Building a Planner: A Survey of Planning Systems Used in Commercial Video Games

Xenija Neufeld*†, Sanaz Mostaghim*, Dario L. Sancho-Pradel†, Sandy Brand†
*Faculty of Computer Science, Otto von Guericke University Magdeburg, Germany
†Crytek GmbH, Frankfurt, Germany

***Abstract*—In the last decade, many commercial video games have used planners instead of classical Behavior Trees or Finite State Machines to define agent behaviors. Planners allow looking ahead in time and can prevent some problems of purely reactive systems. Furthermore, some of them allow coordination of multiple agents. However, implementing a planner for highly-dynamic environments like video games is a difficult task. This work aims to provide an overview of different elements of planners and the problems that developers might have when dealing with them. We identify the major areas of plan creation and execution, trying to guide developers through the process of implementing a planner and discuss possible solutions for problems that may arise in the following areas: environment, planning domain, goals, agents, actions, plan creation and plan execution processes. Giving insights into multiple commercial games, we show different possibilities of solving such problems and discuss which solutions are better suited under specific circumstances and why some academic approaches find a limited application in the context of commercial titles.**

***Index Terms*—Planning, Multi-agent Systems, Video Games, Agent Behavior.**

I. Introduction

FOR a long time, most commercial video games have been using Behavior Trees or Finite State Machines to define agent behaviors and to allow agents to make decisions at runtime [1]. These approaches implement reactive agent behaviors, but they cannot be used to look ahead in time and create longer action plans. Furthermore, they are very limited with regard to coordination of actions of multiple agents. For these reasons, in the last decade, many games have implemented different kinds of action planners (see sections II-A,II-B for examples). By using a planner it is possible to overcome some problems of purely reactive systems as it enables us to look further ahead in time and implement context-sensitive behaviors and strategies. The game *F.E.A.R.*[1] is often mentioned as the first published title that successfully used a planner-based approach [1].

This survey aims to provide an exhaustive overview of different modules that need to be taken into consideration when creating a planner for a video game where different Non-Player-Characters (NPCs) need to interact with each other. Furthermore, providing insights into multiple published games, this work shows different possibilities of implementing these modules and explains why some academic approaches find only limited application in the area of commercial video games. With this approach we hope to identify opportunities for further research topics geared towards solving the complex needs of practical applications.

[1]F.E.A.R.: Developer: Monolith Productions; Publisher: Sierra Entertainment. 2005

The remainder of the paper is structured as follows: first, we describe two major planners used in the game industry – the Stanford Research Institute Problem Solver and the Hierarchical Task Network – in section II. Afterwards we divide planner systems into seven major areas that we look into in more detail. The first six areas describe the process of creating the planner itself, whereas the last area handles the process of implementing a plan executor. Combined, these seven areas are the following: the environment that the planner is used in, described in section III; the planning domain that it operates on, described in section IV; the goals, which have to be achieved by agents using the plans, in section V; the agents, that execute plans, described in section VI; the actions that the agents can perform, in section VII; the process of plan creation and the process of plan execution in sections VIII and IX. Additionally, we draw the reader's attention to important tools in section X and provide concluding remarks in section XI.

Trying to guide developers through the process of implementing a planner, we will show different aspects of each of the seven areas in an individual figure containing an Activity Diagram. Note though that these diagrams are parts of a single larger activity diagram which represents the whole process of the planner implementation.

II. Planners Used in Commercial Games

Even though there are many different classical planning techniques in academia, hardly any of them have found proper application in the field of *commercial* video games so far. Game environments are often very complex and highly-dynamic. They differ a lot from the environments that classical planning approaches were originally designed for. As described in more detail in [2], classical planning techniques rely on the following assumptions about the system they deal with: a) the system has a finite set of states, b) the system is fully observable, c) it is deterministic, d) it is static in the sense that its state can only be changed through action of agents known to the planner, e) the only kind of agent goals are attainment goals that are defined as goal states meaning that e.g. states to be avoided are not defined, f) a solution plan is an ordered finite set of actions, g) actions have no duration and lead to instantaneous transitions between states, h) planning is done offline meaning that the planner does not take into consideration any changes of the world state that happen while planning. However, most of these assumptions do not hold in games with the exception of constraints b, e, f, and h which may be attainable depending on the actual implementation of the planner, as we will describe in more detail in next sections. Thus, several changes to these approaches are required for planning in games and other practical planning applications [3]. The two main planners used as bases in most published game titles are described in the following section. To our knowledge, no other type of planners has seen adoption in commercial games today.

A. STRIPS and GOAP

The *Stanford Research Institute Problem Solver* (STRIPS) [4] was developed in 1971 at the Stanford Research Institute. Its main purpose was the creation of plans for a robot that should navigate and re-arrange objects in rooms [4]. Implemented in Lisp, STRIPS belongs to the group of problem solvers that search through the space of world states in order to find a state in which conditions of a given goal are satisfied. Such a state is called the *goal state*. The problem space of STRIPS consists of the following five components: the initial world state, the goal world state, a set of operators, their preconditions and their effects. A world state is represented by a set of well-formed formulas of first-order logic[2]. In the context of planning, these formulas are often called *facts*. They describe some world properties using functions and variables like, for example, *isAtPosition(agent, position)* describes the position of an agent. An instance of this fact could be *isAtPosition(agent1, home)* where the variable *agent* gets the value *agent1* and the variable *position* gets the value *home* meaning that *agent1* is at *home*. Operators are symbolic representations of actions that can be performed by agents and are defined by the preconditions under which they are applicable and by the effects that they have on the world states. In addition to world states, preconditions are defined by instances of facts that have to hold true in the world state before the application of an operator. Effects are subdivided into two types: those that add new knowledge (new instances of facts) to the world state and those that remove knowledge from it. The knowledge to be added is stored in an *add list* and the knowledge to be removed is stored in a *delete list*. Effects are the results of applying both lists to a world state.

The following example shows the operator *GoTo* that can make an agent move from *startPosition* to *destinationPosition*. Its precondition says that the agent has to be at *startPosition* before the action can start and its effects show that he will be at *destinationPosition* once the action concludes.

```
Operator: GoTo(agent, startPosition, destinationPosition)
 Preconditions:
    isAtPosition(agent, startPosition)
 Effects:
    Delete List:
       isAtPosition(agent, startPosition)
    Add List:
       isAtPosition(agent, destinationPosition)
```

It is assumed that for any world state, there exists a set of operators that transform such a state into a different one. To find a valid sequence of operators that lead to the goal state, STRIPS uses an extended theorem-prover. This theorem-prover takes the difference between the initial world state and the goal state, comparing the instances of facts and searches for operators that *resolve away* clauses[3] [4]. If an operator that reduces the difference is found, it is added to the list of *relevant* operators.

When a new operator is added to the ordered list of relevant operators, its preconditions add new subgoals to the goal list. These subgoals have to be achieved in order for the operators to be applicable [6, Chapter 4.4]. For example, assume that we have the previous *GoTo*-operator and the following *PickUp*-operator:

```
Operator : PickUp(agent, object, position)
 Preconditions:
    isAtPosition(agent, position)
    isAtPosition(object, position)
 Effects:
    Delete list
       isAtPosition(object, position)
    Add List:
       carries(agent, object)
```

If the goal state contains the instance of the fact *carries(agent1, objectB)* meaning that *agent1* has to pick up *objectB*, a relevant operator would be *PickUp(agent, object, position)* with *agent* = *agent1* and *object* = *objectB* and *position* = *park* (if the object is in the park). This operator would add the subgoals/conditions *isAtPosition(objectB, park)* and *isAtPosition(agent1, park)* to be satisfied. Assuming that *agent1* is currently at *home*, in the next step, the instance *GoTo(agent1, home, park)* of the *GoTo*-operator would have to be added to the list of relevant operators. For more complex operators and preconditions, the search tree might grow at this point very rapidly. For this reason, STRIPS uses heuristics, such as the number of remaining subgoals, during the operator selection process.

Since an operator that is found relevant might be either applicable to the initial world state or some intermediate state or its effects might directly lead to the goal state, it is not known where this operator will occur in the final plan [4]. For example, if *agent1* from the previous example had to wear a special suit when picking up *objectB* in the park, he could

[2]For more information on first-order logic see [5].

[3]In propositional logic, a clause is a disjunction of literals. A literal is either a term/variable or the negation of it [5].

either first put on that suit (for example using the operator Wear(agent, suit)) and then use the *GoTo*-operator or first go to the park and then put on the suit before picking up the object.

One of the earliest examples of the usage of planners for behavioral modeling in commercial games was *F.E.A.R.* Although STRIPS has proven itself as a very powerful tool to be able to reason about big search spaces in classical planning environments, the authors of *F.E.A.R.* adopted a modified version of STRIPS in order to make the planner more controllable, performant and better applicable to a real-time game dropping some of the assumptions of classical planning mentioned in section II. This resulted in the so called *Goal Oriented Action Planner* (GOAP) [7] which included four major differences from the original approach.

First, every action/operator got a cost value assigned. These values were used as heuristic for the A* algorithm that searched through the space of world states. The second difference was the representation of the actions' preconditions and effects. Instead of using add and delete lists, GOAP represented both the preconditions and effects as fixed-sized arrays of world state variables. This way, it was faster to find an action that could reduce the difference between the initial and the goal world state through direct comparison of the arrays.

The third change introduced procedural preconditions with which the system could reason about more complex logic at run-time. So after the initial check on fixed-sized arrays of world state variables, preconditions could now call functions to perform certain checks when necessary (as discussed later in section VIII-A). For example, instead of keeping track of whether or not a path exists or if an enemy is visible, a path-finding function or a ray-caster could be called on-demand.

The final change to the original planner was the introduction of the so-called procedural effects [7] which were used to define which behaviors should be executed by the agents applying the GOAP actions. Therefore, the states of a Finite State Machine (FSM) were used to handle different behaviors of an agent and the actions of a plan represented transitions between these states defining when to enter and leave them [8].

After the successful use of GOAP in *F.E.A.R.*, many other games implemented their own versions of STRIPS-like planners. In this paper, we will look into more details to the planner implementations of games like *F.E.A.R.*, *Dirty Harry*[4] [9], *Tomb Raider*[5] and *Middle-earth: Shadow of Mordor*[6][10].

B. HTN

A few years after the release of *F.E.A.R.* (2005), the next milestone in the history of planning in video games was the game *Killzone 2*[7] (2009) which was the first one to implement a Hierarchical Task Network (HTN) [6, Chapter 11.5], [11].

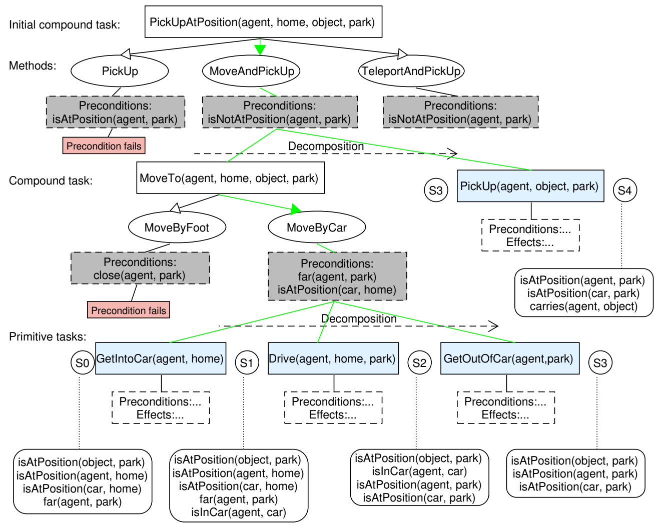
Initial compound task:
PickUpAtPosition(agent, home, object, park)
Methods:
PickUp
MoveAndPickUp
TeleportAndPickUp
Preconditions: isAtPosition(agent, park)
Preconditions: isNotAtPosition(agent, park)
Preconditions: isNotAtPosition(agent, park)
Precondition fails
Decomposition
Compound task:
MoveTo(agent, home, object, park)
S3
PickUp(agent, object, park)
S4
MoveByFoot
MoveByCar
Preconditions:... Effects:...
Preconditions: close(agent, park)
Preconditions: far(agent, park) isAtPosition(car, home)
isAtPosition(agent, park) isAtPosition(car, park) carries(agent, object)
Precondition fails
Decomposition
Primitive tasks:
S0
GetIntoCar(agent, home)
S1
Drive(agent, home, park)
S2
GetOutOfCar(agent,park)
S3
Preconditions:... Effects:...
Preconditions:... Effects:...
Preconditions:... Effects:...
isAtPosition(object, park) isAtPosition(agent, home) isAtPosition(car, home) far(agent, park)
isAtPosition(object, park) isAtPosition(agent, home) isAtPosition(car, home) far(agent, park) isInCar(agent, car)
isAtPosition(object, park) isInCar(agent, car) isAtPosition(agent, park) isAtPosition(car, park)
isAtPosition(object, park) isAtPosition(agent, park) isAtPosition(car, park)

Fig. 1. HTN planning example.

Following its example, many more games adopted this approach and have shown interesting agent behaviors especially in terms of coordination of multiple agents. In this work, we give some insights into games like *Killzone 2 and 3*[8] [12], *Transformers: Fall of Cybertron*[9] [13], *Dying Light*[10] [14] and *PlannedAssault*[11] which is a web-based mission generator for *ARMA II*[12] [15].

Although there are some similarities between HTN and STRIPS, they work in very different ways. Similarly to STRIPS, the world state is described by a set of facts in HTN. However, the main elements of HTN are tasks, which do not stand on their own but build a network. As the name of this approach suggests, this network represents a hierarchy of the so called *compound tasks* and *primitive tasks*. Compound tasks represent the higher levels of the network hierarchy and can be decomposed into further tasks (either compound or primitive). Primitive tasks are the leaves of the network and contain operators which can be compared with the operators of STRIPS. These operators are defined by preconditions and effects. The four operators shown in Figure 1 are the *PickUp*-operator described for STRIPS in section II-A and the following three operators:

```
Operator : GetIntoCar(agent, position)
    Preconditions:
        isAtPosition(agent, position)
        isAtPosition(car, position)
    Effects:
        Add List:
            isInCar(agent, car)
```

[4]Dirty Harry: Developer: The Collective, Publisher: Warner Bros. Interactive, cancelled

[5]Tomb Raider: Developer: Crystal Dynamics, Publisher: Square Enix, 2013

[6]Middle-earth: Shadow of Mordor: Developer: Monolith Productions, Publisher: Warner Bros. Interactive, 2014

[7]Killzone 2: Developer: Guerrilla Games, Publisher: Sony Computer Entertainment, 2009

[8]Killzone 3: Developer: Guerrilla Games, Publisher: Sony Computer Entertainment, 2011

[9]Transformers: Fall of Cybertron: Developer: High Moon Studios, Publisher: Activision, 2012

[10]Dying Light: Developer: Techland, Publisher: Warner Bros. Interactive, 2015

[11]PlannedAssault: plannedassault.com

[12]Arma II: Developer/Publisher: Bohemia Interactive, 2009

```
Operator : Drive(agent, startPosition, destinationPosition)
    Preconditions:
        isAtPosition(agent, startPosition)
        isAtPosition(car, startPosition)
        isInCar(agent, car)
    Effects:
        Delete list
            isAtPosition(agent, startPosition)
            isAtPosition(car, startPosition)
        Add List:
            isAtPosition(agent, destinationPosition)
            isAtPosition(car, destinationPosition)

Operator : GetOutOfCar(agent, position)
    Preconditions:
        isAtPosition(agent, position)
        isAtPosition(car, position)
        isInCar(agent, car)
    Effects:
        Delete List:
            isInCar(agent, car)
```

Additionally, there are *methods* used to describe how a compound task can be decomposed. There may be multiple methods able to decompose the same task providing different ways of solving a task. Therefore, the methods – as well as operators – have preconditions describing under which circumstances they are applicable. Using the previous example where an agent has to pick up an object, the uppermost compound task shown in Figure 1 is *PickUpAtPosition*. This task can be decomposed by one of the following methods: *PickUp*, *MoveAndPickUp* and *TeleportAndPickUp*. Usually, a method is described in the following way:

```
Method : MoveAndPickUp
    Task: PickUpAtPosition(agent, agentPosition,
                                      object, objectPosition)
    Preconditions:
        isAtPosition(agent, objectPosition)
    Subtasks:
        ⟨MoveTo(agent, agentPosition, object, objectPosition),
         PickUp(agent, object, objectPosition) ⟩
```

We sequentially check the preconditions of these methods until a valid method is found by which the task can be further *decomposed*. This process continues until all compound tasks are decomposed and the plan contains only primitive tasks. For example, assuming that the *agent* is currently at position *home* and the *object* is at position *park*, the precondition *isAtPosition(agent, park)* of the method *PickUp* fails. The precondition *isNotAtPosition(agent, park)* of the next method *MoveAndPickUp* holds, so that this method is used to further decompose the task. The remaining methods (in this case *TeleportAndPickUp*) are not checked anymore, unless we fail to completely decompose the entire compound task with the initially selected method. This way, the search space is recursively pruned.

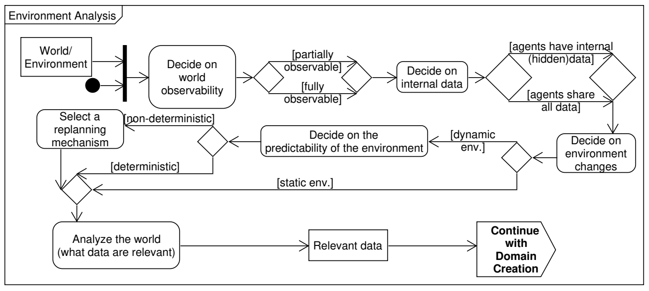
Environment Analysis
World/ Environment
Decide on world observability
[partially observable]
[fully observable]
Decide on internal data
[agents have internal (hidden)data]
[agents share all data]
Decide on environment changes
[dynamic env.]
Decide on the predictability of the environment
[non-deterministic]
Select a replanning mechanism
[deterministic]
[static env.]
Analyze the world (what data are relevant)
Relevant data
Continue with Domain Creation

Fig. 2. Environment Analysis: aspects to consider about the environment.

Note that, theoretically, the methods can be checked in any order. However, the most known approach to HTN is the Simple Hierarchical Ordered Planner (SHOP) [16], variations of which were used in most of the games mentioned in this work. Here, the methods are usually *checked* in the order that they are *presented* into the system. Thus, this order plays an important role. If we were to change the order of the methods presented in Figure 1 and put *TeleportAndPickUp* before *MoveAndPickUp*, its precondition would be checked first and it would hold. For this reason, the *TeleportAndPickUp* would be used to decompose the task and the final plan would be different.

Furthermore, the tasks are decomposed in *total order* in SHOP. This means that the final plan consists of operators which are *listed* in the same order, in which they will be *executed* (as opposed to partial-order decomposition, both of which are described in more detail in section VIII-F). In our example, the plan consists of the following operators: *GetIntoCar(agent, home), Drive(agent, home, park), GetOutOfCar(agent, park), PickUp(agent, object, park)*. With total order, these operators will be executed by the agent in that very same order.

Every time that the planner reaches a primitive task, it applies the effects of the task's operator to its inner world state representation by adding and deleting facts in a similar way as in STRIPS. As shown in Figure 1, after the operator *GetIntoCar* is applied, the planner's world state changes from *S0* to *S1*, so that the preconditions of the *Drive*-operator are checked in this new state. If the planner fails to continue planning using a method, effects that were already applied in this method are reversed and the planner backtracks to the previous compound task. For example, if after applying the effects of the operator *GetIntoCar* the operator *Drive* could not be applied for some reason, the method *MoveByCar* would fail and the effects of *GetIntoCar* would be reversed before checking further methods of the compound task *MoveTo*. Since both methods of this compound task failed, the method *MoveAndPickUp* which led to this task would also fail and the planner would try to decompose the initial task *PickUpAtPosition* with the *TeleportAndPickUp*-method. Another detailed description of the usage of an HTN in a game environment is provided in [17].

III. Environment Analysis

Every video game represents a unique environment. In this section, we describe how the environment is analyzed in order to create and execute plans. This process is summarized in Figure 2.

A. Observability

The observability of the environment is one of the first aspects to take into account. In contrast to most real-world applications, it is often possible for a game agent to have full observability of the game environment. Nevertheless, many modern games try to limit it in order to reflect a more human-like agent behavior. For example, a human player is not able too see through walls and so neither should an agent.

B. Agent's Internal Data

If the game does not grant full observability to the agents, it is important to decide whether they can share environmental information. Information sharing has to be done in a way that makes sense (and its communicated accordingly) to the player. Some games, such as *F.E.A.R.* [18], used verbal communication between agents, which the player could also hear. Due to these dialogues, some players of *F.E.A.R.* were even thinking that the agents were following each others voice commands [19].

Independently from whether or not the game world is fully observable, some data about it *has* to be saved for the planner to have a symbolic representation of the world. These data can be abstracted to some extent, as long as it is sufficient for the planner to make decisions and simulate changes in the world. Saving information in a game is often done with the help of *blackboards* [20], [21] which can be accessed by one or multiple agents. That way it is possible, for instance, to let commanders have different knowledge than soldiers, as described in [22]. The content of a blackboard can either be dynamic or static meaning that the types and the amount of information saved in a blackboard either might change at runtime or remain fixed throughout the game [23]. The squad-based military simulation *SquadSmart*, which implemented an HTN planner, contained a single blackboard that included individual agent's and global squad's knowledge [24]. In contrast, in *F.E.A.R.* each agent had his own blackboard that was updated with only environmental facts it perceived personally [18].

C. Environment Dynamics

Besides taking into consideration the observability of the environment, it is also important to think about its dynamics. In commercial games AI agents usually act in a dynamic environment that is changed by multiple factors. Some of these factors are deterministic (for example the change of the time of the day), but some of them are non-deterministic and cannot be modeled/precomputed in a reliable way (for example the player actions). To be able to handle potential plan failures that can occur due to uncertainties, it is very important to consider re-planning strategies. More detailed descriptions of re-planning strategies can be found in sections IX-C and IX-D.

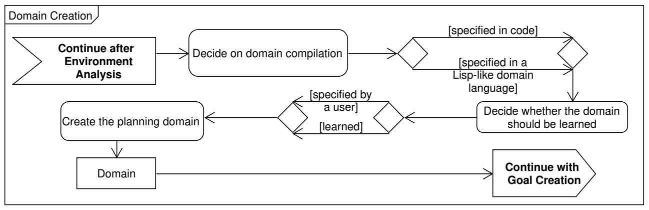
Domain Creation
Continue after Environment Analysis
Decide on domain compilation
[specified in code]
[specified in a Lisp-like domain language]
Decide whether the domain should be learned
[specified by a user]
[learned]
Create the planning domain
Domain
Continue with Goal Creation

Fig. 3. Domain Creation: aspects to consider about the planning domain.

D. Data Relevance

Once the dynamics of the environment are clear, it is important to analyze which data are relevant for the planner. As Jeff Orkin suggests: *start with the minimal possible description of a [world] state, and only make additions carefully while monitoring performance* [18].

IV. Domain Creation

Having an overview of the environment and its relevant data, the *planning domain* can be created. A planning domain can be regarded as a library where knowledge about a domain (the game world) is represented using planner components described in section II (operators with preconditions and effects, facts about world states and methods in case of a domain for an HTN planner). Therefore, developers should think about *how* to create a planning domain in terms of readability and extensibility. This process of analyzing important aspects of domain creation is shown in Figure 3. Some possible approaches are described in the following section.

A. Domain compilation

The planning domain contains important information about the environment, the agent's actions, their preconditions and the effects that they have on the environment which need to be represented as data structures and code functions, so they could be used by a planner algorithm. One possible way to do so is by defining the domain directly in the programming language that is used by the planner. Another possibility is the usage of an intermediary descriptive language that can be compiled into code for the desired programming language.

Since planning domains are often created by programmers, it is natural that they are defined in the code directly, as it was done for *Transformers: Fall of Cybertron* [13]. Also for *Dying Light*, all methods and activities of the HTN planner were described by programmers directly in C++, whereas the game designers only tweaked a few exposed parameters to balance the game. These parameters were saved in a separate data set for every agent [14]. Defining the domain directly in code saves the effort of developing a domain compiler and its related user interface. The disadvantage of this approach is its lack of flexibility and the slower iteration times when compared to a system where designers could directly modify the domain.

An alternative way to define the domain is to use a high-level language like Lisp [25] and compile the domain automatically from this definition into code. The simulation

SquadSmart [24] and the HTN planner that won the *Capture The Flag AI Competition* at AIGameDev.com [26], show that this approach can be used successfully for the domain compilation of HTN planners.

Although this technology might be very useful especially for games with complex behaviors and large game worlds, it is not widely used in the industry. Its major drawback is the high difficulty level that is presented by the task of building a planner compiler that is able to get custom domain definitions as input [27].

B. Learning the Domain

An interesting approach to create a planning domain from a researcher's point of view is learning the domain instead of being specified by the developer. Some learning methods are applied to planning domains in the academia (for example [28], [29], [30]). However, these approaches are tested in very simple and mostly deterministic environments. Even though the environments of commercial video games are much more complex, learning a domain for these environments might be an interesting challenge not only for developers but also for researchers. A possible learning framework might be provided by *Learning Classifier Systems* [31] which are already used to learn behavior rules in games. Resulting plans would be most likely very different from those defined by a developer.

The disadvantage of this approach might be that the developers would not know whether such a domain would produce the desired agent behaviors in all cases. Furthermore, the description of a learned domain might be more difficult to understand, debug and change for developers as it would not necessarily correspond to the developer's intended logic. Another important aspect in this case are the presumably long learning times that would be required after finishing all parts of the game that a learning mechanism should take into consideration in order to learn a planning domain. Since time is usually a very limited resource in the process of game development, thorough considerations have to be made before deciding to adopt a learning process as part of building an AI for an offline game. However, there is great potential in learning planning domains for agents in online games. Such domains could be learned using the available online telemetry – for example re-play data of human players – and adapted continually later on.

In addition to the planning domain, it is possible to learn additional components of the planning algorithm even in very complex environments. An interesting solution was applied for the game *Tomb Raider*. Here, the domain for its GOAP was defined manually but the action costs that were used as heuristics for the planner were learned and adapted during game play [10] (for more details see section VIII-G).

V. Goal Creation

Knowing the environment and the way in which to define the planning domain, the next major topic to think about is the definition of goals. No matter what kind of a planner is being used in the end, plans are always created to achieve some goals or bring the game world into a certain goal state. There are several ways to define and select goals. In the following section we describe some of the main aspects to consider at this point and show the decision process in Figure 4.

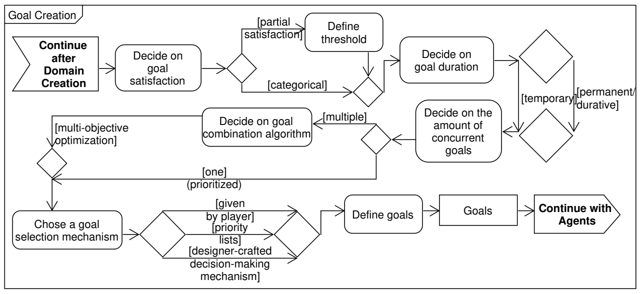
Goal Creation
Continue after Domain Creation
Decide on goal satisfaction
[partial satisfaction]
Define threshold
[categorical]
Decide on goal duration
[temporary]
[permanent/ durative]
Decide on the amount of concurrent goals
[multiple]
Decide on goal combination algorithm
[multi-objective optimization]
[one] (prioritized)
Chose a goal selection mechanism
[given by player]
[priority lists]
[designer-crafted decision-making mechanism]
Define goals
Goals
Continue with Agents

Fig. 4. Goal Creation: aspects to consider about the agents' goals.

A. Goal Satisfaction

Some of the goals that an agent should achieve, can be defined as categorical goals such as "*move to point A.*" As long as the agent is not at point A, he should create and follow a plan that would bring him there. However, sometimes it is sufficient if a goal is satisfied to a certain degree. For example, an agent would need to *collect a sufficient amount of health potions.*

There are several ways how to handle partial satisfaction of goals. Developers could define some thresholds to say when a certain goal is *sufficiently* satisfied and a new goal can be selected. Usually, these values would be defined and tweaked by game designers. To bring some variety into agent's behavior these thresholds might be defined in a fuzzy way.

B. Goal Duration

In our previous example the temporary goal "*move to point A*" is satisfied as soon as the agent reaches point A. However, often, it is desired that the agent "*follows the player,*" "*stays in his view*" or simply "*stays alive as long as possible.*" These goals cannot be declared as *satisfied* at any certain point of time. They are *durative* or *permanent* and can only be aborted in favor of other goals. When creating a planning domain and deciding on the planner type, it is important to think about how to handle such goals as well as how to handle persistent actions (see section VII-B). This decision does not only have an impact on the goal selection mechanism, which is described in section V-D, but also on the frequency of plan re-evaluation and re-planning (section IX-C).

C. Number of Concurrent Goals

Another important factor is the number of goals that can be pursued by an agent at the same time. Often it is important to follow multiple goals simultaneously, for example while an agent should "*move to point A*" he should also "*stay in cover.*" Sometimes these goals can even be competing. For example it could be problematic for an agent A to "*stay outside the view* *of agent B*" and "*stay inside the player's view*" if the player is close to agent B.

Although there are several academic approaches for multi-objective optimization, many of which are described in [32], [33], accepting multiple goals in practice leads to additional challenges because, in contrast to academic problems, a game agent's goals usually cannot be mathematically described by objective functions. Thus, at this point, developers need to decide *how* to represent combined goals in a planning domain and *how* to create plans for them. Often objectives/goals cannot be represented by a measurable value. Moreover, not every planner type allows to combine goals or create new ones. For example, in a Hierarchical Task Network, goals are represented by high-level tasks. Those tasks are then decomposed into lower-level tasks which together create a plan. Thus, plans can only be created for those goals/tasks that have a pre-defined decomposition, meaning that no *combined* goals can be decomposed.

A possible solution for this problem is the simultaneous use of multiple domains. As described in [13], characters of *Transformers: the Fall of Cybertron* used different planning domains for their upper and their lower body. That way, the lower body handled navigation-related actions and the upper body could play additional animations, for example the character could shoot at a target while moving towards a target position. However, this solution carries many risks, since not all action combinations for the different body parts might make sense and would require some workarounds. Furthermore, it is difficult to ensure that both planners/domains are synchronized [17].

Otherwise, a game agent might be restricted to follow only one single goal at a time, like it was done in *F.E.A.R.* The relevance of the current goal was re-evaluated whenever a significant change in the environment happened and the planner selected the next goal [19]. Some care has to be taken at this point for those cases when two successive goals could lead to unnatural character behavior, such as rapid oscillations. One potential solution to this problem is giving agents some *hysteresis buffer* in the form of a log of past goals. This information could be used to bias the planner to stick to certain goals, assuming the oscillations are triggered because multiple goals are equally relevant.

D. Goal Selection

Knowing how many and what kind of goals the game agents should be following, it is important to create a system that is responsible for goal evaluation and selection. If game agents are supporting the player, it might be possible to allow the player to assign them certain tasks that could be passed to the planner in order to create appropriate behaviors.

In many cases however, agents do not get orders from human players. Instead, they should be able to select their goals autonomously taking multiple facts into account. A simple way to create a goal selection mechanism is to create a pool of possible goals such as *patrol an area, kill an enemy, collect ammo* and select the most relevant goal through prioritization. For example, the default goal might be to patrol, but when the agent sees an enemy he should try to kill him, unless he lacks ammo, which he should then try to collect first. Priorities could be either read out from a prioritized list of goals or a more complex decision-making mechanism that could be defined by designers. In *Tomb Raider*, a designer-authored graph was used for goal selection that took into account the agents' motives, action availability and costs [10].

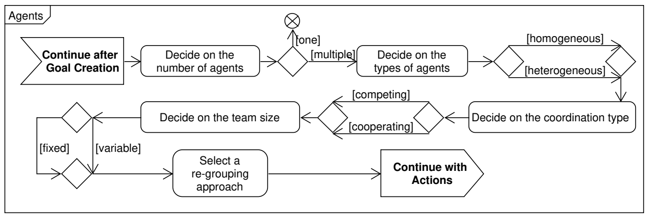
Agents
Continue after Goal Creation
Decide on the number of agents
[one]
[multiple]
Decide on the types of agents
[homogeneous]
[heterogeneous]
Decide on the coordination type
[competing]
[cooperating]
Decide on the team size
[fixed]
[variable]
Select a re-grouping approach
Continue with Actions

Fig. 5. Agents: aspects to consider about plan-executing agents.

Similarly to actions (see section VIII-G), goals might get costs or weights and might be selected using a heuristic. Also *situational* goals might be defined, so that the selection mechanism would only take those goals into account which are applicable in the current situation. For example, in *Shadow of Mordor*, agents were assigned alertness levels like *alert, suspicious, ambient* and goals were sorted and selected based on these [10]. A similar approach was implemented for the game *DEMIGOD* through so-called *master goals* [34]. These represented the upper levels of a goal hierarchy and depending on their current weights, only certain sub-goals on the lower levels of the goal hierarchy could be selected.

Having a squad-like hierarchy of agents might also allow *orders* to be sent from higher hierarchy levels to the lower ones. With this approach, different goal selection mechanisms might be implemented at different levels of the hierarchy whereby more abstract goals might be selected at higher levels and passed as orders to lower levels. For example in the game *F.E.A.R.*, a global squad coordination system created squads dynamically and passed orders to the squad members. These orders were regarded as goals by the agents. Every agent then re-evaluated and prioritized his current goals deciding whether to follow the orders or his own goals [7].

VI. Agents

One of the most important aspects to consider for plan creation are the agents that should execute the plans. Depending on the number of agents, their abilities and whether or not their actions should be coordinated, the basic structure of the planner can vary a lot. Some major questions regarding game agents are shown in Figure 5 and described in the following section.

A. Number of agents

There are multiple games that implement an AI for a single *buddy character* that supports the player. However, especially because of their ability to provide better tactics and coordination, planners are predominantly used in multi-agent games. This approach also introduces other problems however, which is why in the next sections, we concentrate on the issues that are important for *multi-agent* planning.

When deciding on what planner to use in a game, it is important to take into account not only the number of game agents in general, but especially the number of concurrently active agents that could use the planner at the same time. For *Transformers: Fall of Cybertron*, there were up to 17 agents using the HTN planner concurrently [13]. As Eric Jacopin describes in his GDC[13] talk, the GOAP planners in the games *F.E.A.R.* and *Rise of Tomb Raider* were used by a relatively low number of agents. In both games, on average 3 agents (maximum 15 for *F.E.A.R.* and 12 for *Tomb Raider*) were active simultaneously [10], [35]. For *Killzone 3*, the average number was 5 and the maximum number 10 [35]. In contrast to these games, in some scenes of *Shadow of Mordor* the player could see up to 400 active agents. However, creating new plans for all these agents at the same time would be too costly and still remains an unsolved problem in multi-agent planning. In order to avoid this problem in the game, an upper limit of 50 NPCs that could use the planner concurrently was introduced [10]. Though, this is not an optimal solution and developers have to take into consideration that NPCs still need to do *something* while waiting for a plan, or otherwise the scene will look unnatural.

B. Types of Agents

Creating a planning domain for a game where all agents are of the same type and have the same abilities, thus are homogeneous, is already a difficult task. In most video games however, the agents are heterogeneous. Thus, they represent a variety of different agent types, be it members of a combat group that are on different levels of the command hierarchy, zombie types with different degrees of decay or even agents of the same type that have different abilities depending on their equipment. When creating a planning domain, an important decision is how to represent in the planner the different types of agents and their abilities. Typical approaches involve creating a separate domain (or domain parts) for every agent type and ensure that the planner uses the appropriate domain (or domain part) for each of them. This approach is well applicable when the agent type does not change over time, so that there is no need to check for the type in every planning step. This was done for example in *Transformers: Fall of Cybertron* where agents of different types used different task domains [13] and *F.E.A.R.* which used different action sets to represent heterogeneous agents, so that they could accomplish the same tasks in different ways [18]. Alternatively, it is also possible to use only *one* domain and use the agent type as a precondition for a task/action in order to assign actions to the right agent types. This approach is more suitable if the agent type can change over time depending on the agent's current role (in a team) or his equipment.

C. Coordination Type

Having multiple agents in a game usually means that some kind of coordination between them is desired. Their behaviors should be either cooperative or competitive. Often, game agents are the player's enemies and require coordination of their actions *against* the player. They belong to one group of agents. In some games however, the player might get some support from other Non-Player-Characters (NPCs), which should behave cooperatively *towards* the player and each other.

An engineer working on a planner needs to answer the question *how* to achieve these types of coordination. An HTN planner for example, allows for cooperation between multiple agents by definition. It is possible to create task decompositions that require several agents to perform different actions simultaneously. For example the task *TransportGroup* would require one agent to *TakeDriversSeat* and multiple agents to *TakeSeat*. Then, the driver would *DriveToDestination* and every agent would have to *GetOff* of the vehicle at the destination.

However, this kind of coordination is hard to achieve with a STRIPS-like planner *only* since it creates a plan as a sequence of actions and does not deal with concurrent actions and their joint effects. So, some additional coordination mechanism is required to take care of the group behavior. For that purpose, some academic approaches modify the STRIPS-representation of actions allowing for concurrency [36]. However, these approaches require the usage of non-linear planning algorithms instead of the original STRIPS to be able to plan with concurrent actions. Instead, games that use STRIPS-like planners usually implement an additional upper-level coordinator that assigns certain goals to different agents, which then use the planner to achieve these goals. This approach has shown good results in *F.E.A.R.* [18] as well as in *Dirty Harry* [37] which used versions of the GOAP planner. Moreover, according to Jeff Orkin, the combination of the central squad coordinator and the GOAP planner in *F.E.A.R.* led to interesting emergent behaviors. For example, if the squad coordinator ordered two agents to move into cover positions close to the player and there were obstacles (walls) between the agents and the player, the agents would move to the cover positions from different sides. To the player, it appeared as if the agents were executing a coordinated pincer attack, even though they were just moving to closer positions simultaneously [7].

Additional requirements and problems arise when agents should compete with each other instead of cooperating. This can be the case for example in RTS (Real Time Strategy) games where multiple factions compete against each others' and the player's faction. Other examples are sports and race games where multiple factions compete trying to achieve the same goal. In these cases, every faction should take into consideration the possible actions of other factions and try to maximize its own reward or to come closer to the goal preventing its competitors from doing so. Thus, the decisions of every faction depend strongly on the actions of other factions, which results in even more uncertainty being added to the planning problem. For that reason, it is important to think about whether or not a planner should be used in such cases, or whether a reactive approach would be more appropriate to use. Planning with a high level of uncertainty would require some kind of reasoning approach about the competitors' actions in

[13]Game Developers Conference: www.gdconf.com

order to be able to counter them.

D. Varying Team Size

Another important aspect when planning for groups of agents, is the size and structure of these groups and the question of how to handle structural changes. NPCs in a game might be killed, more of them could spawn or they could get more help from nearby comrades. Developers should think about how to take care of agents' deaths and respawns not only in terms of game design but also in terms of re-planning and group re-organization.

When implementing the coordination mechanism – no matter whether it is the planner itself or some upper-level squad manager – these changes should be taken into account. If a planning domain is designed for a certain number of agents in a group, the planner won't be able to find a plan under different circumstances. Even in a football game where the team size is expected to be 11, a player might get a red card and be disqualified. It is important to allow the planner to handle different sizes and types of groups.

Furthermore, imagine the following situation: group A is split up spatially in two rooms. Three members of group A meet members of group B in the room they enter. Both groups are in the same faction and could cooperate against their common enemy. Since they are already in the same room, it might make more sense to re-organize the groups and put the three members of group A with those agents from group B into one combined group. For that purpose, the coordination mechanism should be able to recognize the spacial change and re-assign the agents.

Exchanging the group members dynamically might also mean that the *types* of agents in a group could change. Thus, plans should also change accordingly to the new agents' skills. Such a case would also be in games where the player can pick NPCs as supporters/members of his team. Also, here the planner should provide plans for any number and combination of agent types.

A good example solution for these problems is the dynamic squad manager of *Killzone 2* [12]. Being a team-based game, its bot-squads were re-organized based on the agents' objectives or their distances to the different squad centers. If necessary, new squads could be created and removed at runtime with all free agents being distributed into sensible squads. For that purpose, the types of the agents to select for a squad were defined in a global policy.

VII. Actions

Another important factor to think about when creating a planner are the actions that can be performed by the agents. The types of the actions strongly depend on the planning domain and the planning problem. In games, a lot of actions can be represented by simple animations, whereas other actions are more complicated and cause more uncertainty. In the following section, we describe some aspects of actions that need to be considered when creating a planner and show them in Figure 6.

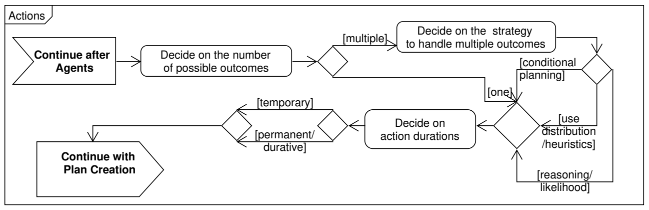
Actions
Continue after Agents
Decide on the number of possible outcomes
[multiple]
Decide on the strategy to handle multiple outcomes
[conditional planning]
[one]
[use distribution /heuristics]
[reasoning/ likelihood]
Decide on action durations
[temporary]
[permanent/ durative]
Continue with Plan Creation

Fig. 6. Actions: aspects to consider about the agents' actions.

A. Determinism of Actions

Planning in a non-deterministic environment is a difficult task. It gets even more difficult when more non-determinism comes from stochastic actions. In a video game domain, some actions might have multiple outcomes instead of a clear one. This might be caused by some predefined randomness of the action itself or some external sources. For example the agent could throw a grenade and it could either hit or miss a moving target.

In classical planning, there are different possibilities for handling multiple action outcomes in the planning process. Most of them try to predict the actual outcome in some way and prevent a possible plan failure. One way is to use a predefined distribution of the probabilities of the different outcomes and plan only for the most probable outcome. Also, there are some ways to use more complex likelihood computations applying some reasoning approach.

For example, Cased Based Reasoning (CBR) could be applied to reason about future action outcomes from past experiences. Reasoning techniques like CBR can also be applied for things such as selecting strategies for Real Time Strategy (RTS) games as described in [38] and [39]. However, in order for CBR to deliver more reliable predictions, a sufficiently large case library would be required.

Another way of reasoning are Bayesian Networks. Using this approach, developers would need to represent the conditional (in-)dependencies between all corresponding world state variables, so that knowing the state of the world before applying an action could provide the likely outcome of it. Further details regarding the possible use of Bayesian Networks in a game environment can be found in [40]. Creating the right network, however, is a very complex task and requires thorough considerations to be made in advance. This and the added debugging complexity, are reasons why Bayesian Networks do not find much support in commercial games.

Another possibility to handle multiple action outcomes which comes from academia is conditional planning [41]. Here the planner does not decide which outcome is the most likely one, but instead creates plans for all possible outcomes. Thus, every branch gets a condition denoted under which it might be executed. When a plan fails, plan recovery may be used to take another plan branch. The obvious disadvantage of this approach is that the branching factor might grow very fast if multiple actions with multiple outcomes would have to be executed in one plan. Due to its complexity and lack of deterministic planning execution time, most planner-based commercial games do not support multiple action outcomes and prefer to build a new plan whenever the selected one fails. In this case, when *executing* the corresponding action, the agent should check whether its actual effects match the outcome that the planner *planned* for. Thus, using effects and preconditions (see VIII-B) in an accurate way might help detecting plan failures and trigger re-planning. As we describe later in section IX-C, re-planning is done very often in the games mentioned in this work. Due to short planning times and frequent plan re-evaluation this approach works well for games like *Killzone 2* or *Transformers: Fall of Cybertron*.

B. Action Durations

Most actions of a video game agent have a clear end point, at which they can be regarded as completed. Many of these actions can be represented by an animation, such as the action *PickUp(agent, object, position)*. When the *PickUp*-animation ends, the agent may continue with the next action from his plan. For example, the majority of the agents' actions in the game *Dying Light* were animations [14].

However, there are also many cases where an agent should perform a certain task for some time until this task is aborted or the agent gets a new goal, as already mentioned in section V-B. In general, such tasks never complete on their own. A good example of such a task is *patrolling*, where an agent should walk around until he sees something suspicious or his watch ends. So, an additional aspect that needs to be taken into consideration when designing the plan creation and execution processes is the handling of such persistent actions.

An important question to be answered about the plan creation process, is how to avoid creating the same plan over and over again while executing a persistent action and at the same time to allow re-planning in general. The game *Tomb Raider* for example, used a GOAP planner and heuristics to select plan candidates [42],[10]. Hereby, persistent actions got some kind of *remaining costs*. Implementing an accurate maintenance of these costs prevented the planner from creating the same plan over and over and re-planning could still take place. At this point, the re-planning strategy (see section IX-C) plays an important role. Also, the planner domain should not be designed in such a way that persistent actions might be in the middle of a plan. Since a plan task is only executed when its predecessor completes, any task following a persistent task would never be executed. Only the creation of a new plan (which is possibly triggered by an event) might provide the agent with new tasks.

Another important aspect is the handling of durative actions and the possibility of planning in time. Imagine, there are multiple agents that are supposed to perform a group task with every agent getting his own sequence of actions to execute. For example, a group of agents has to collect some objects in a level and bring them back to a specific place. As preconditions of the actions, the planner considers only the objects' and the agents' positions. It assigns the objects to the closest agents and thus plans only in space. However, some agents might be faster than others and should get more objects assigned than the slower agents (even if they might be closer to some objects). If the planner does not consider the time, it might provide a plan that actually takes longer, because the slower agents would still be collecting their objects while the faster ones would run out of tasks.

There are multiple approaches that might be applied to prevent this problem. One possibility to actually plan in time is adding time into costs computations while planning. This approach requires some heuristics to be used to select plan candidates which could contain the durations of tasks. The usage of heuristics in combination with STRIPS-like planners in general is possible, however, it is a more complicated task when using HTN, as described in section VIII-G. In most cases, the duration of a higher-level task of an HTN is regarded as the sum of its lower-level tasks' durations. So, in order to compare the high-level tasks' costs it would be necessary to decompose all of them and compute the costs of all of their children, which would increase the computational costs.

Furthermore, using time as a heuristic – be it in combination with an HTN or a STRIPS-like planner – is not that easy because it is not always possible to *know* the exact duration of every task. So, in many cases *approximations* would be needed. For example, the duration of a *GoTo* task could be approximated through the agent's speed and the path length, whereas the duration of a *PickUp* task could be set to the length of the corresponding animation. For some actions, however, approximating their duration might even be impossible, so that some work-arounds would be needed at this point.

An interesting approach from the academia is the combination of HTN with STN (Simple Temporal Networks)[43]. This approach uses HTN with temporal planning, allowing to define temporal preconditions such as *task A has to begin/finish before/during/after task B*. Such an approach could possibly solve the problem described above, where the faster agents would finish *before* the slower ones and could get more tasks to perform. Although this approach is in an early research stage and was, to our knowledge, not applied in any video game, it might be an interesting solution to investigate.

An alternative solution to using temporal planning in games is – as we call it – the *re-plan-and-execute* approach. To avoid all the complications described above, many games re-plan periodically and execute the plan steps immediately. The emphasis hereby lies on the re-planning that is performed very often. That way, the planner always has the most recent information and can change the plans if needed. For example *Killzone 2* re-planned five times per second [44] as described in more detail in section IX-C. Solving the problem with planning in time might be even the main reason to re-plan periodically.

For the example described above, using this solution, there would be no need to create the full plan assigning all the objects to the right agents in advance. Instead, it would be enough to assign the closest objects to all agents first. Then, all agents would start to execute the first task and thus would be occupied for some time. Since the plan would re-plan on a regular basis, at the point when one of the faster agents would finish his tasks, the planner would assign him to the next object, as he would be the available one, preferring him to the slower (occupied) agents.

VIII. Plan Creation Process

When it is clear how all the components of the planner are defined, it is time to decide on some final aspects concerning the plan creation process. In this section, we describe aspects to take into consideration in order to acquire knowledge about the world state, to evaluate world states and to propagate changes during the planning. Additionally, we discuss some optimization strategies for the planner to work more efficiently. These aspects and processes are shown in Figure 7.

A. Precondition Evaluation

Since most planners use *preconditions* for actions, it is important to think about not only how to define them, but also how to evaluate them. Depending on the definition of the preconditions, the evaluation complexity might vary a lot. The expressiveness of the preconditions might play an important role as well during the domain creation as during action evaluation.

One possibility for defining and evaluating preconditions is to follow the example of the original STRIPS (see II-A) and SHOP (see II-B) describing the preconditions as first-order predicates and using a theorem prover. According to Alex Champardard, Guerrilla Games applied this method for the game *Killzone* [44]. Also, the game *Final Fantasy XV*[14] used this approach to represent preconditions of rules for coordination of multiple ambient NPCs [45], [46]. Using this approach, it would be possible to express complex dependencies in the world. However, it would either require additional effort to build a theorem prover or an existing library/middleware program would be needed.

To achieve a more efficient precondition evaluation mechanism, developers might also think about representing the preconditions by data structures that are used in the game itself or by structures more suitable for direct comparison instead of using logical predicates [47]. For example, the world state might be represented as a bit array and fast bit-wise comparisons between the desired world state and the current one might be executed. As already mentioned in section II-A, this approach was used in *F.E.A.R.*, where preconditions, effects and the planner's world state were represented by fixed-size arrays. Also, the game *Transformers: Fall of Cybertron* used simple preconditions in the form of conjunctions of boolean values [13]. This approach might be more efficient and thus more appropriate for video games, according to Dana Nau [44].

Although using only boolean values is less expressive it might significantly decrease the flexibility of the planner. So, in order to be able to perform more complex checks, it might be useful to call external evaluation functions that might be defined in some subsystems instead of the planner itself. The game *F.E.A.R.* applied this method additionally to the fixed-size array representation of preconditions in their implementation of GOAP [7]. Here, they introduced procedural preconditions which required external functions to perform precondition checks such as, for example, ray-casting or path evaluation.

[14]Final Fantasy XV: Developer: Square Enix, Publisher: Square Enix. 2016

Additionally, to the evaluation strategy, developers might think about how to save the binded values of preconditions which were found to be true. These values, assigned to condition variables could be used later during plan execution for plan re-evaluation. Knowing which values were assigned originally, it could be simple to check whether or not the current world state still fits these values and whether the plan can be continued.

B. Effects and World State Propagation

In addition to preconditions, most planners implement *effects*. An effect of an action/task is defined in the planning domain and specifies how this action changes the state of the world. Usually, these effects are denoted in add and delete lists (as described in section II-A). In general, effects are used to have a planner-internal simulation of the actions of the plan. They allow the planner to see how the world would change applying the plan and to create further plan steps by checking the preconditions in the changed world state. Thus, with their usage it is often possible to create more precise plans and to plan further ahead in time. The game *Transformers: Fall of Cybertron* used such a simulation approach in its version of the HTN planner [13]. The effects of its primitive tasks could change the world state and these changes were propagated during the planning process.

Nonetheless, because of the very dynamic and non-deterministic nature of video games, it is very difficult to predict whether the world state during actual plan *execution* will be the same as it was predicted during plan *creation*. It is possible that when an action will be performed, its effects won't be applied because of some outer disturbances. In that case, some sort of re-planning is required (see section IX-D).

For this reason, developers might decide to save the costs of effect propagation. If no propagation were applied, the plan would be created on the current world state which would mean that it would not be possible to validate future actions correctly. Thus, only short plans could be created without being able to chain multiple actions. More specifically, using a STRIPS-like planner, it would mean that only plans of one action could be created since such planners rely on the preconditions and effects of single actions. On the one hand, that would mean that more complex goals that would require longer plans could not be accomplished, basically removing the planner's ability to reason into the future. On the other hand, it would speed up the planner a little because of the smaller plan size. Furthermore, it might be acceptable if re-planning is performed often enough anyway or if not many actions actually do change the world state (such as just performing a simple animation). However, before applying this approach, developers should take into consideration which actions do have effects on the world and then determine the trade-offs between the agents accomplishing complex goals and the possible performance improvement of the planning algorithm. For example, there was no world state propagation in the game *Dying Light* because its plans were re-evaluated periodically and only two of its goals changed the world state [14]. Also, according to Alex Champandard, both *Killzone 2* and *3* did

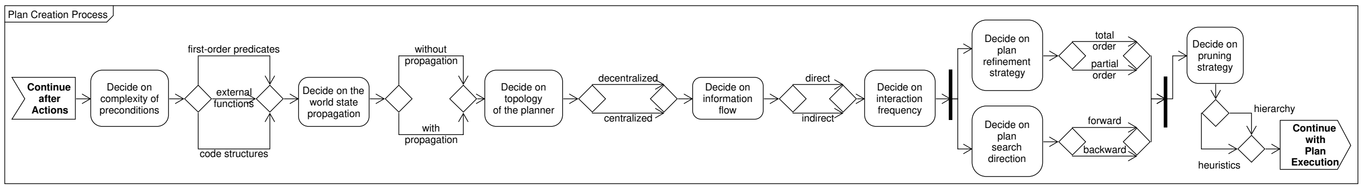
Plan Creation Process
Continue after Actions
Decide on complexity of preconditions
first-order predicates
external functions
code structures
Decide on the world state propagation
without propagation
with propagation
Decide on topology of the planner
decentralized
centralized
Decide on information flow
direct
indirect
Decide on interaction frequency
Decide on plan refinement strategy
total order
partial order
Decide on plan search direction
forward
backward
Decide on pruning strategy
hierarchy
heuristics
Continue with Plan Execution

Fig. 7. Plan Creation Process: aspects to consider about the planning process itself.

not propagate the world state changes in their versions of the HTN planner [14].

Alternatively, it is possible to define reversible effects as described in [44]. This way, instead of copying the complete world state for every plan step and to denote changes in it, one copy of the world state would be enough. Effects could be applied to this world state and if needed, for example in case of a plan failure, they could be reversed.

C. Topology of the Planner

When implementing a multi-agent game, engineers should decide on the topology of the planner. If plans should be created for only one agent or no coordination between the agents is required, a distinct decentralized planner can be implemented for every agent. In this case, every agent would have his own knowledge about the other agents that he would gain, for example, through perception.

However, a decentralized planner can still be used even if coordination between the agents is desired. In this case, every agent would get his own planner and some kind of information flow would be required to signal to other agents the state of an agent's current plan in order to achieve a coordinated behavior. Agents should especially be able to inform their cooperators about (potential) plan failures. Therefore, developers should think about how to implement communication between the agents in a proper way. Especially in the case of video games, performance plays a big role and should not suffer from the reliance on high information exchange rates.

One way to prevent this case is, for example, the implementation of local-only communication, where agents would only talk to agents that are inside a limited radius. Another solution could be a hierarchical communication system where, for example, soldiers could only forward their knowledge to commanders and commanders would pass orders to soldiers, limiting the communication to 2 hierarchy levels as described in [22].

Alternatively, a single coordination system might be implemented on top of multiple decentralized planners. This system could handle the communication between agents and assign goals to them and their planners would take care of their behaviors. That way, less effort would be needed to achieve interesting coordinated behavior than without a coordinator . As described in [48], a central squad coordination provides good synchronization and coordination between agents on a strategic level, but it might have troubles with handling individual needs of members. Therefore, an additional decision-making mechanism might be implemented for every agent in order to prioritize between individual needs and squad-level tasks and make decisions on a tactical level. Although a completely decentralized approach could theoretically be implemented to provide coordinated behavior between multiple game agents, a centralized approach might be much more efficient and much easier to implement. As already discussed in section VI-C, when using a centralized planner such as HTN, coordination can be directly defined in its planning domain. A big advance of a *centralized* coordinator/planner in a video game is the ability to debug the created plans and understand why some certain behaviors occur. In this case there is only one source of the agents' knowledge/orders in contrast to the decentralized case with local communication between agents where any piece of information might come from many sources. As Jeff Orkin describes in [18], it was easier to implement and debug a GOAP planner combined with a squad coordinator in *F.E.A.R.* than just having agents try to collaborate locally.

D. Information Flow

The way information between agents is organized is strongly dependent on the topology of the planner. Having a centralized planner or some central squad manager usually implies that this system is responsible for decision making and therefore needs to have access to all the information. Thus, there is no need to have any kind of direct information flow between the agents. It is sufficient to store and share the knowledge indirectly in some central memory (as described in section III-B).

It might be a different case when using a completely decentralized approach. If every agent had his own planner without any central coordination system, direct exchange of information could be a possible option. However, as already described in the previous section, communication should not unnecessarily affect the performance of the game, so that some approaches might be needed to limit things like the communication radius of the agents.

E. Interaction Frequency

Similar to the topology of the information flow, interaction frequency is another important aspect to take into consideration. In order to create more reliable plans, a planner should always have the most recent knowledge about the world state. Thus, it is important to find a balance between keeping the planner up-to-date and avoiding unnecessarily frequent information exchange.

Again, depending on the topology of the planner, the interaction might take place either between multiple agents and a central planner or between different agents (and their individual planners). In the first case, it is important to think about both when the agents should update the planner's knowledge and when exactly it will be needed/read out. Usually, information about the world state is *needed* whenever a plan is re-evaluated. Thus, this rate depends on the re-evaluation frequency which is discussed in more detail in section IX-C.

However, this does not imply that this information should be *updated* at the moment of re-evaluation. In some cases, it might make sense to update the world state in-between the plan evaluation queries (e.g. whenever sensors detected a change) and only to read out the information when needed, as was done in *F.E.A.R.*[8]. Such time-slicing might save some costly checks to be done on-demand.

In the case of decentralized planners, another important point in addition to the aspects mentioned above, is how often agents should exchange knowledge with each other. Developers should also keep in mind aspects like whether or not the interaction should be *bi-directional*, whether knowledge exchange in both directions should be *synchronized* and whether the knowledge should be spread between the agents in a certain *order*. These aspects may have an impact on how often a planner of a single agent should re-plan and thus, depending on the number of agents, on the required computational resources. For example a very inefficient way would be to re-plan *every time* after an agent gets new information from another agent. Getting first the information from *all* agents in a synchronized way and only then create a plan might be a more efficient approach. Additionally, depending on the direction and the order of interactions, some data might accidentally get lost and thus, an agent's knowledge about the world might be wrong and plans created upon it would fail.

F. Search Direction and Plan Refinement strategy

Using a planner means dealing with a search problem that consists of the planning domain, the initial world state and the goal which is represented either as a goal world state (for a STRIPS-like planner) or as a goal (compound) task (for an HTN planner). Depending on the planning technique, developers might distinguish the search approaches either by the search direction or by the decomposition order [49, Chapter 11]. Whenever using a search algorithm like for example A*, developers have the choice between forward search and backward search. In planning, forward search would start at the initial world state and select tasks whose preconditions are satisfied in that world state. This approach would create tasks in the order that they should be executed. However, due to the combinatorics problem, without applying any pruning strategies (see section VIII-G), the following two problems might arise: first, the search might become very exhaustive and second, it might not necessarily lead to the goal state [8].

Backward search can solve the second problem at this point, starting with the goal world state and searching for tasks whose effects could satisfy the goal. This approach is generally used by STRIPS-like planners. However, this approach has the disadvantage of potentially still being too exhaustive as well. Here, with every new task that is added to the plan, new sub goals might be added for the search algorithm to satisfy. At this point, however, it is easier to find pruning strategies that might help prevent the growth of the search space. Some optimization strategies for both search directions are described in the next section.

In some cases, like for example when using an HTN, it is more suitable to think about the decomposition order of tasks rather then the direction of the search. Most of the games that used an HTN, like *Killzone 2* or *Transformers: Fall of Cybertron* followed the SHOP approach and applied total-order forward decomposition [44], [13]. As already mentioned in section II-B, using total-order decomposition means that the tasks are decomposed in the order they are defined in and added to the plan in the same order, that they will be executed later. This is more intuitive and similar to human reasoning. Total-order HTNs combine the advantages of both forward search, taking only those actions into consideration that are applicable in the current state, and backward search, considering only actions that are relevant for the goal state [6, Chapter 11.3].

An alternative possibility, that is used less often in context of HTNs, is partial-order decomposition. In this case, the final plan may interleave subtasks from different tasks as described in [50]. With this approach, it is possible to define and consider the order of only some certain tasks without constraining the order for others. That way, the decomposition of more critical tasks can be preferred to others and potential plan failures in later steps of the plan can be recognized earlier. For example, if we have a method that decomposes into the following two compound tasks: *Re-group* and *MoveAsGroup*, it might make more sense to first check the more expensive/critical preconditions of the *MoveAsGroup*-task. If the path-finder fails to find a path for this task, we would save the time checking the preconditions and decomposing the Re-group-task and could use a different method instead. Of course, without having simulated the changes in the world caused by the Re-group-task, our assumptions for the MoveAsGroup-task would be less accurate. So, at this point, developers can decide whether they need more accurate plans or whether it is enough to have less accurate plans but a faster plan creation process. The partial-order decomposition approach was, for example, used by the mission generator *PlannedAssault* for the game *ARMA II* which tried to make high-level decisions first decomposing more critical tasks before going into detail with single branches [15].

G. Optimization Strategies

As already described in the previous section, any search algorithm used for a planner might have both advantages and disadvantages. In order to optimize the search, however, there are different strategies that developers might take into consideration. In general, the usage of action preconditions in a planner already leads to an optimized search in terms of the size of the search space. This way, when trying to find a sequence of actions that leads to the current goal, the planner takes into account only those actions for which the preconditions are true, skipping infeasible combinations. Thus, an accurate definition of preconditions plays an important role in the development process of a planner.

Additionally, developers might introduce further limitations on the search space which would exclude some actions or goals in certain situations. For example, in the game *Shadow of Mordor*, agents were assigned specific roles that instructed the planner to only create plans limited to the current role – for example an *investigator* would only investigate conspicuous objects – ignoring other actions during the search process [10].

In addition to the size of the search space, the search can be optimized in terms of the quality of the delivered plans. Alex Champardard distinguishes between two main approaches for search optimization [51]. One approach includes the usage of heuristics, the other one includes hierarchy.

A flat planner like STRIPS searches through all actions for a sequence that results in a feasible plan. Using preconditions and effects, the planner can identify which actions should follow one another. Since multiple actions might lead to similar effects that would match another actions' preconditions, multiple plans might be valid at the same time.

In order to select better plans from all the valid ones, actions could be evaluated for their costs as part of heuristics computations. Using a good heuristic and a breadth-first approach in a planner might lead to optimal solutions since all possible solutions would be considered. Although it might also lead to a higher memory consumption [51].

Defining a heuristic is a complex task that the developers should be aware of. There are different ways to define action costs. For example, the game *F.E.A.R.* used the number of unsatisfied preconditions as heuristics for A* [19]. Additionally, different agent types might get different costs for the same action based on their stats and abilities. Also, these costs might change depending on some outer influences, as in the game *Tomb Raider* where there were the so called *situational costs* implemented for actions which were adapted during game play [10]. This was done by tracking the success rates of goals and actions for each agent type and saving them with game data. This way, it was possible to have different costs and thus different plans for different agent types. An important decision to be made in this case, is whether or not the learned values should be saved until the end of the game. Doing so could mean that *boss enemies* could exhibit very advanced behaviors which in turn could over-strain the player. To prevent this, the learned action costs could be reset at certain intervals [10].

The second approach involves hierarchy instead of heuristics. Here, the search space contains partial plans instead of single actions and it is limited to predefined solutions. This approach corresponds more to human reasoning, decomposing a problem into its subproblems. However, the quality of generated plans strongly depends on the definition of the hierarchy, to a point that generated plans might be suboptimal. A good knowledge of the domain is required to create sufficient task networks. Also, depending on the decomposition order of the hierarchy, better plan candidates might accidentally be discarded from the search. On the other hand, for games, it is often not necessary to have the *optimal* plan, usually it is enough to execute *some* plan that is feasible and leads to the goal. Furthermore, hierarchical planners apply the depth-first approach which usually leads to a lower memory usage and a higher speed, since it does not require comparing multiple plan candidates, as is the case when using a heuristic and breadth-first search [51].

Instead of deciding on whether to use the heuristics approach or the hierarchical one, developers might also think about hybrid solutions. For example while creating mission plans for the game *ARMA II*, the mission generator *PlannedAssault* used an HTN planner combined with A* [15]. Here, *all* methods that were able to decompose a compound task in the hierarchy were applied and provided new plan candidates. The duration from the start of every plan to its end was used as the cost function. The best (shortest) alternative of the generated plans was then added to the open list of A* and further decomposed. Although this approach worked well for mission generation for *ARMA II*, where plans were generated offline (before the game session start) and planning took up to 2 minutes, it might be insufficient in terms of performance for planning in real-time where a planner might have only a few milliseconds to come up with a plan. Another alternative way of mixing the two approaches could involve heuristics and breadth-first search only at lower levels of a hierarchical planner and a depth-first search at higher levels.

Besides the optimization of the plan quality and the size of the search space, developers might also think about improving the general performance of the planner through time slicing. In some cases, the creation of a complete plan might take a long time and cause performance drops in the game. In order to prevent such drops, the planning process might be distributed over multiple frames [8]. Therefore, there should be a possibility of stopping the planner and continuing completion of an existing partial plan later. This way agents could start executing actions that the planner already created, while the planner would still finish creating the current plan. Partial planning was used in the game *Transformers: Fall of Cybertron*[13]. More detailed descriptions of how to use time slicing with HTN can be found in the description of the simulation *SquadSmart*[24] and the HTN planner implemented by the winner of the *Capture The Flag AI Competition*[26].

IX. Plan Execution Process

Besides the implementation of a planner that *creates* a plan, another very important system is the plan executor that is responsible for plan *execution*. Since these two processes take place at different points of time, there are many things that can happen in-between and cause a plan failure - especially in such a dynamic environment as a game world. For this reason, there are many aspects that should be handled with care when implementing a plan executor. These aspects like plan re-evaluation and re-planning are shown in Figure 8 and described in the following section.

A. Reactive Behavior

Having a game agent following a long-term plan without failures is the ideal situation. However, game environments are very dynamic, many entities and forces affect them and game agents need to react properly to changes and unexpected events. Reactive behaviors that could be easily implemented

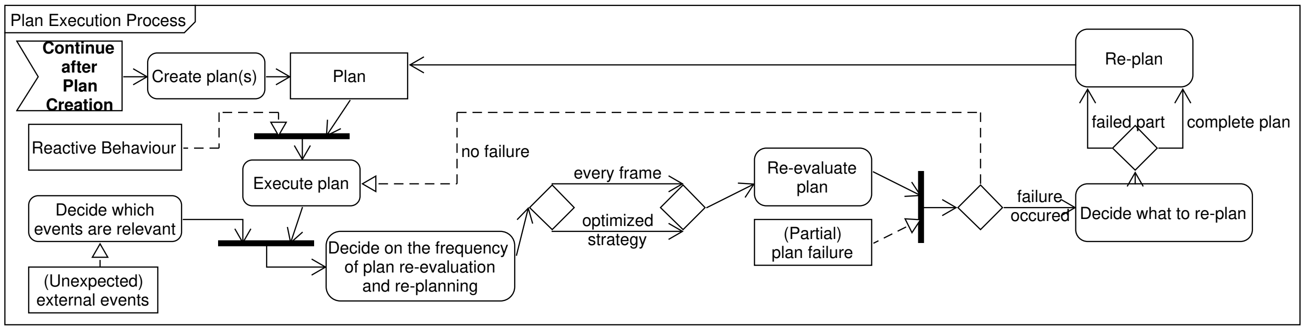

Fig. 8. Plan Execution Process: aspects to consider about the process of plan execution.

with State Machines or Behavior Trees are not directly provided by planners. So, in order to provide such a behavior, developers should think about some additional strategies.

One solution could be the mixed usage of planners and State Machines (as it was done in *F.E.A.R* [8] see section II-A) or Behavior Trees. For example, the primitive tasks of a planner could be implemented as Micro-Behavior Trees. This way, a planner could provide a plan which would contain the task *Goto*. While executing this task, the agent would use a Behavior Tree which would contain some logic for jumping over obstacles, ducking or looking at some target point. A novel solution from the research area of robotics that is described in [52] even proposes to *create* Behavior Trees *automatically* using the *Hybrid Backward Forward* (HBF) algorithm [53]. This approach allows for interleaving planning and acting, combining the advantages of the HBF planner which is able to plan in infinite state spaces and the reactivity of Behavior Trees during execution. For this reason, it shows promise for planning in game environments. An alternative solution would be the creation of new plans (re-planning) whenever necessary. Important aspects of this approach are described in next sections.

B. Extent of Plan Re-evaluation

Usually, while following a plan, this plan should be re-evaluated in order to recognize any changes and prevent a plan failure. One important question hereby is: *what* to re-evaluate? Since re-evaluating the whole plan is usually too costly – especially in the context of video games – in most cases, only the current and the next plan steps are re-evaluated. Thus, if an agent is already in the middle of the plan and finishes one plan step, he should check whether the preconditions of the next task/action still hold before performing this action. This approach would at least make sure that the agent does not fail in the directly following action.

However, in some cases, the plan could fail in some later steps, even though the next step to perform would be still valid. So, in order to foresee plan failure earlier and to re-plan, it might make sense to re-evaluate more than one step ahead. This approach is applied in different planning areas and might be interesting to use in the field of planning in video games as well. Thereby, developers should take into consideration the costs of re-evaluation in relation to the available resources and for example the duration of single plan tasks. For example, if the current task takes a lot of time (like going to a certain position), there might be too many events/changes happening in this time, so that the re-evaluation of the next action might again become less reliable.

C. Frequency of Plan Re-evaluation and Re-planning

Besides thinking about *what* to re-evaluate, it is also important to decide on *how often* to perform the *re-evaluation* and when to *re-plan*. Since many preconditions of tasks may require information, for example, from the sensory system and thus would need some costly ray-casting, re-evaluating the plan in every frame usually would be a bad option. For the same reason, re-evaluating the next task when an agent finishes one task, might lead to undesired delays.

One possibility to prevent such delays is performing re-evaluation on certain events instead of doing it periodically. For example, whenever a sensor perceives some important change it could send a corresponding signal to the plan executor to check the preconditions of the current task and to re-evaluate the main goal of the agent. In case *one*, *all* or only *some* of the preconditions of a plan step did not hold anymore or the goal changed, re-planning should be performed. This approach was used in the game *Dying light* [14] where if the character was hit, for example, the goal changed from *attack* to *react with pain*. Then, a new plan was created replacing the old one. Additionally, a new plan was created whenever the previous one finished. Goal re-evaluation on events was also implemented in *F.E.A.R.* [19]. Also, *SquadSmart* [24] and the game *Dirty Harry* [37] re-planned after certain events.

An alternative solution is proposed by Jeff Orkin in [19]. To not perform some costly computations on-demand, the different checks of subsystems could be distributed over multiple frames (as already mentioned in section VIII-E) and the results of these checks could be saved in a working memory. This way, the re-evaluation of preconditions could be done whenever needed, but instead of querying the systems for information, the latest results could be directly read out from the working memory.

However, re-planning on a regular basis might make sense and even be necessary in some cases, like for example, if there were no effects implemented in the planner and changes in the world state were not propagated during the planning process as discussed in section VIII-B. In this case, the whole plan would be build upon the initial state, so that the world state might quickly become very different from the initial state which could lead to plan failures. Re-planning periodically and very frequently might prevent this problem. This approach was used in *Killzone 2* which did not propagate changes in the world state. Here new plans were created 5 times per second for every character [44]. As an additional side-effect of re-planning periodically, plan re-evaluation might become unnecessary.

D. Re-planning strategy

Often, when a plan step fails, the rest of the plan can still be valid. So, it might make sense to only re-plan the failed part of the plan instead of creating a whole new plan. In academia, there are multiple repairing approaches that allow for partial re-planning. These approaches are especially valuable if, for example, full re-planning takes a lot of resources, created plans are very long or there are other good reasons to keep existing plans.

In video games however, plans are often very short and the planning processes are already optimized so much, that creating a new plan is not a major problem, especially since many actions are represented simply by animations. In this context, it might even become more complicated to re-pair existing plans modifying their parts than creating new ones. According to Dana Nau, one of the pioneers in planning in academia, performing a complete re-planning might be actually the best solution for games [44]. So, at this point, developers should think about whether there is any good reason to keep existing plans and whether it is worth adding a repairing technique to the planner. As already mentioned, games like *Transformers: Fall of Cybertron* and *Killzone 2* applied full re-planning whenever needed [44].

Another important issue related to re-planning is the abortion of running actions and a smooth transition to the new plan. In contrast to applications from other industries, in games, it is important to have a *naturally looking* agent behavior. Thus, some additional mechanisms should be implemented in order to prevent agents from switching between very different actions, especially if certain actions cannot be instantly aborted. *Killzone 2* for example, implemented so-called *continue branches* in its HTN [12]. These (backup-)branches were switched to when an agent's current plan became obsolete, making it do something *seemingly relevant* while waiting for a new plan. *How* an action should be finished using such a branch could be defined for every action separately.

X. Tools

Similar to many other systems in game development, the planner and its domain might change a lot throughout the development process. Designers might ask for more actions, different conditions or effects. In order to be able to introduce changes easily, it might be very helpful to develop editor tools that would allow to tweak planner parameters and define actions. Of course, developing such a tool always leads to additional costs. However, depending on how many different developers should work with the planning domain and whether the planner should be re-used for further projects, the development time might be worth it. Furthermore, in order to introduce heterogeneous agents, it might be a great help to use a user-friendly tool for assigning various attribute values to different agents. Such a tool was, for example, used for the development of the game *Dirty Harry* where designers could assign different goals, action sets and attribute values to different types of agents [37].

Another very important tool for game development in general and especially for a planner is a proper debugging tool. It is already a big problem to understand why an agent is behaving in a certain way when using reactive Behavior Trees, but it is even a bigger problem to understand a behavior caused by a long-term planner. Since a planner does not only take into account the current state of the world, but also makes assumptions about future world states, it is not trivial to track back its search process and understand what exactly caused an undesired behavior. Similar problems arise in the area of robotics [54] where generating plans that are easy to understand by humans is a challenging task that needs to be solved.

Since the world state of a game changes very quickly and the amount of variables used by a planner might be enormous, a simple run-time debugger might not be sufficient at this point. Additional ways to record and playback the gameplay might help to reproduce certain game situations and step through the plan creation process [37]. Even more helpful could be some sort of history to record the agents' goals, the statuses of each agent's plan and executed actions.

XI. Conclusion

Some of the recent commercial video games have used a planning system to define complex agent behavior. Most of them have implemented their own version of either the Stanford Research Institute Problem Solver (STRIPS) or the Hierarchical Task Network (HTN). In this work, we have given an overview of the planner implementations in some of these games and analyzed them in terms of different components of a planner. We defined seven areas that are important to focus on when implementing a planner and identified problems that might arise in these areas. Showing possible solutions to these problems, we have tried to answer questions such as: how to handle high dynamics of a game world preventing plan failures, how to define a planner domain, how to achieve coordination between multiple agents and how to create plans for heterogeneous agents. Furthermore we discussed how to optimize the search in a huge search space.

Even though both HTN and STRIPS have shown good results in some games, they still required extensive modifications in order to combine long term planning with natural looking reactive agent behavior. In our opinion, the blending of planning and execution still remains a major problem in many real-time environments and is the main reason for the low adoption rate of planners in games. With this paper, we hope to provide an insight into this and other problems that developers of planners had to face in the past and to motivate academics to do more research that would resonate with the needs of developers.

REFERENCES

[1] S. Rabin, "#define game_ai," 2009. [Online]. Available: http://gdcvault.com/play/1366/\(307\)-define-GAME

[2] D. S. Nau, "Current trends in automated planning," AI magazine, vol. 28, no. 4, p. 43, 2007.

[3] D. E. Wilkins, "Practical planning: extending the classical ai planning paradigm," 2014.

[4] R. E. Fikes and N. J. Nilsson, "Strips: A new approach to the application of theorem proving to problem solving," Artificial intelligence, vol. 2, no. 3-4, pp. 189–208, 1971.

[5] R. A. Girle and M. Fitting, "First-order logic and automated theorem proving," 1998.

[6] M. Ghallab, D. Nau, and P. Traverso, Automated planning: theory & practice. Elsevier, 2004.

[7] J. Orkin, "Three states and a plan: the ai of fear," in Game Developers Conference, vol. 2006, 2006, p. 4.

[8] ——, "Applying goal-oriented action planning to games," AI Game Programming Wisdom, vol. 2, no. 2004, pp. 217–227, 2004.

[9] A. Champandard, "Planning in games: An overview and lessons learned." [Online]. Available: http://aigamedev.com/open/review/planning-in-games/

[10] P. Higley, "Goal-oriented action planning: Ten years old and no fear!" [Online]. Available: http://www.gdcvault.com/play/1022019/Goal-Oriented-Action-Planning-Ten

[11] I. Georgievski and M. Aiello, "An overview of hierarchical task network planning," arXiv preprint arXiv:1403.7426, 2014.

[12] R. Straatman, "Killzone 2: Multiplayer bots." [Online]. Available: http://files.aigamedev.com/coverage/GAIC09_Killzone2Bots_StraatmanChampandard.pdf

[13] T. Humphreys, "Planning for the fall of cybertron: Ai in transformers." [Online]. Available: http://aigamedev.com/premium/interview/planning-transformers/

[14] M. Kurowski, "Dying lights zombies and htn planning in open worlds." [Online]. Available: http://aigamedev.com/premium/interview/dying-light/

[15] W. Van der Sterren and A. Champandard, "Plan-space hierarchical heuristic search for planned assault in arma ii." [Online]. Available: http://aigamedev.com/members/access.php?article=/premium/interview/planned-assault/

[16] D. Nau, Y. Cao, A. Lotem, and H. Munoz-Avila, "Shop: Simple hierarchical ordered planner," in Proceedings of the 16th international joint conference on Artificial intelligence-Volume 2. Morgan Kaufmann Publishers Inc., 1999, pp. 968–973.

[17] T. Humphreys, "Exploring htn planners through examples," Game AI Pro: Collected Wisdom of Game AI Professionals, vol. 149, 2013.

[18] A. Champandard, "Assaulting f.e.a.r.s ai: 29 tricks to arm your game." [Online]. Available: http://aigamedev.com/open/review/fear-ai/

[19] J. Orkin, "Agent architecture considerations for real-time planning in games." in AIIDE, 2005, pp. 105–110.

[20] D. Isla and B. Blumberg, "Blackboard architectures," AI Game Programming Wisdom, vol. 1, no. 7.1, pp. 333–344, 2002.

[21] J. Orkin and J. Kelly, "Simple techniques for coordinated behavior," AI Game Programing Wisdom, vol. 2, 2004.

[22] J. Reynolds, "Tactical team ai using a command hierarchy," AI Game Programming Wisdom, vol. 1, pp. 260–271, 2002.

[23] A. Champandard, "Using a static blackboard to store world knowledge." [Online]. Available: http://aigamedev.com/open/article/static-blackboard/

[24] P. Gorniak and I. Davis, "Squadsmart: Hierarchical planning and coordinated plan execution for squads of characters." in AIIDE, 2007, pp. 14–19.

[25] G. Steele, Common LISP: the language. Elsevier, 1990.

[26] A. Shafranov and A. Champandard, "Planning domains and compiling htn to c++." [Online]. Available: http://aigamedev.com/premium/interview/plan-compilation/

[27] A. Champandard, "Hierarchical planning and coordinated plan execution for squads of characters." [Online]. Available: http://aigamedev.com/open/review/hierarchical-planning-coordinated-execution/

[28] M. A. Leece, "Unsupervised learning of htns in complex adversarial domains," in Tenth Artificial Intelligence and Interactive Digital Entertainment Conference, 2014.

[29] C. Hogg and U. Kuter, "Learning methods to generate good plans: Integrating htn learning and reinforcement learning." 2010.

[30] H. H. Zhuo, H. Muñoz-Avila, and Q. Yang, "Learning hierarchical task network domains from partially observed plan traces," Artificial intelligence, vol. 212, pp. 134–157, 2014.

[31] K. Shafi and H. A. Abbass, "A survey of learning classifier systems in games [review article]," IEEE Computational Intelligence Magazine, vol. 12, no. 1, pp. 42–55, 2017.

[32] R. T. Marler and J. S. Arora, "Survey of multi-objective optimization methods for engineering," Structural and multidisciplinary optimization, vol. 26, no. 6, pp. 369–395, 2004.

[33] J. Branke, K. Deb, K. Miettinen, and R. Słowiński, "Multiobjective optimization: Interactive and evolutionary approaches," 2008.

[34] D. Staltman, "Demigod's ai from role-playing to real-time strategy." [Online]. Available: http://aigamedev.com/premium/interview/demigod-role-playing/

[35] E. Jacopin, "Game ai planning analytics: The case of three first-person shooters." in AIIDE, 2014.

[36] C. Boutilier, R. I. Brafman et al., "Planning with concurrent interacting actions," in AAAI/IAAI, 1997, pp. 720–726.

[37] D. Iassenev, M. B., J. Orkin, B. Pfeifer, and A. Champandard, "Special report: Goal-oriented action planning." [Online]. Available: http://aigamedev.com/premium/report/goal-oriented-action-planning/

[38] B. G. Weber and M. Mateas, "Case-based reasoning for build order in real-time strategy games." in AIIDE, 2009.

[39] S. Ontañón, K. Mishra, N. Sugandh, and A. Ram, "Case-based planning and execution for real-time strategy games," in International Conference on Case-Based Reasoning. Springer, 2007, pp. 164–178.

[40] P. Tozour, "Introduction to bayesian networks and reasoning under uncertainty," AI game programming wisdom, vol. 1, pp. 345–357, 2002.

[41] J. Blythe, "An overview of planning under uncertainty," in Artificial intelligence today. Springer, 1999, pp. 85–110.

[42] C. Conway, "Goap in tomb raider," 2015. [Online]. Available: http://www.gdcvault.com/play/1022020/Goal-Oriented-Action-Planning-Ten

[43] L. A. Castillo, J. Fernández-Olivares, O. Garcia-Perez, and F. Palao, "Efficiently handling temporal knowledge in an htn planner." in ICAPS, 2006, pp. 63–72.

[44] C. A. Nau, Dana, "Inside hierarchical task network planners." [Online]. Available: http://aigamedev.com/premium/interview/htn-planners/

[45] H. Skubch, "Not just planning: Strips for ambient npc interactions in final fantasy xv," 2015. [Online]. Available: https://archives.nucl.ai/recording/not-just-planning-strips-for-ambient-npc-interactions-in-final-fantasy-xv/

[46] ——, "Ambient interactions: Improving believability by leveraging rule-based ai," Game AI Pro, vol. 3, 2017.

[47] A. Champandard, "Summary of strips: A new approach to the application of theorem proving to problem solving." [Online]. Available: http://aigamedev.com/open/article/strips-theorem-proving-problem-solving/

[48] W. Van Der Sterren, "Squad tactics: Planned maneuvers," AI Game Programming Wisdom, pp. 247–259, 2002.

[49] S. J. Russell and P. Norvig, "Artificial intelligence: A modern approach," 2002.

[50] D. Nau, H. Munoz-Avila, Y. Cao, A. Lotem, and S. Mitchell, "Total-order planning with partially ordered subtasks," in IJCAI, vol. 1, 2001, pp. 425–430.

[51] A. Champandard, "Heuristic vs. hierarchy: Domain knowledge for planners." [Online]. Available: http://aigamedev.com/open/article/heuristic-hierarchy/

[52] M. Colledanchise, D. Almeida, and P. Ögren, "Towards blended reactive planning and acting using behavior trees," arXiv preprint arXiv:1611.00230, 2016.

[53] C. R. Garrett, T. Lozano-Pérez, and L. P. Kaelbling, "Backward-forward search for manipulation planning," in Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on. IEEE, 2015, pp. 6366–6373.

[54] R. Alterovitz, S. Koenig, and M. Likhachev, "Robot planning in the real world: research challenges and opportunities," AI Magazine, vol. 37, no. 2, pp. 76–84, 2016.