Declarative AI design in Unity using Answer Set Programming

Denise Angilica, Giovambattista Ianni, and Francesco Pacenza
DeMaCS - University of Calabria, Italy
Email: *firstname.lastname*@unical.it

***Abstract*—Declarative methods such as Answer Set Programming show potential in cutting down development costs in commercial videogames and real-time applications in general. Many shortcomings, however, prevent their adoption, such as performance and integration gaps. In this work we illustrate our *ThinkEngine*, a framework in which a tight integration of declarative formalisms within the typical game development workflow is made possible in the context of the Unity game engine. *ThinkEngine* allows to wire declarative AI modules to the game logic and to move the computational load of reasoning tasks outside the main game loop using an hybrid deliberative/reactive architecture. In this paper, we illustrate the architecture of the *ThinkEngine* and its role both at design and run-time. Then we show how to program declarative modules in a proof-of-concept game, and report about performance and related work.**

***Index Terms*—Answer Set Programming, Declarative Methods, Game Design, Knowledge Representation and Reasoning, Unity**

I. Introduction

The AI research field shares a past and a present of reciprocal exchange with the game design industry, both whether we are talking of inductive/machine learning-based techniques or knowledge-based, deductive techniques. Although very useful in several respects, the reception of the machine learning revolution in game design had so far been limited by the fact that in this approach one has to deal with black-box AI modules which are not easily "tunable" and configurable at will, and have non-negligible design-time costs (the reader can refer to [1] for a survey of AI techniques used in videogames).

In order to overcome the above limits, one can consider the introduction of declarative knowledge representation techniques. Declarative methods potentially allow to specify parts of the game logic in a few lines of high-level statements: high-level declarative specifications can then be processed at run-time by a specific external engine, whose outputs can be converted back to effects in the game environment. Possible applications range from defining the general game logic, to describing non-player characters, programming tactic and/or strategic credible AI behaviors, to defining path planning desiderata, non-player resource management policies and so on. These methods have some appeal in videogame design, since their integration can potentially result in requiring much lesser design effort. Declarative AI modules promise to be much easier to manage, tune and configure, in that it is fair to define them as "glass-box" methods by construction.

We are specifically interested in the usage of knowledge-based declarative techniques: although the historical example of the F.E.A.R. game [2], which used STRIPS-based planning [3], inspired many other games using forms of declarative planning, such as Halo [4] and Black & White [5], known performance and integration shortcomings limited so far the usage of these techniques in videogames.

We can see knowledge-based declarative techniques as lying in the middle between commercial AI engines, where little or no coding is necessary, yet flexibility is constrained to what the AI design GUI allows (unless some coding effort is added), and ad-hoc coding, where great flexibility comes at the price of consuming much development time. This promising versatility motivated the proposal of game modeling tools based on PDDL [6]–[8], fuzzy logic [9] and event calculus [10] in videogames.

Among declarative approaches, Answer Set Programming (ASP) has been successfully used as a declarative tool to model planning problems, robotics, computational biology as well as some industrial applications [11]. ASP has also been experimentally used in videogames to various extents, such as declaratively generating level maps [12] and artificial architectural buildings [13]; it has been used as an alternative for specifying general game playing [14], to define artificial players [15] for the Angry Birds AI competition [16], and for modelling resource production in real-time strategy games [17]. ASP makes no exception with respect to the shortcomings that restrict the utility of declarative approaches. Two aspects are of concern: performance in real-time contexts and integration, i.e., ease of wiring with other standard parts of the game logic. Concerning the first, much research has been done recently, whose outputs are competitive, incremental ASP engines capable of evaluating fast-paced and repeated decision-making tasks [18]–[20]. These contributions enlarged the range of applications for ASP to high-performance settings like stream reasoning [21]. It is thus appealing to explore the role of ASP as tool for designing AI real-time decision-making modules for videogames.

Although welcome, performance improvements might not be sufficient when one has to implement complex decision making processes. Recall that a videogame is typically executed by running the so called game loop routine [22]. The game loop consists in a single-threaded repeated execution of update operations to the current game scene: within each step of this cycle, user input is processed, AI decision-making operations and physics simulations are made, and the game scene is modified accordingly. The operations performed within the

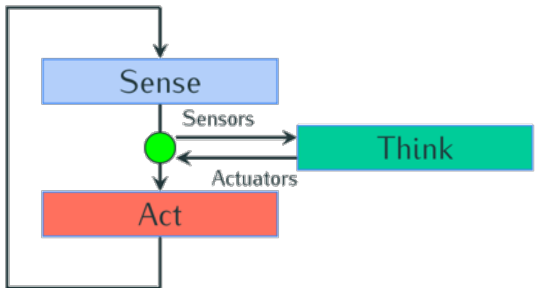


Fig. 1: Hybrid Deliberative/Reactive (HDR) schema.

game loop share many similarities with the reactive sense-think-act cycle typical of agents and robotic systems [23], in which an agent cyclically reads the surrounding environment with sensors, makes some decision making task and then acts on the environment itself.

Reactive AI modules coexist in general very well with the above architecture, in that they are usually implemented using relatively fast techniques, such as behavior trees, finite state automata, rule sets or combinations thereof [1], and can possibly be executed in one iteration of the main game-loop. This type of module comes into play when quick "instinctive" decisions need to be taken by artificial characters: think e.g., at programmed fight responses when an artificial soldier is attacked.

However, when one considers common videogame AI modules such as path-planning, hierarchical task planning and resource management, longer running computing jobs are required. The execution of time consuming jobs within the default game-loop requires to spread processing over several iterations using either coroutines or multi-threading. In order to accomodate both reactive and non reactive AIs, it comes natural to consider the so called hybrid deliberative/reactive paradigm [23] (HDR), in which fast "instinctive" behaviors are combined in a proper way with long term reasoning tasks such as plan generation.

In the HDR scheme (Figure 1) long running reasoning tasks are offloaded, and thus they do not enforce heavy computational loads on the main game loop. A typical AI which is programmed using this scheme can follow a fast sense-think-act loop, in which the think step is fast and reactive (green dot in the figure); one can however introduce offline, non-reactive, reasoning tasks. The outcome of non-reactive tasks can be used to act directly on the game logic, or to reprogram reactive behaviors of game characters.

In this paper we report about the latest version of our system *ThinkEngine*, an asset working for the Unity game engine. Our tool enables the possibility of introducing ASP-based reasoning modules within Unity-made games. Our contributions are the following:

- We describe an architecture were AI decision making-modules, defined in terms of ASP specifications, can be integrated and used within a game engine, in particular the Unity game engine.
- AI modules can be either reactive or deliberative. In this latter case we propose an hybrid-deliberative scheme in which multiple competing plans can be computed outside of the main game loop; plans can be selected according to programmable priorities. Also they can be aborted, replaced or restarted.
- A proper information passing layer to and from the offloaded reasoning jobs is proposed, including also data type mapping facilities and an adaptive computational load control for those parts of our *ThinkEngine* working in the main loop. The language of AI modules is not necessarily restricted to ASP, as we provide data type mapping facilities useful to incorporate other declarative languages (e.g., PDDL).
- We illustrate how *ThinkEngine* can be used for developing deliberative strategies in the context of a Space Invaders-like proof-of-concept game.
- We report about the impact of *ThinkEngine*-based modules on run-time frame rate and on reaction times of artificial players.
- We finally compare in better detail with related work and then draw conclusions.

II. THE *ThinkEngine*: OVERVIEW

The *ThinkEngine* works as a plugin for Unity [24], the known cross-platform game engine primarily used to develop videogames or simulations for more than 20 different platforms like mobile, computers and consoles.

The large share of Unity in the game engine market is justified by its rapid development speed, ease of learning, the availability of a very active community, cross-platform integration, and the wide availability of *assets* such as 2D and 3D models, scripts, shaders and other extensions. The Unity community offers a wide range of re-usable assets, some of them aimed to provide *Artificial Intelligence* capabilities[1]. In particular, at the time of writing, the usage of knowledge-based declarative tools stays substantially unexplored.

The main purpose of the *ThinkEngine* plugin is to offer the possibility of declaratively programming AI modules called *Brains*. Brains can be attached at will to non-player-characters, they can drive the overall game logic, and can be used in general for delivering AI at the tactical or strategic level within the game at hand. Differently from earlier prototypes of the *ThinkEngine*, brains can synthesise and prioritize plans, i.e., sets of actions to be executed in a programmed order.

In the following, we will first briefly introduce Answer Set Programming, then we will overview how the *ThinkEngine* interacts with the Unity game loop at run-time. The section is concluded by describing how *ThinkEngine* interacts with Unity at design-time.

A. Answer Set Programming

Answer Set Programming (ASP) is a prominent representative of declarative approaches to modelling. ASP specifications are composed of set of rules, hard and soft constraints, by

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

means of which it is possible to express qualitative and quantitative statements defining desirable outcomes of a *reasoning task*. A set of input values F (called *facts*), describing the current state of the world, are fed together with an ASP specification S to a *solver*. Solvers in turn produce one or more output $A(S \cup F)$ called *Answer Set*. Answer sets contain the result of the decision-making process at hand in terms of logical assertions which, depending on the applicative domain at hand, encode buttons to be pushed, shifts to be scheduled, protein sequences, and so on.

B. Run-time interaction between Unity and the ThinkEngine.

The Unity game loop is single-threaded, and it is based on a game scene update cycle which is run periodically with a pace depending on the target frame rate of choice [25].

The run-time game world consists of a collection of game objects (GOs in the following), which are subject to iterative updates. Modifications to the game scene depend on user input, on the physics simulation of the game world, and on the game logic enforced by the game designer. A GO consists of a recursive hierarchy of basic properties, such as numeric, string and boolean fields, and complex properties, such as matrices, collections, nested objects, etc.

Game designers can customize the game behavior by implementing specific user callback functions, which are executed within the main thread. For instance, one can provide custom code for the `Update` block of specific GOs, or provide custom coroutines. *Coroutines* constitute a way for implementing asynchronous, single-threaded cooperative multitasking. Additional threads can be introduced, although they don't have direct access to the game scene and are subject to other limitations.

The run-time architecture of the *ThinkEngine* is shown in Figure 2. The *ThinkEngine* makes use of some coroutines working in the Unity main loop, but it also uses additional threads for offloading decision tasks. Intuitively, one or more programmable *Brain* is in charge of decision-making tasks which run in separate *ThinkEngine* threads. *Brains* are connected to the game scene using *sensors*, *actuators* and *plans*. Sensors allow read access to desired parts of the current game state, whereas actuators and plans allow to make changes to the game scene itself. The *ThinkEngine* consists of:
- The *reasoning layer*. This layer is in charge of collecting, processing and executing reasoning jobs. A reasoning job J, in the form of an ASP specification S and a set of encoded sensor values F is elaborated by an answer set solver which produces decisions, encoded in the form of *answer sets*. Two types of decisions can be produced: deliberative ones (i.e., *plans*), which, in the terminology of our *ThinkEngine*, are generic sets of actions to be executed in a programmable order, or *reactive actions* which can have immediate impact on the game scene. These are respectively dealt with by the *Planning Executors* and the *Reactive Executors* which in turn submit reasoning jobs to the ASP solver. The ASP solver is reached using the EMBASP library, a multi solver engine capable of bridging ASP solvers and PDDL solvers [26]. EMBASP is also available in a specialized version for Unity [27]. This layer runs exclusively in auxiliary thread(s) so that time-consuming operations do not affect the game performance.
- The *information passing layer*. This layer buffers data passing between the reasoning layer and the actual game state. Sensors correspond to parts of the game data structures which are visible from the reasoning layer. These are buffered in the *sensor data* store. On the other hand, *actuators* and *plans* data stores collect decisions taken by the reasoning layer and are used to modify the game state in the Unity run-time.
- The *reflection layer*, in which some *Plan Schedulers*, a *Sensors Manager* and an *Actuators Manager* keep the mapping between the game world data structures and the lower layers. The *Sensors Manager* cyclically reads selected game world data which, this way, are made accessible to the reasoning layer. The duration of the sensors update cycle is adaptive. An apposite coroutine spreads the update phase on a number of frames, adapting the time available in each frame depending on the current frame rate. The *Plan Schedulers* select and run generated plans (which in turn can act on the game scene), whereas the *Actuators Manager* updates immediately selected parts of the game world depending on the decisions of reactive brains. Both the *Plan Schedulers* and the *Actuators Manager* behave according to input coming from the reasoning layer. All the modules in the reflection layer are coroutines in the main game loop.
- One or more *Brains* can control the three layers. Each brain can access his own view of the world (i.e., a selected collection of sensors), and can be used for programming a separate AI decision-making activity. We have two types of brains: *Planning Brains*, which are meant for deliberative long

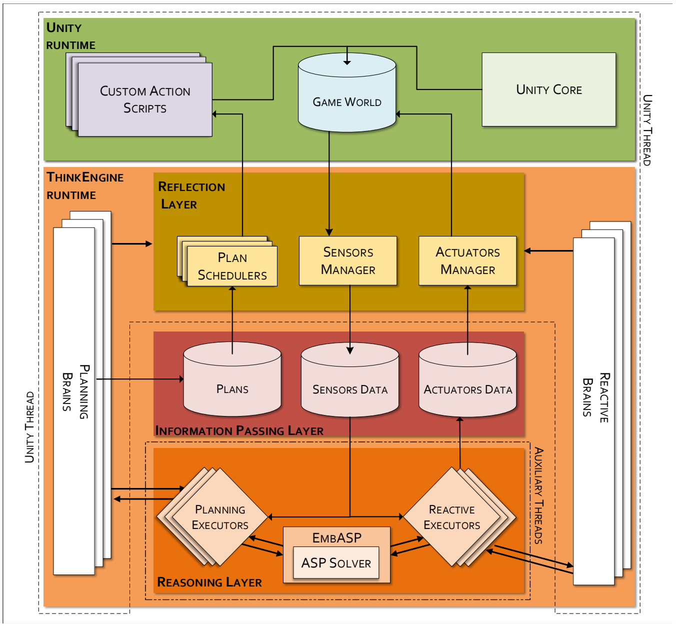


Fig. 2: General run-time architecture of the *ThinkEngine* framework

running reasoning tasks and can generate list of consecutive actions (i.e., plans), and *Reactive Brains*, whose decisions are applied immediately on the game scene. Brains submit their reasoning jobs to the reasoning layer; then, the output of a reasoning job travels back to the reflection layer and modifies the game world in the Unity runtime. A brain is substantially a high-level declarative specification written in the ASP-Core-2 language [28], whose content drives the decision process.

Reasoning tasks, either deliberative or reactive, can be activated in two different ways: on a completed sensors update or on a configurable trigger condition. When an *Executor* is started, it will generate a representation F of sensor readings expressed in terms of logical assertions (set of input facts) and it will invoke the ASP solver. The ASP solver is fed in input with F and with an ASP specification S encoding the AI of the current brain. As soon as the solver provides *decisions* encoded in the form of an answer set, depending on the type of the *Executor*, two things can happen. A *Reactive Executor* populates the actuators associated with the corresponding brain; in this case, the *Actuators Manager* monitors actuators values and updates accordingly the properties of the GO associated with each actuator. The process activated by a *Planning Executor* involves a *Brains Coordinator* and a *Scheduler* and it is described in detail in Subsection II-D.

C. ThinkEngine and Unity at design-time.

At design-time, Unity allows to work on GOs using the above described property-based philosophy, whereas the game logic can be defined by attaching scripted code to specific game events.

In this context, one can use our *ThinkEngine* at design-time by adding, wiring and programming *Brains*. Configuring a brain consists in 1) selecting which sensors are visible as inputs to a given brain; 2) deciding which event triggers a brain reasoning task; 3) crafting a declarative specification S modelling the brain's desired decision-making strategy; 4) for *Reactive Brains*, selecting the actuators and wiring them to the brain at hand; 5) for *Planning Brains*, selecting and programming set of actions to be performed during the execution of plans. The above bindings will be then used by brains at run-time. Brains can be attached to template objects (called *prefabs* in the Unity terminology). Instances of prefabs can have of course their Brain customized at will.

As brains rely on ASP solvers, inputs and outputs need to be bidirectionally converted to the ASP-Core2 format [28]. We thus defined a proper mapping discipline between the game world data and ASP logic assertions. Our mapping rules support basic data types, such as numbers, strings and boolean values. When a game data property p is selected as a sensor value, one can choose a particular *aggregation* function which summarizes consecutive temporal readings of the value of p. For instance, when p is of numeric type, one can choose between *maximum, minimum, average, oldest or newest value*.

In the current *ThinkEngine* version, we support the conversion of arbitrary depth property trees, possibly including lists, mono and bi-dimensional arrays, which are important especially in games based on grids. The statement $playerSensor(plr, objIndex(60), plrs(prevDir(left)))$ can represent, e.g., the value $left$ of the $prevDir$ property that can be found in the component $plrs$ of the GO named plr, uniquely identified by the *ThinkEngine* using the $objIndex$ 60. Note that it is possible to add support for PDDL [29] by just implementing appropriate format mappings, as our solver bridge library EMBASP is already PDDL-aware.

D. Planning Brains

Besides reactive reasoning tasks, the *ThinkEngine* makes possible to declaratively program more complex, deliberative-like, reasoning tasks able to synthetise plans. A plan P is a list of actions $[a_1 \dots, a_n]$ which are supposed to be executed in the given sequence. In turn each action $a_i (1 \leq i \leq n)$ is equipped with a precondition function $PC_i()$. The outcome of $PC_i()$ can be one of {*ready, skip, wait, abort*}, determining, respectively, whether a_i is ready to be executed or it must be skipped, waited on or aborted, in this latter case causing to abort the whole plan. At a given iteration in the game loop, P is *executable* if there is a minimum j for which $PC_j()$ is either *ready* or *wait*, and there is no $k < j$ for which $PC_k() =$ *abort*. The desired outcome of an action a_i on the game scene is obtained implementing the $Do_i()$ function, whereas the function $Done_i()$ is used to define when a_i is completed. One or more *planning brains*, each of which capable of generating a plan, can be associated to a given GO. Plans are associated to a priority value and, once generated by planning brains, they are submitted for execution to a scheduler. A designer can decide how and when a planning brain re-runs its reasoning task. Due to game state changes, re-runs can produce new plans which replace older plan versions.

Planning brains are grouped by GO. Each group has its own *planning scheduler*. A planning scheduler PS has a current plan R and manages a queue of available plans $Q = [P_1, \dots, P_m]$, where subscripts denote priority (lower subscript values denote higher priority). PS can be in the *running* state with a selected plan R, or in *idle* state ($R = null$). R is set according the SCHEDULE coroutine (Algorithm 1). Plans are repeatedly removed from Q (line 5) and either executed or discarded (line 6) according to the following execution policy:
- if PS is *idle* and there exists a minimum j, with $1 \leq j \leq m$ such that P_j is *executable*, then all plans P_z with $z \leq j$ are removed from Q and PS goes in the *running* state. R is set to P_j and it is started.
- If PS is running and $R = P_k$ for some k $(1 \leq k \leq m)$, if there exists a minimum j for which P_j is executable with $j \leq k$, we remove all plans P_z with $z \leq j$ and additionally P_k is aborted and removed from Q. R is set to P_j and is started.

The running plan R is executed according to the coroutine EXECUTEPLAN (Algorithm 2). Let us assume that $R = P_j = [a_1, \dots, a_n]$. EXECUTEPLAN evaluates each precondition $PC_i()$ for $1 \leq i \leq n$. Four possible scenarios may happen: $i)$ if $PC_i() = wait$ the scheduler will pause the current plan until $PC_i()$ changes its outcome; $ii)$ if $PC_i() = ready$ then the function $Do_i()$ is executed and no further steps are performed until $Done_i() = True$; $iii)$ if $PC_i() = abort$, R is set to $null$, thus putting PS in the *idle* state; $iv)$ (implicit) if $PC_i() = skip$ then i is just incremented by one and the next action is taken in consideration. Note that, if a new executable version P'_j of P_j is available, then P_j is aborted and replaced by P'_j. Recall that the **yield** statement suspends the execution of a coroutine until its next call in the game loop. All the instances of the SCHEDULE and the EXECUTEPLAN coroutines run in the main game loop.

```
coroutine SCHEDULE
    while True do
        while $Q$ is not empty and
                ($R$ is null or $Q.first().priority \leq R.priority$) do
            $P = Q.take()$
            if P is executable then
                if R is not null then
                    stopCoroutine(executePlan)
                end if
                $R = P$
                startCoroutine(executePlan)
                break
            end if
        end while
        yield
    end while
end coroutine
```

Algorithm 1: The SCHEDULE coroutine

```
coroutine EXECUTEPLAN
    $i = 1$
    while $i < |R|$ do
        while $PC_i() = wait$ do
            yield
        end while
        if $PC_i()$ = ready then
            $Do_i()$
            while not $Done_i()$ do
                yield
            end while
        else if $PC_i()$ = abort then
            break
        end if
        $i = i + 1$
    end while
    $R = null$
end coroutine
```

Algorithm 2: The EXECUTEPLAN coroutine

III. Proof-of-concept game

In order to give an idea of how AI declarative modules can be integrated within applications developed in Unity we herein report about an extended version of the classic game *Space Invaders*, in which we added a sample automated player whose artificial intelligence is managed via some planning brains. More sample games, including a Tower Defence, Pacman, Frogger, and a Traffic Simulator are available on Github[2].

In the historical *Space Invaders* game, a human player moves a laser cannon horizontally across the bottom of the screen and fires at aliens overhead. The aliens begin in an arrangement of five rows of eleven opponents moving left and right as a group, and shifting downward each time they reach a screen edge. The goal is to eliminate all of the aliens by shooting at them. Although the player has three lives, the game ends immediately if the invaders reach the bottom of the screen. The aliens attempt to destroy the player's cannon by firing projectiles. The laser cannon is partially protected by static defense bunkers which are gradually destroyed from the top by the aliens or, if the player fires when beneath one, from the bottom.

We were interested in real-time gameplay where quickly generating relatively short plans might make sense: in this respect, *Space Invaders* is an ideal simple, yet dynamic, controlled environment for experimenting with multiple competing strategies. Indeed, on the one hand, the player can be programmed with a long term strategy aimed at firing aliens per column, thus reducing the number of bounces on screen borders. This strategy can be overridden by higher priority ones, e.g., when an alien is too close to the screen's bottom, or a projectile is close and in sight of the player's cannon, an emergency plan could take priority. We briefly describe next how our framework has been set up, configured and integrated in the Unity game scene.

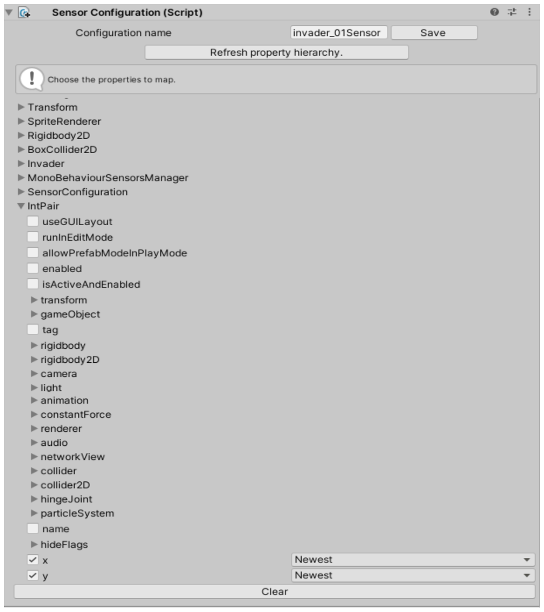
When the developer wants to attach a sensor to a given GO g, it is enough to add a *SensorConfigurator* component instance. As shown in Figure 3, when configuring a new sensor, it is possible to browse g's objects and select which properties are mapped on the reasoning side. In our case we selected the x and y properties. In a similar way, one can configure actuators and brains with specific behaviour for each component.

A. Declarative ASP specification

In this section we briefly describe the strategy we designed by means of ASP. For space reasons we omit the full programs and we focus on those parts having a major impact on the game strategy. The main idea is to generate multiple competing plans that the player will apply; the actions that can be combined in a plan are *MoveAction* and *FireAction* used to let the player move and fire in the game scene, respectively.

Our game strategy combines three *Planning Brains* of different priorities: The first one is called the *Emergency Planner*; this brain is at the highest priority and it is triggered when a missile is in the same horizontal range of the player. The second and third brain, called the *Strategic Planner*, and *Offensive Planner*, are triggered respectively when the number of invaders on screen is more or less than an half of the

[2] https://github.com/DeMaCS-UNICAL/ThinkEngine-Showcase


Sensor Configuration (Script)

Configuration name: invader_01Sensor

Choose the properties to map.

☑ x: Newest

☑ y: Newest

Fig. 3: Sensor configuration at design-time

initial total amount. All the planners define a sequence of ten consecutive actions and use the expected consequences on the environment to make decisions (e.g., the future position of the player, of the invaders and, in general of all the GOs in the scene). The *Emergency Planner* is defined by choosing the "escape" direction via the following ASP rules:

```
1. direction(left) :- player(X,_,_), missile(X_Left,X_Right
    ,_,_,_), L=X-X_Left, R=X_Right-X, L<R.
2. direction(right) :- not direction(left).
```

Intuitively, rule 1 selects *left* when the missile object is on the player right side; similarly, rule 2 chooses *right* if the missile is coming from the left. The operator $:-$ stands for logical implication, i.e., $direction(X)$ is true whenever all the premises on the right of the operator are true. This plan generates ten consecutive moves in the direction of choice, but it is aborted if the missile disappears from the scene.

The *Strategic Planner* and the *Offensive Planner*, depending on the distance between the player and the invaders, choose to kill enemies by columns or by rows. On the one hand, when the invaders are far away, the player fires by columns: this allows to slow down the speed at which enemies are moving downward, since, with lesser columns, invaders need more steps to reach the edges of the game board. On the other hand, when the enemies are closer and the risk that the player is hurt is higher, a row by row modality is applied and the nearest enemies are killed first by row.

The *Strategic Planner* and the *Offensive Planner* differ on the firing policy; in the *Strategic planner*, the player avoids to fire when it is under a bunker and it prefers to fire only if it is aligned with its current target (i.e., a column or a row of invaders); with the *Offensive Planner*, the presence of bunkers is ignored and while the player moves towards the targets, it fires also other enemies having high probability to be hit.

The *Strategic Planner* is modeled with ASP statements like:

```
% Choose a possible action to be applied at a time T
3. 1<={applyAction(T,A) : action(A)}<=1 :- step(T).
4. 1<={actionArgument(T,move,A) : move(A)}<=1 :- step(T),
    applyAction(T,"MoveAction").

% Do not choose to fire at time T_Next if there is already
    an active fire at previous time
5. :- applyAction(T_Next,"FireAction"), laser(_,_,_,T),
    T_Next=T+1, T_Next<=T_Max, maxTime(T_Max).

% Defines the distance between the player and the most left
     column of invaders
6. distance_left_column(X,T) :- not invaders_near_player(T)
    , player(X1,_,T), most_left_invader(X2,T), X = X1-X2,
    X1 >= X2.
7. distance_left_column(X,T) :- not invaders_near_player(T)
    , player(X1,_,T), most_left_invader(X2,T), X = X2-X1,
    X1 < X2.
8. :~ distance_left_column(X,T). [X@4,X,T]

% Defines the X position of the invaders nearest to the
    player w.r.t. Y coord
9. distance_player_invader(X,T) :- invaders_near_player(T),
     player(X1,_,T), nearest_y_invader(X2,_,T), X = X1-X2,
    X1 >= X2.
10. distance_player_invader(X,T) :- invaders_near_player(T)
    , player(X1,_,T), nearest_y_invader(X2,_,T), X = X2-X1,
     X1 < X2.
11. :~ distance_player_invader(X,T). [X@4,X,T]

% Estimates if the player is under a bunker
12. player_under_bunker(T) :- player(X,Y,T), bunker(X_Left,
    X_Right), X >= X_Left, X <= X_Right.
13. :~ applyAction(T_Next,"FireAction"), player(X,_,T), not
     invaders_near_player(T_Next), bunker(X_Left,X_Right),
    X >= X_Left, X <= X_Right, T_Next = T+1. [1@5,X,T,
    X_Left,X_Right,T_Next]
14. :~ applyAction(T,"FireAction"), player(X,_,T), not
    invaders_near_player(T), bunker(X_Left,X_Right), X >=
    X_Left, X <= X_Right. [1@5,X,T,X_Left,X_Right]
```

Rule 3 is a *choice rule* stating that at each step T the solver has to choose exactly one action to perform (among the set of possible $action(A)$ values). Then, rule 4 selects the direction that can be $left$ or $right$ when the chosen action is a $MoveAction$. This information is used to estimate the player's future position in the game scene; similarly, also the enemies' position for each time T are computed and used in order to take decisions. Moreover, the ASP encoding takes in consideration the constraints of the classical *Space Invaders* game; for example, the *hard constraint* 5 is used to avoid generating plans containing two consecutive $FireAction$ since multiple fires are prohibited by the game itself. Rules 6 and 7 define the logical assertion $distance\_left\_column(X,T)$, which holds if at a future time step T, the distance between the current position of the player and the leftmost column in which there is an invader will be X, and there will be no invaders near the player, i.e., it holds $invaders\_near\_player(T)$. Rule 8 is a *soft constraint*, which adds a cost X to possible *plans* in which the statement $distance\_player\_invader(X,T)$ holds. Note that the ASP solver looks for minimum cost answer sets encoding plans. Similarly, when invaders are very close to the player w.r.t. its Y coordinate, rules 9 and 10 define the distance X from the nearest invader at future step T. The soft constraint 11 adds a cost X to this distance, thus suggesting the ASP solver to find plans in which this distance is minimized. In other words, the group of rules $6-8$ and the group $9-11$ respectively instruct the player to select and minimize different distances depending on whether it is

#Sensors	Avg. Frames
100	1.5
2000	7.70
3000	9.30
10000	37.34
20000	69.75

TABLE I: Average number of frames for a sensors update cycle.

#Actions	Avg. Time (ms)
10	68.38
50	74.35
100	81.5
250	153.67
500	574.75
1000	3710

TABLE II: Average time generation for one plan with different sizes.

preferable to target invaders on edges or the lowermost ones. Rule 12 is used to estimate in which step T the player will be under a bunker; then, rules 13 and 14 use this information in order to add an higher priority cost to actions which fire and destroy some bunker. Soft constraints indeed allow to express preferences among answer sets in term of costs, but also in terms of priority levels. The higher is the priority level, the more important is the cost associated to the constraint. In our case, rules 13 and 14 has priority 5 whereas in rules 8 and 11 priority is 4; hence, solutions where the player does not fire to a bunker are preferable, no matter of the costs at priority 4. The *Offensive Planner* strategy similar to the *Strategic Planner*, but with differently tuned soft constraints.

IV. Performance

In order to assess the practical usability of *ThinkEngine*, we were interested in measuring: $i)$ the frame rate, the most common metric in the videogame field, representing the number of screen updates per second that can be achieved given the computational burden of the game implementation at hand; $ii)$ the required number of frames to achieve a complete sensors update cycle; this depends on the number of sensors in the scene; $iii)$ the required time to compute a new plan; this affects the delay between the event triggering the generation of a plan and the execution of the first action.

Tests were performed on a desktop machine equipped with an Intel CPU i7-4700MQ with 16GB of RAM. Using the *Space Invaders* example, we compared the frame rate of the game when played by a human agent and the frame rate obtained using the *ThinkEngine* asset. As we were interested in assessing the viability of the *ThinkEngine* approach we were not interested in comparing the score of the AI with respect to human players. Concerning frame rate, we measured 58FPS, in contrast with 60FPS for human playing. This was expected since the CPU-consuming tasks are executed either in external threads (*Executor*) or in coroutines (like the sensors update cycle) thus, the frame rate of the game is almost the same in both configurations.

Additionally, we measured the time required to compute a new plan, measured from the moment a reasoning job is triggered. The three Planner reasoning jobs took on the average $83.5ms$ (Emergency planner), $189.25ms$ (Strategic planner), and $203ms$ (Offensive planner). As expected the *Emergency Planner* was faster than the others as its set of statements was purposely kept as simple as possible in order to achieve a better reaction time. We observed some oscillation in times depending on the number of invaders still on the screen. It is worth noting that to fill the time gap between two different plan execution, we configured both the *Strategic* and the *Offensive Planners* to early trigger new plan generations when it is being applied the third-to-last action of the current plan.

Since the sensor update cycle can add some delay on triggering reasoning jobs we tested its impact by artificially generating additional sensors. The performance of this cycle can have also some impact on the coherence of sensors data as they can be updated many frames apart. As we can see from Table I, the higher the number of sensors in the scene the more frames the update cycle coroutine is spread on.

As a last performance measure we were interested in assessing the duration of reasoning jobs with varying plan lengths. We experimented in a scenario with a single Planning Brain producing plans of variable length. The generation time increased as reported in Table II. As expected, the frame rate was not affected by the plan length. Although there is room for improvement, performance is fairly satisfactory if one considers real settings where game characters are expected to follow plans of no more than a few dozen of actions.

V. Comparison with related work

Our proposal compares on the one hand with similar research aimed to show the general potential of declarative methods in videogames; and, on the other hand, there are specific points of contact with planning techniques adapted to the videogame realm, commonly named Goal-Oriented Action Planning. Concerning the introduction of declarative methods in the broadest sense, the best known example is the General Game Playing Description Language (GDL) [30], based on logic programming, whose original purpose is providing a descriptive standard for games. In turn, the Video Game Description Language [31] shares with GDL the descriptive purposes, although VGDL is not strictly logic-based nor fully declarative, but aims at describing playable real-time videogames. We also recall the experience of the Ludocore engine [10], where event calculus was the basis of a game engine in which almost any feature could be described using logic statements. The availability of logic-based definitions allowed to state clear descriptions of game features, and also enabled the possibility of analysing and querying consistent logical traces of gameplays. The usage of declarative planning techniques in videogames stems from the usage of STRIPS [3] in the F.E.A.R. videogame [2], which inspired a generation of further work. The usage of the modern Planning Domain Definition Language (PDDL) [29] in the context of videogames has been proposed in [6], where the possibility of achieving real-time planning of player's actions was first shown in the context of the Iceblox and Video Battle Arena 2 games. The focus of the General Mediation Engine of [8] is instead the usage of PDDL for describing gameplay narratives which can be (re)-generated on-the-fly, while PDDL descriptions are part of a procedural content generation pipeline in [32]. It must be noted that ASP can be seen as a general purpose declarative language which is not strictly tailored to just the goal-oriented planning paradigm. For instance, the Angry Birds artificial player of [15] makes usage of ASP for expressing quantitative physics statements which drive the artificial player choices. In [14], ASP has been proposed as an alternative for specifying general game playing, leveraging declarative features of ASP such as the possibility of expressing temporal statements, defining arbitrary search spaces, and expressing soft and hard constraints on the game. ASP is the declarative core for expressing preferences used for procedural content generation in [33], [34] and [12]. Other examples are modelling resource production in real-time strategy games [17], expressing requirements for auto-generated architectural buildings [13] or for the placement thereof [36].

VI. Conclusions

To the best of our knowledge, our contribution is the first attempt at introducing declarative methods in the general setting of commercial game engines. Since its first prototype [35], as a distinctive feature, the *ThinkEngine* proposes an hybrid architecture in which procedural and declarative parts coexists, and where integration and computational offload issues are explicitly addressed. In future work we aim to improve the integration level of *ThinkEngine* by cutting down the need for manually writing glue code and to reduce the general design effort. Using declarative paradigms like ASP can be not so natural for game developers: overcoming this obstacle deserves further research. Concerning performance, we aim to: reduce the cost of the sensor update cycle, add other incremental reasoning job evaluation features, and experiment in more resource-demanding videogames. Our *ThinkEngine* is publicly available[3].

Acknowledgments

We thank the reviewers of this paper, whose constructive comments helped to improve our work. This work has been partially supported by the Italian University and Research Ministry (MIUR) under project "Declarative Reasoning over Streams" (CUP H24I17000080001) – PRIN 2017.

References

[1] J. Pfau *et al.*, "The case for usable AI: what industry professionals make of academic AI in video games", in *CHI PLAY 2020* .

[2] J. Orkin, "Three states and a plan: the AI of F.E.A.R.", in *GDC 2006.*

[3] N. Nilsson, "STRIPS planning systems", *Artificial Intelligence: A New Synthesis*, pp. 373–400, 1998.

[4] "Halo", 2001. [Online]. Available: https://www.xbox.com/en-US/games/halo

[5] "Black & White", 2001. [Online]. Available: https://www.ea.com/games/black-and-white

[6] O. Bartheye and E. Jacopin, "A real-time PDDL-based planning component for video games", in *AIIDE 2009.*

[7] O. Bartheye and E. Jacopin, "A PDDL-based planning architecture to support arcade game playing", in *AGS 2009, LNCS vol. 5920.*

[8] J. Robertson and R. Young, "The general mediation engine", in *AIIDE 2014. Experimental AI in Games Workshop.*

[9] C. R.-Manzano and M. P.-Fariña, "Declarative computational perceptions networks for automatically generating excerpts in computer games by using bousi prolog", in *FUZZ-IEEE 2017.*

[10] A. M. Smith *et al.*, "LUDOCORE: A logical game engine for modeling videogames", in *CIG 2010.*

[11] E. Erdem, M. Gelfond, and N. Leone, "Applications of answer set programming", *AI Magazine*, vol. 37, no. 3, pp. 53–68, 2016.

[12] A. M. Smith and M. Mateas, "Answer set programming for procedural content generation: A design space approach", *IEEE Trans. Comput. Intell. AI Games*, vol. 3, no. 3, pp. 187–200, 2011.

[13] L. van Aanholt and R. Bidarra, "Declarative procedural generation of architecture with semantic architectural profiles", in *CoG 2020.*

[14] M. Thielscher, "Answer set programming for single-player games in general game playing", in *ICLP 2009, LNCS vol. 5920.*

[15] F. Calimeri, M. Fink, S. Germano, A. Humenberger, G. Ianni, C. Redl, D. Stepanova, A. Tucci, and A. Wimmer, "Angry-HEX: An artificial player for angry birds based on declarative knowledge bases", *IEEE Trans. Comput. Intell. AI Games*, vol. 8, no. 2, pp. 128–139, 2016.

[16] J. Renz *et al.*, "The Angry Birds AI competition", *AI Mag. 2015.*

[17] M. Stanescu and M. Certický, "Predicting opponent's production in real-time strategy games with answer set programming", *IEEE TCIAIG 2016.*

[18] M. Gebser *et al.*, "Multi-shot ASP solving with clingo", *TPLP 2019.*

[19] G. Ianni *et al.*, "Incremental maintenance of overgrounded logic programs with tailored simplifications", *TPLP*, vol. 20, no. 5, pp. 719–734, 2020.

[20] H. Beck *et al.*, "Ticker: A system for incremental asp-based stream reasoning", *TPLP*, vol. 17, no. 5-6, pp. 744–763, 2017.

[21] C. Dodaro, T. Eiter, P. Ogris, and K. Schekotihin, "Managing caching strategies for stream reasoning with reinforcement learning", *TPLP*, vol. 20, no. 5, pp. 625–640, 2020.

[22] M. Joselli *et al.*, "An adaptative game loop architecture with automatic distribution of tasks between CPU and GPU", *Comput. Entertain.*, 2009.

[23] R. R. Murphy, *Introduction to AI Robotics*. MIT Press, 2000.

[24] F. Messaoudi, G. Simon, A. Ksentini, "Dissecting games engines: The case of Unity3D", in *NetGames 2015.*

[25] "Unity, Order of execution for event functions". Unity Documentation, available at: https://docs.unity3d.com/Manual/ExecutionOrder.html (Accessed May 2022).

[26] F. Calimeri *et al.*, "A framework for easing the development of applications embedding answer set programming", *PPDP 2017.*

[27] F. Calimeri, S. Germano, G. Ianni, F. Pacenza, S. Perri, and J. Zangari, "Integrating rule-based AI tools into mainstream game development", in *RuleML+RR*, LNCS vol. 11092.

[28] F. Calimeri, W. Faber, M. Gebser, G. Ianni, R. Kaminski, T. Krennwallner, N. Leone, M. Maratea, F. Ricca, and T. Schaub, "ASP-Core-2 input language format", *TPLP*, vol. 20, no. 2, pp. 294–309, 2020.

[29] M. Ghallab, A. Howe, C. Knoblock, D. Mcdermott, A. Ram, M. Veloso, *et al.*, "PDDL—The Planning Domain Definition Language", *AIPS Planning Competition 1998.*

[30] M. R. Genesereth, N. Love, and B. Pell, "General game playing: Overview of the AAAI competition", *AI Mag.*, vol. 26, no. 2, pp. 62–72, 2005.

[31] T. Schaul, "A video game description language for model-based or interactive learning", in *CIG 2013.*

[32] J. Robertson and R. M. Young, "Automated gameplay generation from declarative world representations", in *AIIDE 2015.*

[33] F. Calimeri, G. Ianni *et al.*, "Answer set programming for declarative content specification: A scalable partitioning-based approach", in *AI*IA 2018.*

[34] X. Neufeld, S. Mostaghim, and D. Perez-Liebana, "Procedural level generation with answer set programming for general video game playing", in *CEEC 2015.*

[35] D. Angilica, G. Ianni, F. Pacenza, "Tight Integration of Rule-Based Tools in Game Development", in *AI*IA 2019.*

[36] M. Certicky, "Implementing a Wall-In Building Placement in StarCraft with Declarative Programming". *CoRR abs/1306.4460.*

[3] https://github.com/DeMaCS-UNICAL/ThinkEngine