Logic Programming and Knowledge Representation in Computer Games

Jozef Šiška, Michal Turček and Milutín Krištofič

KAI FMFI UK, Mlynská dolina, 842 48 Bratislava
siska@ii.fmph.uniba.sk

Abstract. Certain kinds of computer games (especially role-playing and adventure games) offer an exact, coherent and relatively small and simple world description. This includes the background knowledge about the world in question, static description of the world state and description of actions and their consequences. Such system can be seen as a multi-agent system and presents a variety of knowledge representation problems.
Logic Programming and specifically Answer Set Programming are formalisms for knowledge representation and reasoning based on classical logic. They allow a clean and declarative characterization of many KR based problems.
In this article we present two applications of Logic Programming in the area of computer games: a simple interactive game and an ASP based world evaluation module for an existing game. We also present a framework for dynamic modularization of logic programs and describe the implementations.

1 Introduction

The aim of this work is to present two applications of Logic Programming in the area of computer games. First is a simple interactive game used to test the ability of ASP to describe game world and to help development of such descriptions. The second is an ASP based world evaluation module for an existing game, that enhances a computer game scripting engine with the ability to declaratively describe the game world and quests. We also present a framework for dynamic modularization of logic programs, which splits the programs into modules and allows to dynamically specify when are they to be included. Finally we describe the implementations being developed.

Logic Programming (LP) is one of the knowledge representation formalisms that are strongly bound to classical logic. It provides the ability do declaratively specify knowledge and problems. With the addition of negation as failure it is also able to do non-monotonic reasoning. Although logic programming is aimed

* We acknowledge support from Slovak Agency for Promotion of Research and Development projects, contract no. APVV-20-P04805 and APVV-99-PO5005; and project VEGA of Slovak Ministry of Education and Slovak Academy of Sciences no. 1/0173/03.

primarily at the representation of static knowledge, it had been used in many approaches to define dynamic domains such as planning and multi-agent systems. Several extensions to logic programming have been also proposed to allow a more simple description of such dynamic domains.

Computer games have used the most recent advancements in computer graphics and physics simulations. However the area of *artificial intelligence* in games (meaning planning and deciding for the computer controlled objects – items, computer opponents, ...) is not very advanced. Average game producers care about things that sell the games – nice graphics or realistic physics of the world. The AI part of the game started to get attention only recently. Although it is very easy to identify problems from the areas of multi-agent systems and knowledge representation in some kinds of computer games, theoretical knowledge from these areas is scarcely used.

Every computer game takes place in a specific world. Some kinds of games, mainly role-playing (RPG) and adventure games are based primarily on player's interaction with such a world. Creativity plays the main part in such interaction. The more ways to finish a task there are (and the more creative and full of fun), the better the game is. It is therefore a constant goal to make the game world as open and generic as possible.

Most games are usually scripted using game engines, which provide communication with the player and a scripting interface. The more complex the quests are, the harder the creation of a game becomes. It is very common that a game does not consider a task finished even when it is from the players perspective. Such *scripting glitches* then negatively influence player's experience and perception of the game. Declarative description of such world and tasks can therefore be much easier to create and maintain.

The paper is organized as follows: in sections two and three we present the logic programming formalism and a basic description of computer games. Section four describes an ASP based game that allows to independently develop, test and evaluate declarative descriptions of game worlds. Section five describes the application of such declarative description in an existing game engine. Section six presents a dynamic modularization of logic programs that reduces the complexity of evaluated programs and shortly describes the implementations. The final section includes comparison to other work and possible future directions.

2 Logic Programming

In this paper we will concentrate on a class of logic programs called *Dynamic Logic Programs*[8] which are a generalization of Logic Programs. First we recall some basic concepts and notation. Then we characterize some classes of logic program 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 $\mathcal{L} = \{A, \mathbf{not}\ A | A \in \mathcal{A}\}$ the set of all literals. For a literal $L \in \mathcal{L}$, we use **not** L to denote its *opposite* counterpart. Thus $\mathbf{not}\ L = \mathbf{not}\ A$ iff $L = A$ and $\mathbf{not}\ L = A$ iff $L = \mathbf{not}\ A$. We call L and **not** L *conflicting* literals and denote it by $L \bowtie \textbf{not}\ L$. 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. When 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 $\text{head}(r)$. Similarly by $\text{body}(r)$ we denote the set $\{L_1, L_2, \ldots, L_n\}$ and call it the *body* of r. If $\text{body}(r) = \emptyset$ then r is called a *fact*. Two rules r and r' are called *conflicting* (opposite) if $\text{head}(r)$ and $\text{head}(r')$ 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 $\textbf{not}\ A \in I$. 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 $I \models \text{body}(r)$ implies $I \models \text{head}(r)$. A total interpretation M is called a model of 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 $\text{least}(P)$ the least model of the definite logic program P' obtained from P 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

$$M = \text{least}(P \cup M^-).$$

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 ovverides 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 $\mathcal{P} = (P_1, P_2, \ldots P_n)$. 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 stable model of $\mathcal{P}$ iff

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

where

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

The set $\mathit{Rej}(\mathcal{P}, M)$ contains all *rejected* rules from $\mathcal{P}$ i.e. rules for which there is a conflicting rule satisfied in M in more important program. The set $\mathit{Def}(M)$ 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, 8, 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[8]. 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 solutions for LP.

3 Computer games

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.

Although all computer games take place in a certain game world, there are types of games which highly depend on a really open and *intelligent* interaction of the player with such a world. These include mostly Role Playing Games (RPGs) and Adventure games. These are the games we are mostly interested in this paper. It is clear that such *intelligent interaction* requires a good ability of the game engine to reason about the game world and thus an appropriate knowledge representation of the world. Therefore in this paper we provide an overview of a game (game engine) from the side of the game world representation and common tasks that are required to be done over it.

In an RPG game, the player assumes the role of a fictional character, guiding it through the game world, solving quests and task and building a storytelling background for his character. An adventure game is very similar to an RPG game, but the action elements are more reduced and the game is more based on puzzle-solving and interaction with the surrounding world. One of the main distinction between RPG and adventure games is a very detailed system describing how the player's character evolves and improves through the course of an RPG game. On the other hand in adventure games the abilities of a player character do not change that much and are not that important during the game.

3.1 Game engines

Computer games are not written from scratch. Computer game engines are used, which contain various common parts used in multiple games and final games are built on top of them. Existing game engines vary in the level of generalization, from very specific with most of the game mechanics and other things hardcoded, to general engines resembling only an user 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 requires a complex script. The quality and consistency of these scripts then forms a major factor of games' quality and impression.

There are two levels of game mechanics in an RPG game engine: one responsible for technical gameplay and a roleplaying part that creates an illusion of story and meaning in the game. The first one is very simple: moving around the world and performing few predefined actions or using dialogs, where user chooses one of the presented options. Game designers than use the scripting abilities of the game engine to give meaning to these simple actions and create the storytelling layer presenting the player with a story behind the game.

The game engines always provide some kind of interface to the representation of the game world ranging from direct access to the whole knowledge about the world to specifically prepared and processed knowledge for a specific NPC. They also offer different kinds of possible actions to be accessible from the scripting interface. These include ability to directly change the world representation, limited ways of changing properties of game objects or selecting actions for computer controlled characters.

3.2 World of a computer game

World of RPG and adventure games consist of *objects*, such as player character, non-player characters (NPCs), items or even abstract items (sounds,light). Objects can be *passive* or *active*. Active objects can execute *actions* and passive objects are only 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: specific properties are recorded on some items and they are checked later in the game.

Because of the size, game 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.

A set of rules that describes the basic events, actions and interactions int the game world is often referred to as game rules or game mechanics. Specific games (including game story and quests) are usually scripted on top of these. Given the base mechanics of the game, the game world is exactly described. Such world is relatively small compared to real world applications and is scarcely ambiguous.

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

4 An ASP based game

Because the creation of the logic programs that describe game mechanics and quests is not an easy task, a project was started to allow easy, incremental way of creating and testing ASP-based representation of a computer game world – a small computer game using exclusively ASP and a simple interpreter and user interface[17].

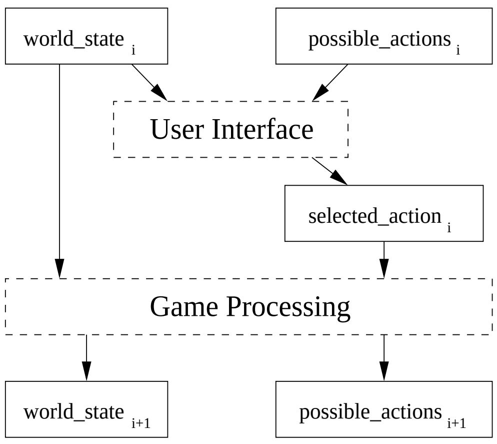

Fig. 1. Iterative game processing

In the most simple case the interpreter constructs a logic program (or a dynamic logic program) from the description of the current state of the game world, the players action and the game and evaluation rules. It then computes the answer sets of this program, selects one of them and extracts the new world state and the list of possible actions. The interpreter then presents these actions and the description of this world state to the player, who selects one of the actions. After this the whole process, as shown in fig 1, repeats itself.

The state of the world is represented using fluents in a similar way as in action languages and planning. Actions are also used in almost the same way as in planning and one iteration of the game resembles very closely one step in the inference of the results of an action in planning problems.

The evaluation of the world state is currently strictly reactive, meaning that effects of players actions including the actions of any active in-game objects are computed directly in one step. There are plans however to embed a simple planning framework in the respective ASP representation to allow simple planning.

5 Game world evaluation

One of the common knowledge representation tasks in a computer game is the evaluation of the state the game world is in. The game engine has to decide if the player has achieved his goals (quests) or if some objectives were fulfilled that allow the game to advance.

This may seem easy and trivial in most games, but there are games which depend on an open interactivity between the player and the world. In such games a multitude of possibilities that satisfy the objectives can arise and scripting them all cat get very hard. Declarative logic programming can thus present a viable alternative to scripted evaluation. Declarative descriptions of a goal would be much more concise and easier to maintain.

On the other hand, evaluation of the world state is a relatively simple. The encoding of the relevant (static) knowledge about the world can be easily done through logic programs and posing queries to such knowledge base is easy. Because of this such an application presents a perfect opportunity for starting the use of LP in computer games, possibly extending it to more dynamic applications like planning the actions for computer controlled characters.

Application of this kind has been proposed in [16] along with an experimental implementation in [18] and we will include only a short characterization here. The main goal is to enhance a scripting subsystem of a game engine allowing scripts to pose DynLoP-based queries of the world state and use them to decide fulfilment of objectives or quests. An overview of such system can be seen in fig. 2.

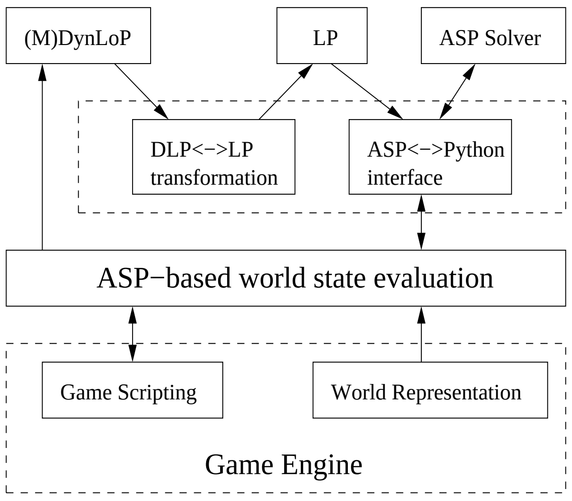

Fig. 2. DynLoP based world state evaluation in a computer game

Whenever a query from the scripting system is generated, a DynLoP program is assembled in the world state evaluation module from the game rules, quest description and the current game world data. This DynLoP program is then transformed into a LP program according to the transformational semantics and passed to the ASP solver. Models of this programs are read by the ASP-python interface module and the result of the query is passed back to the scripting system.

Using the expressiveness of DynLoP queries can be easily represented by dynamic logic programs describing the knowledge about the world, game mechanics and quests in a hierarchic way. The semantics of DynLoP then can automatically resolve possible conflicts between specific quests, core game mechanics and background knowledge. This way, although the base game rules would state that dead people cannot talk, the game designer creating a specific quest concerning ghosts can easily and consistently override that for this single quest.

Such quest description can be represented by a dynamic logic program $\mathcal{P} = (P_0, P_1, P_2, \ldots, P_n, P_g, P_q)$ and a set of atoms Q_a. 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 – whether they are modeled by some or all models of $\mathcal{P}$.

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

$$P_0 = \{ \quad \ldots\ \mathit{alive}(\mathit{King});\ \mathit{alive}(\mathit{General});\ \mathit{speak}(\mathit{King});\ \mathit{speak}(\mathit{General});\ \ldots \quad \}$$

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

$$P_g = \left\{ \begin{array}{c} \ldots\ \mathbf{not}\ \mathit{alive}(X) \leftarrow \mathit{killed}(Y, X); \qquad \mathit{alive}(X) \leftarrow \mathit{ressurect}(Y, X);\ \ldots \\ \ldots \mathbf{not}\ \mathit{speak}(X) \leftarrow \mathit{spell}(\mathit{Silence}, X); \ldots \end{array} \right\}$$

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

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.

Seeing that the game mechanics contains rules about the ability to speak, we can specify our quest more precisely by a small change:

$$P_q = \{\ldots \mathit{influenced}(\mathit{King}, X) \leftarrow \mathit{alive}(X), \mathit{speak}(X), \mathit{wants\_war}(X); \ldots\}.$$

which only states the obvious need to be able to communiacate with someone to influence him. Now the quest can be solved by other means, such as $\mathit{castSpell}(\mathit{Player}, \mathit{General}, \mathit{Silence})$.

6 Modularization of logic programs and Implementation

In this section we first describe a framework that splits logic programs into modules and allows them to be loaded only when they are needed. Then we present a short description of the implementation of presented applications.

6.1 Modularization

Although game worlds are simpler and smaller than real world they are still relatively large. Computer game programmers came up with different techniques to overcame the difficulties of a large game world. Most simple and efficient of these has been introduced in section three: the world is divided into multiple parts: levels, modules, areas or locations and only one or some of these parts are active at a time. Many current games use this technique together with an on-demand, in-background loading of new parts to provide an illusion of a continuous game world.

Modularization is also an interesting concept in logic programming. In the simplest way modules represented by different logic programs are just joined together to form one big logic program. Theoretical research has been done in different directions regarding modularization, whether to resolve conflicts between modules or to define certain interfaces for modules to improve communication between them.

We describe here a simple framework that allows to split logic programs into different modules and to declaratively specify when are they to be loaded. There are two main approaches that are considered: an iterative approach and a simple two-level approach. Both approaches are based on the introduction of a special predicate, say *load_module*/1, whose presence in a stable model of program states that a specific module should be loaded.

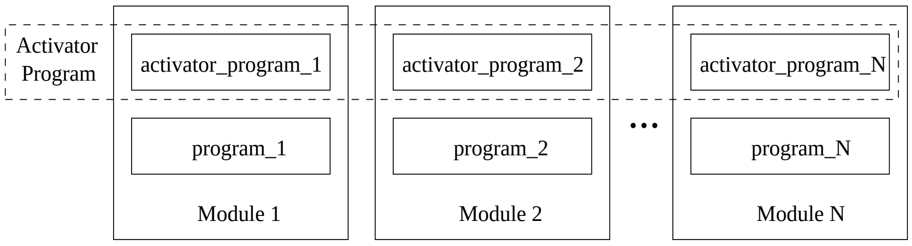

Fig. 3. Modules

In the first approach a certain initial program is taken (a certain module) and its stable models are computed. If a model contains the predicate *load_module*(X) the modules X is added to the program and the process is iterated over. Because a program can have multiple stable models, either one

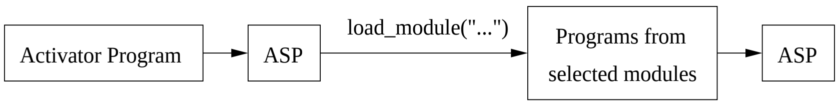

Fig. 4. Modules Processing

can be chosen, or all possibilities can be exploited in parallel depending on the application.

In the second approach a module consists of two programs: a main program and an activator program as shown in fig. 3. First the activator programs from all modules are put together and executed with a query. The stable models are again used to decide which modules should be loaded. In second step the selected modules are added together and the query is executed as shown in fig. 4.

6.2 Implementation

In this section we provide a short technical description of the implementations of the presented applications. An implementation of the game evaluation module from the section five is described [18] and a simplified implementations of the modularization framework is in [19]. These lead to the creation of a general python package pyASP[13].

The choice of the Python language was influenced mainly because it is often included as a scripting languages in game engines and other projects[1]. The package supports DLV [5] and SMODELS [15] as possible ASP solvers. Although both these systems provide C++ or Java interfaces, we decided to use them as standalone programs and read the models through shell pipes from their standard output. This offers greater flexibility for example to add new solvers.

The package provides a class LPParser, which can be initialized with input files and solver options (which specify the solver and its special options). It can be either used to load whole models into memory, callback functions for different atoms can be defined and they will be called as the model is parsed.

The two step modularization approach described above is implemented by a class MLPParser derived from LPParser. It expects input files containing the activator and module part and hides the process of evaluating the activator files and selecting the correct modules.

A simple web-based implementation of the ASP based game described in section four is available at [17]. The non-ASP part consists of two simple scripts `eddom.py` and `eddom.php`. The first one uses an instance of LPParser from pyASP to calculate a stable model of the game description, saved game state and a given action, prints the relevant information and possible actions obtained from corresponding atoms and then saves the games state to a temporary file. The second script formats the output as a web page and manages different prefixes to temporary files to allow concurrent usage.

7 Conclusion

We described two possible applications of Logic Programming and Answer Set Programming in the area of Computer Games. The first one is a simple reactive game created to experiment with game world descriptions in ASP. Second presented application is a game world evaluation framework that can be used to enhance the scripting subsystem of a computer game with the ability to declaratively query the representation of the game world.

We also presented a simple modularization framework, that allows on-demand loading of ASP modules and a short description of the implementation.

There has been other work in the area of Logic Programming that uses computer games as an application field [14, 12] Thinking of a computer game as a multiagent system, these approaches try to enhance the agents with reasoning abilities The game engine – multiagent framework – that coordinates these agents remains unaware of such abilities. In our approach we try to enhance the engine itself, thus giving it better control of the game world.

In our approach we use Dynamic Logic Programming to handle hierarchical knowledge. In other approaches, such as Evolp, DynLoP is used to simulate the change of knowledge over time, especially in planning. Presented applications work with statical knowledge – description of the actual state of the game world. A natural area for future work is therefore the possibility to to support planning and thus control of the objects(agents) in the game world. This can be interesting especially in connection with Evolp and Multi-dimensional DynLoP[9, 8].

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. J.A.Leite. Evolving Knowledge Bases, volume 81 of Frontiers in Artifical Inteligence and Applications. IOS Press, 2003.
9. J.A.Leite, J.J.Alferes, and L.M.Pereira. Mutli-dimensional Dynamic Knowledge Representation. In Procs. of LPNMR'01, volume 2173 of LNAI, Springer, 2001
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. J. Leite and L. Soares. Evolving Characters in Role-Playing Games, In R. Trappl (ed.), Cybernetics and Systems 2006, 18th European Meeting on Cybernetics and Systems Research (EMCSR 2006), vol 2, pp. 515-520, Vienna, Austria, Austrian Society for Cybernetic Studies, 2006
13. PyASP. Python package implementing interface to ASP solvers and various functions, http://ii.fmph.uniba.sk/ siska/pyasp/
14. 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.
15. The SMODELS system. http://www.tcs.hut.fi/Software/smodels/.
16. J. Šiška. Dynamic Logic Programming and world state evaluation in computer games. In Procs. of WLP06, 2006.
17. EDDOM. An attempt at a game written (almost) entirely using an ASP solver, http://www.ii.fmph.uniba.sk/ siska/eddom/.
18. M. Turček. Application of Dynamic Logic Programming in Evaluation of Computer Games World State. Master thesis, Comenius University, Faculty of Mathematics, Physics and Informatics, Bratislava, June 2007.
19. M. Krištofič. Modulárna implementácia hier v Answer Set Programovaní. Bakalárska práca , Comenius University, Faculty of Mathematics, Physics and Informatics, Bratislava, June 2007.