SHPE: HTN Planning for Video Games

Alexandre Menif[1], Éric Jacopin[2], and Tristan Cazenave[3]

[1] Sagem Défense et Sécurité, 100 Avenue de Paris, 91300 Massy Cedex, France
[2] MACCLIA, CREC Saint Cyr, Écoles de Coëtquidan, F-56381 GUER Cedex, France
[3] LAMSADE, Université Paris-Dauphine, 75016, Paris, France

Abstract. This article describes SHPE (Simple Hierarchical Planning Engine), a hierarchical task network planning system designed to generate dynamic behaviours for real-time video games. SHPE is based on a combination of domain compilation and procedural task application/decomposition techniques in order to compute plans in a very short time-frame. The planner has been able to return relevant plans in less than three milliseconds for several problem instances of the *SimpleFPS* planning domain.

1 Introduction

Automated planning is now being used in popular games to satisfy the need of more realistic behaviours for Artificial Intelligence (AI) agents. The advantage of automated planning is twofold for the game industry. First, planners dynamically generate sequences of actions by reasoning according to goals, and thus go beyond simple reactive behaviours. Secondly, the use of a planner improves software maintenance, as an AI designer would only have to define sets of goals and actions, without worrying too much about the interactions between them. Planning has also drawbacks: it is known to require significant CPU time, while modern game engines are already consuming most resources of common gaming hardware. Early games implemented the STRIPS-like GOAP (Goal-Oriented Action Planning) system developed for the AI of F.E.A.R. [9], but nowadays several games have switched to Hierarchical Task Network (HTN) [2] planners. The latter requires to model and maintain an additional amount of planning knowledge, but also achieves better performance. However, the use of planning in games is still limited.

A recent study conducted in some popular video games [4], brings an interesting insight into two games implementing HTN based planning: *Killzone 3* (2011) and *Transformers 3: Fall of Cybertron* (2012). The study reveals the current performance of both games: (i) plan lengths hardly exceed 4 actions (longest plans, up to 12 actions, may appear, but they are rare), (ii) the number of Non-Playable Characters (NPCs) simultaneously handled by the planning system is below the size of a squad (less than 12 AI entities), and (iii) approximately one plan per second and per NPC is generated in average. As a comparison, our goal is to simulate the tactical behaviour of an entire platoon of soldiers (nearly 30

to 40 NPCs), with plans potentially more complex than sequences of 4 actions. To preserve playability and immersion, the planner must be able to return plans for any agent in less than one second of real time. AI usually benefits from 10% of the overall computation time in a game (about 100 ms), thus only 2 to 3 ms per NPC are available in order to plan for a platoon.

This paper introduces the Simple Hierarchical Planning Engine (SHPE). This planning system is based on the same HTN planning techniques currently implemented in games, but achieves better performance thanks to an alternative encoding of planning data. Section 2 describes the main features of the system, some algorithmic details and how to operate the planner. Section 3 focuses on some additional features currently under study. Finally, section 4 evaluates the performance of the planner using the *SimpleFPS* domain [11], a planning domain designed to emulate a game-world environment and test planners according to game mechanics.

2 SHPE: Simple Hierarchical Planning Engine

2.1 Planning Data Representation

Planning data are classically represented by a first-order logical language. A state of the world is described by a set of predicates, which represent properties holding for some objects of the world. Possible changes are described with operators. An operator is defined with a precondition, some effects, and may also have a numerical cost. Preconditions and effects are logical formulas on predicates, and they respectively represent the condition to apply the operator and the changes occurring on a state after the application of this operator. The set of definitions for predicates and operators is called a planning domain. Plans can be sequences or partially ordered sets of operators, and an optimisation criterion can be defined on the costs of these operators. A classical planner expects as input a goal condition on the predicates as well as an initial state, and returns a plan transforming the initial state into one satisfying the goal. A standard planning domain language conforming to these principles is PDDL [8]. Figure 1 shows some samples of planning data expressed in this language.

HTN planning introduces tasks into the planning domain. There are two types of tasks. Primitive tasks are associated with operators, while compound ones are designed to be decomposed into a partial plan made of subtasks, called a task network. For each compound task, alternative decompositions are defined through several HTN methods. This paper adopts the formalism for methods of the Simple Hierarchical Ordered Planner (SHOP [5]) and its successor SHOP2 [6]. Thus, our methods are defined as successive pairs of preconditions and task networks (each pair is called a branch, and the planner only decomposes the method for the first satisfied branch). Figure 2 presents some examples of such methods. The purpose of an HTN planner is different from a classical one: instead of building a plan fulfilling a goal from the initial state, an HTN planner decomposes an initial task network down to a plan only made of primitive tasks, and applicable to the initial state.

```
; a definition of an action in PDDL, here this action
; applies to a NPC that uses a gun to shoot at the player
(:action shoot-player
 :parameters   (?npc ?gun ?wpt1 ?wpt2)
 ; to perform the action, the NPC needs a loaded gun and
 ; a clear line of sight to the player
 :precondition (and (at ?npc ?wpt1) (player-at ?wpt2)
                    (holding ?npc ?gun) (loaded ?gun)
                    (visible ?wpt1 ?wpt2))
 ; the player is wounded and the gun is no longer loaded
 :effect       (and (not (loaded ?gun)) (player-wounded)))

[...]

; a description of the initial situation
(:init (at npc0 wpt0) (player-at wpt1) (visible wpt0 wpt1)
       (visible wpt1 wpt0) (gun gun0) (loaded gun0)
       (holding npc0 gun0) [...])
; the goal we want to achieve
(:goal (player-wounded))
```

Fig. 1. An operator (action) described in the PDDL planning representation language, along with an initial state and a goal formula. Here, *(shoot-player npc0 gun0)* is a valid plan according to the initial situation and the goal.

```
; method for attacking the player at range
(:method (attack-player ?npc)
 ; a first branch, when the NPC already has
 ; visibility with the player
 (and (at ?npc ?wpt1) (player-at ?wpt2) (visible ?wpt1 ?wpt2)
      (gun ?gun) (holding ?npc ?gun))
 ((!shoot-player ?npc ?gun))
 ; a second branch that moves the NPC to a waypoint where it
 ; will have visibility to the player location
 (and (waypoint ?wpt1) (player-at ?wpt2) (visible ?wpt1 ?wpt2)
      (gun ?gun) (holding ?npc ?gun))
 ((move ?npc ?wpt1) (!shoot-player ?npc ?gun)))

; method for attacking the player in melee combat
(:method (attack-player ?npc)
 (and (at ?npc ?wpt) (player-at ?wpt) (knife ?knife)
      (holding ?npc ?knife))
 ((!stab-player ?npc ?knife)))
```

Fig. 2. Two methods that define alternative decompositions for the compound task *(attack-player ?npc)*. The first method provides the behaviour to attack the player with a ranged weapon, while the second one makes the NPC use a close combat weapon. The LISP-like syntax is the one used by SHOP for its input data, where the exclamation mark "!" denotes a primitive task.

From a programming viewpoint, logical formulas (preconditions, effects) in operators and methods are generally encoded as lists of atomic propositions evaluated by an inference engine. Pyhop [7], a SHOP-like HTN planner coded in Python, follows another approach supposed to be more suited to games. First, world states are represented as Python data structures containing state-variables, an alternative representation for facts in planning, instead of sets of predicates. Secondly, operators and methods are Python functions taking a state as input, and respectively returning a new state and a sequence of subtasks. Figure 3 provides an insight on what such functions look like. The entire domain is therefore written in Python, and not with a logical language as usual. For SHPE, we decided to follow the same rules for reason of both system simplicity and expectation of a runtime improvement. Indeed, there is no need to code an inference engine and state-variables can be instantly accessed in a structure to read or modify their values, while we have to find a predicate to add or delete it from a state. But unlike Pyhop, we did not choose Python but C++ in order to implement our planning system. First, this is a very commonly used language for video games and we expect our choice to ease the integration of the planner in most game engines. Secondly, being a low-level compiled language, C++ seemed to be an appropriate option to achieve the best runtime.

```
def shoot_player(state, npc, gun):
    if state.visible[state.at[npc]][state.player_at] and \
       state.holding[npc][gun] and state.loaded[gun]:
        state.loaded[gun] = False
        state.player_wounded = True
        return state
    else: return False
```

Fig. 3. The same operator as defined in Figure 1, but encoded in Python this time.

2.2 Algorithm

SHOP, Pyhop and SHPE use the same algorithmic principle. They conduct a depth-first, backtracking search in the space of partially decomposed task networks, combined with a forward state-space search. When the planner selects the next task to process, it always picks one that has no predecessor in the task network. Doing so results in constructing the plan in the same order as it will be executed. The planner always has a full description of the current state of the world at its disposal, thus logical expressions in operators and methods can be written with very expressive logical formulas. For instance, the domain modeling language of SHOP2 allows for existential and universal quantifiers, disjunction, implication and even more specific expressions [6]. Not only does this type of search provide the domain modeler with a powerful way to encode good strategies

for decomposing task networks, but it also justifies why operators and methods can be encoded as functions in Pyhop.

However, the implementation of SHPE differs from Pyhop in several ways. We do not actually define operators and methods, and associate them with tasks. As primitive tasks are in one-to-one correspondence with operators, we simply blend both of them together into a single primitive task definition. We do not define each method separately either. Instead, they are all gathered in the body of a single function named *decompose*. This function is defined for all compound task types, and it returns a list of all the decompositions for each method. Besides, an operator/primitive task is not considered as being a single function, but is split into three parts:

1. The function *applicable* returns the evaluation of the precondition (a boolean value).
2. The function *apply* returns the new value of the state after the application of all effects.
3. The function *cost* returns the cost of the operator/task.

Another difference is the iterative structure of the algorithm implementation. It provides the ability to interrupt and resume the planner, which is a nice feature to have with some game engines in order to time-slice planning through multiple frames. The last addition is the "branch-and-bound" optimization technique implemented for SHOP2 [6] in order to search for a least-cost plan, along with the standard procedure returning the first plan found. The optimal version, combined with the ability to interrupt the planner, can work more or less as an "anytime-like" algorithm [1]: when a first solution is found, possibly not optimal, it keeps running in order to find a better plan as long as it has not been interrupted. The procedure executed at each iteration of the planner is described in Algorithm 1. Algorithms 2 and 3 are respectively the standard and optimal procedures to run the planner (without interrupting it).

2.3 Operating the planner

SHPE is implemented as a C++ template library. The planner is provided in a template *Planner<MyState>* class, and the template parameter must be specialized with the C++ structure type defined for the state. The implementation of the planner provides the necessary member functions to be run in different ways: *run*, and *run_best* are used to operate the planner directly in the standard and optimize mode, while the function *next* is publicly visible to enable the integration of the planner according to one's requirements. For example, it can be used to run the planner for a limited amount of iterations or time.

All primitive and compound tasks must be implemented by inheriting the provided *Task<MyState>* virtual class and overloading its virtual member functions: either *applicable*, *apply* and *cost* if this is a primitive task, or *decompose* if it is a compound one. Also, a task is supposed to be implemented with all its parameters as class member attributes. Due to the use of polymorphism,

Algorithm 1 next(*stack*, *best_plan*, *best_cost*)

```
(plan, cost, state, task_network) ← top(stack)
pop stack
if cost ≥ best_cost then
  return
end if
if task_network is empty then
  best_plan ← plan
  best_cost ← cost
  return
end if
tasks ← all tasks in task_network without predecessor
for all t ∈ tasks do
  if t is a primitive task then
    if t.applicable(state) then
      plan ← append t to plan
      cost ← cost + t.cost(state)
      remove t from task_network
      push (plan, cost, t.apply(state), task_network) on stack
    end if
  end if
  if t is a compound task then
    for all tn ∈ t.decompose(state) do
      replace t in task_network with tn
      push (plan, cost, state, task_network) on stack
    end for
  end if
end for
```

Algorithm 2 find_first_plan(*state*, *task_network*)

```
stack ← []
best_plan ← []
best_cost ← ∞
push ([], 0, state, task_network) on stack
while best_cost = ∞ and stack is not empty do
  next(stack, best_plan, best_cost)
end while
```

Algorithm 3 find_best_plan(*state*, *task_network*)

```
stack ← []
best_plan ← []
best_cost ← ∞
push ([], 0, state, task_network) on stack
while stack is not empty do
  next(stack, best_plan, best_cost)
end while
```

an instance of the *Planner<MyState>* class only deals with references to tasks, therefore the actual task objects need to be stored in a global memory space. For this purpose, and also to limit dynamic heap allocation, a caching system registers all allocated instances of a type of task. But if a task class only has few different instances, they can also be stored in static attributes of this class. Figure 4 provides a practical example of how a task can be implemented for SHPE.

So, in order to get plans from SHPE, one needs to execute the following steps:

1. Define a C++ structure for the state (*MyState*). Some variables of the domain are never modified and can be easily identified as they do not appear in any effects; thus a good practice is to define a constant state structure, and make all constant variables from a state pointing to the constant structure. This technique significantly reduces memory usage and runtime as states are copied many times in the planner's stack.
2. Define all primitive and compound tasks of the domain as C++ classes inheriting from *Task<MyState>*.
3. Include the C++ files for your domain and the ones from SHPE within a project, and write some code to instantiate and use the specialized instance of the *Planner<MyState>* class.
4. Compile and run this domain-specific planning program.

3 Planning Domain Design and Pre-compilation

3.1 High Level Modeling Language

When it comes to modeling a planning domain, C++ is anything but an appropriate language. Being a low-level programming language, it is already quite verbose. Besides, our way to implement tasks does not help either. On several occasions, a domain written with a few hundred lines of LISP code was expanded into a few thousand of C++ lines. Thus, a much more convenient high-level planning domain modeling language is needed. Neither PDDL nor the LISP syntax of SHOP were appropriate as this language should support state-variable representation and HTN decompositions. By contrast, the ANML language [10], currently under development at NASA, provides these elements. Therefore, we have started to design a language based on ANML as a tool for domain modeling.

However, ANML is quite a comprehensive language for planning and already supports many features. As some of these features are out of scope for SHPE, they were simply discarded. The removal of temporal qualifications on preconditions is probably the most noticeable change (indeed SHPE does not support temporal networks, but this may be a further improvement). Some minor changes on the syntax were also included in order to conform the language to the task-based HTN formalism of SHOP. Figure 5 provides an insight on some elements of a planning domain expressed with this language.

The language was also expanded with *sort-by* and *first* preconditions, two features available in SHOP and SHOP2. *sort-by* preconditions allow to sort the

```
class ShootPlayer : public Task<MyState> {
public:
    // the class constructor
    ShootPlayer(const Npc& npc, const Gun& gun) : primitive(true),
                                                  npc(npc),
                                                  gun(gun)
    {
    }

    // evaluate the precondition according to the current state
    bool applicable(const MyState& state) const
    {
        return state.visible[state.at[npc]][state.player_at] and
               state.holding[npc][gun] and state.loaded[gun];
    }

    // apply the effects of this task on the current state
    void apply(MyState& state) const
    {
        state.loaded[gun] = false;
        state.player_wounded = true;
    }
protected:
    // print this task for debugging purpose
    std::ostream& print(std::ostream& out) const
    {
        return out << "ShootPlayer(" << npc << ", " << gun << ")";
    }
private:
    Npc npc;
    Gun gun;
};
```

Fig. 4. An example of a C++ definition for the primitive task *ShootPlayer(npc, gun)*. The *cost* function is not overloaded here. In this case, the default implementation of this function, inherited from *Task<MyState>*, returns 1. It is a primitive task, so the virtual function *decompose* is not overloaded either and a call to this function will return an empty set of decompositions.

```
task ShootPlayer(Npc npc, Gun gun) {
    cost := 1;
    {
        visible(at(npc), player_at);
        holding(npc, gun);
        loaded(gun) == true :-> false;
        player_wounded := true;
    }
}

task AttackPlayer(Npc npc) {
    // method for attacking the player at range
    method {
        // a first branch, when the NPC already has
        // visibility with the player
        branch {
            exists (Gun gun) {
                visible(at(npc), player_at);
                holding(npc, gun);
                ordered(ShootPlayer(npc, gun));
            }
        }
        // a second branch that moves the NPC to a waypoint
        // where it will have visibility to the player location
        branch {
            exists (Waypoint wpt, Gun gun) {
                visible(wpt, player_at);
                holding(npc, gun);
                ordered(Move(npc, wpt), ShootPlayer(npc, gun));
            }
        }
    }
    // method for attacking the player in melee combat
    method {
        exists (Knife knife) {
            at(npc) == player_at;
            holding(npc, knife);
            ordered(StabPlayer(npc, knife));
        }
    }
}
```

Fig. 5. Primitive and compound tasks defined with a high-level modeling language, in a state-variable based representation.

variable bindings satisfying the precondition according to the value of a numerical expression and *first* preconditions allow to consider only the first binding (Figure 6). The ability to insert calls to external user-defined functions is also under study.

```
task RestoreHealth(Npc npc) {
    method {
        // a first branch, when the NPC already has a medikit
        branch {
            first (Medikit medikit) {
                holding(npc, medikit);
                ordered(UseMedikit(npc, medikit));
            }
        }
        // a second branch when the NPC needs to
        // find a medikit
        branch {
            sort-by (Medikit medikit; distance(at(npc),
                                               at(medikit)); <) {
                ordered(Move(npc, at(medikit)),
                        UseMedikit(npc, medikit));
            }
        }
    }
}
```

Fig. 6. A method using both *sort-by* and *first* expressions. In the first branch, it would be pointless to generate a decomposition for each medikit the NPC holds as the choice of the medikit would not alter the quality of the plan, so it makes sense to consider the first satisfier only. In the second one, a medikit in the neighborhood is more likely to be the best option in order to find one in fewer steps, so exploring this option first is a good heuristic.

3.2 Domain Pre-compilation

Having a convenient language for domain modeling is one thing, but currently the domain still requires to be translated by hand into C++ code. Depending on the size of the domain, this task quickly becomes tedious and error-prone, and completely goes against the requirement for a system simple enough to be used by a non-programmer (for example by a game designer). Therefore, an additional piece of software is required to parse the domain from the high-level modeling language and generate a C++ domain definition automatically. At the end of the process, the generated C++ classes containing the domain elements as well as the specialized planner would be integrated in a game project or even compiled

as an independent dynamic link library to achieve modularity. Nevertheless, this tool has not been implemented yet.

4 Performance Evaluations

4.1 The *SimpleFPS* Planning Domain

The *SimpleFPS* domain [11] has been designed specially to produce planning problems that could serve as benchmarks to evaluate planners according to FPS (First-Person Shooter) game mechanics. *SimpleFPS* problems stage a NPC and a player in a game level made of several areas connected to each other, and each area includes various types of points of interest (items such as weapons, medikits or keys, doors between areas, cover-points...). A comprehensive description of the domain can be found in the original paper.

As this domain is provided in a PDDL format, it was necessary to convert the predicate based representation into a state-variable one. It was also necessary to add tasks and methods in order to operate the domain with an HTN planner, as *SimpleFPS* has only been designed to evaluate goal-oriented classical planning techniques. The hierarchical structure of the domain makes use of the tools provided by SHOP2 to encode heuristics in methods (*sort-by* expressions...) in order to make the planner more likely to find near-optimal solutions. To achieve this, we added a *distance* predicate/state-variable to encode the distances between all areas (this distance is used as the criterion to sort the areas and the point of interest in the methods). In an actual game environment, this information could be computed with the euclidean distance between points of interest.

4.2 Experiments

For the first set of experiments, we wanted to compare the performance of SHPE with a similar planner. We selected JSHOP2 [3], a java implementation of SHOP2 [6]. JSHOP2 is a problem specific planner: a specialized instance of the planner is compiled for each domain and problem. This feature provides JSHOP2 with an advantage over SHPE, which can only be optimized for the planning domain. Besides, JSHOP2 already compared favorably to other more academic implementations (it has been shown to run by a polynomial order of magnitude faster than SHOP2). So it revealed itself to be an appropriate candidate to compete with SHPE in our benchmark. Several problems with various numbers of points of interest were randomly generated (the number of areas is set to 10).

The computer used for all experiments is equipped with an Intel core i5 CPU (2.66 GHz), 4 GB of RAM, and it runs a 64 bits version of Debian 7.0. This configuration is equivalent to an average *Steam Box*, a new PC-based gaming concept currently developed by *Valve*. The running-time results for SHPE and JSHOP2 are presented in Figure 7: each measure is the average running time for a collection of one hundred samples of the same size. The results indicate a

clear advantage on the side of SHPE, which solves each set of problems 10 to 15 times faster than JSHOP2. Moreover, the running times are short enough (less than 3 ms for each instance) to assume that SHPE should be able to plan for several squads of NPCs in a game or a simulator.

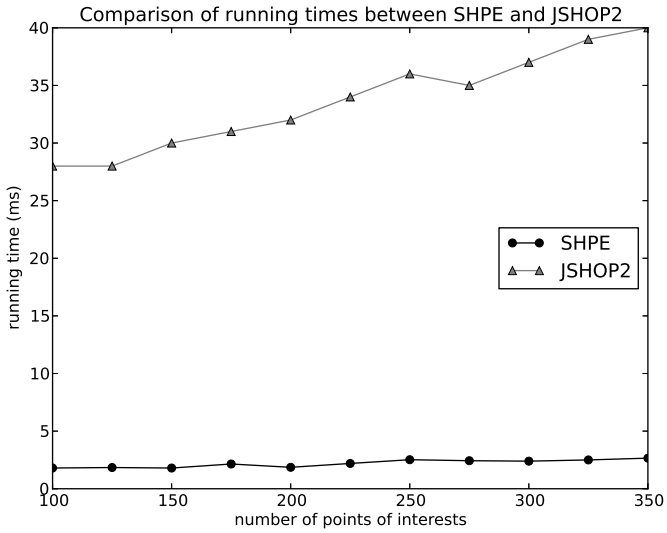

Fig. 7. SHPE and JSHOP2 performance comparison on different instances of problems from the *SimpleFPS* domain. The instances were generated with ten areas and a varying amount of points of interest. For these scenarios, SHPE outperforms JSHOP2 by at least a factor of 10.

The results show that SHPE runs quite fast, but what about the quality of the plans? There is actually no reason to evaluate SHPE on this aspect against JSHOP2. Indeed plan quality is related to the designed decomposition hierarchy, and both planners share the same. Our hierarchy generally performs well: it provides a satisfactory near-optimal plan in most cases, and sometimes it even returns an optimal one. However there are situations when it does not: for instance, a sequence of 70 actions is returned when the optimal plan only contains 30 of them. In this case, will the "branch-and-bound" optimization technique be of any help? In order to get an idea about it, we ran the planner until it had returned an optimal solution for two scenarios. In the first one, the planner first returned a plan far from being optimal; thus it was interesting to measure how long the game would have to wait for a satisfactory solution. In the second scenario, the returned plan was already satisfactory, but could still be improved. The answer is shown in Figure 8. In both cases, the optimal solution is out of reach: it requires several seconds in the already near optimal case, and several

minutes in the other case. When the initial solution is far from being optimal, it also requires several minutes to get an acceptable one. So this optimization option does not seem very useful and the ability to obtain a good solution from the planner mainly relies on the designed decomposition methods.

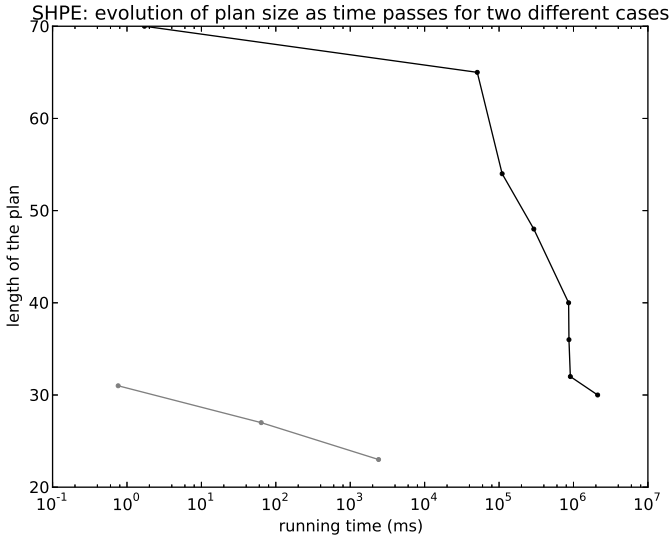

Fig. 8. SHPE using the "branch-and-bound" optimization techniques to search for the best solution in two scenarios. One when the first plan found is far from being optimal (the uppermost graph, starting with 70 actions), and another case with a first plan almost optimal, but still improvable (the lower graph, starting with 31 actions).

5 Conclusion

The ideas introduced with Pyhop [7] were put into application in SHPE in order to provide fast planning capabilities to video games. The system has reached the targeted performance: it has been evaluated against various problems with properties similar to FPS games, and was able to solve them in a few milliseconds. In addition, a high-level modeling language can be used to design the planning knowledge required to operate the planner efficiently. To bridge the gap between this high-level language and the C++ encoding expected by the planner, a pre-compiler should be implemented. This component would enable AI designers to modify the behaviour of game characters, without any skills in low-level C++ programming.

But even if this system achieves its initial goal in terms of performance and maintenance, it does not address issues like real-time re-planning in dynamic

environments such as games. Besides, even if plans containing more than twenty actions can be computed in a few milliseconds, this time is still partially a waste, as it is likely that most of the plan will no longer be relevant to the evolution of the situation. Thus, our future plan for this system is to study and incorporate partial and delayed decomposition. A hierarchical planner including these features could decompose a plan into primitive tasks for imminent acting only, keep the more distant tasks at a more abstract level and eventually expand them at the appropriate time.

References

1. Thomas L. Dean and Mark S. Boddy: An Analysis of Time-Dependent Planning. In *AAAI*. vol. 88, p. 49–54 (1988)
2. Malik Ghallab, Dana Nau, and Paolo Traverso: *Automated Planning: Theory and Practice*. Morgan Kaufmann (2004)
3. Okhtay Ilghami and Dana S. Nau: A General Approach to Synthesize Problem-Specic Planners. Technical Report CS-TR-4597 and UMIACS-TR-2004-40, University of Maryland (2003)
4. Éric Jacopin: Game AI Planning Analytics: Evaluation and Comparison of the AI Planning in three First-Person Shooters. To be published in *AIIDE* (2014)
5. Dana Nau et al.: SHOP: Simple Hierarchical Ordered Planner. In *IJCAI-99*, p. 968–975 (1999)
6. Dana Nau et al.: SHOP2: An HTN Planning System. In *Journal of Articial Intelligence Research (JAIR)*. vol. 20, p. 379-404 (2003)
7. Dana Nau: Game Applications of HTN Planning with State Variables. In *ICAPS Workshop on Planning in Games*. Invited talk (2013)
8. Drew McDermott et al.: PDDL - The Planning Domain Definition Language (1998)
9. Jeff Orkin: Three States and a Plan: The AI of F.E.A.R. In *Game Developer's Conference (GDC)* (2006)
10. David E.Smith, Jeremy Frank and William Cushing: The ANML Language. In *ICAPS-08* (2008)
11. Stavros Vassos and Michail Papakonstantinou: The SimpleFPS Planning Domain: A PDDL Benchmark for Proactive NPCs. In *AIIDE Workshop: Intelligent Narrative Technologies* (2011)