Qsmodels: ASP Planning in Interactive Gaming Environment*

Luca Padovani[1] and Alessandro Provetti[2]

[1] M^2AG: Milan-Messina Action Group
DSI–Università degli studi di Milano. Milan, I-20135 Italy
luca@mag.dsi.unimi.it.
http://mag.dsi.unimi.it/
[2] M^2AG: Milan-Messina Action Group
Dip. di Fisica–Università degli studi di Messina. Messina, I-98166 Italy
ale@unime.it

Abstract. Qsmodels is a novel application of Answer Set Programming to interactive gaming environment. We describe a software architecture by which the behavior of a bot acting inside the Quake 3 Arena can be controlled by a planner. The planner is written as an Answer Set Program and is interpreted by the Smodels solver.

This article describes the Qsmodels project, which grew out of a graduation project [3] is currently under development. The aim of this project is twofold.

First, we want to demonstrate the viability of using *Answer Set Programming* [1] (ASP) in an interactive environment. The chosen environment is the *Quake 3 Arena* (Q3A) game from *id Software;* recently most of the source codes have been released to the public. *Q3A* is a *first person shooter*: the player's goal is to kill enemies using weapons and upgrades found inside the game field (normally a labyrinth). The human-like enemies found within *Q3A* are called *BOTs.* Like in the most computer games, Q3A bots behave according to the rules of a finite-state automaton (FSM) defined by expert game programmers.

The second objective is to implement and experiment with the high-level agent architecture described by Baral, Gelfond and Provetti [2]. Such schema consists in the following loop: *Observe–Select Goal–Plan–Execute.*

The Qsmodels architecture consists of two layers: a *high level,* responsible for mid- and long-term planning, and a (*low level*) in charge of plan execution and emergency state reactions. The high level has been developed mainly in ASP on the Smodels platform. I.e., `smodels` computes the *answer sets* of a logic program which characterizes all successful plans of a given length, following the more or less standard encoding found in [1]. The computed answer set is passed to the low-level layer that inspects it, extracts relevant syntactic informations and *executes* the required actions.

* Work supported by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2001-37004 WASP project.

The high-level layer of our project realizes a *Q3A* agent which, starting with the knowledge about the game field similar to that of an intermediate-level human player, tries to beat his opponents by facing them only when in a better condition for the attack. To achieve this result, we have added to the planner a very simple learning system which keeps track of opponent's behavior in order to better guess future moves.

Even though we are still in the experimental phase, we submit that our architecture has several advantages over the traditional schema for the *AI* part of games. Namely, our solution is easier to develop and keeps the AI at higher level of abstraction. The easiness in development is reached by keeping the planning rules separated from the *world model description* rules, so that they can be written even by AI beginners. Also, Qsmodels could be used for virtual-reality AI experiments; the use of a computer game as the laboratory environment allows choosing the level of abstraction of the physical model while, -at the same time- giving a useful visual feedback of agent's actions.

Finally, this project may help in evaluating the feasibility of using `smodels` in near real-time applications and environments. Indeed, we noticed a high computational demand to achieve realistic real-time behaviors. More code analysis and optimization is in demand.

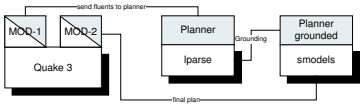

Figure 1. The Qsmodels Architecture

The Qsmodels software components can be divided in 3 parts: i) the ASP planner, ii) the *Q3A C++* interface, which implements *sensing* and plan execution, and iii) the *C++* low-level AI.

The execution model of Q3A, shown in Fig. 1, is summarized as follows. First, `MOD-1` does the *sensing* phase by inspecting some *Q3A* memory areas; then, the computed informations is translated into high-level *fluent values* and added to the planner. The planner is first grounded by `lparse` then passed to `smodels`, which computes one of its answer sets. Finally, `MOD-2` extracts the plan from the answer set and executes it by calling the relative *Q3A traps*.

Methodology

Our implementation required a lot of work and experiments in order to interface the existing software components, *Q3A* and `smodels`. The development of the agent required getting an in-depth knowledge of *Q3A* internal functions, most of which are not documented. `Smodels`, on the other hand, has been used as an external process, invoked by system calls. We are planning to switch to an API interface soon.

The two layers of Q3A work are executed concurrently: while the high level does the planning the low level is responsible for plan execution and reactive behavior in emergency situations. Events are deemed unforeseen when their occur makes the status of the domain incompatible with the assumptions made during the planning phase. It should be noticed that the low-level layer *inherits* some powerful functionality from *Q3A*, such as the *combat* and *shooting* actions, which are seen as atomic from the upper level.

To make our agent act realistically in its domain, we set the frequency of *sensing* at ten times per second. This measure has to do with the way actions are executed: each action may consists of several repeated calls, until the *goal* of the specific action is reached. So, since the plan execution is more associated with the *frame frequency* of the game than with the plan actions, sensing needed to be executed even during action execution.

To realize a *reaction behavior,* we have introduced so-called *pre-emption rules* which describe emergency behaviors in accordance with the environment and status of the plan. Pre-emption rules will be described in later sections.

Execution Cycle

The execution cycle of our application is shown in Figure 2. The first step is *sensing,* where we access *Q3A* memory searching for informations such as the agent state (position, health level ...) and availability of bonuses (health and ammunition tokes). Then, we check whether any emergency is happening, e.g., the agent is under attack, or he/she is facing the enemy, or he/she is behind the enemy etc. If any of these situations holds, then we execute the *pre-emption rules* to find those that apply to the present emergency and state. If no pre-emption rule applies then the execution cycle resumes.

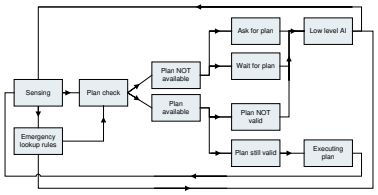

Figure 2 - Execution Cycle.

After sensing, if no emergency is detected we check whether a plan is currently available; if not, then we ask for a new one. Thus, we first translate the *Q3A* memory states in *fluents,* which are add as facts to the planner itself. Finally, we pass the augmented logic program to the external component *QsmodelsServer,* which is in charge of the `smodels` interface. The information embodied in the new fluents includes the agent's position, it's health and weapons state and the positions of known active objects. We include also a couple of atoms describing the *last known enemy position — expected new enemy position* to try to find usual routes taken by the enemy.

If a plan is available, then we have to check if any of the agents or enemies actions have invalidated the assumptions made at the planning phase. Indeed, since actions take some seconds to be executed and also the smodels computation can take several seconds, this situation would frequently happen, e.g., if the enemy takes a weapon that the agent was supposed to go get, the plan has to be invalidated since the weapon is not available anymore.

Let now consider plan execution. Each action available to the agent has been associated to a *trap* to *Q3A* system calls. The available actions are of course at very high level. This way, we have been able to reuse most of the basic AI work made by id Software: such as path finding and aiming. So, the available actions are: `move_towards` – `pick_health` – `pick_ammo` – `attack` and `elude`. All these actions except for `attack` are variations of `move_towards`, since to get an object

we have to reach it. Action `attack` simply passes the control to the low-level AI in a situation where the agent will certainly has to attack the enemy.

The use of pre-emption rules

The game field of Q3A can be described as very dynamic. So, it would be unfeasible to recompute the plan each time some aspect of the environment changes. In this sense, the introduction of the so called *Pre-emption rules* has probably been the most important step toward the realization of believable Q3A bots,

Pre-emptive rules allows describing a high-level reaction system in which we specify the reaction behavior of the agent and let `smodels` compute the actual reaction rules linked to the current plan. For each considered emergency situation and each time frame of the generated plan we get an appropriate reaction rule.

When an emergency happens our modification to *Q3A* searches the corresponding rule (time and event) through the rules and executes the action inside the body of the rule. As a result, pre-emption rules dictate a behavior somewhat similar to that of a FSM. However, in our case, the reaction is integrated in the planner and evaluated by the same inferential engine. Therefore, we couple a time-consuming planning system for long-term reasoning to a more efficient reaction system for quick reactions.

Application Experience

Our testing platform consists of a set of *Q3A* standard game levels in which our agent engages a duel against a human player. We require the game server (in our case an Intel P4 2.0GHz)to be run on a separate machine than that of the human player, due to the high computational power required by `smodels`.

The plan extraction phase can require up to 6/7 seconds, depending on the plan length. This delay is almost transparent to the human opponent, since during the planning our agent tries to hide. Should the agent meet the enemy then, the *pre-emption rules* together with the low-level AI will make it act quickly, usually avoiding the confrontation.

In Qsmodels plans the last action is always `attack` since the overall goal is to kill the enemy. However, the last action seldom gets executed since when emergency situations happen the *pre-emption rules* take control of the bot, canceling the residual part of the plan.

References

1. C. Baral, 2003. *Knowledge Representation, Reasoning and Declarative Problem Solving.* Cambridge University Press.
2. C. Baral M. Gelfond and A. Provetti, 1997. *Representing Actions: Laws, Observations and Hypotheses.* Journal of Logic Programming, 31(1-3).
3. L. Padovani, 2004. *Answer Set Programming in Interactive Gaming Environment.* Graduation project in Informatics (in Italian). University of Milan. Available from *http://mag.usr.dsi.unimi.it/*
4. Web location of the `smodels` solver: *http://www.tcs.hut.fi/Software/smodels/*