Dynamic Logic Programming and world state evaluation in computer games

Jozef Šiška

Department of Applied Informatics, Faculty of Mathematics, Physics and Informatics,
Comenius University, Bratislava, Slovakia,
siska@ii.fmph.uniba.sk

Abstract. In this paper we propose a framework for world state evaluation in computer games based on Dynamic Logic Programming (DynLoP).
Computer games (especially role-playing and adventure games) offer an exact, coherent and relatively small and simple world description and are usually built on game engines, which provide scripting capabilities. A common task of game scripting involves evaluation of the world state in a game, usually to find out whether a player has already completed a task (game quest).
In this paper we describe a framework for building a Dynamic Logic Program based on a description of a game world, quests and events in the game. Stable models of such a program are then used to determine the status of a quest or the whole game. Because of declarative nature of DynLoP, its use allows easier and simpler queries and quest (task) characterization than current imperative scripting languages used in game engines. Furthermore, the world of a computer game provides an excellent environment for evaluation of DynLoP and its various semantics.

1 Introduction and Motivation

Thinking about research in the area of computer games, most people imagine algorithms and hardware for fast and nice 3D graphics or realistic real-time simulations. Although that is really the largest portion of research done in this area, there are communities with other goals, some of them more tied to the area of artificial intelligence and, more specifically, knowledge representation, planning and natural language processing. These share the ultimate goal of a virtual world with artificial computer controlled characters, which would behave and could be talked to just like humans.

Logic programming (LP) is a formalism used in knowledge representation. Although there are few applications of LP in the area of computer games, they usually demonstrate possibilities of LP from a users(players) point of view[12]. In this paper we introduce a framework that uses Dynamic Logic Programming (DynLoP) – an extension of Logic Programming – to enhance and ease development of a certain type of computer games

Every computer game takes place in a specific world. Some kinds of games, mainly role-playing (RPG) and adventure games are based primarily on players interaction with such a world. In RPG games the player assumes the “role” of a character in a made up world and controls how the character interacts with the world, influencing it. In Adventure games player’s interaction with surrounding world is more constricted, as the player tries to solve puzzles and task. In both kinds of games however creativity plays the main part. The more ways to finish a task (and the more creative and full of fun) the better the game. It is therefore a constant goal to make the game world as open as possible and generic.

Games are not always written from scratch. Game engines are used, which provide communication with the player and a scripting interface in which specific games are ”scripted”. The more complex the quests are, the harder the creation of the game becomes. It is very common that a game does not consider a task finished, even when from players perspective it is. Such ”scripting glitches” then negatively influence players experience and perception of the game.

In this article we propose a framework for game world evaluation using dynamic logic programming. Game mechanics, world and quest definitions are represented by dynamic logic programs. Stable models are then used to determine the state of the game and quests. As the semantics of DynLoP handles inference and conflicts, the creation and maintenance of rules – programs is much simpler and very complex quests are easier to describe. The main goal of this framework is to incorporate LP into game engines and thus provide a base for further LP research like planning in a game world.

The rest of the paper is organized as follows: in the next section we recall basic definitions from the area of DynLoP as well as the stable model semantics for DynLoP. Then in the third section we describe the world of a computer game in detail relating it to the usual setting of multi-agent systems and also describe common game engine setups as well as scripting systems. The fourth section contains the description of our presented framework along with examples and a discussion. In the fifth section we provide some considerations for practical implementations. The last section presents some possibilities of future work and adds some concluding remarks.

2 Dynamic Logic Programming

In this paper we concentrate on a class of logic programs called *Dynamic Logic Programs* [9] which are a generalization of Logic Programs. First we recall some basic concepts and notation. Then we characterize some classes of logic programs and recall the definition of the class of Dynamic Logic Programs.

Let $\mathcal{A}$ be a set of propositional atoms (also called objective literals). For an atom A, not A is called a *default literal*. We denote by $\mathcal{L} = \{A, \text{not } A | A \in \mathcal{A}\}$ the set of all literals. For a literal $L \in \mathcal{L}$, we use not L to denote its *opposite* counterpart. Thus not L = not A iff $L = A$ and not $L = A$ iff L = not A. We call L and not L *conflicting* literals and denote it by $L \bowtie \text{not } L$. For a set of literals $M \subseteq \mathcal{L}$ we denote M^+ the set of all objective literals and M^- the set of all default literals from M. A set of literals M is called *consistent* if it does not contain two conflicting literals, otherwise it is *inconsistent*.

A *generalized logic program* P is a countable set of *rules* of the form $L \leftarrow L_1, L_2, \ldots, L_n$, where L and each of L_i are literals. If all literals are objective, P is called a *definite logic program*.

For a rule r of the form $L \leftarrow L_1, L_2, \ldots, L_n$ we call L the *head* of r and denote by *head*(r). Similarly by *body*(r) we denote the set $\{L_1, L_2, \ldots, L_n\}$ and call it the *body* of r. If $body(r) = \emptyset$ then r is called a *fact*. Throughout this paper we will use interchangeably the sets of literals and sets of facts having these literals as heads, although they are formally different. Two rules r and r' are called *conflicting* (opposite) if *head*(r) and *head*(r') are conflicting and we denote this by $r \bowtie r'$.

An *interpretation* is any consistent set of literals I. I is called a *total* interpretation if for each $A \in \mathcal{A}$ either $A \in I$ or not $A \in I$. A literal L is *satisfied* in an interpretation I ($I \models L$) if $L \in I$. A set of literals $S \subset \mathcal{L}$ is satisfied in I ($I \models S$) if each literal $L \in S$ is satisfied in I. A rule r is satisfied in I ($I \models r$) if $I \models body(r)$ implies $I \models head(r)$. A total interpretation M is called a model of the logic program P if for each rule $r \in P$ $M \models r$. We also say that M *models* P and write $M \models P$.

For a generalized logic program P we denote *least*(P) the least model of the definite logic program P and by $\tau(P)$ ($\tau(M)$) the program (model) obtained from P (M) by changing each default literal not A into a new atom *not_A*. According to [4] we can define *stable models* of a generalized logic program as those models M for which

$$\tau(M) = least\left(\tau\left(P \cup M^-\right)\right).$$

Dynamic Logic Programs as an extension of Logic Programs provide a way to express changing knowledge. Knowledge is structured into a sequence of logic programs. Conflicts between rules are resolved based on causal rejection of rules – for two conflicting rules, the more recent one overrides the older. Thus more recent programs contain more important rules which override older rules.

A *Dynamic Logic Program* is a sequence of generalized logic programs $\mathcal{P} = (P_1, P_2, \dots P_n)$. We use $\rho(\mathcal{P})$ to denote the multiset of all rules $\rho(\mathcal{P}) = \bigcup_{1 \le i \le n} P_i$.

A total interpretation M is a *Refined Dynamic Stable Model* of $\mathcal{P}$ iff

$$M = least\,([\rho(\mathcal{P}) \setminus Rej(\mathcal{P}, M)] \cup Def(\rho(\mathcal{P}), M))$$

where

$$\begin{aligned} Rej(\mathcal{P}, M) &= \{r \mid r \in P_i, \exists r' \in P_j : r \bowtie r', i \le j, M \models body(r')\} \\ Def(\rho(\mathcal{P}), M) &= \{\text{not } A \mid A \in \mathcal{L}, \nexists r \in \rho(\mathcal{P}) : M \models body(r), \\ &\qquad head(r) = A\} \end{aligned}$$

The set $Rej(\mathcal{P}, M)$ contains all *rejected* rules from $\mathcal{P}$ i.e. rules for which there is a conflicting rule satisfied in M in a more important program. The set $Def(M)$ contains default negations of all unsupported atoms.

The previous definition defines the *Refined Dynamic Stable Model semantics* (RDSM) [3]. All other semantics for DynLoP based on causal rejection of rules [2, 9, 6, 7, 10, 11] can be achieved by corresponding definitions of the sets $Rej(\mathcal{P}, s, M_s)$ and $Def(\rho(\mathcal{P}_s, M_s)$.

A transformational semantics equivalent to the DSM semantics can be defined for DynLoP[9]. This semantics transforms the DynLoP into a generalized logic program with a set of models that corresponds to that of the DynLoP. This allows for direct implementation of the DSM semantics for DynLoP by using existing LP solvers.

3 The world of a computer game

In this section we take a closer look on the world of a computer game. We describe the basic elements, how game engines usually handle them and compare them to the elements of a multi-agent system.

Throughout this paper we will be interested in role-playing and adventure computer games. World of such games consist of *objects*, such as characters (NPCs), items or even abstract items (sounds,light). Objects can be *passive* or *active*. Active objects can execute *actions*. Passive objects are usually just targeted by actions, thus changing their properties.

A special active object in a game is the *player object*, which is controlled by the player. It represents usually the character or item the player controls, although there are games where player controls more characters. Or in the case of multiplayer games, more players can play the game at a time.

Game objects have *properties*, which can be changed as a result of actions. Some properties and their changes are given by the basic mechanics of the game (position, motion, ...). Other properties are determined specially, by game scripts. This usually includes tracking the progress of the game or quest.

Because of the size, world is generally divided into *locations*. Most objects are active only in one (or very few) location(s) and disregarded elsewhere[1]. Only player objects and game-important objects persist between the locations. Information about quests that span over multiple locations is therefore usually recorded as properties of player object itself in some condensed form.

Given a base mechanics of the game, the game world is exactly described. Although such a world can be large and can make a challenge for smooth and realtime play, it is still relatively small compared to real world applications. One important aspect of such worlds is that they are scarcely ambiguous.

Existing game engines usually vary in the level of generalization, from very specific with most of the game mechanics and other things hardcoded, to general engines resembling just an user

[1] There are games which give an illusion of continuous world. These is however almost always achieved by switching the locations in background.

interface. Engines also offer various levels of scripting support, ranging from very simple and limited to full featured, object oriented, event driven scripting languages. All are however procedural and usually every non-trivial event in a game has to have a complex script associated with it. The quality and consistency of these scripts then forms a major factor of games' quality and impression.

As evaluation of more complex quests or game states can become very complicated a well designed game usually uses a kind of in place inference. Scripts infer small descriptions of game state (atoms) and store them as properties of objects. The evaluation of a more complex quest then resembles a logic program represented ad-hoc in the scripting language.

4 Game world evaluation

In this section we describe a simple use of Logic Programs in evaluation of the state of a game world. We represent the state of the world using the language of LP. Then we can pose queries as logic programs. As the state of the game world changes over time, we can use the benefits of DynLoP.

When posing queries in LP, usually two logic programs are considered: a database P and a query program Q. These are put together as one logic program $P \cup Q$ and it's stable models are determined. The answers are then based on these models, usually the truth value of an atom, or the absence or presence of a specific model is considered.

This however requires that all three programs P, Q and $P \cup Q$ are coherent. With Dynamic Logic Programming, only P and Q themselves need to be coherent. Inconsistencies between these two programs will be resolved preferring information from the latter one. Furthermore, using DynLoP we can combine more programs in such a way that the more specific ones override the others.

With a game world in mind, the process involves creating logic programs for the game mechanics and quests and using those together with program(s) representing current state of the game. These are then combined into a DynLoP and its dynamic stable models are determined.

A dynamic logic view of a game world with regard to a certain quest/task can be represented as follows:

Consider the DynLoP $\mathcal{P}$ and a set of atoms Q_a, where

$$\mathcal{P} = \{P_0, P_1, P_2, \ldots, P_n, P_g, P_q\}$$

P_0 represents the initial configuration of the game (facts at the beginning). P_1 through P_n represent the updates of facts (either as results of player actions or game created events). P_g is a program containing all the rules describing the general game mechanics and P_q represents the rules special for the evaluation of this quest (i.e., describing the quest). The set Q_a contains the atoms we will be interested in when deciding the result of the query.

The program containing the game mechanics is easily developed once for the whole game, quest programs are written for each quests and are used only when determining the state of that particular quest, most probably through a query from a script.

Programs P_1 through P_n represent a dynamic view of the changes in game world. These programs would contain only facts, and could be easily transformed into one program preserving the latest observation about atoms, thus reducing the complexity of the whole dynamic program. This makes even more sense in real applications, where the state of the game will be held internally in the engine, not in a DynLoP, and the logic program would be created from the internal representation just before the execution of a query. In such setup it makes more sense to construct just one program containing the current game state. Note that P_0 is still needed as it contains initial configuration of the game and as such may contain information that is not included in the game's state. This leads to a simpler program

$$\mathcal{P} = (P_0, P_1, P_g, P_q)$$

Game engines are used in creation of more games than one. Although they provide some basic mechanics of the game world, they allow game creators to design also their own rules for the specific game. These rules are not part of the engine enforced ruleset, but still are global to the game when compared with rules specific to the quests. Thus a more refined layout of the DynLoP makes more sense:

$$\mathcal{P} = (P_0, P_1, P_e, P_g, P_q)$$

Program P_e now refers to the rules of the engine, while P_g retains the rules specific for the game. This way a program for the basic mechanics of the engine can be clearly separated and attached to the engine. Also game specific rules can easily override engine provided rules as they are now more preferred (and any inconsistency will be resolved according to the DynLoP semantics).

Dynamic stable models of $\mathcal{P}$ are then used to determine the state of the quest. This is done by considering the number of models and how many of them satisfy Q_a, i.e., whether the atoms we are interested in are satisfied in all possible explanations of the world, in some of them or in none. The game engine can then decide whether to accept a quest if there is just a possible explanation (like a quest to persuade an NPC about something) or may require complete certainty.

Example 1. Consider the following DynLoP $\mathcal{P} = (P_0, P_1, P_g, P_q)$:

$$P_0 = \{\ \dots\ alive(King);\ alive(General);\ speak(King);\ speak(General);\ \dots\}$$

$$P_1 = \{\ killed(Player, King);\ \}$$

$$P_g = \left\{\begin{array}{c} \dots\ \text{not } alive(X) \leftarrow killed(Y,X); \quad alive(X) \leftarrow ressurect(Y,X);\ \dots \\ \dots \text{not } speak(X) \leftarrow spell(Silence, X); \dots \end{array}\right\}$$

$$P_q = \left\{\begin{array}{c} war \leftarrow alive(King), influenced(King, X); \\ influenced(King, X) \leftarrow alive(X), wants\_war(X); \\ wants\_war(General); \end{array}\right\}$$

The program P_0 contains the initialization of the game. P_1 contains the description of the current state. P_g are the common engine rules for the game mechanics. P_q contains the description of a quest to prevent a kingdom to go into a war.

Example 2.

$$P_0 = \{\ \dots\ alive(King);\ alive(General);\ speak(King);\ speak(General);\ \dots\}$$

$$P_1 = \{\ cast\_spell(Player, Silence, General);\ \}$$

$$P_g = \left\{\begin{array}{c} \dots\ \text{not } alive(X) \leftarrow killed(Y,X); \quad alive(X) \leftarrow ressurect(Y,X);\ \dots \\ \text{not } speak(X) \leftarrow spell(Silence, X); \dots \end{array}\right\}$$

$$P_q = \left\{\begin{array}{c} war \leftarrow alive(King), influenced(King, X); \\ influenced(King, X) \leftarrow alive(X), wants\_war(X), speak(X); \\ wants\_war(General); \end{array}\right\}$$

This example shows the advantages of LP representation. A small change (the prerequisite to speak to be able to influence) adds more possibilities of solving a quest. While this is nothing monumental, adding same extensions to scripts might easily become unbearable.

5 Implementation considerations

In this section we put forward a few implementation specific considerations. An implementation of the framework described in this paper consists basically of two parts: a software architecture to allow queries to the DynLoP engine from the game engine and the representation of game mechanics and quest information written as a set of dynamic logic programs. A most simple approach to this would take an existing game engine and extend its scripting language with calls to the LP system.

Two most popular systems for Logic Programming are DLV[5] and SMODELS[13]. Both of these systems support Logic Programs with both kinds of negations and also disjunctive programs. ¿From an evaluational point of view, a possibility to switch between these implementations is important. Unfortunately these systems differ slightly in syntax of the input programs and thus switching between them as the backend engine for LP solving is not straightforward. A class of Logic Programs that has common syntax in both systems has to be chosen or a transformation of the input programs for each syntax has to be implemented.

Many game engines are developed as open source projects, thus making it feasible to add new functionality to their scripting languages. Such project also frequently include relatively small examples of games which can be built on them. Such games present a better opportunity to use logic programming, than working with large games or starting a new game from scratch.

Examples of such open source engines include Adonthell[1] or GemRB[8]. The first is an example of a pure general engine, the latter being a free remake of commercial Infinity engine used Black Isle and Bioware games.

The implementation should make it possible for the designer to independently create a set of logic programs with a defined ordering – background knowledge, common quest information, quest specific programs. The scripting interface should then allow to designate a subset of the programs and ask the LP engine for results. These should be provided in the form of a number of all models and a number of models satisfying the queried atoms.

6 Conclusion

We described the world of (a certain kind of) a computer game from the point of view of a multi-agent system and provided a framework for its evaluation using dynamic logic programming. The main goal of this framework is to find a way to incorporate logic programming into game engines thus allowing the research of more complicated uses of LP.

The framework is very simple and can be easily added on top of existing scripting systems in game engines, allowing game designers to use the full power of LP when identifying situations in the game world. Being a simple and straightforward application of DynLoP in game design, it opens possibilities for further research.

The framework addresses static knowledge about the game world observed in a certain moment. One area of possible improvement would thus be to consider the dynamic game world and formalism as temporal logic.

Another natural area to explore is LP based planning in a game world. Even the simplest tasks in computer games, such as pathfinding (calculating a path for a character to take through the game world, avoiding obstacles) provide interesting challenges. Such a world with NPCs and other active objects is then an excellent place to research planning and decision making.

Acknowledgements: This work was supported by Research and Development Support Agency under the contract No. APVV-20-P04805. We further acknowledge financial support from the grant No. 314/2005 awarded by Comenius University and VEGA grant No. 1/0173/03.

References

1. Adonthell Game Engine, `http://adonthell.linuxgames.com/`
2. J.J.Alferes, J.A.Leite, L.M.Pereira, H.Przymusinska, and T.C.Przymusinski. Dynamic logic programming. In Procs. of KR'98. Morgan Kaufmann, 1998.
3. J. J. Alferes, F. Banti, A. Brogi and J. A. Leite. The Refined Extension Principle for Semantics of Dynamic Logic Programming. Studia Logica 79(1): 7-32, 2005
4. J.J.Alferes, J.A.Leite, L.M.Pereira, H.Przymusinska, and T.C.Przymusinsky. Dynamic updates of non-monotonic knowledge bases. The Journal of Logic Programming, 45(1-3):43-70, September/October 2000

5. A disjunctive datalog system. `http://www.dbai.tuwien.ac.at/proj/dlv`.
6. T.Eiter, M.Fink,G.Sabbatini, and H.Tompits. On Updates of Logic Programs: Semantics and Properties. INFSYS Research Report 1843-00-08, 2001.
7. T.Eiter, M.Fink,G.Sabbatini, and H.Tompits. On properties of update sequences based on causal rejection. Theory and Practice of Logic Programming, 2(6), 2002.
8. GemRB Game Engine, `http://gemrb.sourceforge.net/`
9. J.A.Leite. Evolving Knowledge Bases, volume 81 of Frontiers in Artifical Inteligence and Applications. IOS Press, 2003.
10. J.A.Leite, L.M.Pereira. Generalizing updates: From models to programs. In Procs. of LPKR'97, volume 1471 of LNAI. Springer Verlag, 1997.
11. J.A.Leite, L.M.Pereira. Iterated logic program updates. In Procs. of JICSLP'98. MIT Press, 1998.
12. L.Padovani, A.Proverti. Qsmodels: ASP Planning in Interactive Gaming Environment. In Procs. of the 9th European Conference on Logics in Artificial Intelligence (JELIA'04), Springer-Verlag, LNAI 3229, 2004.
13. The SMODELS system. `http://www.tcs.hut.fi/Software/smodels/`.