
19. Laird, J.E., Assanie, M., Bachelor, B., Benninghoff, N., Enam, S., Jones, B., Kerfoot, A., Lauver, C., Magerko, B., Sheiman, J., Stokes, D., Wallace, S.: A test bed for developing intelligent synthetic characters. In: Spring Symposium on Artificial Intelligence and Interactive Entertainment (AAAI 2002) (2002)
20. Best, B.J., Lebiere, C.: Teamwork, Communication, and Planning in ACT-R. In: Proceedings of the 2003 IJCAI Workshop on Cognitive Modeling of Agents and Multi-Agent Interactions, pp. 64–72 (2003)
21. SWI Prolog, `http://www.swi-prolog.org/` (Accessed 30, January 2010)
22. Hindriks, K.V.: Programming Rational Agents in Goal. In: Multi-Agent Programming Languages, Tools and Applications, pp. 119–157. Springer, Heidelberg (2009)
23. Hindriks, K.V.: Goal Programming Guide (2010), Can be downloaded from `http://mmi.tudelft.nl/~koen/goal`
24. Bordini, R., Dastani, M., Dix, J., Seghrouchni, A.E.F.: Multi-Agent Programming Languages, Platforms and Applications. Springer, Heidelberg (2005)
25. Bordini, R., Dastani, M., Dix, J., Seghrouchni, A.E.F.: Multi-Agent Programming Languages, Tools and Applications. Springer, Heidelberg (2009)
26. Newell, A.: The Knowledge Level. Artificial Intelligence 18(1), 87–127 (1982)
27. Bellifemine, F.L., Caire, G., Greenwood, D.: Developing Multi-Agent Systems with JADE. Wiley, Chichester (2007)
28. Hindriks, K.V., van Riemsdijk, M.B., Jonker, C.M.: An empirical study of patterns in agent programs: An Unreal Tournament case study in Goal. In: Proceedings of the 13th International Conference on Principles and Practice of Multi-Agent Systems (PRIMA 2010) (2010)
29. Gemrot, J., Brom, C., Plch, T.: A periphery of pogamut: from bots to agents and back again. In: Dignum, F. (ed.) Agents for Games and Simulations II. LNCS (LNAI), vol. 6525, pp. 19–37. Springer, Heidelberg (2011)
30. Bartish, A., Thevathayan, C.: BDI Agents for Game Development. In: Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2002), pp. 668–669 (2002)
31. Hirsch, B., Fricke, S., Kroll-Peters, O., Konnerth, T.: Agent programming in practise - experiences with the jiac iv agent framework. In: Sixth International Workshop AT2AI-6: From Agent Theory to Agent Implementation, pp. 93–99 (2008)