A Real-Time PDDL-Based Planning Component for Video Games

Olivier Bartheye and Éric Jacopin
MACCLIA
CREC Saint-Cyr
Écoles de Coëtquidan
F-56381 GUER Cedex
{olivier.bartheye,eric.jacopin}@st-cyr.terre-net.defense.gouv.fr

Abstract

In this paper, we argue that PDDL-based real-time planning can be achieved for today's video-games. PDDL is an expressive language which offers to represent many planning knowledge while freeing from the plan search algorithms. We present two case studies where we connected a PDDL-based AI Planner to a video-game. In each case, we report on the engineering decisions which turned out to be crucial to achieve real-time playability.

Introduction

Bugs are a crucial side effect of computer science: when we cannot immediately correct them, we are left with finding news ways to achieve our goals. And often, there are many ways to achieve what we want to do. In a way, bugs force us to be creative: as we do not expect them to arise, we eventually improvise to achieve our intended results.

The purpose of planning is precisely to search for ways, called plans, to achieve goals. Plans are combinations of actions which will hopefully solve our problem. The various combinations of actions to achieve goals entail that goals are not once for all paired with a unique sequence of actions.

Once we are given a plan, we are left with its execution. Unfortunately, a plan is not a cure for bugs which can arise again, in particular during the execution of (what we thought was) a good plan. Monitoring the execution of a plan is not an easy task: should we try to adapt our current plan so as to turn around the new bug or should we call for a complete new plan? When it is possible to do both in real-time, this situation becomes a matter of when to plan.

Unexpected situations or events arise during the game experience making game/level objectives harder to achieve in a way similar to bugs for everyday software: quickness and sometime random enemy moves in yesterday's 2D arcade games produced numerous unexpected situations and today's best 3D video games possess a carefully designed scenario with many surprises and unexpected events. Every video game player has faced this situation where things don't go as initially planned.

Planning needs in video-games follows the previous informal discussion. For instance, the Goal Oriented Action Planning architecture (GOAP) (Orkin 2003) successfully introduced the decoupling of goals and actions (Orkin 2006): instead of pairing a sequence of actions with a set of goals, planning proposes a sequence of actions which can vary according to the game situation assessment. Plans are indeed built during an initial phase prior to their execution but the dynamic environments of video-games can require the execution phase to adapt the plan with respect to the current game situation assessment: one can expect real-time plan construction for video-games to be a two phases process.

The Planning Data Description Language (PDDL) is the mandatory language for planners competing at the International Planning Competition (IPC) (International Planning Competition 1998 2009). Consequently, many Artificial Intelligence (AI) planners with various functionalities are now freely available. A planning problem is made of an initial state, a final state and a set of operators; a state is a set of predicates and operators can be seen as a couple of states: preconditions which must be true to allow execution and effects which are true after execution. The next two sections give illustrations in the context of an arcade game and of a first person shooter (FPS). Planning is expected to build a sequence of operators which shall transform the initial state into the final state. Beyond its wide acceptance within the AI Planning community, PDDL is an expressive language which allows representations of resources, preferences, constraints, etc. It must be clear, however, that the computational complexity of AI Planning is high, even for simple forms of PDDL-based planning problems (Erol, Nau, and Subrahmanian 1995) and this is why, for instance, (Orkin 2006) reduces the expressiveness of the effects of operators and moreover allows procedural information to be included. However, we show in this paper that real-time video game PDDL-based planning can be achieved.

The rest of this paper is organized as follows. The next section describes how we achieved real-time PDDL-based planning for the Iceblox arcade game (Hornell 1996). The third section reports the same for the serious game VBS2 (Bohemia-Interactive-Australia 2006 2009). The fourth section presents a reverse (game) engineering approach to the design of a real-time PDDL-based planning

component. Finally, we conclude with a brief discussion and perspectives.

Iceblox

We first describe Iceblox and then present the engineering decisions we made to achieve real-time. We then detail two planning operators in order to illustrate the planning predicates we designed and eventually discuss plan length. Many planning operators can be designed for a given set of planning predicates; design decisions mainly concern the amount of information pushed to the plan execution module to avoid computations during execution.

Description In the Iceblox video game (Hornell 1996), the player presses the arrow keys to move a penguin horizontally and vertically in rectangular mazes made of ice blocks and rocks, in order to collect coins. But flames randomly patrol the maze at the speed of the penguin which can be killed in a collision; moreover, each coin is iced inside an ice block which must be cracked before the coin is ready for collection. The player can push (space bar) an ice block which will slide until it collides with another ice block, a rock or any four side of the game and kills a flame when passing over it. If the player pushes an ice block which is next to another ice block, a rock or a side of the game, it cannot slide freely and shall begin to crack. Once cracked, an ice block cannot move any more and thus pushing a cracked ice block eventually results, after seven pushes, in its destruction. An ice block which contains a coin slides and cracks as does an ordinary ice block. However, a coin cannot slide; it can only be collected once revealed after the seventh push. The player reaches the next, randomly generated, level when all coins have been collected. The number of flames and rocks increases during several levels and then cycle back to their initial value. Finally, the player gets 200 points for each collected coin, 50 points for each killed flames and begins the game with three spare lives.

Although Karl Hornell's Iceblox java code is open and has been widely available for some time now (e.g. (Bartlett, Simkin, and Stranc 1996)), we wrote a C++ version with Microsoft's DirectX (Microsoft 2007) to ease the integration of the C/C++ AI Planners such as FF (Hoffmann 2001) which are available through the IPC.

Iceblox Real-time planning does not mean an instantaneous plan construction or monitoring. It means determining the time limit beyond which the current Iceblox situation is so dangerous that an action has to be taken right away, thus changing the initial state of the planning problem and consequently asking for re-planning and therefore making the current planning activity useless. Iceblox has a good playability when the frame rate is about 30 frames per second, that is, when the flames coordinates (in pixel) are updated about 30 times per second. Luckily, the sprites are 30 by 30 pixels; it then takes about 1 second for a flame to move from one crossroad to another. According to the Iceblox code, a flame gets a random new direction between 1 and 4 crossroads, while keeping in mind that the new direction at the fourth crossroad might just be the same as the previous one. We can then consider that the limit is when the penguin is 4 crossroads away from a flame, which gives a plan search runtime of at most 4 seconds. When a plan has been found then it is to be executed, which undoubtedly takes time to trigger. Consequently, the flame should not reach the fourth crossroad and since movement between two crossroads is not interruptible, the flame should not reach the third crossroad before the plan search ends, which gives us a time limit of 3 seconds.

We did not investigate to allow more time to planning by having the penguin taking cover during planning to escape danger.

Achieving real-time planning for Iceblox The player intentionally starts the planning activity by pressing a designated key (e.g. the "p" key). Of course, planning does not stop the Iceblox game: flames shall patrol the maze and sliding ice blocks shall continue to slide while the plan formation activity is running. The player is warned when the activity ends, either with success or else failure (e.g. a different sound for each case). At any time before the end of the planning activity, the pressing of an arrow key or else the space bar terminates the activity. Instead of having several basic move actions in a plan, we collapsed them into a unique Move operator recording only the coordinates of the departure and of the destination: this drastically reduces the plan length and consequently the plan search runtime. AI Planning has been using such an abstract Move operator since the inception of Strips operators with the GoThru operator (Fikes, Hart, and Nilsson 1972, page 256) and still does, e.g. the Move operator of the storage domain (International Planning Competition 1998 2009). Of course, this delegates the details of the moves in Iceblox to the plan execution module.

If the details of path planning are left to the plan execution module then actions like avoiding or fleeing can be delegated as well. Fighting flames requires both basic moves and pushing ice blocks, so it is already handled. Finally, following the design of the abstract Move operator, we collapsed the 7 basic push actions in an abstract coin Extraction operator.

The plan execution module receives a plan from the AI Planner and perform corresponding actions in the Iceblox game. Execution of a plan means executing the players actions corresponding to the operators of the plan, in an order compliant with their order of appearance in the plan. As discussed above, the plan execution module decides of several basic moves (including the pushing of ice blocks) to perform a Move operator in the plan. The plan execution module also decides to take advantage of the current Iceblox situation to avoid pushing an ice block when a flame is no longer aligned with it, to choose a new weapon or to avoid unexpected flames. These decisions result from a local analysis and no global situation assessment is realized to take such advantages; that is, these decisions do not result from planning. The plan execution module eventually warn the user when it terminates (whatever the outcome, success

or failure), thus giving the game control back to the player. However, the player can terminate the plan execution at any time by pressing any arrow key or else the space bar.

Planning problems predicates used to describe Iceblox PDDL planning problems are the following (crossroad$_{i,j}$ denotes the location at the intersection of line i and column j):

- (*at* i j): a sprite (penguin, ice block, iced coined, weapon) is at the crossroad$_{i,j}$.
- (*extracted* i j): the coin at the crossroad$_{i,j}$ has been collected by the player.
- (*guard* i_1 j_1 i_2 j_2): the flame at the crossroad$_{i_1,j_1}$ guards the coin at the crossroad$_{i_2,j_2}$. The idea of guarding a coin is linked to the 3 seconds time constraint (see the planners requirements): a flame makes dangerous the collection of a coin when it is within a range of 3 crossroads.
- (*iced-coin* i j): there is an ice block at the crossroad$_{i,j}$ which contains a coin.
- (*protected-cell* i j): there exists a path towards the crossroad$_{i,j}$. This path is safe: no flame makes this path dangerous
- (*reachable-cell* i j): there exists a path towards the crossroad$_{i,j}$; there exists at least one flame putting this path in danger.
- (*weapon* i_1 j_1 i_2 j_2 i_3 j_3): there exists a weapon at the crossroad$_{i_2,j_2}$; the penguin should push this weapon from the crossroad$_{i_1,j_1}$. The weapon shall stop sliding at the crossroad$_{i_3,j_3}$; this is useful information when an ice block needs more than one push before the final kick.

Operators use two more predicates. A path towards a crossroad always exists in a maze of crushable ice blocks and the only real matter is whether it is dangerous; bus as crushing an ice block takes time and as an ice block can be used as a weapon, it is worth noting such facts:

- (*blocked-path* i j): there is an ice block on the path to crossroad$_{i,j}$.
- (*blocked-by-weapon* i_1 j_1 i_2 j_2): the weapon at crossroad$_{i_2,j_2}$ is on the path to crossroad$_{i_1,j_1}$.

Over all the operators we designed, 4 proved to be critical for Iceblox game playing: move-to-crossroad, destroy-weapon, kick-to-kill-guard and extract. We hope their names are self explanatory. Basic Iceblox playing is impossible if your design does not capture the meaning of those 4 operators: moving to iced coins, extracting coins from ice blocks and fighting flames from time to time is the essence of the Iceblox gameplay. Due to space limitations, we only give the PDDL code of the last two:

```
(:action extract
   :parameters (?coinx - coord-i ?coiny - coord-j)
   :precondition (and (protected-cell ?coinx ?coiny)
         (iced-coin ?coinx ?coiny)
         (at ?coinx ?coiny)
         (reachable-cell ?coinx ?coiny))
   :effect (and (extracted ?coinx ?coiny)
         (not (iced-coin ?coinx ?coiny))
         (not (protected-cell ?coinx ?coiny))
         (not (reachable-cell ?coinx ?coiny)))))

(:action kick-to-kill-guard
   :parameters (?reachablewx - coord-i ?reachablewy - coord-j
         ?weaponx - coord-i ?weapony - coord-j
         ?newweaponx - coord-i ?newweapony - coord-j
         ?guardx - coord-i ?guardy - coord-j
         ?coinx - coord-i ?coiny - coord-j
         ?blockedx - coord-i ?blockedy - coord-j)
   :precondition (and (iced-coin ?coinx ?coiny)
      (at ?reachablewx ?reachablewy)
      (guard ?guardx ?guardy ?coinx ?coiny)
      (weapon ?reachablewx ?reachablewy ?weaponx ?weapony
            ?newweaponx ?newweapony)
      (reachable-cell ?weaponx ?weapony)
      (protected-cell ?weaponx ?weapony))
   :effect (and (at ?weaponx ?weapony)
         (protected-cell ?coinx ?coiny)
         (blocked-by-weapon ?blockedx ?blockedy
         ?newweaponx ?newweapony)
         (reachable-cell ?newweaponx ?newweapony)
         (protected-cell ?newweaponx ?newweapony)
         (not (reachable-cell ?weaponx ?weapony))
         (not (protected-cell ?weaponx ?weapony))
         (not (reachable-cell ?blockedx ?blockedy))
         (not (guard ?guardx ?guardy ?coinx ?coiny))
         (not (weapon ?reachablewx ?reachablewy ?weaponx ?weapony
               ?newweaponx ?newweapony))))
```

These operators can be much simpler and we indeed designed and used several versions of various complexity.

We eventually plugged the following PDDL-based planners in Iceblox: FF (Hoffmann 2001) and Qweak (Bartheye and Jacopin 2005). FF achieves our real-time objective in most of the Iceblox game situations; it is written in C and its code is simple to understand. Qweak is a real-time planner for Iceblox, but it is written in Prolog and its code is not simple. FF heuristically applies operators from the initial state, hoping to reach the final state at some point of its search. Qweak associates time intervals to both states and operators predicates and uses operators to generate arithmetic constraints over the time intervals; a plan is found when the lower and upper bounds of the time intervals satisfy the arithmetic constraints. The reader is referred to the references for further details. Other planners available from the IPC web site either failed to understand our PDDL problems or else failed to satisfy our need for real-time better than FF and Qweak (Bartheye and Jacopin 2008).

A plan is generated for one coin at a time. Basic Iceblox playing means that 2 actions are needed to kill a flame (move to a weapon and push it) and 2 actions are needed to collect a coin (move to an iced coin and extract it). Consequently, the length of the generated plans is at least 2 plus 2 times the number of dangerous flames; note that on average, the generation of random levels produces 1 or 2 dangerous flames per coin. There are less basic situations when a primary push is needed to align a weapon with a dangerous flame; and of course, ice blocks preventing from reaching a coin must be destroyed, thus linearly (one action for each destruction plus one move to the next ice block) increasing the plan length.

Planning for VBS2

Let's face it, our success with Iceblox does not entail an immediate reuse of our work with other games. But which other games should we try? The addition of planning functionalities to a game is subject to accessing the game loop, which is generally impossible in commercial games. Fortunately, commercial serious games do provide such hooks. We consequently chose Virtual Battle Space 2 (Bohemia-Interactive-Australia 2006 2009) as our next test-bed for AI Planning: it is vastly different from Iceblox, allows external calls and is a popular commercial serious game.

This section is similar to the previous section on Iceblox. We start with a brief description of VBS2, explain what we mean by real-time planning for VBS2 and report on the decisions we took. We end with a tactical planning example: we give the generated PDDL problem, a PDDL operator crucial to solve the problem and VBS2 scripts generated on the fly for this problem.

Description VBS2 is a serious game aiming at training and after action review in a military setting; a toolkit allows professional users to edit terrains, characters, weapons, vehicles, buildings,..., and a scarcely documented scripting language provides commands for scenario editing. Many real world military assets are available and concrete cases can be implemented to train troops in realistic scenarios. As a game, VBS2 looks like a first person shooter: you control a character in a subjective or a third person view and assign orders to other characters of your troop; then, shooting and destroying can be part of your mission objectives.

A VBS2 session begins with the selection of a map and continues by loading a mission. A scenario is then activated by the preview command.

VBS2 Real-Time Planning is less constrained than in the case of Iceblox. This because taking cover has the military use, among others, of being safe while waiting to synchronize your soldiers before the execution of a tactical operation. And there is, as in real world combat, a huge need for taking cover: VBS2 enemies shall otherwise efficiently open fire at you with no other objective than killing you. Consequently, we decided to allow planning only when troops are safe. Of course, being safe does not mean that you can spend all day planning the next tactical actions of your troops: following our work on Iceblox, we allowed at most 4 seconds to plan search.

Achieving real-time planning for VBS2 The player intentionally starts the planning activity by pressing a designated key (e.g. the "p" key). Obviously, planning does not stop the VBS2 game. The player is warned when the activity ends, either with success or else failure (e.g. a different hint on screen for each case). The player can stop the activity at any time planning by pressing a designated key. Goals are generated according to the current game situation assessment: enemies neutralized, building entered, civilian and military units rescued, etc. The example below illustrates the rescue of a hostage while neutralizing a guard.

We studied two means to activate planning within a scenario. At each frame produced by the graphic engine, VBS2 calls the function `OnSimulationStep` which can then call your own code. VBS2 also provides the function `PluginFunction` which can be called within a script.

We used `OnSimulationStep` for three reasons: (i) the hook in Iceblox was provided at each graphic frame so this is similar to our work on Iceblox, (ii) `OnSimulationStep` does not suffer from the overhead of launching a script and, moreover, (iii) `PluginFunction` seems to only accept strings for both parameters and returned data, which adds overhead to the generation of a PDDL planning problem.

We packed our planners in a Dynamically Loadable Library (DLL). DLLs must be dropped in the "plugins" folder of the VBS2 folder and are loaded when VBS2 is launched thus entailing no activation overhead in the playing of the scenario. Then, everything works as in Iceblox: `OnSimulationStep` checks for the keystroke "p", which first generates a PDDL planning problem, then calls for planning and eventually activates plan execution.

We discovered that VBS2 provides high level scriptable actions such as `move-unit`, `escort-unit` (*unit* is the VBS2 term for every VBS2 human agent) and many others: this is a backward validation of our decision to plan for abstract actions in Iceblox. Therefore, we mapped VBS2 high level actions to PDDL operators. We accordingly decided to let the VBS2 game engine manage path planning and only used waypoints as game parameters for our PDDL operators: paths always existed in the scenarios we tested.

An example of elementary tactical planning We take as an elementary example the case of rescuing a hostage in a fictive landscape. Two blue force soldiers, a machine gunner called the *hero* (which is, at start up, the player and consequently cannot be directed by AI script statements) and a grenadier called the *companion*, must escort a *hostage* to a vehicle, a *desert car*. An *enemy/* is located not too far from the hostage and therefore needs to be neutralized.

The PDDL planning problem generated for this example is the following (most of the predicates should be self-explanatory):

```
(define
    (problem plan-auto)
    (:domain vbs2-strips)
    (:objects companion - unit hero - unit hostage - unit enemy - unit
        desert-car - vehicle)
    (:init
        (near-object hero hero)
        (soldier-unit hero)
        (soldier-unit companion)
        (hostage-unit hostage)
        (enemy-unit enemy)
        (has-grenade companion)
        (guard-unit enemy hostage)
        (unsafe-unit hostage)
        (reachable-unit hostage)
        (reachable-unit hero)
        (safe-unit hostage))
```

```
(:goal (and (liberated-unit hostage)
            (near-object hostage desert-car)))
```

The tactical goal is to rescue the hostage; the planning **:goal** is to establish the predicates (*liberated-unit* hostage) and (*near-object* hostage desert-car). The unit named *hero* represents the player; the predicate (*player-near-unit* hero) means that the player starts at the hero's location.

Operators are designed from the high level scriptable actions of VBS2 and then extended to deal with the tactical goals of a particular mission: we designed extra predicates, such as the previous unary *liberated-unit*, which we included in our PDDL operators so as to achieve a mission:

```
(:action escort-unit
    :parameters (?soldier - unit ?hostage - unit ?vehicle - vehicle)
    :precondition (and (near-object ?soldier ?hostage)
        (soldier-unit ?soldier)
        (hostage-unit ?hostage)
        (safe-unit ?hostage)
        (reachable-unit ?hostage) )
    :effect (and (liberated-unit ?hostage)
        (near-object ?hostage ?vehicle)
        (not (near-object ?soldier ?hostage))
        (not (hostage-unit ?hostage))
        (not (safe-unit ?hostage))
        (not (reachable-unit ?hostage))))
```

Once a plan is found, we are left with its execution. A script is automatically generated to let VBS2 monitor (role assignment to currently available units, game trigger conditions for the execution of actions) the execution:

```
[companion, enemy] execVM "kill_unit.sqf"
    selectPlayer companion
    [hero, position hostage, str hostage ] execVM "move.sqf"
    _trg=createTrigger["EmptyDetector", position hostage];
    _trg setTriggerArea[3,3,0,false];
    _trg setTriggerActivation["WEST","PRESENT",true];
    _statement='[hero, hostage, desert_car ] execVM "escort_unit.sqf"'
    _trg setTriggerStatements["this", _statement, ""];
```

We also automatically generate as many files as operators in the plan, in order to read actual values of the parameters from the game, activate necessary animations and perform necessary computations such as where to throw a grenade. Here is a part of the "kill_unit.sqf" generated script file which makes the grenadier throw a grenade to neutralize the enemy:

```
_Soldier = _this select 0;
_Enemy = _this select 1;

// On screen printing
hint "kill-guard companion enemy" ;
// Performs a "throw" animation
_Soldier switchmove "AwopPercMstpSgthWnonDnon_end";
// Grenade data
_array=[getPos _Enemy, "VBS2_ammo_G_40x46mm_HE",15,4,3,1,0.75,1,5];
_height = _array select 2 ; _coords = _array select 0 ;
// Maximum Distance for shots off the Aimed Point
_radius = _array select 8 ;
_tmp setpos [(_coords select 0) + (random _radius) - (_radius / 2),
             (_coords select 1) + (random _radius) - (_radius / 2), _height];
```

We did not investigate the handling of emergency situations as we did for Iceblox where we were able to avoid unexpected flames and find new weapons for them. Also, as we did not access VBS2 low level instructions (e.g. path planning), squad tactics that would, for instance, place soldiers in front of a building according to the current military doctrine, are not achievable. We hope to tackle these in the future.

Reverse Engineering

The previous two case studies illustrate how real-time PDDL-based planning can be achieved without affecting game playability. This section presents a brief reverse engineering of the planning developments made during these two case studies; two UML activity diagrams (Douglass 2000) document this step.

Figure 1 presents a UML activity diagram of the planning component as reverse engineered from the previous two case studies. Round corners rectangles represent procedures which we implemented as threads. This is a classical pipeline architecture where each thread waits for the previous thread for its input parameters. The pipeline is enabled when goals are received. Goal selection is then activated so as to simplify plan search. For instance, coins in Iceblox are processed one at a time, according to distance from the penguin and the number of flames less than four crossroads away. Distance and danger criteria are also relevant in the setting of VBS2. It is possible to enhance this diagram with a Take Cover procedure whose objective would be to protect the characters involved in the plan during plan search so that these characters stay alive to carry out the plan. As we mentioned earlier, we avoided such a protection step in our two case studies.

Figure 2 presents a UML subsidiary activity diagram of the plan execution thread which appears in Figure 1. The rectangle on the top side is the input parameter and the two rectangles on the right side are two values of the output parameter: as outlined in Figure 1, the plan execution receives a plan to execute and returns whether this execution is a success. As in Figure 1, round corners rectangles represent subsidiary activities which we implemented as threads. Each high level action is compiled into game instructions; for instance path planning data must be generated in Iceblox while appropriate animations and waypoints must be generated in VBS2. High level action compiling must be done on the fly, one high level action at a time: further game instructions become useless when an emergency game situation arises. The compilation time thus saved improves playability and allows to test for emergency game situations.

Conclusion

We reported two cases studies where we were able to achieve PDDL-based real-time planning; the first case study is an arcade game and the second is a commercial serious game. PDDL is an expressive language which easily maps to high level video-game actions. Among many engineering decisions, multi-threading, goal selection and on the fly compilation of high level actions proved to be necessary steps

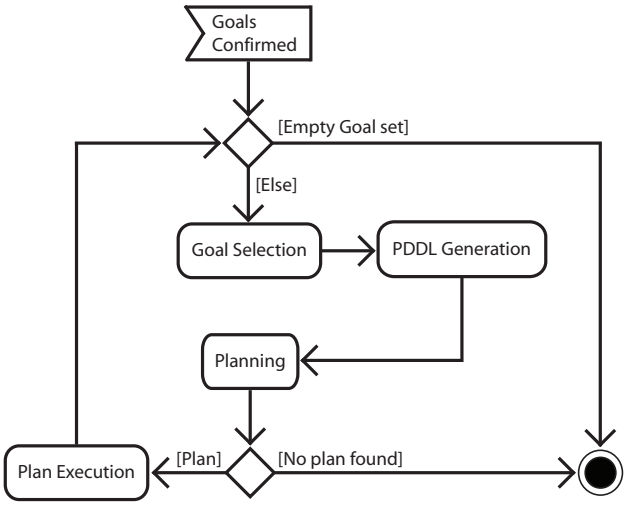

Figure 1: UML activity diagram of a PDDL-based Planning Component. Round rectangles represent subsidiary activities. The component begins with accepting the signal of goals confirmation and ends when this goal set is empty or else when no plan has been found. When the execution of a plan is a success, its goal are removed from the goal set by the subsidiary activity Plan Execution (see Figure 2).

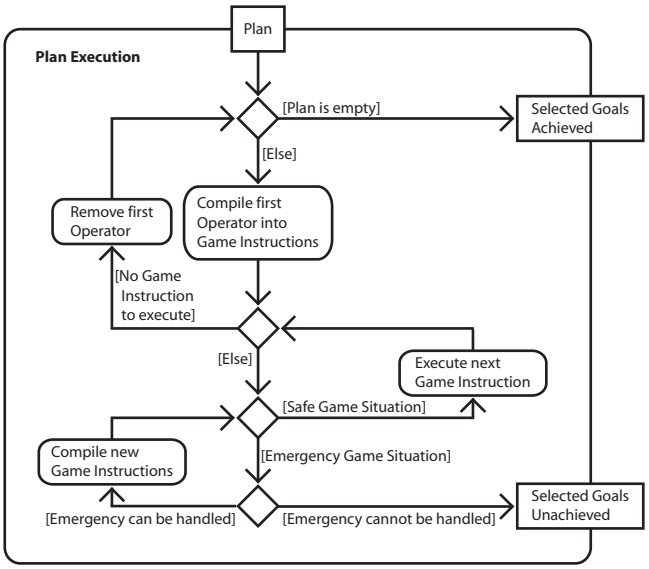

Figure 2: UML subsidiary activity diagram of the Plan Execution from Figure 1; it receives a plan as an input parameter and returns whether the plan's goals have been achieved. Operators from the plan are compiled, one at a time, into low-level game instructions such as path planning. These instructions are in turn executed one at a time, thus providing an opportunity to test for emergency game situations.

to cope with the untractability of PDDL-based plan search. We eventually gather those lessons in UML diagrams which must be understood as design documents for a real-time PDDL-based planning component.

Are two cases studies enough? A 2D arcade game is really different from a 3D commercial serious game but as incredible as this may sound, everything from Iceblox was reused or adapted for VBS2. In fact, these two games are close in terms of gaming experience and the previous question should be instead: what kind of game genre next? Good question, isn't it?

Acknowledgements

This work is part of a 3 year project founded by the Saint-Cyr Foundation. Thanks to Rick Alterman, Jon Gratch, David Kirsh, Carlos Linarès, Jeff Orkin and Adrian Smith for helpful discussions; and to two anonymous reviewers for their constructive reviews.

References

Bartheye, O., and Jacopin, É. 2005. New results for arithmetic constraints partial order planning. 24^{th} Workshop of the UK Planning and Scheduling Special Interest Group, London, UK, Dec. 15-16.

Bartheye, O., and Jacopin, É. 2008. Connecting pddl-based off the shelf planners to an arcade game. ECAI Workshop on AI in Games, Patras, GR, Jul. 21.

Bartlett, N.; Simkin, S.; and Stranc, C. 1996. *Java Game Programming*. Coriolis Group Books.

Bohemia-Interactive-Australia. 2006–2009. Virtual battle space 2. `http://www.vbs2.com/`.

Douglass, B. 2000. *Real-Time UML (2^{nd} Edition)*. Addison-Wesley.

Erol, K.; Nau, D.; and Subrahmanian, V. 1995. Complexity, decidability and undecidability results for domain-independent planning. *Artificial Intelligence **76**(1-2)* 75–88.

Fikes, R.; Hart, P.; and Nilsson, N. 1972. Learning and executing generalized robot plans. *Artificial Intelligence **3**(4)* 251–288.

Hoffmann, J. 2001. FF: The Fast-Forward planning system. *AI Magazine **22**(3)* 57–62.

Hornell, K. 1996. Iceblox. `http://www.-javaonthebrain.com/java/iceblox/`.

International Planning Competition. 1998–2009. `http://ipc.icaps-conference.org/`.

Microsoft. 2007. Directx 9. `http://msdn.microsoft.com/en-us/directx`.

Orkin, J. 2003. Applying goal-oriented action planning to games. In Rabin, S., ed., *AI Game Programming Wisdom 2*. Charles River Media. chapter 3.4, 217–227.

Orkin, J. 2006. Three States and a Plan: The A.I. of F.E.A.R. In *Proceedings of the Game Developper Conference*, 17 pages.