Institute of Applied Informatics
Faculty of Mathematics, Physics and Informatics
Comenius University, Bratislava

Application of Dynamic Logic Programming in Evaluation of Computer Games' World State

(Diploma Thesis)

Michal Turček

Supervisor:
Mgr. Jozef Šiška

Bratislava, 2007

I honourably proclaim, that I wrote the thesis by myself and I did not use any other than referenced resources.

. . . .

I would like to thank my supervisor Mgr. Jozef Šiška for such interesting thesis theme and his guidance during this work. Also, I would like to express my gratitude to a friend, Mgr. Tomáš Záthurecký. Not only for providing development environment and overall technical support but for all his help through the course of my studies.

Abstract (English)

Application of Dynamic Logic Programming in Evaluation of Computer Games' World State

Aim of this work is to make use of Dynamic Logic Programming (DynLoP) in making of computer games. More specifically - to enhance some existing game creation toolkit (game engine) with the capability of employing DynLoP for evaluation of game-world state. This should be an implementation of the original idea by Mgr. Jozef Šiška [1].

key words: knowledge representation, computer games, Dynamic Logic Programming

Abstrakt (Slovensky)

Využitie Dynamického Logického Programovania pri vyhodnocovaní stavu sveta v počítačových hrách

Práca si kladie za cieľ umožniť uplatnenie Dynamického Logického Programovania (DynLoP) pri tvorbe počítačových hier. Presnejšie - rozšíriť nejakú existujúcu sadu nástrojov na tvorbu hier (herný engine) o možnosť použiť DynLoP pre vyhodnocovanie stavu herného sveta. Ide o implementáciu pôvodnej ideii Mgr. Jozefa Šišku [1].

kľúčové slová: reprezentácia znalostí, počítačové hry, Dynamické Logické Programovanie

Contents

Chapter 1

Introduction

1.1 Motivation

What we do in this work, is making a connection between computer games and knowledge representation theories. The reason behind is that these two areas might be significantly beneficial to each other.

How could computer games profit from theoretical artificial intelligence (AI) is quite straightforward. Trends in game industry have so far lead us to moment, where virtual worlds are an eye-candy. Visuals are rendered in realistic graphics. Objects in game environments react accordingly to the laws of physics. Everything is cast in perfect animations and surround sound. What often spoils the overall impression is the intelligence of computer controlled agents. It is disturbingly mechanical, too simple or even flawed. Very frequently it does not correspond to the expected behaviour. This situation is unarguably caused mainly by the fact the intelligence can not be promptly perceived by the main sense - vision. And that is of course what sells the game. Another reason might be the lack of more general tools for programming AI, so the development would be rather costly. This is where we might help. In comparison, for each of the features already mentioned (graphics, physical simulations, sound) there exist both software libraries (OpenGL, DirectX, etc.) and dedicated hardware. Exactly these hardware accelerator cards make using Central Processing Unit for computing artificial intelligence possible.

The opposite direction - games helping science - is even more important. Application of theoretical models into practical use is always of major significance. Game development might prove to be another such source of valuable feedback. Logic-based representation of dynamic knowledge is still relatively young, evolving concept. As such, providing testbed and playground for people studying it might be helpful. Particularly for the research of its various semantics, their correspondence to intuition, etc. But also more generally, for problem solving by means of Dynamic Logic Programming (DynLoP). The reasons for using computer games for this purpose is that game worlds could serve as an adequate approximation of the real one. It is easy to work with them for their simplicity and always available, exact, explicit, formal description (given by the set of rules of that particular game). On the other hand, game worlds could be complex enough at the same time, to make the results relevant for real world applications.

1.2 Summary of Goals

- development of framework for world state evaluation (WSE), that wold utilize Dynamic Logic Programming
- embedding this WSE unit into an existent game creation toolkit (game engine)
- provide usage demonstration example in actual computer game built on this engine

1.3 What Follows

Before describing our practical implementation, we present a theoretical view on the subject in the following three chapters.

Chapter 2 formally introduces Logic Programming, as a prerequisite for chapt. 3 which deals with Dynamic Logic Programming. These two can be skipped if reader is already familiar with the addressed topics.

Chapter 4 describes on what our focus in computer games lays on. What knowledge representation problems we try to face and how could DynLoP be used to improve the currently most used solutions. Also we take a look at projects similar to ours. That means those, somehow involving both knowledge representation theory and computer games.

Finally, the last chapter covers the actual implementations description.

Chapter 2

Logic Programming

For the purpose of self containment of this thesis we present the knowledge representation formalism called logic programming (LP) here. Note that we aim to describe the class of logic programs that our implementation works with, although the deviations from the standard definitions are only of cosmetic nature. However, the exact syntax depends a lot on the ASP solver used (more in sec. 5.1.2). We provide only the needed basics - more on the subject of LP and DynLop can be found for instance in [3].

2.1 Syntax

Definition 2.1.1 *(Term) A term is either a variable or a constant, where both are symbols drawn from the disjoint sets of variable and constant symbols correspondingly.*

Definition 2.1.2 *(Atom) An atom is an expression of the form* $p(t_1, t_2, \ldots, t_n)$*, where* p *is an predicate symbol with arity* n *and each* t_i *is a term. An atom is called* grounded *if it does not contain any variables.*

The set of all grounded terms is called the *Herbrand universe*. The *Herbrand base* is the set of all grounded atoms.

Definition 2.1.3 *(Literal) A literal is any of these:*

- objective literal - *an atom:* A
- default literal - *an atom with default negation: not* A

Notation 2.1.4 Two literals A, B are called *conflicting* if one of them is negation of the other one, that is either $A = not\ B$ or $B = not\ A$. We denote this fact by $A \bowtie B$

Notation 2.1.5 For a set of literals M we denote by M^+ the set of all objective literals from M and by M^- the set of all default literals from M.

Definition 2.1.6 *(Logic Program) A logic program is a countable set of rules of the form:*

$$L_0 \leftarrow L_1, L_2 \ldots, L_n.$$

where each L_i is a literal. If $n = 0$ we call this rule a fact *and denote it simply:*

$$L_0.$$

Intuitively, every rule represents an implication $L_1 \wedge L_2 \wedge \ldots L_n \Rightarrow L_0$.

Notation 2.1.7 Let r be a rule $L_0 \leftarrow L_1, L_2 \ldots, L_n.$, then by $head(r)$ we refer to L_0 and by $body(r)$ to the set $\{L_1, L_2 \ldots, L_n\}$.

Notation 2.1.8 Two rules r_1, r_2 are called *conflicting* iff $head(r_1) \bowtie head(r_2)$. We denote this fact by $r_1 \bowtie r_2$

2.2 Semantics

Definition 2.2.1 *(Herbrand interpretation) A Herbrand interpretation of a logic program is any subset of its Herbrand base.*

An interpretation I is *minimal* among the set S if there does not exists any $J \in S$ such that $J \subset I$. We say an interpretation I is *least* among the set S if for all $J \in S$ holds $I \subseteq J$.

Notation 2.2.2 A literal L is *satisfied* in an interpretation I (denoted by $I \models L$) iff either L is objective and $L \in I$ or L is defuault (i.e. $L = not\ A$) and $A \notin I$.

A set of literals S is satisfied in I ($I \models S$) iff $\forall L \in S; I \models L$. We also say that I models S.

A rule r is satisfied in I iff $(I \models body(r)) \Rightarrow (I \models head(r))$.

Definition 2.2.3 *(Herbrand model) A Herbrand model of a logic program P is a Herbrand interpretation of P such that it satisfies all rules in P.*

2.2.1 Answer Set Programming

All the logic programming used in this thesis is based on the Answer Set semantics. Therefore, we now proceed to enumerating some basic (syntactical) classes of logic programs and defining their stable models - *answer sets.* We begin with the most simple ones and continue towards the more general ones. Whenever we use the term model, except in formal definitions in chapt. 2 and 3 we mean an answer set.

Definite logic programs

Definition 2.2.4 *(Definite logic program) ...is a countable set of rules of the form* $L_0 \leftarrow L_1, L_2 \ldots, L_n$. *where each* L_i *is an atom.*

Definition 2.2.5 *A stable model (i.e. answer set) of a definite logic program* P *is the least model of* P*.*

Normal logic programs

Definition 2.2.6 *(Normal logic program) ...is a countable set of rules of the form* $L_0 \leftarrow L_1, L_2 \ldots, L_n$. *where* L_0 *is an atom and each* $L_{1 \leq i \leq n}$ *is a literal.*

Definition 2.2.7 *(Gelfond-Lifschitz operator) Let* P *be a normal logic program and* I *an interpretation. The GL-transformation of* P *modulo* I *is the program* P^I *obtained from* P *by:*

- *removing each rule containing a default literal not* A *such that* $A \in I$*.*
- *removing each literal not* A *such that* $A \notin I$ *from remaining rules.*

Since the resulting program P^I *is a definite program, it has a unique least model. We define* $\Gamma(I) = least(P^I)$*.*

Definition 2.2.8 *An interpretation* M *of a normal logic program* P *is a stable model of* P *iff* $\Gamma(M) = M$*.*

Generalized logic programs

Definition 2.2.9 *(Generalized logic program) ...is a countable set of rules of the form $L_0 \leftarrow L_1, L_2 \ldots, L_n$. where each L_i is a literal.*

Definition 2.2.10 *Let $'$ be an operator that replaces all default literals not A by new atoms not_A. An interpretation M of a generalized logic program P is a stable model of P iff*

$$M' = least((P \cup M^-)')$$

Extended logic programs

Intuition behind the two types of negation is, that by explicit negation we claim that we have evidence that something is not true. Whereas by default negation we state that if no evidence is found to support a certain statement we assume it is false.

Definition 2.2.11 *(Extended logic program) ...is a logic program with explicit negation. An explicitly negated atom is called an* extended objective literal *($\neg A$) and not $\neg A$ is an* extended default literal.

Definition 2.2.12 *(Extended Gelfond-Lifschitz operator) Let P be a normal logic program and I an interpretation. The GL-transformation of P modulo I is the program P^I obtained from P by:*

- *removing each rule containing a default literal not A such that $A \in I$.*
- *removing each literal not A such that $A \notin I$ from remaining rules.*

Since the resulting program P^I is a definite program, it has a unique least model. If $least(P^I)$ contains a pair of complementary objective literals (A, $\neg A$), then $\Gamma(I) = Herbrand\ base$. Otherwise, we define $\Gamma(I) = least(P^I)$.

Definition 2.2.13 *An interpretation M of an extended logic program P is a stable model of P iff $\Gamma(M) = M$.*

Chapter 3

Dynamic Logic Programming

Dynamic Logic Programming (DynLoP) is an extension of the Logic Programming, designed for describing knowledge changing in time. Syntactically it is simply a sequence of logic programs. This represents the timeline of knowledge gathering. However it can be viewed generally as a sequence, where any information expressed on a certain position is strictly more important (update, overriding inconsistencies) than anything that precedes it. It might for instance mean knowledge acquired from different sources, sorted by their reliability.

3.1 Semantics

3.1.1 Dynamic Stable Models

Notation 3.1.1 Let $\mathcal{P} = (P_1, P_2, \ldots P_n)$ be a dynamic logic program, then by $\rho(\mathcal{P})$ we mean a logic program obtain by uniting all the rules from $P_1, P_2, \ldots P_n$.

$$\rho(\mathcal{P}) = \bigcup P_i$$

Definition 3.1.2 *(Dynamic Stable Model) An interpretation M is a stable model of a dynamic logic program $\mathcal{P} = (P_1, P_2, \ldots P_n)$ iff*

$$M = least([\rho(\mathcal{P}) \backslash Reject(\mathcal{P}, M)] \cup Default(\rho(\mathcal{P}), M))$$

where

$$Reject(\mathcal{P}, M) = \{r_i | r_i \in P_i, \exists r_j \in P_j, r_i \bowtie r_j, i < j, M \models body(r_j)\}$$

$$Default(\rho(\mathcal{P}), M) = \{not\ A | A \in \mathcal{A}, \nexists r \in \rho(\mathcal{P}), M \models\ body(r), head(r) = A\}$$

$\mathcal{A}$ is the set of all atoms from the DynLoP $\mathcal{P}$.

On the intuitive level, this definition says that we can test whether an interpretation M is a stable model by:

1. uniting all the updates (i.e. programs $P_1, P_2, \ldots P_n$)
2. removing all the rules that are in conflict with a more important rule (i.e. coming from a more recent update and having its body modeled by M)
3. adding facts *not* A for every atom A that is not in the head of any rule that has its body modeled by M.

3.1.2 Transformational Semantics

The previous declarative semantics provide us only with a way of testing whether a certain given interpretation is an answer-set. The need of having a method of computing the stable models is obvious. In this section we present a procedural semantics for DynLoP.

Definition 3.1.3 *(Dynamic Program Update) Let $\mathcal{P} = (P_1, P_2, \ldots P_n)$ be a dynamic logic program, let $T = \{1, 2, \ldots n\}$.*

Let K be the set of all atoms from $\mathcal{P}$. Let $\{A^-, A_s, A_s^-, A_{P_s}, A_{P_s}^- | A \in K, s \in T \cup \{0\}\}$ be a set of new atoms (i.e. disjoint with K).

By the dynamic program update over the sequence of updating programs $\mathcal{P}$ we mean the logic program $\biguplus \mathcal{P}$, which consists of the following rules:

(RP) Rewritten program clauses:

$$A_{P_s} \leftarrow B_1, \ldots, B_m, C_1^-, \ldots, C_n^-$$

$$A_{P_s}^- \leftarrow B_1, \ldots, B_m, C_1^-, \ldots, C_n^-$$

for any clause

$$A \leftarrow B_1, \ldots, B_m, not\ C_1, \ldots, not\ C_n$$

$$not\ A \leftarrow B_1, \ldots, B_m, not\ C_1, \ldots, not\ C_n$$

of $P_s \in \mathcal{P}$ respectively, where $s \in T$.

(UR) Update rules:

$$A_s \leftarrow A_{P_s}$$

$$A_s^- \leftarrow A_{P_s}^-$$

for all atoms $A \in K$ and for $s \in T$. (A is true/false in the state s if it is true/false in the program P_s)

(IR) Inheritance rules:

$$A_s \leftarrow A_{s-1}, not\ A_{P_s}^-$$

$$A_s^- \leftarrow A_{s-1}^-, not\ A_{P_s}$$

for all atoms $A \in K$ and for all $s \in T$. (A is true/false in the state s if it is true/false in the previous state $s-1$ and it is not forced to be false/true by the program P_s)

(DR) Default rules

$$A_0^-$$

for all atoms $A \in K$. (All objective atoms are initially false.)

Definition 3.1.4 *(Dynamic Program Update at a Given State) Given a fixed state $s \in T$, by the dynamic program update at the state s, denoted by $\biguplus_s \mathcal{P}$, we mean the dynamic program update $\biguplus \mathcal{P}$ with the following rules added:*

(CSs) Current State Rules:

$$A \leftarrow A_s$$

$$A^- \leftarrow A_s^-$$

$$not\ A \leftarrow A_s^-$$

for all atoms $A \in K$.

Theorem 3.1.5 *(Soundness and Completeness) Given a Dynamic Logic Program $\mathcal{P}$, the stable models of $\biguplus_s \mathcal{P}$, restricted to the original set of atoms, coincide with the stable models of $\mathcal{P}$ at state s.*

Chapter 4

World State Evaluation in Computer Games

The basic principle of games intended for playing solo or players versus environment (in contrast to player versus player) is accomplishing certain tasks, designed by the creators of the game. Our focus will be on so called adventure and role-playing games (RPGs). This is due to specific type of tasks that the player of such games needs to complete. These are commonly referred to as quests. The difference that make them suitable for our cause is their complexity - implying the effort needed to describe them and the difficulty of checking their completion. However the same concept, described later (sec. 4.3), might probably be successfully used for other types of games as well. Mainly strategic or more complex logical puzzles. By the term “world state evaluation” (WSE) we mean the process of obtaining truth value of arbitrary statements in the context of the in-game world (mostly for the sake of checking whether a certain quest or its part is solved).

4.1 Standard Approach

Tracking the player’s progress through the game involves maintaining a representation of the state the game world is in. The standard approach to do this uses very limited or none inference (reasoning by making conclusions) at all.

Typically for every quest there exists a set of player actions influencing the outcome of that particular quest. This set can be organized as a simple table (list) or some more complicated structure, for instance a tree (generally a graph). The structure represents mutual relations between these facts (pieces of knowledge). Also it often implicitly or explicitly codes the way how to compute the overall result of changing a single fact. When player executes any of those noteworthy actions a simple procedure derives the status of all the related information carrying nodes and ultimately of the quest itself. Nearly never does this process make use of any knowledge representation theories. Describing the game world and its mechanics (inference rules) in this fashion is rather cumbersome and is very frequently source of erroneous (from the players point of view) game behaviour.

In the rapidly changing market of computer games, eliminating the need of programming a game from scratch is essential. Therefore modular architectures and ready-to-use frameworks are a must. These reusable (and often subject to trade) parts of code and sets of development tools are called engines. They range from those for very specific tasks (for instance graphics engines) to the complete prefabricates of a whole game. However they rarely cover the areas of artificial intelligence or knowledge representation. These parts have to be programmed ad hoc. If a development instrument should be put to easy use, embedding it into a game engine is the right way to go. On the other hand for the sake of general applicability it should be also provided as a module usable in other engines or software projects. In this work we target both of these needs.

4.2 Overview of Related Projects

Except [1] (on which this thesis is based on) there are few other works concerning artificial intelligence and computer games. A brief description of those most closely related follows.

Qsmodels: ASP Planning in Interactive Gaming Environment

Qsmodels [8] is a software architecture, that is an interface between a game and *Smodels* [4] inferential engine. It is used for controlling an agent inside the game - an automatic player, often called a bot. Behaviour of the bot is determined by a planner written in the language of Logic Programming under the Answer Set semantics. This program is interpreted by the Smodels solver. The game is a real-time action *'Quake 3: Arena'* and Qsmodels is conceived as its modification.

Evolving Characters in Role Playing Games

In this work [9] authors enhance Java based Multi-Agent Platform madAgents with Evolving Logic Programming (EVOLP). MadAgents are then used in Role Playing Game engine to control the non-player characters (NPCs). EVOLP is a formalism descending from Logic Programming, intended to work with dynamic knowledge, evolving in time. These two attributes (RPG engine and EVOLP) make this project very similar to ours. The differences are as follows. Our focus lays on more basic, more general task than NPC behaviour - the underlying evaluation of game-world state. More than just the NPC activities could be built on this knowledge. The second distinction is that the used game engine (MulE - Multi-player Evolutive Game Engine) is for developing Massive Multi-player Online Role-Playing Games (MMORPGs). Their concept of thousands of players interacting with the same game world simultaneously somewhat limits the depth, volume (processing power issues) and scope (one player heavily influencing the game for all the others would not be desired) of advanced AI methods used within.

Proprietary Solutions

Recently, there is some noticeable raise of interest for artificial intelligence in the game industry. As a pioneer that made a significant advancement should be considered *'The Elder Scrolls IV: Oblivion'* [10]. It is a commercial computer game of the role playing (RPG) genre, released in 2006.

It exhibits certain features that might be achieved by similar means as we present. These features include most notably the non-player characters' reasoning, setting up daily schedules, designating goals and planning how to achieve them. The authors' term for this is *'Radiant AI'*. However, what actual techniques are hidden beneath this is not publicly known.

4.3 DynLoP Approach to WSE

Knowledge needed to decide whether a certain statement is true in the current state of the game, consists of description of the game mechanics plus the query and the state itself. In case of quest completion checking, in addition to general game mechanics we also need specific rules that drive this particular quest. The present state is defined by the initial game configuration and its updates. These updates are actions (or their most direct effects) that the player has performed until the moment of querying. Let's sum it up ...

1. general, background knowledge about game mechanics - rules of the game (P_G)
2. initial configuration (P_0)
3. sequence of configuration updates ($P_1, P_2 \dots P_n$)
4. description of the current context - quest (P_Q)
5. query (Q)

All this knowledge (general mechanics, quest rules, starting state, each individual action and the query) could be represented by separate logic programs. Furthermore, these programs can be sorted linearly according the increasing weight of the statements made within. That means if a statement from a latter program is in conflict with a former one, this should be resolved in favor of the more important one. Thus we obtain a dynamic logic program, that describes the current in-game situation.

$$\mathcal{P} = (P_G, P_0, P_1, P_2, \dots, P_n, P_Q)$$

What we are looking for is the occurrence of Q in the models of $\mathcal{P}$. By checking whether the query Q holds in all, some or none of the models we obtain the answer whether this statement is surely true, might be true (we do not have enough information) or is not true correspondingly.

There exists a transformation that can translate a DynLoP to an equivalent logic program (sect 4.2). After that, one of the existing ASP solvers (for instance DLV[5] or smodels[4]) could be used to obtain the models of $\mathcal{P}$.

Because updates of the world state ($P_0, P_1, P_2, \dots, P_n$) might be quite frequent we would like to remove them from our concept. Registering these changes (or their most direct consequences) is simple and every game already has a mechanism to handle them in place. So all we need to do is borrow the configuration (P_S) from this internal mechanism. This way we have simplified our dynamic logic program to:

$$\mathcal{P} = (P_G, P_S, P_Q)$$

4.3.1 Benefits

Aside from reasons stated in the introduction (sec. 1.1) we try to enumerate the major features of blending the classic WSE concept with Dynamic Logic Programming in this section. The idea is to unite the advantages of both these approaches:

1. swiftness of the traditional mechanisms for the frequent, simple tasks
2. power of DynLoP for the more difficult problems

Thanks to its declarative nature, description of game world (game rules, quests etc.) by means of ASP (DynLoP) has the following attributes:

1. It is much simpler and more intuitive (thus allowing more complexity and bigger scope)
2. less rigid - potentially providing more, different solutions to quests. Even such, that the designer did not think of explicitly. This relates to the next good property, that I would call...
3. robustness - less vulnerability to imperfections and bugs in the quest designs. There is no need for the creator of the quest to encode all the possible ways of finishing the quests. Instead he/she just describes the dependencies and interactions.
4. Easier modification and expansion.
5. It could be a layer that more intricate features might be based on - such as event occurrence detection, computer controlled characters' (so called NPCs - non player characters) decision making and behaviour or planning.

Chapter 5

Implementation

We start this chapter by outlining the general concepts of the implementatation. Then we describe our universal world state evaluator unit in the next section. Afterwards, we show how it was used to enhance the Adonthell [2] RPG engine. The last section presents a manual for the users of our software. It is a simple guide for the game designers, showing how to create quests specification in the language of Logic Programming.

The core of this project is programmed in Python [6] scripting language. This comprises a general world state evaluator, that could be used in any computer game or even similar programs. Everywhere, where we have to deal with evaluating statements in a formally describable world. In the most general point of view it can be employed as a Dynamic Logic Programming solver based on Smodels [4]. Python was chosen for several reasons. The most promiment one beeing that it is quite frequently used in rapid game development. Source code written in python is very clean, easily understandable, in comparison for instance with the C language. This is quite useful in making further modification easy.

Since there exist no Dynamic Logic Programming solvers suitable for our case, we resort to transformational semantics and consecutive solving of the resulting logic program by one of the available ASP solvers. Thus creating our own DynLoP solver.

As discussed in cha. 4, the best way of releasing a programming tool to aid in game development, wold be to embed it into a game engine. Adonthell is well suitable for our purpose. It is an open source single-player Role Playing Game engine. It is quite simple to make the integration easy. But fully functional, with demonstration game available, therefore well suited for

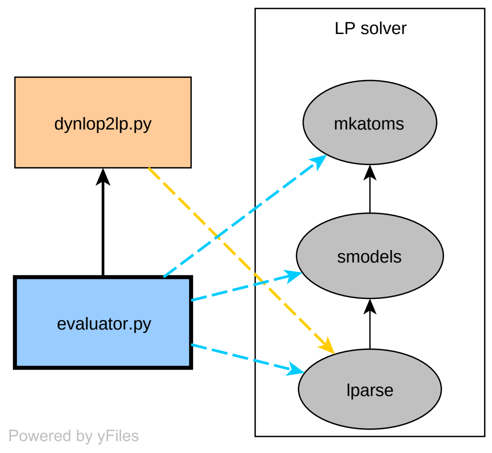

Figure 5.1: World state evaluator's component dependence scheme.

presentation purpose. What makes it particularly favourable is that games based on this engine are intended to be written in Python. Therefore it already contains strong support for this language.

5.1 World State Evaluator

One of the main goals behind this thesis was the development of a framework for world state evaluation that wold utilize Dynamic Logic Programming. Description of two python modules which together fulfill this task follows.

5.1.1 dynlop2lp.py - transformational semantics

This module's main purpose is to provide the Dynamic Logic Program to logic Program transformation. The transformation itself is described in sec. 3.1.2.

constants

- `_tmpfilename = '/tmp/dynlop2lp.tmp'` On a few occasions we need to save auxiliary data to a file. This says what file to use if not specified otherwise. Note that this low-level module does not delete its temporary file.

- `prefix = 'dynlop2lp'` Since we have to generate many unique literals, the most convenient way to achieve this is starting the identifiers with a string, that we hope will not be a prefix of any of the literals from input.

function: remove_explicit

Transforms a Logic Program into an equivalent without explicit negation. This is needed because transformational semantics do not handle explicit negation. All explicitly negated atoms $-A$ are translated into new atoms `prefix + '_expneg_' + A`.

arguments:

- `infilename`

- `outfilename` Name of file to save the output to. This form of output was chosen because the DynLoP to LP transformation is expected to have files as input.

function: get_atoms

For the transformational semantics we need to produce, among others, facts of form A_0^- for every atom A contained in the DynLoP to be transformed. This cannot be done with non-grounded atoms. One of the things that lparse (preprocessor for smodels) does is grounding all the atoms from input and making a list of them. Because lparse does not support default negation in rule heads, what we do is uniting all the logic programs and changing all default literals to objective ones. This way we get a logic program that will yield us the same set of grounded atoms as the original DynLoP.

arguments:

- `infilenames` List of the filenames containing Logic Programs, representing a Dynamic Logic program, that we need to ground.

- `tmpfilename= _tmpfilename` This is an auxiliary file that is fed to lparse for grounding.

return value: List of grounded atoms from the input files.

Table 5.1: Correspondence of new atoms produced by dynlop2lp function to the transformational semantics algorithm described in sec. 3.1.2

A	`A`
A^-	`prefix + '_n_model_'+ A`
A_s	`prefix + '_p_state_' + s + '_' + A`
A_s^-	`prefix + '_n_state_' + s + '_' + A`
A_{P_s}	`prefix + '_p_program_' + s + '_' + A`
$A_{P_s}^-$	`prefix + '_n_program_' + s + '_' + A`

function: dynlop2lp

Transforms a Dynamic Logic Program into an equivalent Logic Program, having the same set of stable models. It is a direct implementation of the algorithm from [3].

arguments:

- `infilenamea` List of the filenames containing Logic Programs, representing a Dynamic Logic program, that we want to transform.
- `outfilename` Name of file to which the resulting LP should be outputted.
- `exp_neg = True` Specifies whether the input logic programs use explicit negation. If they do not, setting this argument to false saves some processing time.
- `tmpfilename = _tmpfilename` This is used as an output for `remove_explicit` and as a temporary file for `get_atoms`.

5.1.2 evaluator.py - world state evaluation

The evaluator class is defined in this module. It is an universal world state evaluator and could also serve as a general python interface for Dynamic Logic Programming.

attributes

- `general_knowledge= None` List of names for files (interpreted as a dynamic logic program) that will be used for every evaluation by the `query` method as a background knowledge. Full path to files must be provided here to allow it to be located somewhere else as quest specific knowledge.

- `explicit_neg= True` Specifies whether explicit negation is used in any logic program evaluator deals with. If it is not, setting this argument to false saves some processing time.

- `reasoning= 'cautious'` The default reasoning style for `evaluate` and `query` metheds. Valid values are:

  - `'cautious'` True is returned only if query holds in all models of the world.
  - `'brave'` True is returned only if query holds at least in one model of the world.
  - `'precise'` Returns pair `(|good_models|, |all_models|)`, where `good_models` are those where query holds.

- `tmpfiles_prefix= ""` Location for temporary files, including trailing (back)slash.

- `tmpfilename_solve= 'evaluator_solve.tmp'` Name of temporary file used by the solve method.

- `tmpfilename_query= 'evaluator_query.tmp'` Name of temporary file used by the query method.

- `tmpfilename_status= None` The default name of file containing logic program, that describes the current status of the world. This is intended to be derived from the game's internal world status.

- `tmpfilename_dynlop2lp= 'evaluator_dynlop2lp.tmp'` Name of temporary file for dynamic logic program to logic program transformation (dynlop2lp.py module).

- `knowledgefiles_prefix= ""` Location of files with logic programs describing quests, including trailing (back)slash.

- `knowledgefiles_suffix= ""` Filename ending common for all quest descriptions (this is meant to allow omitting the extension when specifying file names).

method: __delete__

Deletes all the temporary files.

method: solve

Solves dynamic logic program specified by list of filenames. Note that after translating the DynLoP into a regular logic program, smodels solver is used to compute the models. Although other ASP solvers could be used for this purpose. For instance DLV [5].

arguments:

- `infilenames`

return value: Stream containing models of the inputted DynLoP. Formatted by MKAtoms [7], that means - every literal on a separate line, individual models divided by line with `'::endmodel'` string.

method: evaluate

Checks models of Dynamic Logic Program specified by infilenames for occurrence of `atom`, according to the desired reasoning mode.

arguments:

- `infilenames`
- `atom`
- `reasoning= None` Valid values are:
  - `None` Use the default mode, specified by object's `reasoning` attribute.
  - `'cautious'` True is returned only if `atom` is present in all the models.

  - `'brave'` True is returned only if `atom` is present in at least one of the models.
  - `'precise'` Returns pair `(|good_models|, |all_models|)`, where `good_models` are those containing `atom`.

method: query

Gives answer for the query. This method calls
`self.evaluate(infs, 'evaluator_good_model', reasoning)` and returns its result. `Infs` is a dynamic logic program assembled from:

1. `self.general_knowledge`
2. `knowledge`
3. `status`
4. `evaluator_good_model` ← `query`

by concatenating them in this order.

arguments:

- `knowledge` List of filenames containing logic programs (interpreted as a DynLoP) , describing the context this query refers to. Most commonly, this will be the quest, status of which we want to know. The attributes `knowledgefiles_prefix` and `knowledgefiles_suffix` are added to get the actual name (with location).
- `query` String `"a_1, a_2, ... a_n"`, where each `a_i` is a literal, interpreted as a conjunction $a_1 \land a_2 \land \ldots a_n$.
- `status= None` Name of file containing logic program, that describes the current status of the world. This is intended to be derived from the game's internal world status. If not specified, object's `tmpfilename_solve` attribute is taken as default value. The attribute `tmpfiles_prefix` is added to get the actual name (with location).
- `reasoning= None` If not specified, object's `reasoning` attribute is taken as default value.

5.2 Integration to Game Engine - Adonthell

In most usage situations it would be required (or at least desirable) to write a wrapper for the general world state evaluator (described in the previous section). This is needed to consistently incorporate our extension to the host engine and adopt its specifics. There are two major possibilities how to do this:

1. *Python* The most convenient way would be to write a subclass of the `evaluator` class. Redefining or adding methods as needed. Fig. 6.3 depicts a possible architecture based on this approach.

2. *language of host engine* This is the path that we chose and is described in the next few sections. A scheme of this architecture is shown on the fig. 6.2.

Where is which solution more appropriate differs from case to case. It depends mostly on factors such as the level of support for Python language in the host engine and the intensity of communication needed between the evaluator and rest of the engine (for instance for obtaining the current game-world status from the engine's original data structures). These inter-language calls should be minimized as much as possible because they are usually rather costly.

5.2.1 evaluator.cc - world state evaluation

This is an extension of the Adonthell engine. It allows describing geneneral background and quest specific knowledge in the language of Logic Programming. All this logic programs can refer to internal quest status representations. Evaluator can then give answers whether certain statements are true or not. This can be used to drive the course of the game.

In the `evaluator.h` header file the `evaluator` class is defined. This is a wrapper for the general python evaluator class from `evaluator.py` (sec. 5.1.2). Its purpose is to make calls to its methods from the Adonthell engine easy, as well as adding code to handle Adonthell specific features. It also adds `WSeval`, which is a pointer to instance of the `evaluator` class, to the `data` namespace.

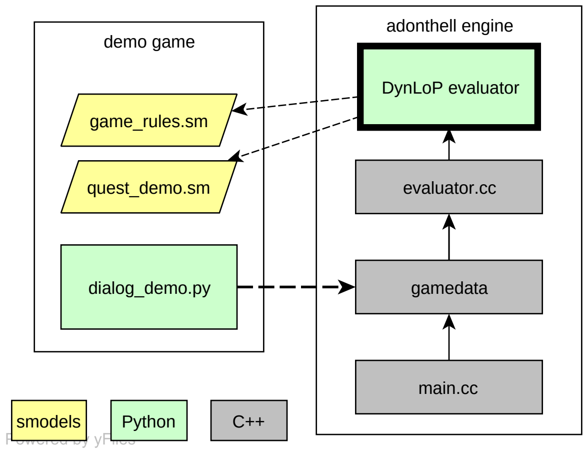

Figure 5.2: Integration of our DynLoP-WSE unit into Adonthell game engine. (C++ wrapper version)

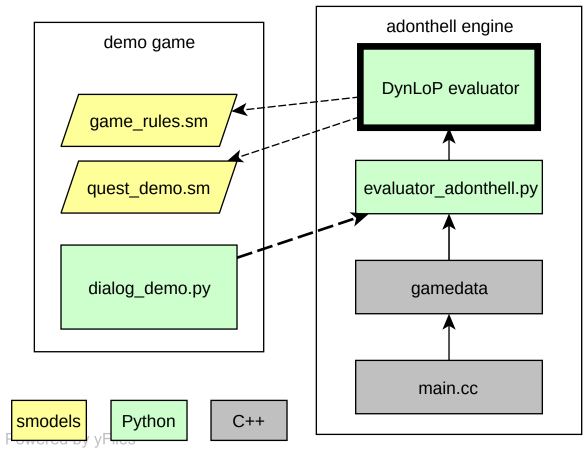

Figure 5.3: Integration of our DynLoP-WSE unit into Adonthell game engine. (Python wrapper version)

attributes

- `private: py_object *evaluator_py` Points to the instantiated python evaluator class.

constructor

Instantiates and initializes the `evaluator_py` attribute.

arguments:

- `(string) general_knowledge` Full path to file containing logic program with background knowledge for all the evaluations by `query` method.
- `(bool) explicit_neg` Specifies whether explicit negation is used in any logic program evaluator deals with.
- `(string) reasoning` What reasoning mode should the evaluator use. Either `''cautious''` (`True` is returned only if query holds in all models of the world) or `''brave''` (`True` is returned only if query holds at least in one model of the world).
- `(string) tmpfiles_prefix` Location for temporary files, including trailing (back)slash.
- `(string) tmpfilename_solve` Name of temporary file for python evaluator's solve method.
- `(string) tmpfilename_query` Name of temporary file for python evaluator's query method.
- `(string) tmpfilename_status` Name of auxiliary file for saving current Adonthell's internal world state as a logic program.
- `(string) tmpfilename_dynlop2lp` Name of temporary file for dynamic logic program to logic program transformation.
- `(string) knowledgefiles_prefix` Location of files with logic programs describing quests, including trailing (back)slash.
- `(string) knowledgefiles_suffix` Filename ending common for all quest descriptions (extension including dot).

destructor

Calls the `evaluator_py`'s `__delete__` method to clean up temporary files.

method: (bool) query

Gives answer for the query.

arguments:

- `(string) knowledge` Name of the quest this question refers to (also filename of its logic programming description).
- `(string) query` Conjunction of literals (in form of comma separated list).

return value: Whethert query holds in the current world state or not.

method: (void) quest2lp

Saves the Adonthell's internal quest representation to a file as a logic program. This means that from `data::quests[questname]` dictionary every entry will be saved as a fact `<key>(<value>).`

arguments:

- `(string) questname`
- `(string) filename`

5.2.2 Other files

Except adding `evaluator.h`, `evaluator.cc`, `dynlop2lp.py`, `evaluator.py` our DynLoP-WSE extension for Adonthell modifies the following original files:

- `main.cc` Instantiates and initializes the `data::WSeval` of the `evaluator` class. Following values are used for the initialization:
  - `general_knowledge =`
    `game::game_data_dir () + "/quests/background.sm"`

  - `explicit_neg = true`
  - `reasoning = "cautious"`
  - `tmpfiles_prefix = "/tmp/"`
  - `tmpfilename_solve = "adonthell_solve.tmp"`
  - `tmpfilename_query = "adonthell_query.tmp"`
  - `tmpfilename_status = "adonthell_status.tmp"`
  - `tmpfilename_dynlop2lp = "adonthell_dynlop2lp.tmp"`
  - `knowledgefiles_prefix = game::game_data_dir () + "/quests/"`
  - `knowledgefiles_suffix = ".sm"`

- `gamedata.h` Defines the `static evaluate` method for Adonthell's `gamedata` class. It has the same two parameters as the `query` method of our `evaluator` class. Only thing it does is calling `data::WSeval->query` with those parameters and returning its result. Its sole purpose is to stay consistent with the design of Adonthell code. This is the function that can be called from the dialogues scripts to get the needed statements evaluated.

- makefiles - to incorporate all these changes to the Adonthell installation package.

5.2.3 Demonstration game - Waste's Edge

Because a simple example is often more helpful than complex documentation we provide this little demo. Its purpose is to show how our Adonthell DynLoP-WSE extension can be used. Not to show what everything can be achieved by writing quests by this method. Therefore it only contains the minimum. Instances of features usage, on both the levels of the engine and the logic programming syntax.

It includes background knowledge (`background.sm`), quest specific knowledge (`demo.sm`) and a dialogue sample (`demo_intro_1.py`) that is querying this quest's status. It is the first dialogue that automatically executes after the start of the Waste's Edge game.

5.2.4 User's manual

This brief guide explains how to install and use our DynLoP-WSE extension for Adonthell game engine.

Installation

1. In order for this software to work you need a python interpreter, lparse, smodels and mkatoms installed. You can download them here:

   `http://www.python.org/download/`

   `http://www.tcs.hut.fi/Software/smodels/`

   `http://www.krlab.cs.ttu.edu/mkatoms/`

2. Get Adonthell and/or Waste's Edge source code distributions. You can download them from:

   `http://adonthell.linuxgames.com/download/index.shtml`

3. Get the DynLoP-WSE package for Adonthell. Either from the bundled CD or from

   `http://www.tulasacra.host.sk/index.php?dir=_diplomovka/`

   Copy the desired directories from this package (depending on whether you want just the engine extension or also the demo quest) over the engine and game sources respectively, overwriting all original files.

4. Run autoconf, then compile and install selected packages according to instructions in their documentation.

Technical issues

For some reason the original Adonthell distribution does not compile with gcc-4. However it works well for instance with versions 3.4.4 or 3.4.6. If you have these older versions installed on your system you can force their usage by entering (before compilation):

```
export CC=gcc-3.4.6 ; export CXX=g++-3.4.6
```

Usage

If you want to design your own quests in the Logic Programming fashion follow these simple steps:

1. Get our DynLoP-WSE engine extension properly installed.
2. Write your quests description in the language of Logic Programming. Name these files `<questname>.sm` and save them to your game data directory, subdirectory `quests`. Note that `<questname>` should be identical to the Adonthell's internal quest identifier.
3. If you want you can specify knowledge common for all the quests in a separate file. It should be named `background.sm` and also placed in `<game data directory>/quests`.
4. When writing dialogues, in the condition statement you can now call:

   ```
   adonthell.gamedata_evaluate(questname, query)
   ```

   to get the query evaluated in the context of the specified quest, where `query` is a string of comma separated literals (representing their conjunction) and `questname` is a string denoting both the quest identifier in the Adonthell's `quests` dictionary and the filename for that quest's logic programming description.

Chapter 6

Resume

Study of logic programming based formalisms for representing dynamic or hierarchically ordered knowledge could benefit from the existence of more practical implementations. The artificial intelligence in computer games needs boosting. These were the problems that we addressed. Application of the more recent approaches from the area of knowledge representation to the game engines seems to be the right step.

The main outputs of this thesis are:

- *implementation of transformational semantics for Dynamic Logic Programming (DynLoP) in Python language* (sec. 5.1.1) We made a function that can transform a dynamic logic program to equivalent logic program. This can subsequently be evaluated by any available Answer Set Programming (ASP) solvers. As a byproduct we obtained a function for *grounding of DynLoP programs* (sec. 5.1.1)

- *Dynamic Logic Programming solver in Python language* (sec. 5.1.2) The `evaluator` class can conveniently be used for computing models of DynLoP programs. Its principle is to make the DynLoP to LP transformation and solving the resulting logic program. Version that we provide, uses Smodels for this purpose. But if needed, it might be easily modified to work with DLV [5] or potentially other ASP solvers.

- *general DynLoP world state evaluator* (sec. 5.1.2) Probably the most important result - the `evaluator` class. It could be used in any computer game engine (although RPGs, adventures, more complicated puzzles or strategies are the intended target domain) or similar applications. It should be well suited whenever we have to deal with evaluating statements in a formally describable (and evolving in time) world.

- *DynLoP-WSE extension for Adonthell game engine* (sec. 5.2.1) One of the main goals was embedding DynLoP world state evaluator into an existent game engine. Adonthell was our choice, mainly because it is opensource, python friendly and of the role playing genre.

- *usage demonstration quest in Waste's Edge game* (sec. 5.2.3) For the most simple and quickest understanding how our Adonthell DynLoP-WSE mod should be used, demo quest was written. It includes background knowledge, quest specific knowledge and a dialogue sample that is querying this quest's status.

Enhancing a component with such underlying nature as a world state evaluator, opens up a space for a lot of possible future work. It could be a basic layer for more advanced applications. Such as event occurrence detection, decision making of computer controlled entities, or more complex behaviour for instance goals acquisition and assembling plans how to achieve them.

Bibliography

[1] Jozef Šiška. Dynamic Logic Programming and world state evaluation in computer games, In Procs. of WLP06, 2006

[2] Adonthell game engine. `http://adonthell.linuxgames.com/`

[3] J.A.Leite. Evolving Knowledge Bases, volume 81 of Frontiers in Artifical Inteligence and Applications. IOS Press, 2003.

[4] Stable model semantics implementation - smodels.
`http://www.tcs.hut.fi/Software/smodels/`

[5] A disjunctive datalog system - DLV.
`http://www.dbai.tuwien.ac.at/proj/dlv`

[6] Python programming language. `http://www.python.org/`

[7] MKAtoms utility for smodels. `http://www.krlab.cs.ttu.edu/mkatoms/`

[8] L.Padovani, A.Proverti. Qsmodels: ASP Planning in an Interactive Gaming Environment. In Procs. of the 9th European Conference on Logics in Artificial Intelligence (JELIA'04), Springer-Verlag, LNAI 3229, 2004

[9] 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

[10] Role-playing game TES4: Oblivion.
`http://www.elderscrolls.com/games/oblivion_overview.htm`