Università della Calabria

Dipartimento di Matematica e Informatica

Dottorato di Ricerca in Matematica e Informatica

XXXI ciclo

Tesi di Dottorato

Tight integration of Artificial Intelligence in Game Development Tools

Settore Scientifico Disciplinare INF/01 – INFORMATICA

Coordinatore: Ch.mo Prof. Gianluigi Greco

Supervisore: Ch.mo Prof. Giovambattista Ianni

Dottorando: Dott.ssa Denise Angilica

Alla mia famiglia

Sommario

In questo lavoro di tesi puntiamo a ridurre le lacune che impediscono l'utilizzo di strumenti dichiarativi all'interno di ambienti in cui lo stato del mondo cambia molto rapidamente e, in particolare, nel contesto dello sviluppo di videogiochi. Integrare moduli di ragionamento automatico basati su specifiche dichiarative, durante le varie fasi dello sviluppo di un videogioco, pone diverse sfide ancora irrisolte, ognuna delle quali richiede soluzioni non banali. Si devono rispettare requisiti di efficienza; la dualità tra codice procedurale e specifiche dichiarative rende l'integrazione non semplice; l'esecuzione concorrente dei task di ragionamento automatico e di quelli di aggiornamento del gioco, richiede l'applicazione di strategie idonee al passaggio di informazioni tra le due parti. In questo contesto, proponiamo un framework utilizzabile all'interno di Unity, un noto ambiente di sviluppo per videogiochi. Il framework, chiamato ThinkEngine, consente di integrare moduli di ragionamento automatico, basati su tecniche di rappresentazione della conoscenza, all'interno della logica di gioco. ThinkEngine, seguendo la filosofia di sviluppo di Unity, è integrato sia in fase di progettazione che in fase di esecuzione del gioco. A dimostrazione delle potenzialità del sistema proposto, è illustrato un caso d'uso.

Abstract

In this thesis we aim to narrow some of the gaps that prevent the adoption of declarative tools within highly dynamically changing environments, with a particular focus to the context of game development. Integrating reasoning modules, based on declarative specifications, within the commercial game development life-cycle, poses a number of unsolved challenges, each with nonobvious solution. It is necessary to cope with strict time performance requirements; the duality between procedural code and declarative specifications prevents easy integration; the concurrent execution of reasoning tasks and game updates requires proper information passing strategies between the two involved sides. In this context, we propose a framework that can be deployed within the well-known Unity game development engine. The so-called ThinkEngine framework allows to embed reasoning modules, based on knowledge representation techniques, within the game logic. ThinkEngine respects the Unity development philosophy, and is properly integrated both

at design-time and at run-time. A use case is reported about, showing the potential of the proposed infrastructure.

Contents

Introduction

In Artificial Intelligence, Knowledge Representation (KR in the following) aims to express information in a transparent, symbolic notation. Knowledge is expected to be encoded in a way suitable for modelling and perform automated reasoning on beliefs, intentions, and value judgments of an intelligent agent. In Logic, knowledge is represented by propositions and it is processed by applying various laws of logic, including a selection of appropriate rules of inference.

When comparing rule-based formalisms with imperative languages, one can notice how it can be much easier to solve a problem using the former approach rather than the latter one in a variety of settings. When using an imperative language, in order to produce a solution for a certain problem statement, developers first have to think to some algorithm to solve the problem at hand and then they have to implement this algorithm in a programming language of choice. This "algorithm design phase" is almost absent when approaching a problem with a declarative language. Indeed, it is sufficient to write down rules that describe how a problem solution should look like; then, a *KR solver* will take care of finding an actual solution. Moreover, when dealing with a declarative encoding of a problem, it is much easier to find errors and, once reached a good starting representation of what is needed, one can increasingly improve his/her knowledge base by means of subsequent refinement.

However, there are multiple limitations that arise when trying to introduce such formalisms in real world applications based on Object Oriented Programming (OOP):

- KR solvers still have a performance bottleneck when the problem at hand has a high complexity, a large input dataset, or both;
- OOP data structures are not available or difficult to model in declara-

tive systems;

- it is unclear how to manage the data exchange flow between modules that use completely different data representation, like when a knowledge representation side and a procedural side are coupled together;
- KR solvers often have limitations on the data types available.

One of the areas that has recently inspired a great interest is the integration of KR in the videogame development workflow. In general, AI techniques have (or can have) a role in numerous task applications in the game industry, ranging from programming the behavior of Non Player Characters to game level and content generation. Especially when considering real-time videogames, this particular context is really challenging for researchers since it means to work within an highly reactive environment, requiring really fast responses from a KR system. Also, for a reasoning module deployed within a videogame logic, it is crucial to have the possibility of stopping and/or quickly restarting reasoning tasks, especially if the KR system is not able to guarantee the required reactivity. This feature happens to be strategic when the videogame world has changed in a way that invalidates the solution that the system is trying to produce starting from an old state of the environment.

Besides the performance problem, a second fundamental issue concerns the technical integration of an external KR module in an object-oriented environment. The two paradigms offer a world representation on different abstraction levels: the level of abstraction is usually higher for the KR module and lower and more operational for the object oriented one. Indeed, reasoning in terms of a low-level abstraction is not feasible, and an abstraction process from this to the high-level is needed. For instance, game maps tipically require a discretization in grid cells, so to avoid to work at the pixel level; floating point physical simulations are simplified and abstracted; and so on.

Multiple strategies can be used to achieve such an integration [1], each one with its advantages and disadvantages depending on the context in which one has to work. Specifically, in a game development engine it is important to clearly distinguish run-time settings from design-time settings. Also it must be noted that, at run-time, a main module that we will call the *procedural side*, takes care of updating the game world: embedding one or more reasoning modules, which will belong to what we will call the *reasoning side*, requires the introduction of multiple concurrent execution flows, in order to prevent lagging in the screen update.

In fact, to make an embedded reasoning module aware of the state of the videogame world and to make the main update module capable of applying decisions provided from the reasoning module, it is needed to devise a proper information passing strategy. Devising such a strategy is not so trivial, mainly because of synchronization issues (the world can change while a reasoner is running), and because of the different data types and representations used respectively on the reasoning side and on the procedural side. This latter problem requires the usage of data reflection techniques, allowing to examine, introspect, and modify data structures and behavior on the procedural side at run-time, and to map logical propositions to and from object oriented data structures. It is thus clear that, the usage of KR systems along with imperative languages, can not be seen as a mere call to an external tool, rather as a tight integration of reasoning capabilities inside the main environment.

In this work, we aim to cover the highlighted gaps by proposing an abstract framework infrastructure for the integration of a KR module in a game engine. An actual implementation for the well-known game development engine Unity (ThinkEngine) is presented. Unity is one of the most used commercial game engines, and offers different facilities for the development of video games. It comes with different tools, called *Assets*, available on its Assets Store (some for free, others on payment), making it easier to manage physics, artificial intelligence and other fundamental aspects of a game project.

The ThinkEngine is a Unity asset that offers an infrastructure to introduce and manage automated reasoning models in a video game. In particular, we choose Answer Set Programming (ASP) as the reasoning model. ASP is a formalism based on the stable model semantics of logic programming [2], featuring a number of desirable advantages. The facilities offered by this framework regard a tight sharing of data structures between the main program, the game, and the ASP reasoner, enhancing reflection techniques for mapping game data to and from ASP logic assertions. ThinkEngine is based on concurrent execution to avoid a drop of performances of the game due to the time consuming reasoning process. Indeed, if a reasoner is executed within the thread that takes care of the graphical update, the game would freeze until the solver finds a solution.

This thesis work is structured as follows. In the first chapter we give an overview of the background context for this thesis, such as the Unity game engine, Declarative Logic Programming, and the earlier solutions proposed in

the literature integrating KR in OOP. Moreover, we overview the main usages of AI in videogames. In the second chapter we discuss the problems arising while integrating a KR module in an object oriented real-time system. The third chapter contains a description of the general architecture implemented with the ThinkEngine asset, focusing on the usage of Sensors and Actuators to share data between the main program and the reasoner solver. In the fourth chapter, we show the usage of the asset by means of its application on the very well known Tetris game. Some performance aspects are discussed.

Chapter 1

Basic concepts and related research

In this chapter we relate our thesis work with existing research efforts and provide the necessary context. It must be noted that the ThinkEngine which we present has points of contact with the stream reasoning field, with declarative logic and object-oriented programming as well as planning and with the broad discipline of artificial intelligence in videogames.

In the following we will overview related literature and introduce some key concepts used throughout this work.

1.1 Object Oriented Programming

Videogame development is the process of creating a videogame. Videogame development is usually based on *Object-Oriented Programming* (OOP), a well-known programming paradigm based on the concept of "object". Objects can contain data, represented in the form of fields, and code, in the form of procedures. Objects can interact with each other but this interaction is based on some principles. Main principles of OOP are:

- *Encapsulation*. Each object keeps its state private within their *class* scope. This means that, on the one hand, different objects of the same class can access fields from each other without limitations. On the other hand, instances of a different class are allowed only to call a list of public methods (functions) of the class, some of which expose safely

the object state. This allows to achieve information hiding in a natural way.

- *Abstraction.* An object-oriented program is usually maintained by different developers for a long time and, during its development cycle, a program is subject to a number of modifications. To associate an abstract concept to a class, actually exposing only *what* it does and not *how* it does it, allows easy interaction between objects but also make developers capable of working on different parts of a program, without caring about what other people are working on.

- *Inheritance.* Object classes can be related each other in two ways: either by means of an *is-a* relation or an *has-a* relation. The inheritance principle is used to express the former, in fact allowing a derived class to reuse logic (thus code) of super classes.

- *Polymorphism.* As an immediate consequence of the inheritance principle, there is the polymorphism (literally "many shapes" in Greek). This principle allows to use an object of a derived class as if it was belonging to one of its super classes, for instance one can invoke on a given object a method of its superclasses, although the behavior of such method can be specialized in each subclass.

1.2 Game engines: Unity

Thanks to the advent of online distribution systems, such as Steam and Origin, as well as the mobile market for Android and IOS devices, the videogame industry has grown rapidly since the early 2000s. In fact, these systems are making it easier to publish indie games, i.e. videogames published without the funding of a publisher. Around this gaming market several platforms are springing up, known as *Game Engines*. These are object-oriented software-development environments designed for people to build video games [3]. They offer the possibility to easily integrate in a software different things, for example:

- physics

- rendering

- artificial intelligence.

There are more than 100 solutions on the market, making it more difficult to choose which one to use when developing a game. Among these solutions, one of the most used is the *Unity* game development engine.[1]

Unity is a cross-platform game engine first announced and released in 2005. Nowadays, the engine has been extended to support over 25 platforms (Windows, Android, iOS and so on). Unity offers facilities for developing both 2D and 3D video games. One can use this framework to assemble assets (audio, special effects and so on) and art into environments and scenes, and concomitantly play, test and edit the game if necessary. Developers can use a primary scripting API in `C#` (for both the Unity editor in the form of plugins, and games themselves) but also visual editing facilities like drag and drop, property editing etc.. Unity makes available an asset store[2] on which developers can put their solutions (editor plugins, models, SDKs, templates...) for selling (some assets are proposed for free). Assets are divided in macro and micro categories offering a rich collection of solutions for different purposes such as, for instance, characters, textures and AI. Within this engine, *Game Objects* are the fundamental data structures that represent characters, props and scenery. Game objects do not accomplish much in themselves but they act as containers for *Components*, which implement the real functionalities. The *Game world*, or *environment*, is the set of all the objects used in a game project. At run-time stage, the game world is highly dynamically changing, requiring that all the processes executed inside the *game logic* (i.e. all the procedures needed to the game execution) must be fast.

One of the main disadvantages of this game engine is that the adoption of multithreading in videogame development is a controversial topic. The usage of threads with this engine, indeed, is not trivial since Unity is not strictly designed to be thread safe [4]. In order to keep thread safety, the Unity APIs are accessible only by the main thread. However, there could be heavy tasks that could slow down the main thread and thus the whole game. These tasks include the execution of AI algorithms, which are of special interest for our thesis. When dealing with the execution of such type of CPU-bound code,

[1] https://unity3d.com/unity
[2] https://www.assetstore.unity3d.com/

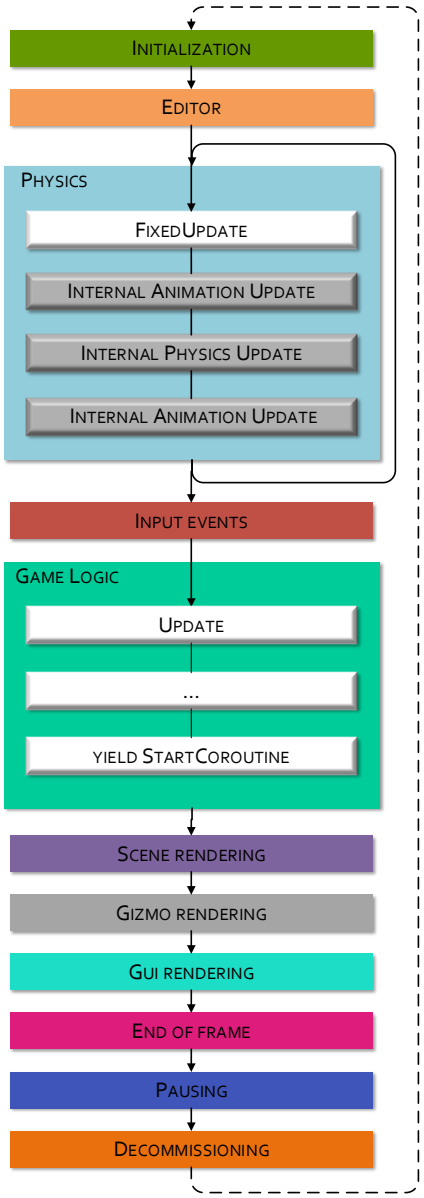


Figure 1.1: Unity workflow

Unity offers the possibility of using *Coroutines*[3] or the *Job System* [4]. This

[3]https://docs.unity3d.com/Manual/Coroutines.html
[4]https://docs.unity3d.com/Manual//JobSystem.html

can avoid dramatic decreases of performance.

Figure 1.1, describes the main run-time execution workflow for a Unity videogame. This workflow is mostly single-threaded, with the expection of some parallel code in the physics engine. Game designers can customize the game behavior by implementing specific user callback functions, which are executed within the main thread. For instance, the game designer can provide her/his own code for the `FixedUpdate` block, or provide her/his own coroutine. *Coroutines* constitute a way for implementing asynchronous cooperative multitasking within a single thread.

The collection of game objects (GOs in the following) constituting the game world, are subject to continuous updates depending on user input; on the physics simulation of the game world, and on the game logic enforced by the game designer. Game objects contain a recursive hierarchy of basic properties, such as numeric, string and boolean fields, and complex properties, such as matrices, collections, nested objects, etc.

At design-time, it is possible to work on game objects using the above property-based philosophy, while the game logic can be edited by attaching scripted code to specific game events.

1.3 Planning Domain Definition Language

Planning is the branch of Artificial Intelligence (AI) that seeks to automate reasoning about plans, most importantly the reasoning that goes into formulating a plan to achieve a given goal in a given situation. With at hand description (or model) of the initial situation, the actions available to change it, and the goal condition, a planning system outputs a plan composed of those actions that will accomplish the goal when executed from the initial situation. In industry, more and more application make use of planning, such as robots and autonomous systems, cognitive assistants and cyber security.

Planning Domain Definition Language, PDDL, is a standard language designed to express classic planning tasks [5]. PDDL is used in a variety of planning system and is mandatory for the International Planning Competition. Several variants of this language are emerging among which PDDL+ [6] is the one pushing forward the use of planning in real-world domains. Indeed, PDDL+ is designed to model dynamical system in which there are both continuous control parameters and discrete logical modes of operation, so called *hybrid systems*. A system that can both flow (described by a differ-

ential equation) and jump (described by a state machine or automaton) is a simple example of a hybrid dynamical system.

1.4 Declarative Logic Programming

Declarative Programming is a programming paradigm meant to narrow the gap between humans and machines when the former must command the latter. In fact, declarative programming is meant to solve problems by describing *what* should be a desired solution of a problem at hand, instead of describing *how* algorithmically build that solution.

Declarative programming is actually a hypernym for different programming paradigms and in particular for Declarative Logic Programming (DLP from now on). DLP is fundamentally based on formal logic. With this paradigm, facts and rules, expressed by sentences in logical form, are used to write some program to solve a given problem.

To highlight the characteristics of languages based on logic, Robert Kowalski has defined the equation: `Algorithm = Logic + Control`. In the case of logical languages, the task of the programmer is limited to the specification of the logical part, while the control of the computation is completely left to the computer. On the contrary, in the case of procedural languages, even the flow of control is specified by the programmer.

In declarative programming, a distinction must be made between *syntax* and *semantics* when discussing of a specific language. The syntax describes the grammar structure of the rules accepted by a language while the semantic explains the meaning of a program in that specific language by means of precise mathematical models. Different languages can share the same syntax but have completely different semantic.

Two of the main dialects belonging to the DLP family are Prolog [7] and Answer Set Programming [2], [8]. These two languages share a part of the syntax but their semantics are quite different. Rules written in these languages express sentences like

$$\texttt{H if B}_1 \texttt{ and ... and B}_n\texttt{.}$$

in the following syntactic form:

$$\texttt{H :- B}_1\texttt{,..., B}_n\texttt{.}$$

In the above expression, `H` is called *head* of the rule while $\texttt{B}_1\texttt{,}\ldots\texttt{,}\texttt{B}_n$ is called *body* of the rule. Rules without body are called *facts* and they can be written simply as

`H.`

Both Prolog and ASP are based on the *Predicate Logic* and, in particular, on the fragment constituted of *Horn Clauses* (i.e. a disjunction of literals of which at most one is positive). Peculiarly, ASP allows also more than one positive literal in a clause. Some common terminology used in these languages follows:

- *constants* are strings (or numbers) starting with lowercase letter, for example *alice* or *bob*. These are the actual individuals of the domain at hand;
- *variables* are strings starting with uppercase letter, for example *X*or *Person*. These are generic placeholders for an individual of the domain;
- *functional terms* are compositions of constants, variables and function names and address other individuals of the domain referred to the given composition. For example, the functional term *mother(alice)* is made of the functional name *mother* and the constant *alice* in fact representing the *"mother of alice"* as an individual of the domain;
- *predicates*, constitute templates of logical assertions and are associated to a fixed number of attributes called *arity*. Arity is a value $k \geq 0$. Predicates are written with lowercase starting letter often followed by their arity (for instance *ready/0* represents a predicate with name *ready* and arity 0, while *isFatherOf/2* is accordingly a predicate of arity 2 named *isFatherOf*);
- *atoms*, are instances of predicate assertions of according arity (e.g. *ready* or *isFatherOf(alice,X)*); *ground atoms* are composed of constant terms only, like *isFatherOf(alice,jack)*.
- *negation as failure*, expressed by the word *not*. Given a predicate p, *not* p is true when p can not be derived as true;
- *literals* are either an atom or a negated one.

1.4.1 Prolog

Prolog emerged from the collaboration of two computer scientists, Alain Colmerauer and Robert Kowalski, in the 70s [9], as probably the first logic programming language. Prolog is still the most known language in this field, with applications ranging from theorem proving to natural language processing (its original intended application field).

One of the tasks that best exploit the potential of Prolog is searching databases (especially when using the Prolog fragment called *Datalog*). In fact, programs are naturally expressed in terms of relations between objects, and this is easily mapped on relational database schemes.

However, due to the highly efficient inference algorithm adopted by Prolog, the *SLD resolution*, which is sound but not complete, multiple difficulties arise. This algorithm, indeed, introduces some limitations on those that are key features of a declarative approach. For instance, changing the order of the rules of a program or changing the order of terms in rules body, could even change query answers or even lead the evaluation to not terminate.

1.4.2 Answer Set Programming

Answer Set Programming (ASP) is a declarative formalism for knowledge representation and reasoning. The language used for ASP is also called AnsProlog.

ASP is based on the stable model semantics of logic programming [10]. The idea of ASP is to employ non-monotonic logic (i.e. a form of logic where the introduction of new axioms can invalidate old theorems) to describe problems of high difficulty (NP-Hard) whose solutions are called *answer sets*.

Contrary to what was said for Prolog, ASP results to be much closer to the idea of a "pure" declarative formalism. In fact the order of the rules of a program, as well as the order of atoms in the body of the rules, does not impact on the answer provided by the inference algorithm. Nonetheless, termination is always guaranteed unless function symbols are involved (indeed, the availability of function symbols, combined with recursion allow to simulate a Turing machine).

An ASP program consists of rules of the type seen in section 1.4. In addition to facts (rules with empty body), there are also (strong) constraints, i.e. rules with an empty head which are expressed as

$$\texttt{:- B}_1\texttt{,...B}_n\texttt{.}$$

With this kind of rules the body (i.e. the conjunction of the $\mathtt{B}_i$) must be false in the models of the program.

In addition to basic concepts that ASP shares with Prolog, ASP is characterized by advanced constructs:

- *aggregates*, i.e. specialized atoms in which the values of aggregate functions, like #max, #min, #sum, #count and #times, can be computed over a conjunction of literals and compared to given bounds;

- *weak constraints*, i.e. constraints that, differently from strong ones, express a desired condition.

In order to give an intuition on the ASP semantic, we will show a practical example while more details will be given in Section 3.3.1. A very well known problem for which ASP offers a simple solution is the graph three-coloring [11].

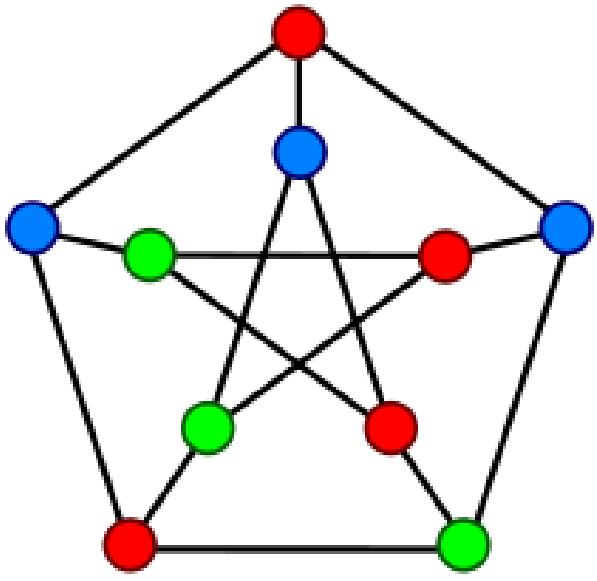


Figure 1.2: A three-coloring example

The graph coloring is defined as follows: given a graph `G=(V,E)`, one can assign a color to each node so that two adjacent vertices do not have the

same color. In the three-coloring problem one can use only three different colors.

Suppose that the graph `G` is stored using facts in the form `node(n)` for each `n` $\in$ `V` and `edge(e)` for each `e` $\in$ `E`, one can model this problem with an ASP program:

1. `color(N,red)` $\vee$ `color(N,blue)` $\vee$ `color(N,green) :- node(N).`
2. `:- color(N,C),color(N1,C),edge(N,N1).`

Rule 1 (containing disjunction in the head) expresses the requirement that every node must be associated with a color. Rule 2, a strong constraint, asserts that two vertices connected by an edge both being associated to the same color are not possible.

Note that since an answer set of a problem is a minimal model of a set of rules, it is not necessary to make explicit a constraint expressing that a node must be associated with only one color. Indeed, the answer set of the above program will contain the smallest subset of literals sufficient to satisfy the program. In this case, Rule 1. is satisfied if and only if, for each node, *at least* one of the three literals of the head is picked as true. In this case, it is sufficient to pick exactly one color for each node to satisfy the rule.

1.5 Stream reasoning

In a world where the Internet became the common thread in almost every daily action, it is possible to collect really large datasets from which one can extract useful information. Smart cities with sensors that track road traffic or environmental pollution, and web traffic revealing customers' preferences can be seen as examples of data sources which allow really interesting analysis. A lot of research can be found in this field, among which *Stream reasoning* is gathering great attention in the artificial intelligence community. The main goal of a stream reasoning system is to be capable of working with datasets that have specific features such as *volume*, *velocity* and *incompleteness*.

A more detailed overview of stream reasoning literature and basic concepts like the above is given in Section 2.3.

1.6 Integrating DLP and OOP

As mentioned above, the most common programming paradigm can be identified in the object-oriented programming. However, there are some problems that, if solved through OOP, require a considerable effort both in terms of lines of code and in terms of time needed for the execution of the solving algorithm. In general using a declarative language makes easier the implementation of pieces of code that are difficult to be designed using imperative programming.

When one tries to integrate the two paradigms, however, one has to face some technical issues, such as the input-output management.

In the following, we will briefly illustrate the earlier proposals for the integration of these two programming paradigms, first for what it concerns declarative logic in general and then focusing on ASP in particular.

1.6.1 General approaches

In more than twenty years of research, numerous attempts have been made to integrate OOP and declarative logic. Even before OOP imposed itself as the main programming paradigm, there has been several proposal of integration of imperative and declarative logic programming.

One of the first attempts aiming at the integration of imperative programming with declarative logic can be found in [12]. In this paper authors presented COP, a language that integrates `C++` with Prolog. The authors present a language that can be seen as `C++` having access to the Prolog goals with a proper compiler capable of detecting several errors deriving from switching between `C++` and Prolog.

In [13], authors present a new imperative language enriched with some features inspired by logic programming (e.g. logical variables, constraints and unification). They show how it can be easier to solve some problems, in particular ones involving research, combining the two paradigms instead of using exclusively one of them.

Two completely different solutions have been proposed in [14] and [15]: JSetL, and Tweety, respectively. They both offer libraries that endow Java with several facilities to support general purpose declarative programming. Tweety already contains over 15 different knowledge representation formalisms ranging from classical logics, over logic programming and computational models for argumentation, to probabilistic modelling approaches,

including ASP.

1.6.2 ASP for OOP

Solutions aiming at the integration of ASP in object-oriented environments have been proposed during the last decade. These solutions can be divided in two major families:

- API's approaches;
- hybrid languages.

The first one, basically, consists of creating a library that exposes APIs for the ASP solver written in a specific object-oriented language. In [16] it has been presented the DLV Java Wrapper, a first attempt in this direction, creating in fact a Java library allowing to embed a disjunctive logic program in a Java program. With this solution, however, a programmer must take care of the integration of ASP in Java. In particular, one has to spend a lot of time in developing customized solutions for the execution of an external solver and, mostly, for converting back and forth from objects' data structures to logic representation. Moreover, a solution based on an API approach obliges a programmer to learn new APIs.

Concerning the second family of solutions, there are different proposals in the literature. They try to solve problems that arise within the previous family of solutions. In [17], authors present a framework for the integration of ASP and Java. JASP is based on an hybrid language that allows ASP programs to access Java variables and, in the opposite direction, answer sets are stored in Java objects (even using Object-Relational Mapping tools like Hibernate).

Clingo4 [18] is a remarkable member of a generation of ASP systems enriched with the possibility of using it via external applications. This is allowed by adding some scripting facilities (*lua* and *python* languages) that make available instructions aiming at controlling the execution of the solver. The main objective of this solution is to make ASP suitable also for dynamic environments and for problems that can take advantage of incremental reasoning.

In [19] the authors have proposed an abstract architecture for a framework for the integration of ASP in external tools for which authors presented a Java implementation. The strength of this proposal is that, since it is

an abstract architecture, it can be easily implemented in any programming language, deployed for multiple platforms and take advantage of different ASP solvers. The Java implementation of EmbASP proposed here, is based on the annotations-guided mapping (i.e. java objects are translated via Java annotation back and forth to the ASP notation), thus solving the problem of feeding an ASP program. EmbASP has been even proposed in a `C#` version, available as an asset for the Unity Engine [20].

Similarly to EmbASP, but quite orthogonal to it is the work presented in [21]. Indeed, in the same way as EmbASP, this solution exploits annotations in order to build the mapping for input and output ASP atom. However, annotations are used in the ASP program instead that on the procedural side and they are used to specify how to parse the input received and how to produce the output for the caller. Thus, contrary to EmbASP, an ASP programmer has to have competence in OOP while an object-oriented programmer can ignore the existence of the ASP module.

The ActHEX language [**?**], uses an input language derived from HEX programs [22] and allows to specify *action atoms* in rules' heads. Action atoms are directly attached to procedural scripts that can be written in arbitrary languages and are executed depending on the obtained answer sets. In a sense, ActHEX programs can be seen as ASP programs with an external procedural side, in the same way in which clingo4 scripts resemble procedural scripts with an external declarative side.

In [23] it has been proposed a different way of making ASP interesting for object-oriented programmers. The authors proposed a visual tool based on model-driven engineering in fact encouraging programmers to use ER diagrams to describe data models. In this way, they can automatically generate constraints in the ASP program deriving from the ER diagram and, also, they offer a graphical interface for data input that can be easily build starting from the ER itself.

1.7 Artificial Intelligence in videogames

The first valuable results in programming aimed at the development of virtual game players (so called game artificial intelligences), date back to the 50s. In [24], Claude Shannon (considered as the founding father of electronic communications age) studied the chess game scenario proposing an interesting approach for a program that could play chess autonomously and

intelligently. Since then, the application of computer programming to game-player programs has undergone continuous evolution. Chess has been one of the most studied games from this point of view. In fact, in the 70s and 80s, this was the main research argument for computer-games scientists. In that period, approaches were focused on the brute-force technique. After a setback regarding the development of a high-performance chess player, computer-game research had a total change of style from the late 80s.

1.7.1 Overview

Start from the 80's of the past century a series of algorithms capable of winning several world competitions intended for humans [25] has been proposed. Particular attention has been focused on "player vs player" games. A famous istance of these winning proposals was *Deep Blue* [26], the first computer chess player able to overcome a human player.

Strategies and programming techniques used to implement a game AI are many. They range from pathfinding to neural-networks, including rule systems and decision-tree learning [27]. All these approaches can be all referred as game AI since they can reproduce a behaviour that a human player expects to observe in his opponent. Nowadays, however, games are no longer limited to involving two players. In fact, for instance, there are more and more titles in the world of massive multiplayer online games.[5] In this kind of games, there are multiple interacting systems, element that completely changes the way of managing the AI of an object of a game.

Moreover, new sectors are emerging in which the usage of AI is considerably simplifying the work of video game developers and designers [28]. In order to arise in the multitude of titles offered by the gaming industry, a new game coming on the market must be able to offer dynamism and, possibly, a wide variety of landscapes or scenarios. Generating ever new environments, diversified game dynamics with the right balance between difficulty and fun, requires a lot of imagination, time and resources. In this context, in order to offer increasingly exciting titles, the use of artificial intelligence is intensifying in many phases of the development of a video game. Actually, one can identify different AI-based game design patterns currently used [29]. Below we propose some examples to illustrate how the AI can be used to develop a video game.

[5] Multiplayer computer games that can withstand hundreds or thousands of players.

The Sims[6]

The Sims series[7] is a strategic life-simulation video-game developed by Maxis and published by Electronic Arts in 2000. In this kind of game, a player can be seen as a God that guides choices of the characters. However, once that the player tells to a character (a *Sim*) what it has to do, an artificial intelligence (improved with every game edition) will act, and make the character perform actions consistent with the instructions received.

Main aspects that use AI techniques in this game:

- Pathfinding: Sims move autonomously, even if they occasionally get stuck in front of an obstacle. The player indicates to the character a point to be reached and the Sim will reach it without further instructions.

- Decision making: in a gameplay of The Sims, players usually have to take care of more than one Sim. It is easy to forget about one of them or not have enough time to control everything. In this case, AI takes control of that character: it will eat, sleep, work or have fun (possibly with other Sims) depending on its needs and duties (but also their personality).

- Social interaction: when two Sims start to interact, they can improve or worsen their relationship. Depending on the score assigned to the current state of this relation, the player (or the AI if the player is not controlling the character) can choose next action for a Sim among a determined set. Each action will cause an increase or decrease in the score.

Minecraft[8]

Minecraft is a sandbox video game created by Swedish game developer Markus Persson and released by Mojang in 2011. In a 3D procedurally generated world, players have access to a variety of different blocks to build, explore, gather resources, craft, and combat with entities called mobs.

In this title, AI is used for two main purposes:

[6]https://www.ea.com/games/the-sims
[7]Last edition released is "The Sims 4".
[8]https://www.minecraft.net

- World Generator: when starting a new game, a map is generated with a pseudo-random procedure. This allows to offer completely different maps simply changing a starting seed chosen at initialization step. There is no human designer of maps. Humans, at game design-time, decided only what characteristics must have a map. Then, at game run-time, maps are auto-generated by an algorithm.

- Mob[9] behavior: each *mobile* entity belongs to a certain category (for instance peaceful, hostile, tameable, neutral and so on). In the same way as the player, mobs are subject to the physics of the game and can be hurt by the same things that are dangerous for a player. Each type of mob (skeleton, creeper, witch and so on) has a certain AI system with different behaviors and mechanics. All mobs will try to avoid a downfall, a danger area and they will move at random until there are no players in their radar. Some hostile mobs can even attack some peaceful ones. Many mobs have an advanced pathfinding system that allows them to traverse complex mazes to get to a desired object or destination. Peaceful and hostile mobs have completely different behavior when facing a player: the first category completely ignores it, while the second one immediately attacks it. Neutral mobs can be seen as intermediate between peaceful and hostile. They ignore a player until they are not attacked. After being attacked, they start to act like an hostile mob.

GVGAI

In academic research, AI for video-games is still a hot topic. In the last few years, in particular, a new competition has been launched: General Video Game AI Competition [30].

The GVGAI framework, available in many languages, provides an object-oriented interface for creating agents that can play in any game defined in Video Game Description Language. VGDL is a language that allows to describe entities and interactions between them that can take place in a game. The framework lets the developer to program dynamics of these objects such as movement and behavior.

The availability of this framework allows multiple challenges[10] which have

[9] https://minecraft.gamepedia.com/Mob

[10] https://github.com/GAIGResearch/GVGAI/wiki/Tracks-Description

been proposed to researchers, often in the form of a competition:

- Single player planning: given a set of VGDL single player games which can be played by some AI player, the goal is to create an artificial intelligence able to play 10 secret test games.

- Level generator: given a VGDL game which can be played by some AI player, the goal is to construct a generator that builds any required number of different levels for that game which are enjoyable for humans to play.

- Rules generator: given a set of game sprites and a level, one has to construct a generator that builds any required number of different games for these sprites and level which are enjoyable for humans to play.

1.7.2 AI in Unity

After perceiving the potential of artificial intelligence applied to video games, the Unity developers began to give more importance to this topic [31]. Tools for *Machine Learning* are offered and it is encouraged the exchange of ideas as well as collaboration between developers and researchers on dedicated forums. A toolkit has been provided, which offers great flexibility and ease-of-use to the growing groups interested in applying machine learning to developing intelligent agents.

In addition to this toolkit offered by Unity, various solutions are available on the unity store for integrating AI within the games.[11] Among all the assets offered in the store, the AI tools occupy only the 0.3% and the principal ones concern:

- Pathfinding, the most popular but frustrating game AI problem [32];

- Dialogue system;

- Behavioral designers, ranging from personality design to fighting style design;

- Action planning.

[11] https://assetstore.unity.com/categories/tools/ai

The majority of these solutions, basically, offers facilities for characterizing an object of a game using previously implemented algorithms just by graphic configuration. This implies almost to avoid code writing but also, unfortunately, to have not so much customization.

1.7.3 Planning in video-games

Planning has been used with interesting results in the video-games field. One of the first attempts at integrating PDDL in video-games environment has been presented in [33]. In this work, PDDL has been used a simple arcade game, Icebox, in order to show the feasibility of using artificial players with underlying planning techniques. Authors presented a PDDL-based planning system featuring a problem generator, a planner and a plan execution module. The system can play, with some exceptions, a real-time arcade game. Further tests have been done by the same authors on a commercial video-game, called Virtual Battle Space 2 [34]. This is a military simulator and the underlying game engine offers multiple features for simulation steps that are useful for the integration of planning techniques in such a game. In [35], authors investigate the usage of STRIPS planning techniques [36] within real-time strategy games. Authors created a new game domain, Smart Workers, in the Unity game engine. This is a typical setting of real-time strategy video games where the user manages various resources, structures, and units with the aim to eliminate all other kingdoms. Multiple methods have been tested and it appears that the sophisticated heuristic methods are the only ones reliable even in small problems.

1.7.4 ASP for video-games

Similarly to the GVGAI competition, there is a competition launched by AAAI regarding General Game Playing. Differently from what it has been for GVGAI, competitors of the AAAI competition are faced with rules of games previously unknown. After a period of time generally ranging from 5 to 20 minutes, the AIs begin to challenge each other having a time limit for each move. During these matches, developers cannot in any way intervene on the AI itself. Rules of these games are modeled with the Game Description Language.[12] Since GDL and ASP have a similar syntax and semantics, in

[12]The language from which VGDL has its origins

[37], authors proposed usage of ASP for the Single player subset of general game playing competition. They tried to create an AI completely based on ASP (exploiting the translation from GDL to ASP), obtaining good performances with most of the games proposed. Limitations of this paradigm arise when the complexity of the game increases. Indeed, complexity often affects the size of the ground program that quickly becomes too large and therefore untreatable. However, it remains a viable option to outsource a part of a general single-player game system to an ASP module to drastically increase performance for a particular scenario. Similarly, ASP has been used in Centurio [38] a General Game Playing system. Authors based their system on Monte Carlo Tree Search (best-first search method that is based on randomized explorations of the search space) for multiplayers games while exploiting ASP for single-player games using incremental grounding techniques to avoid excessive dimension of ground programs.

Another task in which ASP has shown its potential is the Procedural Content Generation, i.e. the auto-generation of the world map of a game. In [39] it has been proposed a general mapping between PCG problems and ASP offering a perspective completely different from state of the art PCG techniques. It has been shown that ASP is a proven method for quickly (in terms of both running time and designing effort) generating good solutions for content generation problems. Similarly, in [40] it has been investigated the usage of partitioning techniques for maze generation via ASP. With the partitioning approach, a given room is partitioned in two new ones and a door is placed on the wall that has been just created. Authors propose a Unity asset and a GVGAI plugin for the generation of multiple purposes mazes. Results obtained in both [39] and [40] are encouraging on continue the research on this task.

Interesting results have been presented in [41]. In this case, ASP has been used to predict unit's production in real-time strategy games (RTS). Such games are distinguished from turn-based strategy (TBS), in which all players take turns when playing. In an RTS, players can position and maneuver units and structures to secure areas of the map and/or destroy their opponents' assets. In a typical RTS, it is possible to create additional units and structures during the course of a game. Authors decided to give ASP a chance in this type of game since its semantics is well suited for reasoning with uncertainty and incomplete knowledge. Based on some assumptions (for instance that unit production requires time and resources, there is a continue supplying of resources and so on), they built a prediction system that has been test on

StarCraft and *WarCraft III* obtaining really good results for mid and late game.

Instead of using ASP to design an AI player, in [42] authors presented a framework in which ASP can be used for application programming. Authors discussed a show case made on the well-known Tetris game in which human player input is processed by an ASP module extended with actions. The actions resulting in a computed answer set are used to update the state of the game. Above all, the main issue arising is the slowness of ASP while dealing with real-time applications.

Chapter 2

Reasoning in real-time systems

In the previous chapter we have discussed the main approaches for the integration of DLP (in particular for ASP) in generic contexts. In the following, we will focus on the issues arising when the context to work in is a *real-time system*.

2.1 Real-time systems

A computation system can be divided in multiple tasks, small pieces of the system that are executable independently from each other. Tasks of real-time computation systems [43] are characterized by three major components: *time, reliability* and *environment*.

Time. It is the most important feature of real-time systems. Real-time tasks must be scheduled to be completed before their *deadline*. When dealing with this kind of systems, beyond the correctness of the logical results computation, one of the most important features one has to consider is the rapidity in providing a response for a certain request. A real-time system changes its state over time, even if its state is not checked by some computation algorithm. This means that if the solution computation is not fast enough the provided solution would be completely useless. Each real-time system has its own behavior and thus a proper maximum deadline.

Reliability. Potentially, a failure of a task in a real-time system can lead to an economical disaster or, worse, to loss of human lives. We can think

for instance to an air traffic control system: if something goes wrong multiple airplanes could get involved in some accidents. For this reason, system reliability is crucial.

Environment. When a self-driving car is actually driving on a highway it needs to sense if there are other cars or obstacles nearby. In order to be capable of cleaning the floor, a vacuum cleaner robot needs to knows where it is located, which parts of the room it has already cleaned. In general, each task of a real-time system need to be aware of the environment that surrounds the system itself.

A real-time application is usually comprised of both *periodic* and *aperiodic* tasks each of which has its own characteristics.

Periodic tasks. Periodic tasks have an important role in real-time systems since they control the standard flow of an application. This kind of tasks are invoked or activated at regular intervals and have a deadline by which they must complete their execution. An example of periodic task is the assisted brake system that is becoming increasingly popular among new cars. Indeed such a system checks periodically the distance between a car equipped with this system and the nearest obstacle on which the vehicle could crash on. If this distance is lower than a certain threshold the system will signal the driver with an acoustic alarm and, if the driver does nothing in order to avoid the crash, it will finally brake by itself.

A feature common to many periodic tasks is that they have severe deadlines and that is why they are also referred as *time-critical* tasks. In the assisted brake example, if the system fails to signal the driver about the imminent danger, a serious accident can happen even causing people death. For this reason, time-critical tasks must have priority over other tasks.

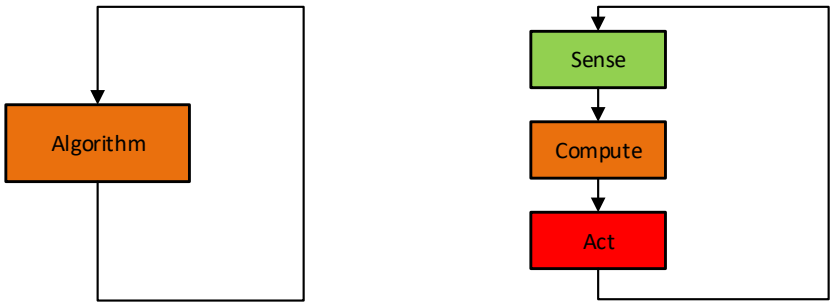
Aperiodic tasks. Along with periodic tasks there exist some other tasks that are activated only when some event occurs. This kind of tasks are referred to as aperiodic since they do not arrive on regular intervals. An example of an aperiodic task is the seat belt reminder system on cars. A dash indicator light or an alarm goes off when there is a passenger in the seat but the passenger herself/himself is not buckled in. This example task is not time-critical and it has not a strict deadline. However it should be

executed as soon as possible without compromising other time-critical tasks.

Depending on how strict a deadline is, it can belong to three major categories: *hard, firm* and *soft*. A deadline is classified as hard when catastrophes occur if the execution is not completed before the deadline is reached. Periodic tasks often happen to belong to this first category. When no major problems occur if the deadline is not met then it is classified as firm. Such kind of deadline represents the moment in which the result produced by the corresponding task loses its usefulness. In this category we find many aperiodic tasks. Finally, soft deadline cover all the tasks that have neither hard nor firm deadline. In these cases, results produced by the corresponding tasks after that the deadline expires will lose their usefulness as time passes.

Beside the time constraints, real-time systems have also constraints that are familiar in non-real-time systems such as resource constraints (I/O devices, files and so on), dependency constraints (a task needs other tasks results in order to start its computation) and finally performance constraints.

2.2 Looping update environment



(a) (b)

Figure 2.1: In a looping update environment some algorithm is executed in a loop cycle (2.1a). Such an algorithm usually can be divided in three phases (2.1b).

The real-time system tasks examples proposed in the previous section, both the periodic and the aperiodic ones, have a common structure constitued by a *sensing* phase, followed by a *computation* one and finally some

results/actions are provided. Such a structure suggests to model a real-time system as a *looping update environment* (Figure 2.1). In a looping update environment some algorithm is executed in a loop cycle and there is a strong connection between the algorithm and the environment in which it runs. This means, that the environment provides the algorithm with multiple information (for instance using sensors, I/O devices, files and so on) and, on the other hand, the algorithm after some computation can apply some actions on the environment. Therefore, this kind of black box algorithm can be actually identified by three specific phases: sense, compute and act. Each of this loop iteration can either be a single task or a collection of multiple tasks. For instance, consider an air conditioning system. At regular intervals, sensors collect information about temperature, humidity, position of people in the room and so on. If the computation algorithm decides that the current conditioner configuration is correct for the room climate target, then no actions are requested and the loop will continue normally. Otherwise, if the configuration must be changed, a new aperiodic task is executed and actions deriving from this latter will change accordingly the conditioner configuration. Only when these actions are performed, the loop cycle can skip to the next iteration.

2.3 The compute step as an automated reasoning task

As stated in previous sections, hard deadlines can be crucial for a real-time system. Recalling the sense-compute-act model, since sense and act phases usually require a short amount of time to complete, to meet a task deadline basically means that the computation phase should be fast as well as correct. Moreover, since an hard deadline task can happen right after a loop iteration that executes either a firm or a soft deadline task, these two latter kinds of task must have a really fast computing phase. Indeed, if this is not the case the system might get stuck in the execution of a soft deadline task and thus an hard one will never start its computation.

Although the quality and the performance of hardware and software components has clearly improved compared to previous years, still there are multiple problems that requires a significant amount of time to be solved (many practical problems are indeed NP-hard [44]). With this in mind, the loop-

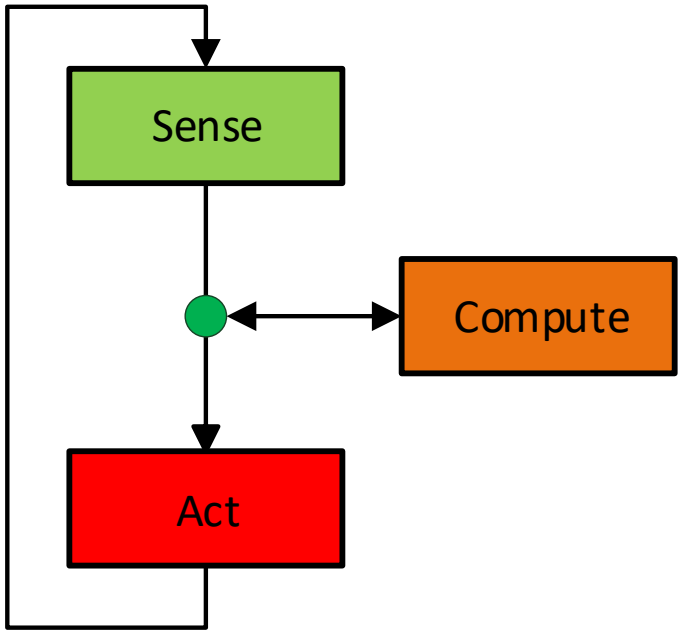


Figure 2.2: The computation phase is moved outside the loop cycle.

ing update environment shown in Figure 2.1b must be reviewed in order to allow the execution of time consuming tasks. Figure 2.2 shows a different model for a looping update environment in which the computation phase is detached from the main process (i.e. the computation phase is executed in asynchronous with respect to the sense and act phases). This approach allows to execute a new task even if the computation phase of a previous one is not completed yet. In this model there must be some synchronization points in which the two processes (the one running the sense-act phases and the one running the computation one) can share information.

In this configuration it is possible to use a declarative paradigm as computation engine while this is unfeasible with the configuration in Figure 2.1b.

Indeed, this kind of paradigm still has a performance bottleneck when either complex problems are involved or a huge input dataset is provided. Moreover, when one actually tries to integrate such formalisms in real-time object-oriented systems new issues arise.

Validity of a solution: stop and restart computation. Since multiple computation phases from different tasks are potentially executed at once, the deadline of some task could change while its own computation is still running. If it happens that a deadline is reached but the computation is not completed yet, then it should be necessary to stop or maybe restart the computation itself. When restarting a reasoning task it can be useful to keep partial results obtained by its previous execution since maybe only minor changes have occurred to the system environment. Moreover, if a new execution of a reasoning program can take advantage from a previous one, the computation of the former will be faster. This sort of reusage of previously obtained results in reasoning is known as *incremental reasoning* [45]. The idea is to store a knowledge base previously obtained and subsequently tuning it by removing or adding logical assertions accordingly to the update in a new input dataset.

Data flow: huge dataset management. Besides the complexity of the problem at hand, the dimension of the input dataset has a huge impact on how fast a solution is provided. Moreover, in a highly dynamic environment there exist a lot of incoming data at different moments: which of these data should the system take into account? In which form? This problem is gaining a great attention from researchers in the field all leading to the *stream reasoning* [46] discipline. Stream reasoning goals consists in being capable of inferring new knowledge in a context that requires a reasoning system to satisfy multiple features, among which:

- handle *volume* and *variety* of data: this means that the system should be capable of processing input datasets whatever their size and structure are. Real-time systems have more and more data sources attached, each one with different data format. All these sources must to be taken into account when performing inference thus the reasoning system should be as flexible as possible;

- handle *velocity* since the previous mentioned data sources provide information at really short time intervals;

- support *incompleteness* of information. Indeed, data sources can get through a break or there could be parts of the environment that are not tracked down at all. This means that the reasoning system should consider that there are some blind spots in its knowledge;

- perform a *fast computation* because of both the velocity of incoming data and the highly dynamic interactions and changes in the sensed environment.

In order to meet such requirements, stream reasoning systems introduced a new operator called the *Window operator* that manage the access to the data stream. A window operator partitions the stream in time-dependent finite blocks, named windows. Such blocks identify data that each reasoning task needs to perform its computation. There exist several windows operators that are based on two fundamental concepts:

- *Landmark windows.* When one aims to decide whether a certain sequence of event appears in a stream or not, it is important to look at the whole data stream, each item that appears in it being labeled with a timestamp. Thus the window on which the reasoning task must be performed includes all the events between the first and the last one occurred.

- *Sliding windows.* In most cases only most recent portions of the stream are interesting for the inferring task. The sliding windows operator creates over time a new window whose width is fixed either in the amount of events that it can contain or in the amount of time units it should cover. Moreover it happens that while the content of a window is important, it is meaningless to take into account the moment in which each event occurred. The *window merge* is an operation that aggregates or filters data contained in a window in fact changing the focus from a temporal approach to an atemporal one. Similar concepts can be found in SQL language extensions created in order to query time-series data [47, 48].

Processes synchronization. Moving the computation phase outside the main loop, thus going from a single to a multi process model, introduces concurrency issues. Since processes share data, which need both shared writing and reading access, there could exist situations leading to inconsistent data.

In order to avoid this, it is necessary to synchronize processes, which means that the shared sources must be managed in a way that limits, or better avoids, the generation of such inconsistent data. There exist a variety of synchronization techniques depending on the specific concurrency problem that one wants to avoid:

- *producer consumer problem*: in this problem two processes, the producer and the consumer, share a common finite buffer to exchange data. The producer put data in the buffer that are then retrieved by the consumer. What happen if the producer tries to add data when the buffer is full? And what are the consequences of the consumer retrieving data from an empty buffer? Underestimate this problem could cause either delays in the execution of certain tasks failing to meet the deadline or system failures. This problem is relevant as shared buffers are at the basis of message passing distributed programming paradigms.

- *shared data structures problem*: a number of processes share resources. If the allocation of these resources is not managed carefully, processes can finally end in a *deadlock* or a *starvation* condition: in the former situation processes get stuck while waiting each other; in the latter, some of the processes, starvers, keep getting access to the resources meanwhile the others, starving, have to wait not being capable to overcome the starvers. In both cases, the computation phase of a task would never end thus causing a failure in the system.

- *readers and writers problem*: as stated, processes can have access to the resources with shared reading or writing access. It should never happen that while one process is reading data, another one modifies (writes) the same resource otherwise some task would work with not valid data thus producing meaningless results.

Chapter 3

Reasoning for game development

As it has been discussed in the previous chapter, tightly coupling of automated reasoning modules in a real-time system is not so trivial.

Let us assume to work with a modular system in which a procedural side and a declarative side can be identified. On the one hand, the procedural side takes care of sensing and acting on the environment surrounding the system; and, on the other hand the declarative side is used to symbolically represent the knowledge of the system, which can be used to take decisions depending on external events and knowledge itself.

The coupling between the two sides involves two different aspects: first, one has to consider how the two sides combine their computational efforts and, second, one has to consider how information is exchanged between the two sides. We will call the two form of coupling *computational coupling* and *data coupling* respectively. Both types of coupling can be said to be either *tight* or *loose* depending on the level of integration between the two sides:

- In *tight computational coupling*, the two systems share CPU resources in a *synchronous* way, i.e. each of the systems waits that the other one finishes its computation to start its own. This approach avoids concurrency problems making the two systems interaction as smooth as possible. On the other hand, in *loose computational coupling*, i.e. each module will run independently from the other one, offering a faster computation at the price of putting much effort on managing concurrency issues.

- In *tight data coupling*, the two systems actually (or virtually) share the same data structures. This integration allows to skip the information passing phase between the two modules usually due to a complete different modeling of the state of the world (object-oriented data structures versus logical assertions). In *loose data coupling*, a message passing or similar infrastructure allows to pass data only on demand and whenever necessary.

Depending on what kind of coupling one wants to achieve, one can end up into four different configurations. The coupling type that one can (or wish) to obtain between procedural and declarative side can be influenced by multiple factors depending on which kind of context one has to work in.

From now on, we will focus on *videogame development*, which is a significant example of real world applications based on imperative languages with strict real-time requirements. In general, AI techniques have (or could have) a role in numerous task applications in the game industry, ranging from programming the behavior of non-player characters to game levels and content generation. Especially when considering real-time videogames, this particular context is really challenging for researchers since it means to work within an highly reactive environment, requiring really fast responses from a KR system.

3.1 Integrating reasoning modules in game engines

When proposing a new tool, like a declarative reasoning engine, for a particular game development environment, one has to deal with the game engine of choice and its runtime execution architecture; the new tool must be performant and easy to use. Making a declarative language module suitable for game development, indeed, is not so trivial. Even if declarative paradigms are expected to be easy to use, game developers usually are specialized only in object-oriented programming. That is why, in addition to the issues exposed in the previous chapter, new obstacles arise in the integration, like

- providing to the declarative side a good representation of the game state, and this means passing from an object-oriented representation to a logical representation;

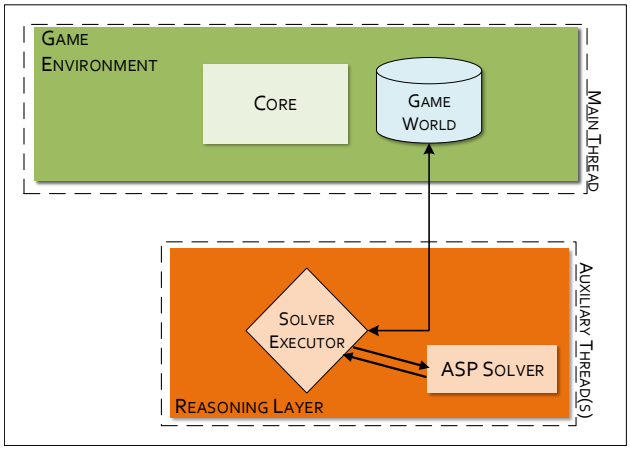


Figure 3.1: Ideal architecture for threads interaction

- returning to the game main program (the procedural side) the results computed on the reasoning side, thus passing from logical assertions to object-oriented data structures;

- keeping high performance of the game while reasoning.

Providing a tool where the above problems have been solved, allows developers to focus on their main goal, i.e. create an high quality AI, instead of dealing with integration aspects. Since reasoning tasks are time-consuming and can easily slow down the game workflow if executed within the main *thread* of the game, we decided to keep a loose CPU coupling, thus letting the reasoning module run in a thread that is not the main one. Unfortunately, a common feature of almost all the game engines is that the main game execution flow is basically *thread unsafe*. A common consequence of this fact is that game engines allow the access to (parts of) the game logic APIs only to the main thread (like in Unreal Engine, Unity, Godot). This feature has an heavy impact on the data coupling level. Indeed, in the ideal configuration for a multi-thread scenario, shown in Figure 3.1, the main thread delegates the execution of a declarative specification to an external thread. Before starting the execution, the external thread gets the current state of the world. Once that the needed information is encoded in logical assertions, the specific solver (i.e. an engine capable of executing declarative specifications) is run. When the solver completes its execution, the main

CPU \ Data	Loose	Tight
Loose		X
Tight		

Table 3.1: Coupling goal for the ThinkEngine framework.

thread is provided with the results found. The main program can now update the objects of the game according with the solution provided by the reasoning side. One can then believe that only a form of loose data coupling is possible.

However, the game logic data structures can be accessed from the declarative side by introducing a transparent information passing layer that allows to achieve a virtual tight data coupling. To this end, we aim to achieve the coupling configuration shown in Table 3.1, i.e. we aim to develop a tool in which we have loose computational coupling, but tight data coupling is obtained nonetheless.

3.1.1 The ThinkEngine framework architecture

Under the previous assumptions, the architecture in Figure 3.1 should be reviewed such that an external thread can be aware about what is happening in the game logic. We thus delegated reasoning tasks to auxiliary threads and introduced an information passing layer allowing the reasoning side to access and act on a representation of the game world. This representation is independent from the game engine APIs and can be accessed separately. The whole run-time ThinkEngine architecture is shown in Figure 3.2.

In particular the ThinkEngine consists of:

1. A *reasoning layer*, in which the game world is accessible and encoded in terms of logical assertions. A reasoning engine can elaborate the current state of the game and produces decisions encoded in its own format.

2. An *information passing layer* which allows to mediate between the reasoning layer and the actual game logic. In this layer, *sensors* store data

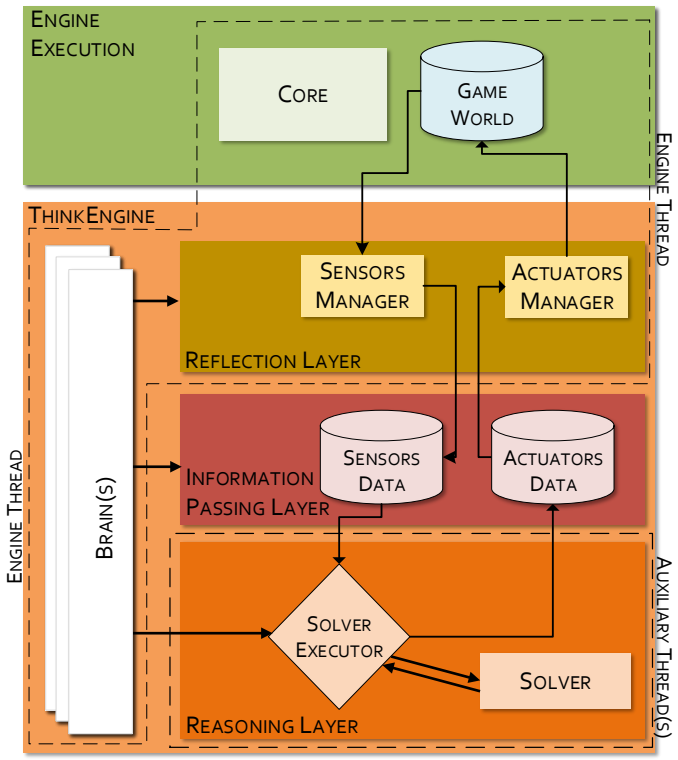


Figure 3.2: General run-time architecture of the ThinkEngine framework

originated from the upper layers. Sensors correspond to parts of the game world data structures which are visible from the reasoning layer. On the other hand, *actuators* collect decisions taken by the reasoning layer and are used to modify the game state.

3. A *reflection layer*, in which a *Sensors Manager* and an *Actuators Manager* keep the mapping between the game world data structures and the lower layers. On the one hand, the sensors manager reads selected game world data which, this way, is made accessible from the reasoning layer. On the other hand, the actuators manager updates selected parts of the game world, based on input coming from the reasoning layer.

4. One or more *brains* that can control the three layers. Each brain can access his own view of the world (i.e. a selected collection of sensors

and actuators), and can be used for programming a separate reasoning activity, like a separate artificial player logic, etc.

A brain interacts with both the sensors manager and the *Solver Executor*. The former is activated periodically or when a trigger condition is met. The sensors manager is responsible of updating all the sensors data mapped to the current brain. Once that a sensors update is completed or a trigger condition is met, the brain starts the solver executor. It will generate a representation W of the world expressed in terms of logical assertions and it will invoke the solver. The solver is fed in input with W and with a logical knowledge base KB encoding the AI of the current brain. As soon as the solver provides *decisions*, the solver executor populates the actuators associated with the corresponding brain. The actuators manager monitors the actuators values and updates accordingly the properties of the game object associated with each actuator.

In order to better understand how sensors and actuators work, in the following we will introduce some formal definitions for the main elements of the framework both for the procedural side and the declarative one. For the the former we will use a pseudo-object-oriented language while for the latter we will stick with the ASP notation.

3.2 Procedural side of the Information Passing Layer

In the following we give some fundamental definitions and then we will formally define both sensors and actuators semantics on the procedural side.

Definition 1. We give the following definitions:

- An *object type* T is a data structure that can include multiple *direct* sub-properties. Each sub-property has a name P_T, and an associated data type D_{P_T}. Property types can be either: a basic type such as string, integer and boolean; an object type itself; or a homogeneous collection. A property P of a sub-property object type S_T of T is said to be an *indirect* property of T and is also denoted, using a dotted notation, by $T.S.P$;

- A collection property models either an array, a list or a vector. The *element type* $\mathcal{T}(C)$ of a collection of objects C is the type of the objects contained in the collection itself. A sub-property P of the i-th element of C is said to be an *implicit* sub-property of T and denoted as $T.C[i].\mathcal{T}(C).P$.

- An *object instance* O of an object type T is a value assignment to all the sub-properties of T. For a basic type property P, a value assignment is a value of the given data type; a value assignment for an object property P' of data type T' is an object instance of type T'; a value assignment for a collection property C is a possibly empty sequence $O_1, \ldots, O_n$ of object instances each of type $\mathcal{T}(C)$.

Definition 2. A *frame* of a video game is a snapshot of the game state taken at (almost) regular intervals. The *frame rate* represents the frequency at which consecutive frames are taken (for instance 60 frames per second).

3.2.1 Sensors definition

Definition 3. It is given an object type T and its properties

$$\mathcal{P} = \{P_T^1, \ldots, P_T^i, T.S.P^1, \ldots, T.S.P^j, T.C.\mathcal{T}(C).P^1, \ldots, T.C.\mathcal{T}(C).P^k\},\ i, j, k \geq 0$$

A *Sensor Configuration* SC^T for T is a subset of $\mathcal{P}$.

Definition 4. It is given a sensor configuration SC^T and an object instance O of type T; a *sensor reading* $v(P)$ of a direct or indirect basic property P such that $P \in SC^T$ is the value assignment of P for the object O at a specific frame of the game.

Definition 5. It is given a sensor configuration SC^T and an object instance O of type T; a *Simple Sensor* $SS(SC^T)$ is a set of sequences of sensor readings $V = \{< v(P_1)_1, \ldots, v(P_1)_n >, \cdots < v(P_m)_1, \ldots, v(P_m)_n >\}$ for each direct and indirect basic property P_i such that $P_i \in SC^T$. Each sensor is associated with a unique name.

Definition 6. It is given a sensor configuration SC^T and an object instance O of type T; an *Advanced Sensor* $AS(SC^T)$ is a simple sensor enriched with a set of n collections of simple sensors $\{C_1, \ldots, C_n\}$ where each C_i corresponds to either a bidimensional array or a list property P_i such that $P_i \in SC^T$.

Each C_i contains a number of simple sensors equal to the number of object instances contained in the value assignment of P_i for the object instance O. We denote by C_i^j each simple sensor storing a single sensor reading for each implicit property $T.P_i[j].\mathcal{T}(P_i).P$.

Example 1. Suppose it is given a game object *player* with the following properties:

- *name* whose type is a string;
- *dead* whose type is boolean;
- *position* which is an object with three integer properties: *x,y,z*;
- *neighbors* which is a list of *player* instances.

The four properties *name*, *dead*, *position* and *neighbors* are direct properties of *player* while *x,y,z* are indirect properties of *player* by means of the object referred by the property *position*. The direct properties of the objects of type *player* contained in *neighbors* are implicit properties of *player* by means of the property *neighbors*. We can create a sensor configuration as

$$SC^{player} = \{name, position.x, position.y, position.z, neighbors.player.dead\}.$$

A sensor $AS(SC^{player})$, besides a name `sensorName`, will include the following data structures:

```
Map<String,List<String>> stringProperties
Map<String,List<Integer>> integerProperties
Map<String,List<SimpleSensor>> listsProperties
```

The map `stringProperties` will contain only one entry

$$< name, \{playerName_1, ..., playerName_n\} >$$

while the `integerProperties` map will contain three different entries

$$< position.x, \{x_1, ..., x_n\} >$$

$$< position.y, \{y_1, ..., y_n\} >$$

$$< position.z, \{z_1, ..., z_n\} >$$

where $playerName_i, x_i, y_i, z_i$ are the sensor readings, respectively, of the properties *name, x,y* and z at the i-th frame. The map `listsProperties` will contain a number of entries equal to the number of objects in the list *neighbors*[1].

3.2.2 Actuators definition

Definition 7. It is given an object type T and its properties

$$\mathcal{P} = \{P_T^1, \ldots, P_T^i, T.S.P^1, \ldots, T.S.P^j\}, \quad i, j \geq 0$$

an *Actuator Configuration* AC^T for T is a subset of $\mathcal{P}$.

Definition 8. It is given an actuator configuration AC^T and an object instance O of type T; an *Actuator* $A(AC^T)$ consists of a single value for each property P such that $P \in AC^T$. These values are used to update the corresponding property of the object instance O. Each actuator is associated with a unique name.

Example 2. Recalling the game object *player* of the Example 1, we can create an actuator configuration as

$$AC^{player} = \{dead, position.x, position.y, position.z\}.$$

An actuator $A(AC^{player})$, besides a name `actuatorName`, will have the following data structures:

```
Map<String,Boolean> boolProperties
Map<String,Integer> integerProperties
```

The map `boolProperties` will contain only one entry

$$< dead, isDead >$$

while the `integerProperties` map will contain three different entries:

$$< position.x, nextX >$$

$$< position.y, nextY >$$

$$< position.z, nextZ >$$

where $isDead, nextX, nextY, nextZ$ are the value computed by the reasoning module and that should be set to the properties *dead, x,y* and z.

[1]Note that our current ThinkEngine implementation does not store historical readings for implicit properties: only one sensor reading at the time is materialized. At each update the previous reading will be overwritten.

3.3 Declarative side of the Information Passing Layer

In order to give a clear idea of what is done in the declarative side of the ThinkEngine framework, we will first give some details on the ASP semantic citing [49] and then we will provide some formal definition for both sensors and actuators on the ASP side .

3.3.1 Syntax and semantic of Answer Set Programming

An ASP program Π is a finite set of rules of the form:

$$a_1 \vee \ldots \vee a_n \leftarrow b_1, \ldots, b_j, not\ b_{j+1}, \ldots, not\ b_m \tag{3.1}$$

where $a_1, \ldots, a_n, b_1, \ldots, b_m$ are atoms and $n \geq 0$, $m \geq j \geq 0$. In particular, an *atom* is an expression of the form $p(t_1, \ldots, t_k)$, where p is a predicate symbol and $t_1, \ldots, t_k$ are *terms*. Simple terms are alphanumeric strings, and are distinguished in variables and constants. A term is either a simple term or a *functional term*. A functional term is in the form $f(t_1, \ldots, t_n)$, where $t_1, \ldots, t_n$ are terms. According to the Prolog's convention, only variables start with an uppercase letter. A *literal* is an atom a_i (positive) or its negation $not\ a_i$ (negative), where *not* denotes the *negation as failure*. Given a rule r of the form (3.1), the disjunction $a_1 \vee \ldots \vee a_n$ is the *head* of r, while $b_1, \ldots, b_j, not\ b_{j+1}, \ldots, not\ b_m$ is the *body* of r, of which $b_1, \ldots, b_j$ is the *positive body*, and $not\ b_{j+1}, \ldots, not\ b_m$ is the *negative body* of r. A rule r of the form (3.1) is called a *fact* if $m = 0$ and a *constraint* if $n = 0$. An object (atom, rule, etc.) is called *ground* or *propositional*, if it contains no variables. Rules and programs are *positive* if they contain no negative literals, and *general* otherwise. Given a program Π, let the *Herbrand Universe* U_Π be the set of all *ground* terms that can be built using constants and function symbols appearing in Π and the *Herbrand Base* B_Π be the set of all possible *ground* atoms which can be constructed from the predicate symbols appearing in Π with the constants of U_Π. Given a rule r, $Ground(r)$ denotes the set of rules obtained by applying all possible substitutions σ from the variables in r to elements of U_Π. Similarly, given a program Π, the *ground instantiation* $Ground(\Pi)$ of Π is the set $\bigcup_{r \in \Pi} Ground(r)$.

For every program Π, its answer sets are defined using its ground instantiation $Ground(\Pi)$ in two steps: first, answer sets of positive programs are

defined, then a reduction of general programs to positive ones is given, which is used to define answer sets of general programs.

A set L of ground literals is said to be *consistent* if, for every literal $\ell \in L$, its negated literal *not* ℓ is not contained in L. Given a set of ground literals L, $L_{|+} \subseteq L$ denotes the set of positive literals in L. An interpretation I for Π is a consistent set of ground literals over atoms in B_Π. A ground literal ℓ is *true* w.r.t. I if $\ell \in I$; ℓ is *false* w.r.t. I if its negated literal is in I; ℓ is *undefined* w.r.t. I if it is neither true nor false w.r.t. I. A constraint c is said to be *violated* by an interpretation I if all literals in the body of c are true. An interpretation I is *total* if, for each atom a in B_Π, either a or *not* a is in I (i.e., no atom in B_Π is undefined w.r.t. I). Otherwise, it is *partial*. A total interpretation M is a *model* for Π if, for every $r \in Ground(\Pi)$, at least one literal in the head of r is true w.r.t. M whenever all literals in the body of r are true w.r.t. M. A model X is an *answer set* for a positive program Π if any other model Y of Π is such that $X_{|+} \subseteq Y_{|+}$.

The *reduct* or *Gelfond-Lifschitz transform* of a general ground program Π w.r.t. an interpretation X is the positive ground program Π^X, obtained from Π by (i) deleting all rules $r \in \Pi$ whose negative body is false w.r.t. X and (ii) deleting the negative body from the remaining rules. An answer set of Π is a model X of Π such that X is an answer set of $Ground(\Pi)^X$. We denote by $AS(\Pi)$ the set of all answer sets of Π, and call Π *coherent* if $AM(\Pi) \neq \emptyset$, *incoherent* otherwise.

3.3.2 Sensors on the declarative side

The ThinkEngine framework offers an ASP representation for each sensor that has been configured. Taking up the idea of the sliding windows discussed in the Section 2.3, the translations from object data structures to logical assertion is preceded by an aggregation phase in which the collection of sensor readings is replaced by the results of some aggregation function.

Definition 9. A *window* is a function $w^{\#n}(l_1) = l_2$ that takes in input a sequence of values $l_1 =< e_1, ..., e_m >$ and returns a new collection l_2 where

$$l_2 = \begin{cases} < e_{m-n+1}, ..., e_m >, & \text{if } m > n \\ l_1, & \text{if } m \leq n \end{cases}$$

Definition 10. An *aggregate function* is a function $f(w^{\#n}(l_1)) = \bar{l}$ that

takes as argument a collection of at most n values $w^{\#n}(l_1)$ and returns a single summary value $\bar{l}$.

Examples of aggregate functions are *maximum*, *minimum*, *average*, *mode*, *median*, *first* and *last*.

Definition 11. It is given a simple sensor $SS(SC^T)$, an aggregate function f_i for each property P_i such that $P_i \in SC^T$ and a window function $w^{\#n}$; the *value* of P_i with respect to $SS(SC^T)$ is defined as $\bar{V_i} = f_i(w^{\#n}(V_i))$.

With this approach, each property tracked by an advanced sensor $AS(SC^T)$ will correspond to exactly one ASP atom a. The syntax of a can be described in an extended Backus–Naur form as follows:

```
atom = sensorName,(,gameObjectName,(, property ,)).
sensorName = lowCaseString
gameObjectName = lowCaseString
property = directBasicProperty|directArrayRank2Property|
           directListProperty|indirectProperty
lowCaseString = lowLetter,{letter|digit|punctuation}
directBasicProperty = lowCaseString,(,propertyValue,)
directArrayRank2Property =  lowCaseString,(,{digit},
         {digit},arrayType,(,directBasicProperty,))
directListProperty =  lowCaseString,(,{digit},
              listType,(,directBasicProperty,))
indirectProperty = lowCaseString ,(,property,)
propertyValue = ",lowCaseString,"|lowCaseString|
             {digit},[",",{digit}]
letter = lowLetter|upLetter
lowLetter = a-z
upLetter = A-Z
digit = 0-9
punctuation = ","|.|!|?
```

Definition 12. It is given an advanced sensor $AS = AS(SC^T)$, with name μ, and an object instance O of type T, with name γ; a *sensor mapper* $M_S(AS, P_i) = s_i$ is a function that takes as arguments AS and a property P_i such that $P_i \in SC^T$ and returns a string s_i representing a logical assertion encoding P_i. M_S behaves differently based on the property type:

- if P_i is a direct property denoted by P_T then $s_i = \mu(\gamma(P_T(v(P_i))))$. where $v(P_i)$ is the value of P_T with respect to AS;

- if P_i is a indirect property denoted by $T.S.P$ then $s_i = \mu(\gamma(T(S(P(v(P_i))))))$. where $v(P_i)$ is the value of $T.S.P$ with respect to AS;
- if P_i is a bidimensional array implicit property denoted by $T.C[j][k].\mathcal{T}(C).P$ then $s_i = \mu(\gamma(T(C(j,k,\mathcal{T}(C)(P(v(P_i)))))))$. where $v(P_i)$ is the value of $T.C[j][k].\mathcal{T}(C).P$ with respect to AS;
- if P_i is a list implicit property denoted by $T.C[j].\mathcal{T}(C).P$ then $s_i = \mu(\gamma(T(C(j,\mathcal{T}(C)(P(v(P_i)))))))$. where $v(P_i)$ is the value of $T.C[j].\mathcal{T}(C).P$ with respect to AS.

Remark. In our current implementation, the size of the window function used by sensors is set to be equal to 200 in order to avoid memory and performance issues. For matrix and list properties it is not needed to apply aggregation functions as a default window of size 1 is taken.

3.3.3 Actuators on the declarative side

Values for the properties of each actuator are retrieved from an answer set resulting from an ASP program execution. The syntax of the atoms mapped to an actuator $A(AC^T)$ can be described in an extended Backus–Naur form as follows:

```
atom = "setOnActuator(",actuatorName,(,gameObjectName,
           (, property ,))).
actuatorName = lowCaseString
gameObjectName = lowCaseString
property = directBasicProperty|indirectProperty
lowCaseString = lowLetter,{letter|digit|punctuation}
directBasicProperty = lowCaseString,(,propertyValue,)
indirectProperty = lowCaseString ,(,property,)
propertyValue = lowCaseString|{digit},[",",{digit}]
letter = lowLetter|upLetter
lowLetter = a-z
upLetter = A-Z
digit = 0-9
punctuation = ","|.|!|?
```

Definition 13. It is given an actuator $A = A(AC^T)$, with name μ, and an object instance O of type T, with name γ; an *actuator mapper* $M_A(A, s_i) = P_i$

is a function that takes as arguments A and a string s_i representing a logical assertion encoding a property P_i such that $P_i \in AC^T$ and returns a value assignment for P_i. M_A behaves differently based on the property type:

- if P_i is a direct property denoted by P_T then $s_i = \mu(\gamma(P_T(v(P_i))))$. where $v(P_i)$ is the value to assign to P_i for the property A;
- if P_i is a indirect property denoted by $T.S.P$ then $s_i = \mu(\gamma(T(S(P(v(P_i))))))$. where $v(P_i)$ is the value to assign to P_i for the property A.

3.3.4 Declarative side semantic

Definition 14. We define a *brain* B as a triple $\langle SC, AC, \Pi \rangle$, where:

- $SC =< SC^{T_1}, \ldots, SC^{T_i} >=< SC_1, \ldots, SC_i >$ is a set of sensor configurations;
- $AC =< AC^{T_{i+1}}, \ldots, AC^{T_n} >=< AC_{i+1}, \ldots, AC_n >$ is a set of actuator configurations;
- Π is an ASP program.

Definition 15. It is given a brain B and

- a set of advanced sensors $AS =< AS(SC_1), \ldots, AS(SCT_i) >=< AS_1, \ldots, AS_i >$;
- a set of actuators $A =< A(AC_{i+1}), \ldots, A(AC_n) >=< A_{i+1}, \ldots, A_n >$;
- a set of sensor mappings $\mathcal{S} = \{M_S(AS_1, P_1^{SC_1}), \ldots, M_S(AS_1, P_j^{SC_1}), \ldots, M_S(AS_i, P_1^{SC_i}), \ldots, M_S(AS_i, P_k^{SC_i})\}$;
- a set of actuator mappings $\mathcal{A} = \{M_A(A_{i+1}, P_1^{AC_{i+1}}), \ldots, M_A(A_{i+1}, P_l^{AC_{i+1}}), \ldots, M_A(A_n, P_1^{AC_n})), \ldots, M_A(A_n, P_m^{AC_n})\}$;

Let

- $\mathcal{F}(\Pi)$ be the set of the input facts of Π;

- $Ans(\Pi \bigcup \mathcal{F}(\Pi))$ be the ordered set of the answer sets of $\Pi \bigcup \mathcal{F}(\Pi)$;
- $Ans(\Pi \bigcup \mathcal{F}(\Pi))[0]$ be the first answer set in $Ans(\Pi \bigcup \mathcal{F}(\Pi))$;
- $\overline{Ans} = Ans(\Pi \bigcup \mathcal{F}(\Pi))[0] \setminus \mathcal{S}$.

then $\mathcal{A}$ is a valid *decision* for B if $\mathcal{A} \subset \overline{Ans}$.

3.4 A ThinkEngine implementation: Unity and ASP

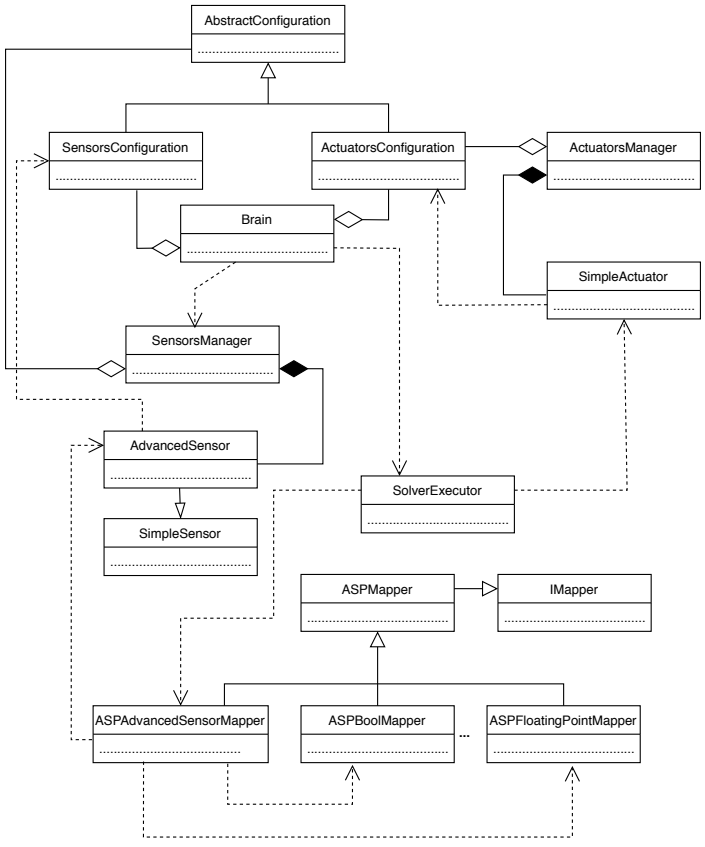


Figure 3.3: The ThinkEngine asset class diagram.

We provided an actual implementation of the ThinkEngine framework deployed in the Unity 3D game engine and featuring an answer set solver at the core of the declarative side. The ThinkEngine has been developed as an Unity asset using the `C#` programming language.

Every element discussed in this chapter, except for the ASP solver, has been implemented as a `C#` class while the object instances introduced in Definition 1 are Unity's game objects. Game developers interact with the ThinkEngine by means of some graphical editor views at design time. Sensor and actuator configurations have been implemented as classes containing the names of the properties (either direct, indirect or implicit) of some `GameObject`. At run-time an instance of the class Brain instantiates an `AdvancedSensor` for each `SensorConfiguration` attached to it. In the same way, it instantiates a `SimpleActuator` for each `ActuatorConfiguration`. The `SensorsManager` keeps a map in which each Brain instance of the game execution is associated with its own sensors. In the same way, the `ActuatorsManager` keeps a map in which each Brain instance is associated with its own actuators. The Brain instance demands a sensors update to the `SensorManager` on a trigger event, or periodically. These trigger events are implemented by polling within a Unity co-routine some trigger boolean function. In the same way, another co-routine checks for others boolean functions in order to demand to a `SolverExecutor` instance an execution of the ASP solver. Each `SolverExecutor` runs in a separated thread that waits to be notified by a brain to start its computation. As soon as the `SolverExecutor` is notified to start the execution, it demands to an `ASPAdvancedSensorMapper` for the logical assertion representation of all the sensors attached to the `Brain instance`. The result is written in a file and the ASP solver is invoked with both the ASP program and the input facts. Once that the solver terminates its own execution, the `SolverExecutor` sends the provided answer set to each actuator of the `Brain`. In this way, each `SimpleActuator` populates its own data structures. The `ASPAdvancedSensorMapper` translate the data structures of an `AdvancedSensor` by means of other mappers (`ASPBoolMapper`, `ASPIntegerMapper` and so on). Each of these latter take care of translating properties of a specific type. Finally, the `ActuatorsManager` periodically checks if there are action (`SimpleActuator`) to apply to the game world. These updates are performed only if some precondition boolean function is satisfied.

Chapter 4

A showcase for ThinkEngine: Tetris

In order to give an idea of how AI declarative modules can be integrated within applications developed in Unity via the ThinkEngine, we developed a showcase application. We started from a public available open-source project[1], inspired from the original Tetris game, and we modified this project to obtain an automated player whose artificial intelligence is managed by an ASP program. Note that we are not proposing a state-of-the-art Tetris player, rather a demonstration of how an AI can be easily developed by means of logical rules and then deployed in Unity.

In the following we briefly describe how our framework has been set up and configured in order to cooperate with the Unity game scene. First, we will show how we configured the *sensors and actuators modules*, then how the *brain component* were set up. Finally we will describe our *ASP encoding* giving some actual game-play example and discussion on performance.

4.1 Sensors and Actuators Configuration

Developers can access a list of the GOs used in the game scene via a custom Unity window editor[2] as in Figure 4.1. It is possible to browse objects and

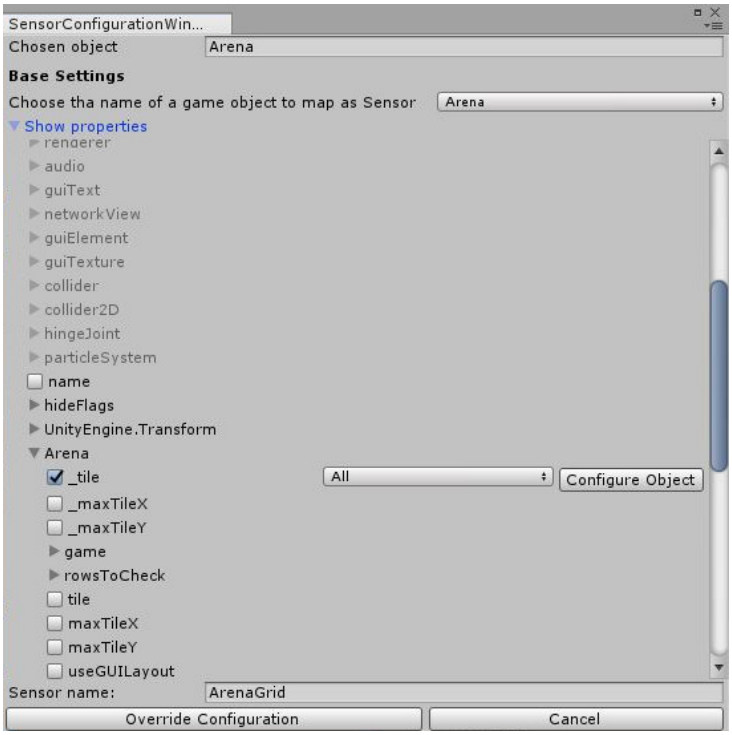


Figure 4.1: Editor window for the sensor configuration

select which properties are mapped on the reasoning side.

We will use next some of the typical terminology used to describe our infrastructure and the Tetris game, as recalled here:

Arena: as shown in Figure 4.1, the arena is a GO that contains all the properties relative to the playable game scene (i.e. a matrix of tiles, the properties *maxTileX*, *maxTileY* etc.);

_tiles: a matrix of GOs of type ArenaTile. This matrix can be expanded by the user in order to configure some extra properties;

Tetromino: a geometric shape composed of four squares;

currentTetromino: in the Tetris game it represents the tetromino that is currently dropping in the Arena;

[1] `https://github.com/MaciejKitowski/Tetris`

[2] I.e. a window similar to the Unity inspector. The inspector displays detailed information about the currently selected game object, including all attached components.

Spawner: a GO that manages the generation of a new tetromino when the previously created one can not drop further down in the Arena.

We bound to the reasoning side, as sensors, the *Arena*, the *currentTetromino* and the *Spawner*, and, in a similar way, we configured the actuators. By means of the *Actuator Configuration Window* one can select the AI script that is needed to be mapped within the ASP module. The single selected actuator, called *player*, contains the properties: *nMove*, *nLatMove*, *nRot*, *typeLatMove*. The meaning of these properties will be explained in subsection 4.1.1.

4.1.1 Brain Component

After configuring the sensors (*arenaGrid*, *tetromino* and *spawner*) and the actuator (*player*), we added to the GOs hierarchy a new GO with an attached component of type `Brain` and a `C#` script called `AIPlayer`. The brain consists in a standard script belonging to the ThinkEngine asset that will coordinate sensors, the actuator and the solver executor.

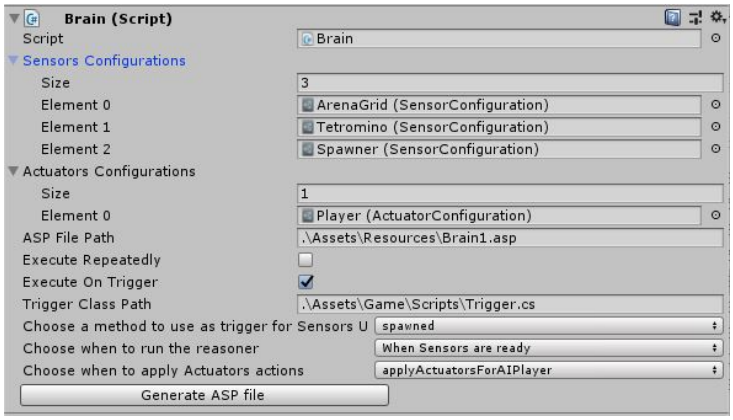


Figure 4.2: Configuration of the brain.

The brain component can be configured via the inspector tab. Sensors, actuators and some other additional features can be attached to a brain via the visual interface. In our example, we setup the conditions[3] to meet in order to *a)* update the sensors every time that a new tetromino is spawned and run

[3]Conditions are selected from a set of boolean functions customized by the developer.

the ASP Solver as soon as sensors have data available; *b)* let the actuators manager apply the actuators actions, if the current tetromino corresponds to the one that was on the grid when the sensors were updated.

When the game starts, thus at run-time, the brain will start updating sensors and will also run an external thread that will execute the ASP solver if sensors have data to share and the solver is not already running. Every time that it is necessary to invoke the solver, the Sensor Mapper produces a representation, in the form of logical assertions, of the filtered sensors values attached to the brain. Then, the ASP solver is invoked by passing it this sensor representation and a knowledge base KB_T expressing the desired brain AI. After the ASP solver ends its execution, its answer sets will be mapped to actuators by means of the actuators manager, thus influencing the game world.

In the setting of the Tetris game, the solver's output encodes the position and orientation in which the current tetromino should be dropped. This is then translated to the corresponding number of rotations and lateral moves of the tetromino. In turn, a corresponding number of simulated swipes is commanded via Unity procedural code in the `AIPlayer` and the tetromino is eventually dropped.

4.1.2 ASP Encoding

The ASP declarative specification KB_T driving the brain decision is based on the *Guess/Check/Optimize* paradigm [50]. The idea is to range in the search space of columns of the Tetris grid and of rotations of the tetromino; to exclude combinations of columns and rotations such that the piece cannot be geometrically placed; choose the optimal combination among the remaining candidates. For the sake of simplicity the optimality criterion looks for positions not leaving holes in the grid, and with lesser priority, lower dropping positions in the grid are preferred.

The *guess* phase is expressed in the rule

```
bestSol(X,Y,C) | notBestSol(X,Y,C):-col(C), availableConfig(X,Y).
```

where the `availableConfig(X,Y)` predicate keeps track of all the possible rotations for the current tetromino. This assertion, combined with the strong constraints

```
:- #count{Y,C: bestSol(X,Y,C)} > 1.
:- #count{Y,C: bestSol(X,Y,C)} = 0.
```

assures that each model produced by the solver will contain exactly one `bestSol`. The lowest row that the tetromino can reach when positioned with a given rotation in a chosen column is described as follows.

```
1. free(R,C,C1):- tile(R,C,true), C1=C+1.
2. free(R,C,C2):- free(R,C,C1), tile(R,C1,true),C2=C1+1.
3. firstEmpty(R):- nCol(C), #max{R1:free(R1,0,C)}=R.
4. canPut(R):- bestSol(X,Y,C), free(R,C,C1), firstEmpty(R), confMaxW(X,Y,W),
              C1=C+W.
5. canPut(R):- bestSol(X,Y,C), canPut(R1), free(R,C,C1), confMaxW(X,Y,W),
             C1=C+W, R=R1+1.
6. freeUpTo(R):- canPut(R), not canPut(R1), R1=R+1.
7. oneMore(R1):- bestSol(X,Y,C), botSpace(X,Y,I,J), freeUpTo(R), R1=R+1,
               free(R1,C1,C2), C1=C+I, C2=C+J.
8. twoMore(R1):-bestSol(X,Y,C), oneMore(R), extraRow(X,Y), botSpace(X,Y,I,J)
    ,           free(R1,C1,C2), R1=R+1, C1=C+I, C2=C+J.
9. bestRow(R):- freeUpTo(R), not oneMore(R2), botSpace(X,Y,0,0),  R2=R+1,
              bestSol(X,Y,_).
10. bestRow(R1):- freeUpTo(R), not oneMore(R2), not extraRow(X,Y),
                 bestSol(X,Y,_),  not botSpace(X,Y,0,0), R1=R-1, R2=R+1.
11. bestRow(R1):-bestSol(X,Y,_), not oneMore(R2), freeUpTo(R), extraRow(X,Y)
    ,             not botSpace(X,Y,0,0), R1=R-2, R2=R+1.
12. bestRow(R):- oneMore(R), not twoMore(R1), bestSol(X,Y,_), R1=R+1,
               not extraRow(X,Y).
13. bestRow(R):- twoMore(R).
14. :-#count{R:bestRow(R)}=0.
```

The `tile`[4] predicate is used to derive in which rows the tetromino can be placed. The space occupied by a tetromino is encoded by a number of assertions, like e.g. `confMaxW(x,y,w)` which expresses that the maximum horizontal amount of cells occupied by the tetromino `x` on which it has been applied the rotation `y` is `w`; other similar assertions are `botSpace(x,y,c,c1)`, `topSpace(x,y,h)`, `leftSpaceWrtSpawn(x,y,l)`, `extraRow(x,y)`.

Rules 1. and 2. describe, for each row of the arena, all the sequences of free slots of the matrix $(0-2, 0-3...0-10, 1-2, ..., 1-10...$, note that the second index is exclusive). Rule 3. derives the highest row in the arena which is completely empty, thus identifies the first row in which the tetromino can be placed in whatever column. Starting from this row, rules from 4. to 8. describe in which row the tetromino, in the chosen rotation configuration, is allowed to be placed, according also with the current tetromino shape. Finally, rules from 9. to 13., describe the lowest line that the tetromino will drop to.

The next set of rules describes the highest row the tetromino will reach

[4]This assertion maps facts derived from the *ArenaGrid* sensor mapped by the predicate `arenaGrid(arena(arena(tiles(X,Y,arenaTile(empty(T))))))`.

once it is placed (rule 15.) and how many holes will remain in the row immediately below (rules from 16. to 20.).

```
15. reach(R):- bestSol(X,Y,_), bestRow(R1), topSpace(X,Y,W), R=R1-W.
16. hole(R,C1):-bestSol(X,Y,C), bestRow(R1), tile(R,C1,true),confMaxW(X,Y,W)
    ,          R=R1+1, C1>=C, C<W1, W1=C+W.
17. hole(R,C1):- bestSol(X,Y,C), botSpace(X,Y,I,J), tile(R,C1,true), L=I+J,
             L>0, C1>=C, C1<C2, C2=C+I, oneMore(R).
18. hole(R,C1):- bestSol(X,Y,C), botSpace(X,Y,I,J), tile(R,C1,true), L=I+J,
             L>0, C1>=C2, C2=C+J, C1<C3, C3=C+W, oneMore(R), confMaxW(X,Y,W)
    .
19. hole(R,C1):- bestSol(X,Y,C), botSpace(X,Y,I,J), tile(R,C1,true), L=I+J,
             L>0, C1>=C, C1<C2, C2=C+I, twoMore(R).
20. hole(R,C1):- bestSol(X,Y,C), botSpace(X,Y,I,J), tile(R,C1,true), L=I+J,
             L>0, C1>=C2, C2=C+J, C1<C3, C3=C+W, twoMore(R), confMaxW(X,Y,W)
    .
```

The next fragment of declarative code represent optimization criteria, which are expressed in terms of *weak constraints*. Roughly speaking, a weak constraint is a condition that, if met, increases the cost of a possible tetromino drop configuration. In the presence of weak constraints, the lowest cost answer set is preferred. Weak constraints can have priority levels.

```
21. :~ #count{R,C:hole(R,C)}=N, #int(N1),#int(N),N1=3*N. [N1:4]
22. :~ bestRow(R),numOfRows(N),D=N-R. [D:4]
23. :~ reach(R),numOfRows(N),D=N-R.  [D:3]
24. :~ bestSol(X,Y,C). [C:2]
25. :~ bestSol(X,Y,C). [Y:1]
```

Weak constraints 21. and 22. have been assigned to the same priority (4) since we want, at the same time, to minimize the number of holes and to maximize the lowest line that the tetromino will drop to. However, since we want to give a bit more importance to the holes, we decided to assign a triple weight with respect to the lowest row optimization criterion. At a lower priority level, we find the minimization of the row reached in height by the tetromino (23.). The last two constraints, 24. and 25., are used to assure that no more than one answer set is produced. Indeed, when having two answer sets with the same costs for respectively the number of holes criterion, for the lowest line criterion and for the top most row criterion, we will choose the solution occupying the leftmost column and requiring the lowest number of rotations.

Note that this artificial player, although not optimal, can be easily modified by changing the heuristic associated to the weight of constraint in 21.; introducing new weak constraints expressing other desiderata; changing the

priority level of the constraints and so on. Finally, actuators' atoms are assigned

```
26. setOnActuator(player(aI(assetsScriptsAIPlayer(numOfLateralMove(N))))):-
    bestSol(X,Y,C), spawnCol(S), spaceToSpawn(X,Y,L), N=S-D,  D=C+L, D<S.
27. setOnActuator(player(aI(assetsScriptsAIPlayer(numOfLateralMove(N))))):-
    bestSol(X,Y,C) ,spawnCol(S), spaceToSpawn(X,Y,L),  N=D-S, D=C+L, D>=S.
28. setOnActuator(player(aI(assetsScriptsAIPlayer(numOfRotation(N))))):-
    bestSol(_,N,_).
29. setOnActuator(player(aI(assetsScriptsAIPlayer(typeOfLateralMove(l))))):-
     bestSol(X,Y,C), spawnCol(S), D=C+L, D<S, spaceToSpawn(X,Y,L).
30. setOnActuator(player(aI(assetsScriptsAIPlayer(typeOfLateralMove(r))))):-
     bestSol(X,Y,C), spawnCol(S), D=C+L, D>=S, spaceToSpawn(X,Y,L).
31. setOnActuator(player(aI(assetsScriptsAIPlayer(numOfMove(X))))):-
    setOnActuator(player(aI(assetsScriptsAIPlayer(numOfLateralMove(N))))),
    setOnActuator(player(aI(assetsScriptsAIPlayer(numOfRotation(N1))))),
 X=N+N1.
```

Rules from 26. to 31. assign the properties' values to be mapped back to the game logic. The `spawnCol` literal represents in which column the tetromino is spawned while the `spaceToSpawn` is the number of the tetromino's squares, in the configuration proposed as solution, that are on the left of the spawn column. The number of lateral moves to apply to the tetromino are modeled in 26. and 27. depending on whether the leftmost square of the tetromino is on the left or on the right of the spawn column. The number of rotations to apply to the tetromino are retrieved directly from the `bestSol`. The type of lateral move, left (l) or right (r), is inferred by rules 29. and 30. with the same logic used in 26. and 27.. Finally, the total number of move that should be performed is derived in 31. as sum of both lateral move and rotations.

The above artificial player, including both the declarative code and all the procedural code, can be downloaded at `https://github.com/DeMaCS-UNICAL/Tetris-AI4Unity`.

4.2 A game-play

In order to better illustrate how Unity and the ASP solver interact via the ThinkEngine framework, we will examine a game configuration that generates some input for the solver and reacts at the answer set provided. Given the Tetris situation in Figure 4.3, the input facts provided by the ThinkEngine to the ASP solver are the following

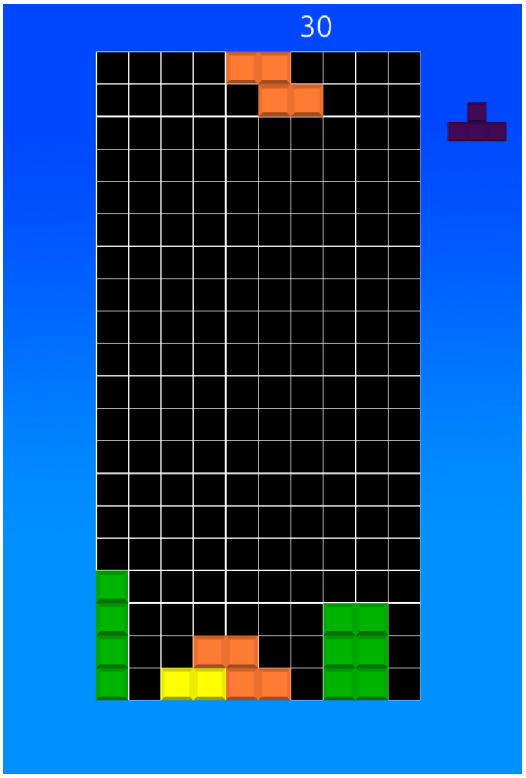


Figure 4.3: Game configuration with the tetromino "Z" just spawned.

```
arenaGrid(arena(arena(tiles(0,0,arenaTile(x(0))))).
arenaGrid(arena(arena(tiles(0,0,arenaTile(y(0))))).
arenaGrid(arena(arena(tiles(0,0,arenaTile(empty(true)))))).
                                        :
arenaGrid(arena(arena(tiles(0,19,arenaTile(empty(false)))))).
                                        :
arenaGrid(arena(arena(tiles(1,18,arenaTile(empty(true)))))).
arenaGrid(arena(arena(tiles(1,19,arenaTile(empty(true)))))).
                                        :
spawner(tetrominoSpawner(tetrominoSpawner(currentTetromino(5)))).
```

Note that the `x` property pointed by the atom `arenaGrid` value increases from the left to the right of the grid, while the `y` increases from the top to the bottom. The `empty` property is `false` if the corresponding `(x,y)` cell is occupied by a piece of some tetromino (the `(0,19)` cell for example). The tetromino just spawned is the "Z" and it is represented with the number 5.

With this input dataset, the decision taken (i.e. the best answer set

produced, filtered on the actuators atoms), is

```
{setOnActuator(player(aI(assetsScriptsAIPlayer(numOfLateralMove(3))))),
setOnActuator(player(aI(assetsScriptsAIPlayer(numOfRotation(1))))),
setOnActuator(player(aI(assetsScriptsAIPlayer(typeOfLateralMove(1))))),
setOnActuator(player(aI(assetsScriptsAIPlayer(numOfMove(4)))))}
```

and its cost is

```
Cost ([Weight:Level]): <[1:1],[1:2],[3:3],[1:4]>.
```

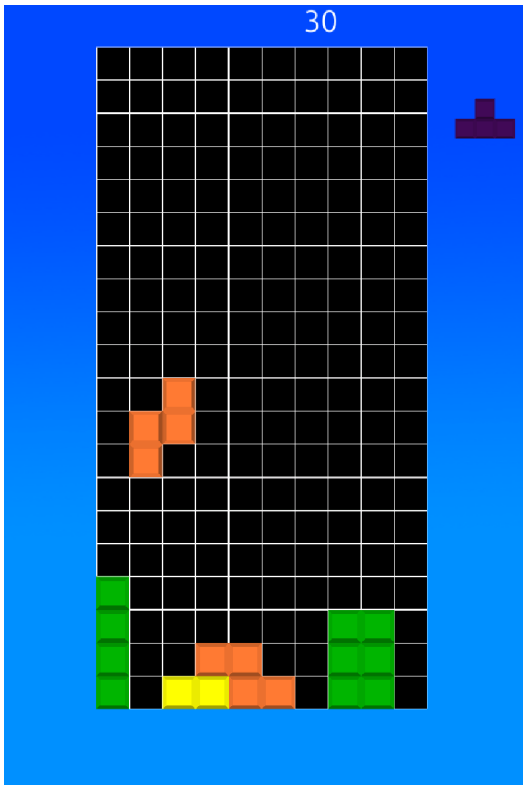


Figure 4.4: Game configuration after that decisions derived from the answer set have been applied.

This decision will lead to the configuration in figure 4.4 since, according to it, the AIPlayer script will reproduce 3 left movements followed by 1 rotation. In fact, this placement results to be exactly the one that we want:

- no holes are left by positioning there the tetromino;

- it is impossible to place the tetromino in a lower place in the grid, indeed it will hit the bottom of the grid once placed.

4.3 Benchmark

One of the most common measuring indicators used for assessing the performance of a videogame is the framerate. i.e. the number of frames that can be displayed in a second.

The framerate appears to be a good measure even for the evaluation of the ThinkEngine impact on the game performance. Using the Tetris showcase, we compared the framerate of the game when played by a human agent and the framerate obtained using the ThinkEngine asset. The performance is expected to be higher when a human agent controls the game, since the thinking phase is absent and substituted by a quick keyboard reading; on the other hand if the game is controlled by ThinkEngine, some impact on performance is expected. Videogames are generally designed in order to keep a constant acceptable framerate, which, in the case of Unity games, is set by default to a target of 60 frames per second. Figure 4.5 shows how, for our setting, the framerate is not constant, but it can vary on each frame. The two curves represent, respectively, the framerate obtained when a human is playing (the blue one) and when the game is controlled by the ThinkEngine (the red one). The two curves generally keep the target framerate, although they present some occasional negative spikes. However, the ThinkEngine framerate has specific negative spikes that are caused by the overhead introduced by the sensors update phase (red crosses in the figure). These spikes do not have a visible impact on the graphical update as they are sufficiently isolated and the moving average (light green curve in figure 4.5) over 25 frames is almost constant. This analysis can be used as an indication for how often one should update the sensors: the game would stall if this is done too often. The actuators update step, instead, has no appreciable impact on the performance of the game (green diamonds in the figure). Obviously, the sensors update needs more time with respect to actuators since they have to track down an entire matrix of values on the game board.

Another aspect that is interesting to look at, is the time that the ThinkEngine needs to auto-generate the input facts for the ASP solver and how fast is this latter in producing a solution. These two measures

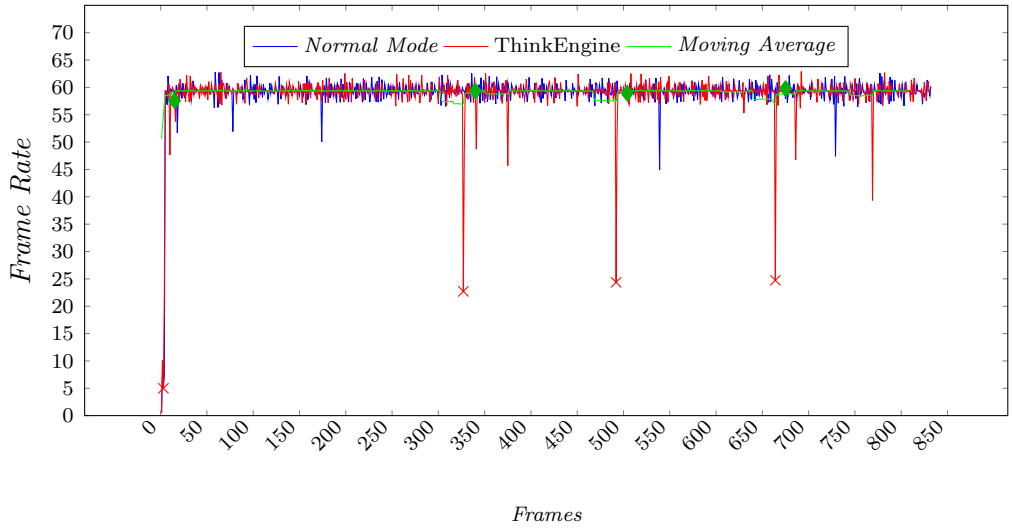


Figure 4.5: Frame rate evaluation on Tetris game.

cannot be tracked in the framerate analysis since the two operations are performed in a separate thread and the main one which is in charge of updating the graphics. However, an intuition of the amount of time elapsed between a sensors update and a solution generation can be spotted in the figure 4.5. Indeed, the number of frames between a sensors update and an actuators update is really low. The table 4.1 shows the time needed on average on a single Tetris match for both facts and answer set generation: the last row is the overall average.

Currently, the ThinkEngine asset has been used within other two games: PacMan and Frogger. In these two games, the obtained performance results are comparable with the Tetris. However, in the PacMan it is clear that the system should be further improved in order to deal with really highly dynamic games. Both Frogger and PacMan Unity projects can be found at `https://github.com/DeMaCS-UNICAL/UnityGames`.

Although the integration approach used in the ThinkEngine framework is improvable, other approaches are almost infeasible. Indeed, recalling what discussed in section 3.1 there are only few ways of integrating an ASP module in a game engine. The ThinkEngine framework is based on a computational-loose and data-tight coupling. In solutions based on a computational-tight coupling the game logic and the ASP solver should be executed in a unique

Run	Facts (ms)	AnswerSets (ms)
1	537.17	628.60
2	548.86	623.00
3	609.00	785.86
4	310.60	444.20
5	228.00	421.00
6	426.20	607.80
7	342.25	544.75
8	493.60	596.25
9	435.80	522.00
AVG	**436.83**	**574.83**

Table 4.1: Generation time

thread. This kind of integration will lead to a dramatic decrease of performance of a game, in fact the graphical update would be slowed down by the ASP solver. On the other hand, computational-loose and data-loose coupling won't help in any way a game developer. Indeed, since game engines are based on a single-thread philosophy, external threads can not access game objects thus a developer should waste time in finding a way to share information. Moreover, without a data integration strategy, developers should work both on the input providing and output retrieving tasks in fact having less focus on designing the underlying artificial intelligence.

Conclusions

The usage of rule-based formalisms within imperative languages, is attracting ever more interest from researchers. Indeed, for some tasks it can be easier to use declarative formalisms than an imperative one, thanks to the simplicity of writing a solution, finding errors and, once reached a good starting representation of what it is needed, one can increasingly improve his/her knowledge base by means of subsequent refinement.

When applied to real-time systems and in particular to game development applications, a really challenging context due to really fast responses required from a KR system, different gaps arise: slowness in producing a solution (then requiring to restart the reasoner when needed), sharing data and mapping between logical assertions and object-oriented data type and representation.

With this work we proposed an abstract architecture for a framework that aims to cover the highlighted gaps when coming to the integration of a declarative formalism in a game engine. We even presented an actual implementation of the framework, called ThinkEngine, integrating an ASP solver in the Unity 3D game engine.

A tight sharing of data structures between the procedural side and the reasoning side, achieved by means of data reflection, make it possible to get rid of the burden of mapping by hand input and output data and to focus only on the encoding of the problem in ASP. The ThinkEngine runs mostly in external threads in order to avoid a drop of performances of the game (think for instance to the graphical update of the game).

After a deep analysis of the architecture of the asset, with an infrastructure based on sensors used to pass data from the game to the ASP solver and on actuators for the inverse process, we have shown how it becomes easy to write an AI playing by itself to Tetris.

We are still working on improving the our infrastructure by adding different features:

- analyzing the formal and technical issues arising when one aims to stop and restart a reasoning task, if needed;
- introduce sequences of actions (plans), and propose a model in which plans are executed transparently and can be aborted, restarted, or modified on-the-fly;
- revise the Actuator model so to better describe actions on the environment and reduce the coding burden on the procedural side;
- add new data type mappings (collections, arrays and so on) for both Sensors and Actuators.

Acknowledgements

I have to thank all the professors of the Department of Mathematics and Computer Science of the University of Calabria. In more than ten years I have learned so much from so many people that it is impossible to thank everyone. However, I cannot forget what Professor Giovambattista Ianni has done and continues to do for me. I could not have finished this path without him.

I want to thank all the employees of the DLVSystem: everyone has been really supportive every day I spent with them.

Thanks to all my friends, especially Bernardo who has always been by my side.

Finally, thanks to Francesco and Lara. In the worst days you were the motivation that made me not giving up.

Bibliography

[1] Esra Erdem, Volkan Patoglu, and Peter Schüller. A systematic analysis of levels of integration between high-level task planning and low-level feasibility checks. *AI Commun.*, 29(2):319–349, 2016.

[2] Thomas Eiter, Giovambattista Ianni, and Thomas Krennwallner. Answer set programming: A primer. In *Reasoning Web. Semantic Technologies for Information Systems, 5th International Summer School 2009, Brixen-Bressanone, Italy, August 30 - September 4, 2009, Tutorial Lectures*, pages 40–110, 2009.

[3] Anis Zarrad. Game engine solutions. In *Simulation and Gaming*. IntechOpen, 2018.

[4] Jordi Bonastre. Why should i use threads instead of coroutines? `https://support.unity3d.com/hc/en-us/articles/208707516-Why-should-I-use-Threads-instead-of-Coroutines-`.

[5] Patrik Haslum, Nir Lipovetzky, Daniele Magazzeni, and Christian Muise. *An Introduction to the Planning Domain Definition Language*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2019.

[6] Michael Cashmore, Maria Fox, Derek Long, and Daniele Magazzeni. A compilation of the full PDDL+ language into SMT. In *Planning for Hybrid Systems, Papers from the 2016 AAAI Workshop, Phoenix, Arizona, USA, February 13, 2016*, 2016.

[7] William F Clocksin and Christopher S Mellish. *Programming in Prolog: Using the ISO standard*. Springer Science & Business Media, 2012.

[8] Francesco Calimeri, Wolfgang Faber, Martin Gebser, Giovambattista Ianni, Roland Kaminski, Thomas Krennwallner, Nicola Leone, Marco Maratea, Francesco Ricca, and Torsten Schaub. Asp-core-2 input language format. *CoRR*, abs/1911.04326, 2019.

[9] Robert Kowalski. Algorithm = logic + control. *Commun. ACM*, 22(7):424–436, July 1979.

[10] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, USA, August 15-19, 1988 (2 Volumes)*, pages 1070–1080, 1988.

[11] Stefan Boettcher and Allon G Percus. Extremal optimization at the phase transition of the three-coloring problem. *Physical Review E*, 69(6):066703, 2004.

[12] C-A Brunet and R Gonzalez Rubio. Cop: A simple way to integrate imperative programming and declarative programming. In *Proceedings 1995 Canadian Conference on Electrical and Computer Engineering*, volume 2, pages 1034–1037. IEEE, 1995.

[13] Krzysztof R. Apt, Jacob Brunekreef, Vincent Partington, and Andrea Schaerf. Alma-0: An imperative language that supports declarative programming. Technical report, Amsterdam, The Netherlands, The Netherlands, 1997.

[14] Gianfranco Rossi, E. Panegai, and Elisabetta Poleo. Jsetl: a java library for supporting declarative programming in java. *Softw., Pract. Exper.*, 37(2):115–149, 2007.

[15] Matthias Thimm. Tweety: A comprehensive collection of java libraries for logical aspects of artificial intelligence and knowledge representation. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Fourteenth International Conference, KR 2014, Vienna, Austria, July 20-24, 2014*, 2014.

[16] Francesco Ricca. The DLV java wrapper. In *2003 Joint Conference on Declarative Programming, AGP-2003, Reggio Calabria, Italy, September 3-5, 2003*, pages 263–274, 2003.

[17] Onofrio Febbraro, Nicola Leone, Giovanni Grasso, and Francesco Ricca. JASP: A framework for integrating answer set programming with java. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Thirteenth International Conference, KR 2012, Rome, Italy, June 10-14, 2012*, 2012.

[18] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Clingo = ASP + control: Preliminary report. *CoRR*, abs/1405.3694, 2014.

[19] Francesco Calimeri, Davide Fuscà, Stefano Germano, Simona Perri, and Jessica Zangari. A framework for easing the development of applications embedding answer set programming. volume abs/1707.06959, 2017.

[20] Francesco Calimeri, Stefano Germano, Giovambattista Ianni, Francesco Pacenza, Simona Perri, and Jessica Zangari. Integrating rule-based AI tools into mainstream game development. In *Rules and Reasoning - Second International Joint Conference, RuleML+RR 2018, Luxembourg, September 18-21, 2018, Proceedings*, pages 310–317, 2018.

[21] Jakob Rath and Christoph Redl. Integrating answer set programming with object-oriented languages. In *Practical Aspects of Declarative Languages - 19th International Symposium, PADL 2017, Paris, France, January 16-17, 2017, Proceedings*, pages 50–67, 2017.

[22] Thomas Eiter, Michael Fink, Giovambattista Ianni, Thomas Krennwallner, Christoph Redl, and Peter Schüller. A model building framework for answer set programming with external computations. *TPLP*, 16(4):418–464, 2016.

[23] Johannes Oetsch, Jörg Pührer, Martina Seidl, Hans Tompits, and Patrick Zwickl. VIDEAS: A development tool for answer-set programs based on model-driven engineering technology. In *Logic Programming and Nonmonotonic Reasoning - 11th International Conference, LPNMR 2011, Vancouver, Canada, May 16-19, 2011. Proceedings*, pages 382–387, 2011.

[24] Claude E Shannon. Programming a computer for playing chess. In *Computer chess compendium*, pages 2–13. Springer, 1988.

[25] Jonathan Schaeffer and H. Jaap van den Herik. Games, computers, and artificial intelligence. *Artif. Intell.*, 134(1-2):1–7, 2002.

[26] Feng-Hsiung Hsu. *Behind Deep Blue: Building the computer that defeated the world chess champion.* Princeton University Press, 2004.

[27] Michael Mateas. Expressive AI: games and artificial intelligence. In *Digital Games Research Conference 2003, 4-6 November 2003, University of Utrecht, The Netherlands*, 2003.

[28] Georgios N. Yannakakis. Game AI revisited. In *Proceedings of the Computing Frontiers Conference, CF'12, Caligari, Italy - May 15 - 17, 2012*, pages 285–292, 2012.

[29] Mike Treanor, Alexander Zook, Mirjam P. Eladhari, Julian Togelius, Gillian Smith, Michael Cook, Tommy Thompson, Brian Magerko, John Levine, and Adam M. Smith. Ai-based game design patterns. 2015.

[30] Diego Perez Liebana, Spyridon Samothrakis, Julian Togelius, Tom Schaul, and Simon M. Lucas. General video game AI: competition, challenges and opportunities. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*, pages 4335–4337, 2016.

[31] Arthur Juliani, Vincent-Pierre Berges, Esh Vckay, Yuan Gao, Hunter Henry, Marwan Mattar, and Danny Lange. Unity: A general platform for intelligent agents. *CoRR*, abs/1809.02627, 2018.

[32] Jie Hu, Wang gen Wan, and Xiaoqing Yu. A pathfinding algorithm in real-time strategy game based on unity3d. In *2012 International Conference on Audio, Language and Image Processing*, pages 1159–1162. IEEE, 2012.

[33] Olivier Bartheye and Eric Jacopin. A pddl-based planning architecture to support arcade game playing. In *Agents for Games and Simulations, Trends in Techniques, Concepts and Design [AGS 2009, The First International Workshop on Agents for Games and Simulations, May 11, 2009, Budapest, Hungary]*, pages 170–189, 2009.

[34] Olivier Bartheye and Eric Jacopin. A real-time pddl-based planning component for video games. In *Proceedings of the Fifth Artificial Intelligence and Interactive Digital Entertainment Conference, AIIDE 2009, October 14-16, 2009, Stanford, California, USA*, 2009.

[35] Ioannis Vlachopoulos, Stavros Vassos, and Manolis Koubarakis. Flexible behavior for worker units in real-time strategy games using STRIPS planning. In *Artificial Intelligence: Methods and Applications - 8th Hellenic Conference on AI, SETN 2014, Ioannina, Greece, May 15-17, 2014. Proceedings*, pages 555–568, 2014.

[36] Richard Fikes and Nils J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artif. Intell.*, 2(3/4):189–208, 1971.

[37] Michael Thielscher. Answer set programming for single-player games in general game playing. In *Logic Programming, 25th International Conference, ICLP 2009, Pasadena, CA, USA, July 14-17, 2009. Proceedings*, pages 327–341, 2009.

[38] Maximilian Möller, Marius Thomas Schneider, Martin Wegner, and Torsten Schaub. Centurio, a general game player: Parallel, java- and asp-based. *KI*, 25(1):17–24, 2011.

[39] Adam M. Smith and Michael Mateas. Answer set programming for procedural content generation: A design space approach. *IEEE Trans. Comput. Intellig. and AI in Games*, 3(3):187–200, 2011.

[40] Francesco Calimeri, Stefano Germano, Giovambattista Ianni, Francesco Pacenza, Armando Pezzimenti, and Andrea Tucci. Answer set programming for declarative content specification: A scalable partitioning-based approach. In *AI*IA 2018 - Advances in Artificial Intelligence - XVIIth International Conference of the Italian Association for Artificial Intelligence, Trento, Italy, November 20-23, 2018, Proceedings*, pages 225–237, 2018.

[41] Marius Stanescu and Michal Certický. Predicting opponent's production in real-time strategy games with answer set programming. *IEEE Trans. Comput. Intellig. and AI in Games*, 8(1):89–94, 2016.

[42] Peter Schüller and Antonius Weinzierl. Answer set application programming: a case study on tetris. In *Proceedings of the Technical Communications of the 31st International Conference on Logic Programming (ICLP 2015), Cork, Ireland, August 31 - September 4, 2015*, 2015.

[43] Kang G Shin and Parameswaran Ramanathan. Real-time computing: A new discipline of computer science and engineering. *Proceedings of the IEEE*, 82(1):6–24, 1994.

[44] M. R. Garey, David S. Johnson, and Larry J. Stockmeyer. Some simplified np-complete problems. In *Proceedings of the 6th Annual ACM Symposium on Theory of Computing, April 30 - May 2, 1974, Seattle, Washington, USA*, pages 47–63, 1974.

[45] Raphael Volz, Steffen Staab, and Boris Motik. Incrementally maintaining materializations of ontologies stored in logic databases. volume 2, pages 1–34. 2005.

[46] Daniele Dell'Aglio, Emanuele Della Valle, Frank van Harmelen, and Abraham Bernstein. Stream reasoning: A survey and outlook. *Data Sci.*, 1(1-2):59–83, 2017.

[47] Esteban Zimanyi. Streaming databases & pipelinedb.

[48] Andre Bolles, Marco Grawunder, and Jonas Jacobi. Streaming SPARQL - extending SPARQL to process data streams. In *The Semantic Web: Research and Applications, 5th European Semantic Web Conference, ESWC 2008, Tenerife, Canary Islands, Spain, June 1-5, 2008, Proceedings*, pages 448–462, 2008.

[49] Bernardo Cuteri, Carmine Dodaro, Francesco Ricca, and Peter Schüller. Constraints, lazy constraints, or propagators in ASP solving: An empirical analysis. *TPLP*, 17(5-6):780–799, 2017.

[50] Francesco Buccafurri, Nicola Leone, and Pasquale Rullo. Strong and weak constraints in disjunctive datalog. In *Logic Programming and Nonmonotonic Reasoning, 4th International Conference, LPNMR'97, Dagstuhl Castle, Germany, July 28-31, 1997, Proceedings*, pages 2–17, 1997.