Long-Term Planning and Reactive Execution in Highly Dynamic Environments

DISSERTATION

zur Erlangung des akademischen Grades

Doktoringenieurin (Dr.-Ing.)

angenommen durch die Fakultät für Informatik
der Otto-von-Guericke-Universität Magdeburg

von M.Sc. Xenija Neufeld

geb. am 18.01.1990 in Nikolskoje

Gutachterinnen/Gutachter

Prof. Dr.-Ing. habil. Sanaz Mostaghim
Prof. Dr. Simon Lucas
Prof. Dr. Mike Preuss

Magdeburg, den 17.12.2020

Abstract

In many highly dynamic environments artificial agents need to follow long-term goals and therefore are required to reason and to plan far into the future. At the same time, while following long-term plans, the agents are also required to stay reactive to environmental changes and to act deliberately while always maintaining robust and secure behaviors. In many cases, such agents act as parts of a larger system and need to collaborate while coordinating their actions. Generating agent behaviors that allow for long-term planning *and* reactive acting is a complex task, which becomes even more challenging with an increasing number of agents and an increasing size of the search space. This thesis focuses on video games as highly dynamic multi-agent environments proposing a solution that allows to combine long-term planning with reactive execution.

On the one hand, existing literature proposes a variety of different planning approaches. However, plans that are executed in highly-dynamic environments are very likely to fail during their execution. This can lead to high replanning frequencies and delayed execution. On the other hand, there are various reactive decision-making approaches, which allow agents to quickly adjust their behaviors to environmental changes. However, usually such approaches do not allow for long-term planning.

Inspired by various approaches observed in areas such as spacecraft control, robotics, and video games, this thesis proposes a hybrid approach. The general idea of the hybrid solution combines a Hierarchical Task Network (HTN) planner and a reactive approach in a three-layer architecture. The approach separates the decision-making responsibilities between a planner, which is responsible for abstract long-term planning, and a reactive approach that is responsible for local decision-making and task refinement at execution time. The major contribution of this work, which allows for such interleaved decision-making and continuous execution of long-term plans, is an extension of the plan tasks, which is used by the reactive approach at execution time.

The thesis describes two different implementations of this solution using either Behavior Trees or Monte Carlo Tree Search (MCTS) as reactive approaches. It examines the effects of the interleaved decision-making in two different highly dynamic video game environments and evaluates the performance of agents using the hybrid approaches comparing them to existing benchmark agents. Additionally, it proposes a possibility to automatically improve the execution of long-term tasks using an Evolutionary Algorithm.

The results of the performed experiments show that the proposed solutions allow to reduce the global replanning frequencies and decrease the total execution time of multi-agent long-term plans while increasing the success rates of their execution when compared to a pure planning approach. Furthermore, the use of the extended high-level plan tasks allows to guide the search process of MCTS resulting in emergent agent behaviors, which can be further improved by a learning mechanism such as an Evolutionary Algorithm.

Zusammenfassung

In vielen hochdynamischen Umgebungen müssen Agenten Langzeitziele verfolgen und dafür weit in die Zukunft planen können. Während sie Langzeitpläne ausführen, müssen sie schnell auf Veränderungen in ihrer Umgebung reagieren können und stets bewusstes, robustes und sicheres Verhalten zeigen. In vielen Fällen agieren sie als Teile eines größeren Systems und müssen ihre Handlungen koordinieren. Die Generierung von Agentenverhalten, die sowohl die Verfolgung von Langzeitplänen als auch reaktives Handeln ermöglichen, ist eine große Herausforderung, die mit steigender Agentenanzahl und steigender Größe des Suchraums noch komplexer wird. In dieser Thesis werden Videospiele als hochdynamische Multiagentenumgebungen untersucht und eine Lösung vorgeschlagen, die es erlaubt, die Verfolgung von Langzeitzielen mit reaktivem Handeln zu kombinieren.

Einerseits beschreibt existierende Literatur eine Vielzahl an unterschiedlichen Planungsansätzen, jedoch scheitern Langzeitpläne oft bei ihrer Ausführung in hochdynamischen Umgebungen. Dies kann zu häufigen Neuplanungen führen und potenziell die Ausführung der Pläne verzögern. Andererseits existieren viele reaktive Entscheidungssysteme, die schnelle Anpassungen an Agentenverhalten ermöglichen, jedoch nicht weit in die Zukunft planen können.

Inspiriert von unterschiedlichen Ansätzen aus den Bereichen der Raumfahrt, der Robotik und der Videospiele wird in dieser Thesis ein hybrider Ansatz vorgeschlagen. In seiner Grundidee kombiniert der Ansatz einen Hierarchical Task Network Planer und ein reaktives Entscheidungssystem in einer Drei-Schichten-Architektur. Die Entscheidungsverantwortung wird zwischen dem Planer, welcher für abstrakte Langzeitplanung verantwortlich ist, und einem reaktiven System, welches lokale Entscheidungen trifft und die abstrakten Aufgaben während der Ausführung verfeinert, aufgeteilt. Der Hauptbeitrag dieser Arbeit, der eine gekoppelte Entscheidungsfindung and eine ununterbrochene Ausführung ermöglicht, ist eine Erweiterung der Planungsdomäne, welche während der Ausführung von dem reaktiven System benutzt wird.

Die Thesis beschreibt zwei konkrete Umsetzungen der vorgeschlagenen Lösung, die entweder Behavior Trees oder Monte Carlo Tree Search (MCTS) als reaktive Systeme einsetzen. Die Auswirkungen der kombinierten Entscheidungsfindung werden in zwei unterschiedlichen hochdynamischen Videospielumgebungen untersucht und die hybriden Agenten mit existierenden Benchmark-Agenten anhand ihrer Spielleistung verglichen. Außerdem wird eine Möglichkeit vorgeschlagen, die Ausführung von Langzeitaufgaben durch einen evolutionären Algorithmus zu verbessern.

Experimentergebnisse zeigen, dass die vorgeschlagene Lösung die globale Häufigkeit der Neuplanungen sowie die Ausführzeit von Langzeitplänen im Vergleich zu einem reinen Planungsansatz verringern kann während die Erfolgsrate der Ausführungen erhöht wird. Die Erweiterung der Planungsdomäne erlaubt es außerdem den Suchprozess von MCTS zu lenken, wodurch emergente Verhalten entstehen, die durch einen Lernmechanismus wie einen evolutionären Algorithmus noch weiter angepasst werden können.

Contents

Chapter 1

Introduction

In many virtual as well as real-world environments, artificial agents are required to operate deliberately and pursue long-term goals under quickly changing environment conditions. The difficulty of achieving such goals while staying reactive increases if further requirements such as coordination or cooperation are added. This work focuses on such problems and proposes different solutions to them. This chapter explains the motivation of this work in more details and describes fundamental characteristics of the kind of environments considered in this work. It clarifies the problems and challenges arising from such environments and, finally, defines the goals of this thesis.

1.1 Motivation

Virtual and physical artificial agents are operating in many environments such as video games, simulation environments, smart factory environments, and various robotics applications. Often, they are contributing to an overall high-level goal while being part of a larger system. In order to help achieving such goals, the agents are required to reason far into the future searching for feasible (and potentially optimal) sequences of actions and to execute them in a robust way over a potentially long period of time. At the same time, while pursuing *long-term* goals, the agents are equally required to act in a *reactive* way, which includes detecting environmental changes, recognizing and managing potential failures, and adapting their behaviors accordingly. These two requirements are usually addressed separately by dedicated solutions.

The process of reasoning far into the future and searching for long-term solutions is called *planning* and sequences of actions to be executed by an agent are called *plans*. There is a high variety of different planning techniques in academia. However, classical planning techniques (described in Section 2.2.1) assume that the agent is operating in a static environment and that it is the only entity that changes the environment. This allows for a *planner* to create a full plan in a *planning phase* followed by an *execution phase*, in which the agent can execute the created plan from the beginning to the end. However, the clear separation between the two phases and a full execution of a plan are not possible in highly dynamic environments for several reasons.

The major challenges of planning in highly dynamic environments are caused by non-determinism, uncertainties, and partial observability. Trying to reason and to plan into the future, an agent is likely to be lacking some necessary information, which can only become known at the execution time, during the planning phase. This requires the agent to actively gather information and to make use of it by potentially requesting a more optimal plan (also called *replanning*) during the execution. Furthermore, the information that was already available at plan time can change along with environmental changes during the execution invalidating the previously created plan. Such plan failures also require replanning. Another reason for replanning can be a change or re-prioritization of the agent's goals. For these reasons, planning and execution can no longer be separated and have to be tightly interleaved ensuring that the agent is always following an updated plan.

Other challenges of highly dynamic environments considered in this work are the large search spaces that an agent has to deal with. Searching far into the future and generating a detailed long-term plan can be computationally intensive and take a long time. Stopping the execution of a previous plan while waiting for a new plan can potentially violate robustness and safety requirements of an agent, endanger the agent and other actors, or simply lead to unnatural and undesired agent behaviors. That means, on the one hand, the agent is required to create a long-term plan, which accomplishes a high-level goal but the computation of which can be very expensive and consequently delay its execution. On the other hand, it requires reactive execution that considers the current state of the environment and ensures robust and safe behavior of the agent in all possible situations and cannot be delayed by a long search process. Therefore, when interleaving planning and execution, it is not desirable to perform a full replanning very frequently. Instead, the replanning frequency and replanning times should be minimized in order to not delay the execution.

In contrast to planners, which search for long sequences of actions, there are different reactive approaches that allow for fast decision-making at execution time. Among such approaches are Behavior Trees as richer extensions of Finite State Machines and Monte Carlo Techniques that approximate a full search. Usually, reactive approaches focus on a local search considering the current state of the environment and searching for a single optimal action to be executed in the current moment. They do not provide further actions to be executed afterwards and, in many cases, neither consider previous states nor reason into the future. Limiting the search space in this way allows for a fast and efficient search, which is sufficient for purely reactive behaviors and tasks. However, if no previous and no future states are considered, following high-level long-term goals becomes problematic.

In order to bridge the gap between purely reactive decision-making and pure planning, there are different hybrid approaches that combine various techniques. As described in more details in Section 2.3.1, a major challenge in combining planners and reactive approaches is the fact that they operate on different domain descriptions. Reactive techniques usually operate on either a predefined domain, which describes rules for decision-making (for example Behavior Trees as described in Section 2.1.1) or they require a (simulation) model of the world on which to perform a search (for example Monte Carlo Tree Search as described in Section 2.1.2). Manual as well as automated creation of both domains and models is difficult to a certain degree, however not fully impossible.

More challenging is the combination of these domains with planning domains, which usually have a different structure. For that reason, many existing hybrid approaches are domain-specific and consequently cannot be used in other domains. Furthermore, most of these approaches are used in environments that are less complex and less dynamic than the highly dynamic environments that are considered in this work. Therefore, there remains the need for a solution that can interleave long-term planning and reactive execution that is general enough to be used in various environments and is applicable to highly dynamic environments according to their description in the following section.

1.2 Highly Dynamic Environments

As the name suggests, *highly dynamic environments* change very frequently. Naturally, these are *real-time* environments and therefore *time* is an important variable when making decisions and acting in such environments. For artificial agents, the dynamics especially affect the requirements and the restrictions on the computational time budget that is available to make a decision and execute a program step. When speaking about *highly* dynamic environments, we consider scenarios in which the computational budget lies in the milliseconds range.

In addition to the computational time, another important aspect to be considered in highly dynamic environments is the *impact of the changes* in an environment on the agents' behaviors. Dynamic environments can be categorized by the *severity* of changes and the *frequency* of changes [1]. The severity indicates how much an environmental changes and whether these changes are only small local changes that do or do not affect an agent's behavior or whether they are large changes that affect the global situation. The frequency indicates how often environmental changes happen and consequently how often an agent has to adapt its behavior. For highly dynamic environments, we assume a very high (up to continuous) frequency of small environmental changes and a high frequency of severe changes that happen in the range of seconds to minutes.

There are various *causes* of environmental changes such as (natural) changes coming from the environment itself (for example time of day or weather changes) or changes that are caused by other actors[1] acting in the environment. With a growing number of actors that can affect an environment, its dynamics increase. Therefore, in highly dynamic environments, we expect a high number of actors. For that reason, in this work, we focus on *multi-agent* scenarios with multiple agents being controlled by our approaches and additional agents acting independently contributing to environmental changes.

When considering multi-agent scenarios, there are certain challenges that do not necessarily arise from highly dynamic environments as such but do result from the multi-agent setting and make decision-making in such environments even more complex. For example, in addition to other actors, which simply do not directly communicate with the agents, there can be actors who act as adversaries of the agents. In both cases, they add more uncertainty and non-determinism to the environment. However, in the adversarial setting, they can deliberately manipulate the environment and try to invalidate an agent's plan. Therefore, in highly dynamic *adversarial* environments, we expect extremely high uncertainty.

An important requirement for agents is *cooperation*. If the agents act as parts of a larger system or a group, they are required to cooperate while following a common goal. Alternatively, if no cooperation is required, each agent simply follows its own goals. Cooperation usually implies a distribution of tasks. Consequently, another requirement can be an optimal assignment of tasks to agents, which sometimes happens through *role assignment*. The challenge of an optimal role assignment becomes, again, more complex if the agents are heterogeneous with different types of agents being able to execute only certain tasks. Therefore, working with heterogeneous agents makes the problem of decision-making *combinatorial* and increases the size of the search space even further. For that reason, when speaking about highly dynamic multi-agent environments, which are in most cases continuous, we assume *large search spaces*.

[1]Throughout this work, we use the word *agent* for artificial agents, while an *actor* can be both an artificial or a natural entity acting in the environment.

Another challenge comes from the requirements or restrictions on the agent *control*. One important aspect here is the way that multiple agents are controlled, which can be achieved by using a *centralized* system or each agent making its own decisions in a *decentralized* manner or there can be a kind of *hierarchy* among the agents. The type of multi-agent control can imply whether the agents use a *shared knowledge* and how they can *communicate* with each other or with the central system.

These aspects affect the planning and acting of agents in highly dynamic environments in multiple ways. For example, having one centralized system that plans far into the future for multiple agents can lead to long planning times and possibly cause delays in the execution. On the other hand, maintaining a shared knowledge base can save some communication between the agents and allow for more precise plans. In contrast, distributed planning can be more time-efficient but requires more communication between the agents, which, again, can cause delays.

Besides planning times and possible delays, another important aspect in the context of control in highly dynamic environments is the extent to which the change of one agent's behavior affects other agents' plans. Depending on how the agents are controlled, how they communicate, and the severity of changes, their frequency of decision-making and behavior adaptation can vary. For example, with a shared knowledge base and a central decision-making system, a failure of one agent's task can be recognized by the central system immediately and the system can decide whether a new global plan is required depending on the severity of the change. Consequently, with an increasing number of agents, the frequency of behavior adaptation for every agent can grow. On the other hand, it allows every agent to operate on the most recent plan. In contrast, in a decentralised system, each agent can decide for itself whether the change of the other agent's behavior is severe enough to change its own plan, which does not necessarily lead to a globally optimal plan execution. Additionally, an environmental change can lead to a change of the common goal requiring a global adaptation of all agents' behaviors.

Independent from the question whether or not the agents cooperate, when acting in the same environment and using the same resources, they can be required to *coordinate* their actions. Such coordination can be required even if every agent is following its own goals. An example for such coordinated behavior are multiple agents requiring to use the same resource one after another, such as one agent walking through a narrow corridor with another agent waiting at the other end not blocking the way. Coordinated behaviors are even more likely in cooperative scenarios, for example, if multiple agents have to use the same resource simultaneously in order to contribute to the common goal. Carrying an object together is a common example for such cooperative and coordinated behavior.

When considering coordination between multiple agents in a scenario with planning and execution, it is important to distinguish between the time phases at which the coordination is required. For example, it can be required at plan-time only if the agents have to coordinate either the distribution or the order of their tasks but do otherwise not depend on each other at execution-time. On the other hand, it is possible that there is no (time) dependency between the agents' plans at plan-time but the requirement for coordination arises during the execution. This holds for the corridor example mentioned above.

Alternatively, in the cooperative and coordinated scenario where two agents are supposed to carry an object, coordination is required both at plan time and at execution time. At plan time, the agents need to coordinate their plans in such a way that both of them move next to the object before picking it up together. At execution time, they need to coordinate their actions locally in order to simultaneously pick it up and balance the object trying not to drop it during the transportation.

Another important question in the context of coordination is whether it happens in an explicit or an implicit way. Implicit coordination requires less communication between the agents (and the central control system) than explicit coordination. It can be enabled through a shared knowledge base or through observations of the other agents and the environment.

When creating artificial agents for a certain environment, a very important aspect is the degree of autonomy: how much control over the agents is desired as opposed to how much autonomy they can be granted with. The solution to this depends, in first place, on the required robustness level of the agents' behaviors and potentially on the availability of expert knowledge about the environment. Especially in highly dynamic environments where the agents are making decisions under limited time constraints, it is important to decide how much autonomy they can be provided with. In certain critical or dangerous environments, full control over an agent's decision-making process can be granted by providing it with predefined rules. Selecting and following such rules does not only ensure specific behaviors in dangerous situations but also decreases the decision-making time in comparison to a full autonomous search.

In contrast, in less critical environments or when not enough expert knowledge is available or if such knowledge cannot be encoded properly, the agents can be granted more autonomy. In this case, they can search for feasible actions, potentially simulating them into the future. Simulation, however, requires a model of the environment, which, in the extreme case, the agents can be required to learn from scratch. A possible drawback of full autonomy, however, are undesired behaviors. The level of autonomy can also vary depending on the situation or the granularity of the decision to be made. Varying the provided level of autonomy is possible with hybrid solutions consisting of different decision algorithms used either at different points of time (depending on the situation) or on different control levels (depending on the decision granularity).

The described challenges are present in many different real-time environments and make decision-making and acting in highly dynamic environments especially complex. As already mentioned, all these challenges do not apply in classical planning. Therefore, if there is a need for long-term planning in such environments, it requires a flexible and yet powerful approach that is able to deal with the presented complexity of the search space and the high dynamics during the execution. In the remainder of this section, we present examples of highly dynamic environments and classify the environments used in this thesis according to the described challenges.

With the increasing amount of automation that can be observed in various industries, there arise more dynamic environments, in which artificial agents are used. A large part of dynamic environments represents the industrial sector, where different kinds of machines work as part of a factory system. In many cases, such agents work next to human workers and are referred to as *cobots* (collaborating robots). Another example of robots that operate in interaction with humans are the so called *social* robots. These machines can be used in different service sectors such as retirement homes or hotels. Further examples are robots that are used in rescue missions where the degree of uncertainty is particularly high and the severity of dynamics are critical. More real-world environments for robotic planning as well as their challenges and opportunities are described in [2].

Since interaction between machines and humans is challenging and due to the unpredictability of human behaviors, the uncertainty of an environment from an agent's perspective increases with the number of human actors in it. Additionally, with an increasing size of the environment and the search space, the possible severity levels of environmental changes increase. Therefore, we assume open spaces with a high number of human actors to have generally higher dynamics than closed environments with a low number of non-controllable actors.

A good example for a highly dynamic real-world environment are future traffic scenarios, in which autonomous cars are the artificial agents operating among many human actors. When driving from point A to point B, an autonomous car is required to have a long-term plan containing different sub-sections of a path. However, it is not possible to create this plan on a very detailed level upfront. Instead, while following the original plan, the agent is required to observe the environment, react to its changes, and refine its plan according to the situation ensuring safety and robustness of all actors.

In regards to the challenges mentioned above, the agent, in this case, is operating in a multi-agent environment where it is following its own goals. It is using shared resources (the road) with other actors and therefore is required to reason about their intentions and coordinate its actions. According to most of the visions of the future of autonomous driving, it can have the possibility to communicate with other (autonomous) cars but it cannot communicate with human road users. Since it is acting as an individual and not as part of a group, it is using its own decentralized decision-making approach. Additionally, in case of smart cities, it can communicate with some central system, which is responsible, for example, for an optimal distribution of traffic within the city through traffic light and navigation system control. Since it is impossible to manually encode a detailed rule set for every possible situation in such scenarios, the agent is required to act fully autonomously. For this, it usually uses a simulation model, the learning of which is one of the major challenges nowadays.

Similarly to the described *real-world* environments, there is a large number of *virtual* highly dynamic environments, namely video games. In the last two decades, the graphical quality of video games has improved, reaching almost photo-realistic visualizations. The constantly improving hardware allows for large open-world games with large numbers of simultaneously playing players. These aspects increase the size of the search spaces of artificial agents acting in such games. At the same time, the expectations of players towards many game aspects grow, including the believability of the artificial agents' behaviors.

In many games, artificial agents can collaborate with players as so-called *buddy characters*. Although, in most cases, they act as adversaries of the player. In both cases, the agents can be required to show deliberate long-term behaviors as well as to quickly react to environmental changes. In different game genres, ranging from so-called shooter games to Real-Time Strategy (RTS) games, multiple agents can be operating as part of a group. In such cases, an expectation of a player is a visible coordination of the agents' actions. Especially in strategic games, the requirement for visible long-term strategies is very high. From the perspective of the game industry, the player is the most important variable in the design process of a game. For that reason, an improvement of the agents' (collaborative) behaviors is interesting to the industry.

From the research perspective, on the other hand, video games represent highly dynamic environments whose complexity can even exceed the complexity of the currently available real-world settings. At the same time, simulating game environments and testing academic approaches in them is less expensive, faster, and less dangerous than using real-world environments or creating specific simulation environments for tests. Video games represent the previously described challenges to different degrees, as described in more details in our survey [3]. They are multi-agent environments, in which the agents can be required to act in a cooperative and adversarial manner at the same time. The number of cooperating agents can reach multiple hundreds while the size of the simulated environment can reach multiple square kilometers.

The size of the already large search space can increase even further with heterogeneous agents and large numbers of players. Since players are the major source of uncertainty from the agents' perspective, the uncertainty is particularly high in multi-player games. At the same time, most games are high-pace environments where both the frequency and the severity of environmental changes can be very high. Both the agents and the players are able to execute a large number of actions and to cross long distances within seconds combining different forms of movement (running, driving, flying).

For example, in shooter games, a player can pick up some objects, run a hundred meters, and hide from an agent's view in less than a minute, forcing the agent to adapt its behavior multiple times during this time span. In RTS games, the severity of the changes that can happen within the same time is even higher. For example, if an agent is controlling an army of 100 military units and its opponent is moving its army of a similar size, then each of the 200 units can cross certain distances and execute multiple actions within one minute. This can possibly change not only the local situations of each army but also lead to a global game change. Such large numbers of artificial cooperating agents are not (yet) usual for real-world scenarios.

In addition to all mentioned challenges, a major challenge is represented by the requirements on the computational performance of a game. Due to the improving possibilities of gaming hardware and the increasing expectations of players, modern games are expected to run with a so-called frame rate of 60 to 120 frames per second. The frame rate represent the frequency of game updates, leaving only 8 to 16 milliseconds for a full computation of certain game systems. Although there can be systems that are allowed to run their computations less frequently and therefore can distribute one computation over multiple frames, the systems responsible for artificial agents are usually required to update their logic in every frame. Taking into consideration that the provided milliseconds are shared among multiple systems, it leaves even less time for decision-making. Therefore, video games are not only highly dynamic but also very restricted in the provided computation time.

Parameter	Chapter 3	Chapters 5	Chapter 6
Type of environment	academic	commercial	academic
Simulation model available	✓	✗	✓
Number of controlled agents	1	2 – 9	1 – 42
Type of agents	–	homogeneous	heterogeneous
Cooperation required	–	✓	✓
Coordination required	–	✓	✓
Adversarial setting	✓	✓	✓

Table 1.1: Characterization of the environments used in Chapters 3, 5, 6.

An important aspect that counterbalances these limitations is that, from the industry perspective, the agents are usually not required to find *optimal* solutions during decision-making. Since the time and reactivity aspects outweigh other aspects, it is usually enough to find a *feasible* but not necessarily optimal solution. Furthermore, optimal solutions are often even undesired by game developers since they can provide an advantage to the agent and make it much stronger than the player. This can lead to the player loosing the game and increase their frustration. However, as already mentioned, player enjoyment in a game is the highest priority.

Nevertheless, video games are not only used for entertainment. Meanwhile, there are many academic game environments that are used as highly dynamic benchmark environments for artificial agents. These games usually offer simpler graphics than commercial games but focus more on the aspect of agent control. In contrast to commercial games, academic game environments encourage strong (adversarial) agent behaviors and are used in multiple competitions where the performance of different agents can be measured against each other.

Considering the advantages and challenges that they provide, in this work, we focus on video games as highly dynamic virtual multi-agent environments. We concentrate on three different environments, which offer different challenges. The characteristics of these environments are summarized in Table 1.1. The environment used in Chapter 5 resembles a commercial game, whereas the environments of Chapter 3 and 6 are academic game environments. The environments of Chapter 3 and Chapter 6 provide their own simulation models, which can be used by the agents to perform a search, while the agents in Chapter 5 are not provided with such a model. The agents in Chapter 6 are granted full autonomy on low levels of a decision hierarchy, while the higher levels are controlled by predefined rules. In Chapters 3 and 5, we keep full control over the agent's decision logic. The number of agents that are controlled by the proposed solutions vary between 1 (in Chapter 3) and 42 (in Chapter 6). Only the agents in the latter environment are heterogeneous. In Chapters 5 and 6, the agents are required to cooperate and to coordinate their actions while using shared resources. All environments are adversarial and contain other actors that are not controlled by our approaches.

1.3 Goals of the Thesis

This thesis aims to propose a framework that allows for a combination of long-term (strategic) planning and reactive (tactical) execution in highly dynamic environments. Given the problems and challenges described in Sections 1.1 and 1.2, we specifically focus on multi-agent environments. Agents controlled by the proposed framework will be able to continue following a long-term plan while staying reactive and adjusting their behaviors according to the current environmental situation. In order to fulfill this goal, the following sub-goals are defined:

Separation of decision levels and interleaved decision-making Provided the large search spaces existent in highly dynamic multi-agent environments, the major sub-goal is the introduction of a *hybrid* approach that allows for an interleaved use of a planner and a reactive decision-making and execution technique. With this combination, we aim to separate the responsibility and the complexity of decision-making on different levels of a decision hierarchy, allowing the planner to make abstract high-level decisions and the reactive approach to refine these decisions on a lower level.

Reduction of the planning complexity and the replanning rate Due to very limited computational time budgets in highly dynamic environments, an important sub-goal of the proposed solution is the reduction of both the planning complexity and the replanning frequency on high decision levels.

Improvement of the execution In order to ensure robust behaviors, it is important to avoid any delays in the execution. Therefore, another important sub-goal is the improvement of reactive decision-making and the reduction of the overall execution time.

Guidance of the reactive decision-making from the planner In order to ensure a smooth interplay between the two control approaches and to allow the agents to focus on long-term goals even when making reactive decisions, we aim to provide a possibility for the high-level planner to guide the decision-making process of the reactive approach.

Automatic improvement of the guidance In case of guided decision-making of the reactive approach, another sub-goal is the proposal of a possibility to learn and automatically improve this guidance.

The following sub-goals are defined in terms of the research concept of this thesis:

- Proposal of a general, domain-independent approach for planning and execution that can be adapted to any environment.
- Proposal of concrete implementations of this approach.
- Evaluation of the concrete approach implementations.

Intuitive use and maintenance A secondary goal for the concrete implementations is the use of existing decision-making approaches without major changes to them. The intention behind this is the maintenance of the available expertise on such techniques and the desire for an intuitive use of the new approach.

1.4 Structure of the Thesis

The stated goals are addressed in multiple steps in this thesis. First, Chapter 2 introduces basic concepts of the two major areas of this work, planning and reactive decision-making approaches. It gives an overview of the existing work from different research areas and different types of environments describing known problems and available solutions to them.

In the next step, Chapter 3 describes a pre-study on the usage of a pure planner but no reactive approach in a highly dynamic single-agent environment. As a follow-up, Chapter 4 describes the general idea of a hybrid approach that combines a long-term planner with a reactive decision-making and execution approach. It points out major challenges of such a combination and introduces a new solution that is used in further chapters in the form of concrete implementations. Chapters 5 and 6 describe two different hybrid approaches based on the general idea.

The focus of Chapter 5 is on the evaluation of the advantages of a hybrid approach in comparison to a pure planning approach. The hybrid approach of this chapter uses Behavior Trees as a reactive decision-making technique in combination with a planner. The proposed solution is evaluated in a multi-agent scenario where it is controlling up to 9 collaborating agents simultaneously.

Chapter 6 transfers the general idea into a multi-agent environment where it is required to control tens of agents, while dealing with a large search-space. It proposes a hybrid solution combining a planner and Monte Carlo Tree Search granting more agent autonomy on the reactive decision levels. This chapter describes a possibility to represent descriptive plan tasks in a way that can be used by Monte Carlo Tree Search. Furthermore, it proposes a way to automatically improve reactive behaviors under the consideration of the high-level plan tasks.

Finally, Chapter 7 concludes this work summarizing its ideas and insights. It discusses remaining questions and limitations and outlines possible directions for future work.

Chapter 2

Background and Related Work

This chapter describes the two areas that are the focus of this work – reactive decision-making and planning. For both areas, it gives insights into existing work underlining challenges faced in different highly dynamic environments and describing possible solutions. Finally it concludes with a summary of the related work.

2.1 Reactive Decision-Making Approaches

Reactive decision-making approaches are commonly used in highly dynamic environments. In general, such approaches are able to make decisions at run-time. By constantly monitoring the environment and making decisions a on a regular basis, such approaches are able to instantly *react* to changes in the environment and adapt the agent's behaviors accordingly. However, such decisions have to be made within a limited time in order to ensure a fluent execution. For that reason, most reactive approaches do not perform a deep search or plan ahead in time. Instead, they make decisions based on the current environment state and potentially on some limited historical data searching for an optimal solution for the current state.

Well known reactive approaches are Finite State Machines (FSMs) and Markovian Decision Processes (MDPs). Both approaches allow to encode state-action transitions that enable quick *decision-making*. However, they are limited in their functionalities when it comes to search in large search spaces and *execution*. In the following sections we describe two reactive approaches that are used in this work. First, Behavior Trees (BTs) are described as a powerful extension of FSMs. Afterwards, Monte Carlo Tree Search (MCTS) is described as an approach that allows for quick near-optimal search through MDPs with a large search space. We regard the usage of BTs as suitable whenever full control over an agent's behavior is required, for example due to safety or robustness requirements. On the other hand, MCTS is rather suitable for situations where more autonomy from the agent is desired and where instead of fully encoding the decision rules, the decision problem can be formulated as an optimization problem.

2.1.1 Behavior Trees

Behavior Trees (BTs) represent a reactive control approach that originates from the video games industry [4]. Extending Finite State Machines, Behavior Trees are able to constantly monitor the environment, make decisions in every time step and execute durative actions. This way, they offer high reactivity and allow for deliberative behaviors, which are very important when controlling Non-Player Characters (NPCs) in real-time video games [5].

Behavior Trees are a popular choice amongst game developers and have been widely used in most commercial games [6–9]. Being very modular systems, BTs can be composed of different stand-alone modules that can be easily read by humans. Furthermore, BTs preserve safety, robustness, and efficiency [10]. For these reasons, they have been gaining an increasing interest in the area of robotics in the last few years [11–15].

A detailed introduction to Behavior Trees (BTs) can be found in [8], but we will give a short overview here. The traversal of a Behavior Tree starts from a *root node* and the tree is updated in every update cycle - the so called *tick*. The frequency of ticks can be set by developers depending on the dynamics of the environment and the desired computational performance.

In each tick, the tree is updated from the top nodes to the bottom in depth-first order. Each node can either be *running*, which allows for durative actions and looping behaviors, or its execution can *succeed* or *fail* after which it will remain dormant until the parent node decides to start ticking it again. Each node is running as long as at least one of its children is running. A success or failure of a node is propagated to its parent and the parent reacts to this change according to its own node type. BT nodes can represent *conditions, actions* or *composite* nodes that allow for sequential (*sequence* node), parallel (*parallel* node) or selective (*selector* node) execution of behaviors. Furthermore, many implementations of BTs have additional *decorator nodes*, which allow for complex structures such as loops, timers, wait nodes (waiting for a signal) or other, potentially domain-specific functionalities.

In the following, the most common types of Behavior Tree nodes are explained according to [8]:

- Condition:
  This node is used to check information. It can run with the following options: *instant check* or *monitoring.*
  With the *instant check*, the information is checked once the node is initialized and depending on the outcome the node *succeeds* or *fails* immediately.
  With the *monitoring* option, the node *runs* and checks the information as long as the condition is true. Once the outcome becomes false, the node *fails.*
  (Usually, both options can be negated.)

- Action: This node executes an action. It is *running* as long as the action is executed and *succeeds* or *fails* depending on the success of the action execution.
- Composite Nodes:

  Sequence:
  This node is responsible for a sequential execution of its children. It is *running* as long as one of its children is running. It starts with the initialization of its first child and switches to the next child when the first one succeeds. If a child fails, the sequence node *fails* and will not execute the remaining children following the failed node. It *succeeds* when all its children succeed.

  Selector (also known as Fallback):
  This node represents an OR node allowing for different fallback behaviors. The execution starts with the initialization of its first child and the node is *running* as long as one of its children is running. If a child fails, the next child is initialized and tried. A selector node *succeeds* when one of its children succeeds and *fails* when all of its children fail.

  Parallel:
  This node is responsible for the parallel execution of its children. All of its children are initialized at once. Usually, the node implements different possible options for running and termination criteria. Depending on the selected options, the node can either *succeed* of *fail* when either one or all of its children succeed or fail. Until then, then node will be *running.*
- Decorator Node:
  Decorator nodes have only one child and are used to add more logic to the tree - for example, a *Loop.*

  Loop:
  This node repeats the execution of its child. Depending on the selected options, it can either run the child n times, or as long as the child keeps succeeding or failing. As long as its termination condition is not met, this node will be *running.*

Combinations of these nodes allow for very advanced behaviors. For example, checking a condition in a loop leads to a constant monitoring of the environment, whereas executing an action in parallel to such a monitoring branch, allows to react to environmental changes and switch to different behaviors aborting the current action. For example, the Behavior Tree in Figure 2.1 can be read as follows: if there is a free seat, then sit down. Otherwise wait for as long as the seat is occupied and sit down afterwards. Then, if there is a book and some coffee available, read and take a sip of coffee afterwards. Repeat reading and drinking coffee until neither book nor coffee is available.

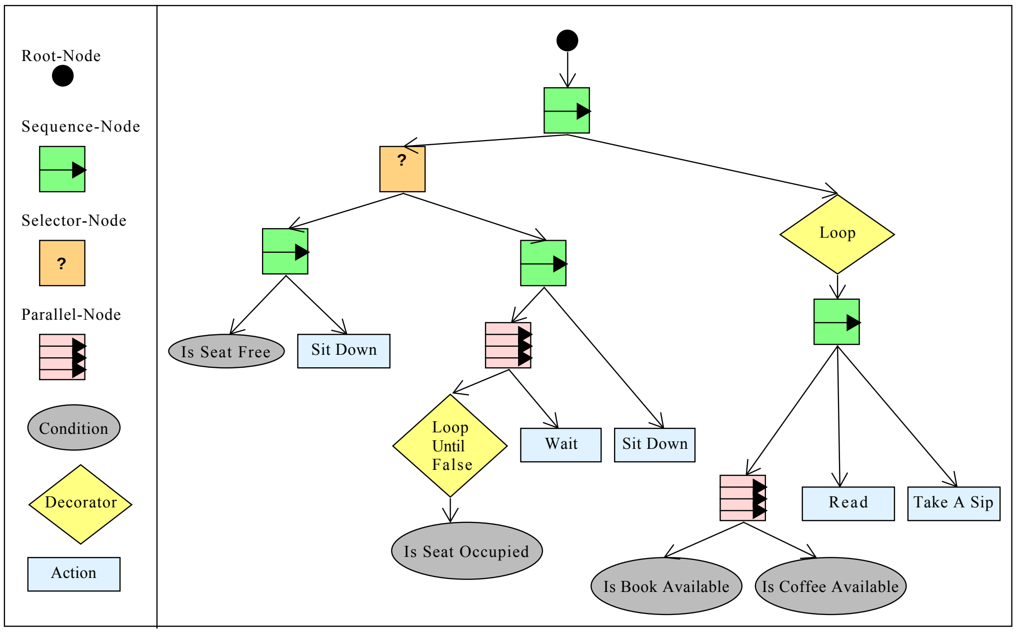


Figure 2.1: Example of a Behavior Tree.

There are different works that extend the functionalities of standard BT nodes or add new node types. For example, the work described in [16] optimizes the Selector node learning the success probabilities of its children and sorting them accordingly. A new Decorator node that enables synchronization between multiple agents executing a common task is added in [17].

Coming from the area of game development, BTs are usually fully hand-crafted by developers offering full control over the agent's behavior. This is an advantage in environments where the agent should not show any unexpected behaviors. In games, full control is usually desired due to two reasons: debug-ability and maintainability. First, knowing the structure of its Behavior Tree, it is easier to debug an agent's behavior and to understand its reasoning. Second, knowing exactly under which circumstances a certain behavior is (not) executed, developers can easily tweak and adapt it to certain situations. For these reasons, BTs can be applied in other environments where full control is required - such as for example environments where unexpected behaviors lead to dangerous situations.

Nevertheless, there are some works that allow for automatically learned behaviors executed through BTs. One possibility is to combine hand-crafted and learned behaviors ensuring that safety criteria are met. For example, the work described in [18] creates Behavior Trees that are responsible for switching between different controllers depending on the task progress and the safety status of the agent. The tree can switch between a reliable model-based controller, a high performing Neural Network controller, and a safe emergency controller. Combining these controllers in a BT, this approach provides efficient execution ensuring that the safety requirement is met during the control of the well-known pole-cart control problem (inverted pendulum problem). Further alternatives to incorporate automatically learned behaviors into BTs are described in the following section.

Automatic Generation and Learning of Behavior Trees

Automatic generation of BTs can be performed with the aim to save the time that a human developer would require for the generation of BTs. Another goal is to find behavior variations that a human cannot find otherwise. Such behaviors can be more efficient, more robust or even more creative than those generated by a human.

One way to create or improve BTs automatically are Evolutionary Algorithms (EAs). Here, an *individual* is usually represented by a single Behavior Tree. In order to automatically evolve the trees, the developer needs to define which leaf nodes are legal, which can be either *condition* or *action* nodes. Thus, defining which conditions/variables an agent is able to check in the given environment and which actions it can execute. One of the pioneering works that applied an Evolutionary Algorithm to evolve BTs for an RTS game is described in [19].

However, the main challenge of applying a simple Evolutionary Algorithm to BTs, is the fact that resulting trees can lose their structure and syntactic correctness and can grow to undesired sizes. This makes the trees unreadable and their execution computationally expensive. For example, an inappropriate selection of cross-over points can lead to infinite loops, wrong type of leaf nodes or checks of contradicting conditions within the same branch.

In order to prevent the creation of such trees, the work described in [20, 21] proposes using grammars to encode syntactical rules and domain knowledge. These grammars can be used for Grammatical Evolution creating BTs that follow the encoded rules.

Alternatively, enforcing a correct structure and an acceptable size on evolved trees can be performed through additional control mechanisms that are used during or after genetic operators. For example, the work described in [22] allows nodes to be changed to other nodes of the same type (for example only composite nodes) during mutation to prevent illegal tree configurations. Additionally it applies an anti-bloat mechanism to the final trees - those trees that achieve a given goal - pruning and removing redundant nodes. Another approach that enforces constraints during evolution is described in [23]. Additionally, in order to prevent the trees from growing to unreadable sizes, *selective pressure* can be used to prefer shorter trees [24].

An initial *population* of BTs can be initialized either manually by a human expert or automatically. In case the EA uses constraints or rules in its genetic operators, it is important to ensure that the initial *population* of trees conforms to the same rules. For the automatic initialization, it is common to *grow* them starting from very small trees (only the root node) and iteratively adding nodes to them either at random [25] or greedily trying to achieve some goal [22]. A more detailed description of an algorithm to iteratively refine a Behavior Tree with the help of a planning algorithm is given in [26].

Further problems evolving BTs can arise if the agent's memory or execution times are limited. For example, a BT can evolve towards the optimal fitness values but result in undesirably long execution times [25]. Similarly, given only a small internal memory, which is usually the case for small mobile robots, the number of variables, tasks, and tree nodes available to it might need to be kept to a minimum [24].

Besides EAs, there are some works that use other learning approaches to either create BTs automatically or to learn how to execute certain behaviors. For example, the works described in [27–29] introduce new types of so-called *learning* nodes. These nodes use Reinforcement Learning to either learn *how* to execute an action [27, 28] or *when/under which conditions* to execute certain behaviors (actions or sub-trees) [28, 29]. Combined with expert knowledge regarding the reward and state representation, these nodes are used within a human-designed BT to improve the execution of simple tasks.

2.1.2 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a best-first tree search algorithm that searches for a near-optimal solution in a large stochastic state space [30, 31]. When used as a decision-making algorithm for an agent, a solution represents the agent's action that is supposed to lead to the optimal outcome. Since the algorithm is usually used in every time step (also called frame or tick), the agent performs the search on the most recent environment data and thus can dynamically react to environmental changes. MCTS was first introduced to solve classic board games and led to great interest in the research area of computational intelligence in video games. In the last decade it has been applied to multiple academic video game environments [32–38]

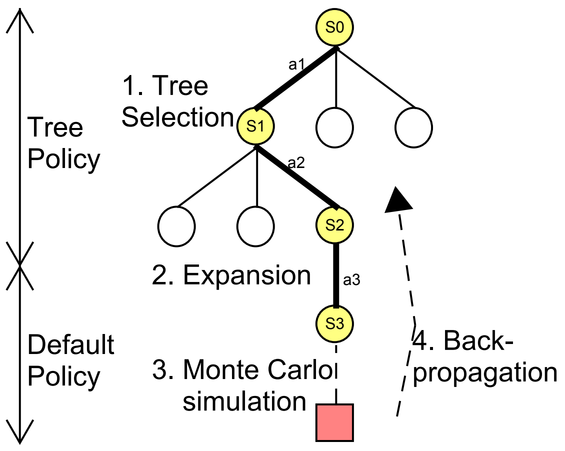


Figure 2.2: Workflow of the Monte Carlo Tree Search Algorithm.

MCTS works under the assumption that the search space can be represented by a Markovian Decision Process (MDP) and a model of this process is available. In the context of MCTS this model is usually called the *forward* or *simulation* model since it is used to simulate an agent's actions forward into the future.

The search process of MCTS is shown in Figure 2.2. The search for an optimal agent action starts with the initial world state $s0$ represented by the root node. Actions are represented by branches and states are represented by nodes. The search tree is built by iteratively executing the following four steps: *selection*, *expansion*, *Monte Carlo simulation*, and *back-propagation*. The search starts by *selecting* an action that is available in the initial state. The selection is done following a *tree policy*. By simulating the selected action, the tree is *expanded* by the *new* node that represents the new state $s1$. Having reached a node from which not all actions have been selected (here the node with the state $s2$), a *Monte Carlo simulation* is performed from this node. Following a *default policy*, which usually selects actions at random, the simulation runs until either reaching a final (game) state or another terminal criterion. The new state reached by the simulation ($s3$) is then *evaluated* with the help of an *evaluation function* (EF, also called reward function) and the resulting reward is *back-propagated* updating the average rewards of all preceding nodes.

The described steps are repeated until reaching a final criterion, for example until exceeding a computational time budget. By repeating the steps, an asymmetric tree is built up iteratively and the final decision is made with the help of the average rewards of actions available in the initial state. For this purpose, each node stores the number of times that its state s was visited ($N(s)$), the number of times that an action a was selected from this state ($N(s,a)$), and the average reward obtained from selecting action a in state s ($Q(s,a)$). These values are updated in each iteration.

In the selection step, each action can be selected uniformly at random. However, usually the selection follows a more complex *tree policy* taking into consideration the stored number of visits and the received rewards. Assuming that multiple actions can lead to near-optimal solutions, most tree policies try to balance between exploration of actions that were simulated less often and exploitation of more promising search branches [39].

Such policies are typical for Multi-Armed Bandit (MAB) problems. A MAB problem presents multiple slot machines. Selecting an arm of one of these machines leads to a random reward and the reward distribution is unknown in advance. The goal is to minimize the *regret*, which is the difference between the theoretical optimal reward and the cumulative reward obtained from selecting arms. When using MCTS, every action represents an arm of a MAB and the average rewards stored in each node represent the payoff of selecting this arm.

A well-known MAB policy is the Upper Confidence Bound (UCB1) [40] or Upper Confidence Tree (UCT) [30], when applied to MCTS. As shown in Equation 2.1, this policy tries to balance between exploitation (left part of the term) and exploration (right part of the term) through the parameter C selecting the action with the highest upper confidence bound.

$$a^* = arg\ max_{a \in A(s)} \left\{ Q(s,a) + C\sqrt{\frac{ln\ N(s)}{N(s,a)}} \right\} \tag{2.1}$$

Multiple early works on MCTS in different RTS games have used the UCT sampling strategy as presented in [33, 35, 37]. These works have shown promising first results. However, their performance deteriorated with the growing number of units.

NaïveMCTS

There are certain environments and problems where a centralized approach is desired to control multiple agents, such as a group of robots that are controlled as one swarm or all units of one player in an RTS game. When a tree search algorithm is applied to solve such a problem, its goal is to find the optimal *combination* of multiple agents' actions. Thus, the branching factor of the search tree becomes combinatorial and the MAB problem is then formulated as a Combinatorial Multi-Armed Bandit (CMAB) problem [34, 41, 42].

Following the notions used in [34, 42], a CMAB is defined by:

- A set of variables $X = \{X_1, ..., X_n\}$ with each variable X_i taking one of the K_i values $\mathcal{X}_i = \{v_i^1, ..., v_i^{K_i}\}$.
- A reward distribution $R : \mathcal{X}_i \times ... \times \mathcal{X}_n \to \mathbb{R}$ that depends on all variables.
- A function $V : \mathcal{X}_i \times ... \times \mathcal{X}_n \to \{true, false\}$ determining which variable value combinations are legal.

When used to solve a common task in a multi-agent environment, each variable $X_i \in X$ represents a single agent and its values represent the agent's actions.

The goal of solving a CMAB problem is to maximize the reward over a legal[1] *combination* of variables ($R(x_1, ..., x_n)$) as opposed to a MAB problem where the reward is dependant on a *single* variable ($R(x)$). However, most tree search algorithms struggle to find optimal solutions of combinatorial problems. A naïve approach to solve a CMAB problem is to assume that the variables (actions of different agents) are independent from each other and the reward over all variables can be approximated as the sum of rewards over single variables ($R(x_1, ..., x_n) = \sum_{i=1}^{i=n} R_i(x_i)$). In this case, each legal *combination* of variable values can be considered to be a different arm of a *global* MAB (MAB_g). Whereas each of these arms (combinations) on its own represents a *local* MAB problem (MAB_i). Each MAB_i selects a value for each of the *variables* $X_i \in X$ maximizing the sum of rewards over single variables. Additionally, MAB_g selects the optimal combination of variable values.

[1]A combination of variables is *legal* when it does not contain variable values that are not possible or not allowed. An example of an illegal combination of actions is two agents trying to simultaneously make a move into the same location/grid cell.

Applied as a tree policy to MCTS, this approach is known as *naïveMCTS* [34,42]. When selecting a legal combination of actions, it first uses a policy π_0 in order to balance between exploration (simulation of less-explored action combinations) and exploitation (selection of the best action combination). If exploration is chosen, a local policy π_l is used to solve MAB_i by selecting an action for each agent independently. Resulting new action combinations are added to MAB_g as new arms. If exploitation is chosen, a global policy π_g is used to select a combination of actions from MAB_g. In [34,42], each of these policies is an ϵ-greedy policy with different ϵ probabilities to select exploration and $1 - \epsilon$ probability to select exploitation.

Since in most real-time environments an agent's actions are durative and multiple agents can execute actions in parallel, naïveMCTS considers these facts when selecting and simulating actions. The selection of actions is only done when there is at least one agent that can execute a new action. Afterwards, the world state is *simulated* applying the selected actions until reaching the new decision point (next time when one of the agents is free to start a new action). Furthermore, in the case of adversarial environments such as two-player zero sum games, the sampling strategy distinguishes between the agent side and tries to maximize or minimize the reward accordingly.

As the results of [34, 42] show, with a growing branching factor found in microRTS (which is described in Section 6.2), naïveMCTS is able to outperform UCT as well as other CMAB sampling strategies such as Matching Learning with Polynomial Storage (MLPS) [41], and Linear Side Information (LSI) [43]. Furthermore, it outperforms other tree search strategies such as α-β Considering Durations (ABCD) search [44].

Further Enhancements to MCTS (and its Apllications to Games)

Although naïveMCTS is able to outperform some earlier sampling techniques in microRTS, in its pure form it is still not able to find optimal solutions in large search spaces, such as those presented in RTS games. The main reason for finding sub-optimal solutions is that given a large action space, the algorithm is usually able to look only a few steps ahead within the very limited avalable time (as for example 100 milliseconds in microRTS). Therefore, it is not able to reach a final game state in order to sufficiently evaluate possible action sequences. In order to overcome these difficulties, different enhancements to MCTS have been proposed.

One solution to avoid exploring actions that are less effective in certain situations is to embed some domain knowledge into the sampling strategy. For example, when exploring the action space in microRTS, the work described in [45] considers a probability distribution of actions played by a strong player. This way, the search is guided by the probability distribution in addition to the reward function. Another way to reduce the considered actions is described in [46]. Here, following the asymmetric action abstraction approach [47], the sampling strategy only considers a limited sub-set of actions of each unit. The sub-sets are created in accordance to the unit type (similar to an agent type in a heterogeneous multi-agent environment) and other unit-specific parameters. Both approaches are able to outperform a pure naïveMCTS in microRTS.

Other approaches that use action abstractions in combination with MCTS are described in [37, 48]. The former work uses an Upper Confidence Tree Considering Durations (UCTCD) tree policy and the latter an ϵ-greedy policy in the game StarCraft. Both works use abstract actions to control *groups* of units instead of giving orders to single units. In addition to abstract group actions, MCTS used in [48] is operating with abstract game states and abstract evaluation functions. These works have shown that abstractions can decrease the size of the search space while still leading to good results.

A different way to manipulate the search process of MCTS is to tweak its parameters such as the maximum search depth, the maximum simulation time, or the parameter that is used to balance between exploration and exploitation [38]. Additionally it is possible to change the default policy used in the Monte Carlo simulation [49] or to adapt the evaluation function used for the evaluation of (game) states guiding the search towards certain solution [50, 51].

2.2 Planning Approaches

In contrast to purely reactive approaches, planning approaches do not only search for a single action that is optimal in the current situation but try to create a plan that is supposed to lead to a long-term goal. Thus, a plan is usually a sequence of actions. To find a solution, a planner uses a planning domain that is predefined by a human expert and contains a model of the environment. Depending on the planning approach, a plan can be created in different ways. For example, a planner can be flat or hierarchical. The following section underlines the differences between classical planning and real-world planning and introduces two hierarchical planners, which are used as a base for the Hierarchical Task Network with Postconditions (HTN_p) planner proposed later in this work.

2.2.1 Classical Planning

In the research area of *automated planning* a *plan* represents a sequence of *actions* (also called operators) to be executed by an agent in order to achieve a *goal*. The creation of such a plan requires the definition of a *planning domain*, which provides relevant information about the environment. In particular, a planning domain usually contains a set of state variables F (also called *facts*), and a set of *actions* A that an agent can execute. Additionally, in hierarchical planning, a domain contains further information about the task hierarchy (see Section 2.2.3).

An action is usually represented by the tuple (pre, add, del). *Preconditions* $pre \in 2^F$ define the set of facts that have to (not) be true in a state for the action to be applicable. The set of preconditions can be divided into two sets: facts that have to be true and those that have to be false before applying the action. However, in the major part of literature, preconditions are represented by one common set pre. For that reason, we stick to this definition. Additionally, *effects* of an action describe which facts are *added* to or *deleted* from a state after executing the action. They are usually represented by the two sets $add \in 2^F$ and $del \in 2^F$ respectively. Preconditions and effects allow a planner to *simulate* changes in the world during the planning phase that will be caused by the execution of an action.

A well-known classical planner that provides the basics for many other planners is the Stanford Research Institute Problem Solver (STRIPS) [52]. It belongs to the planners that search through the *space of states* (in this work the term *state* and *world state* are used as synonyms). Following the STRIPS formulation, a *planning problem* is defined by the tuple $p = (F, A, s_I, s_G)$ with the set of facts F, the set of actions A, an *initial state* $s_I \in 2^F$, and the *goal state* $s_G \in 2^F$.

A widespread definition language for planning problems is the Planning Domain Definition Language (PDDL) [53], which was first used in the 1998 International Planning Competition. Since then, it received multiple extensions such as the possibility to handle durative actions and continuous effects [54] or the combination of hard and soft constraints [55].

A *space-state* planner aims to find a sequence of actions that lead from the initial state to the goal state. It reasons about state transitions checking the preconditions of actions and simulating their effects. A STRIPS-like planner performs a backwards-search starting from the goal state. Trying to resolve the difference of facts between the initial state and the goal state it first searches for an action whose effects lead to one of the facts that exist in the goal state but not in the initial state. When such an action is found, it is added to the ordered plan and its preconditions are added as further sub-goals to be achieved in the state preceding the goal state. Iteratively, the planner adds further actions to the plan until all sub-goals are reached – thus, until the planning state corresponds to the initial state and the plan can be applied starting from it [56, Chapter 4.4], [3].

Classical planners like STRIPS operate on classical planning domains, which underlie the following limitations [3, 57]: a) the set of possible states is finite, b) the environment is fully observable, c) all state transitions are deterministic, d) the system is static so that only the agent's actions can lead to a transition in states, e) the only kind of goals are attainment goals defined as goal states meaning that e.g. states to be avoided are not defined, f) a solution plan is an ordered finite sequence of actions, g) actions are instantaneous transitions between states and have no duration, h) planning is done offline without considering any state changes that appear during the planning time. However, most of these assumptions do not hold in real-world environments and a lot of research has been done focusing on challenges presented in such environments. Some of these challenges are described in more details in the following section.

2.2.2 Real-Time Planning

In contrast to classical planning domains, most real-time environments are non- deterministic, highly dynamic, and partially-observable. For these reasons, plans are likely to fail during execution. Possible approaches for plan failure management and planning under uncertainty are summarized in [58] and include conditional or *contingency* planning [59–61] and probabilistic planning through Bayesian Networks [62] .

Alternatively, online planning and interleaved planning and execution can be used to recognize and deal with plan failures at execution-time (as described in more details in Section 2.3.1). In these cases either plan repairing approaches or full replanning can be used [63]. The former allows to keep a plan and only repair the failed parts of it, whereas the latter generates a new plan from scratch. In the area of video games, full replanning is preferred over plan repair. One reason for this is the additional engineering effort required for a repair approach. Another reason is the potentially high computational time required for a proper repair. In contrast, planners used in video games are usually optimized enough to allow computationally cheap replanning [3].

A popular planning approach used in multiple commercial video games is the Goal Oriented Action Planner (GOAP) [64–67], which is an adapted version of STRIPS. Since its first implementation in the game *F.E.A.R*[2], it has been used in further games such as *Rise of the Tomb Raider*[3] and *Middle-earth: Shadow of Mordor*[4] [3, 68]. In order to deal with the challenges existent in the highly dynamic real-time games, is has made four major changes to STRIPS.

First, it added costs to actions and used them as heuristics in the search process. Second, instead of representing an action's preconditions and effects by lists of facts (as described above), it represented them by fixed-sized arrays allowing for faster comparisons of world states. Third, it added the possibility to not only check variables as preconditions, but also to call code functions in order to perform more complex checks (such as finding a path). And lastly, instead of applying effects as instantaneous state changes during the plan execution, a Finite State Machine was controlling the states of the agent and updating the planner's world state.

In order to reason about time and deal with time-dependent and parallel actions, there are different *temporal planning* techniques [69–72]. However, temporal planning is – to the best of our knowledge – not used in the area of video games and is out of scope of this thesis. Instead of considering temporal constraints during the planning phase, we rely on recognizing and dealing with plan failures caused by temporal dependencies during the execution phase.

Another challenge when planning for real-world environments, is the large (continuous) state space. In order to deal with such a search space in real-time, it can be beneficial to reduce its size by embedding some domain knowledge into the planning algorithm itself. To achieve this, it is possible to create *domain-specific* planners [57]. As opposed to *domain-independent* planners, which can work with *any* planning domain, or *domain-configurable* planner that receive the planning domain description *and* the planning problem description as input, domain-specific planners encode domain information in themselves. Using the information to decrease the size of the search space, they are able to work in a more efficient way in the specified domain. However, they cannot be used for a different domain.

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

[3]Rise of the Tomb Raider: Developer: Crystal Dynamics, Publisher: Square Enix, 2015

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

Further disadvantage of domain-specific planners is the maintenance of the domain. If it frequently changes or needs to be corrected, it requires programmers to change the planner code directly. Whereas for a domain-configurable planner, whose domains are defined in some high-level planning language (such as PDDL), anyone can change the planning domain without modifying the planner itself. Thus, depending on the size of the planning domain and the planning problems, the available manpower, the expected requirement in maintenance on the domain, and the desired performance of the planning algorithm a domain-configurable or a domain-specific planner can be a better choice. In commercial video games, both approaches have been successfully used so far [3]. Although domain-independent planners are less popular due to the difficulty of building a domain compiler that can interpret the high-level planning language, there are some successful applications of such planners [73]. On the other hand, most academic planners are domain-independent.

An important aspect when planning for real-time environments is the coordination of multiple possibly heterogeneous agents. Whether in video games, service robotics, evacuation planning or autonomous vehicle control, the agents are usually acting within a larger ecosystem and have to either coordinate their actions or at least interpret and react to the other agents' actions. Thus, not only planning but also the execution is an important aspect at this point. For that reason, the topic of multi-agent planning (and execution) is discussed in Section 2.3.2.

2.2.3 Hierarchical Task Network Planning

In contrast to flat planners that search the space of states, *hierarchical planners* search the space of plans. In particular, a *Hierarchical Task Network (HTN) planner* is based on the idea of task decomposition. Instead of searching for a sequence of actions that leads to a *goal state*, an HTN planner tries to find a sequence of subplans that *perform a certain task.* For this purpose, it is searching for a feasible decomposition of the task.

In HTN planning, tasks can be recursively decomposed into further subtasks and thus build a hierarchical network. Those tasks that can be further decomposed are called *compound* tasks (or *abstract* tasks), whereas *primitive* tasks represent the actions that can be directly executed by an agent and thus cannot be further decomposed. (Some related work uses the term *operator* to describe what kind of actions can be executed by an agent [74–76]. In this case, a primitive task represents a *ground* instance of an operator. Thus it is an operator instance with values assigned to its parameters. In this work, we use the terms *operator* and *primitive task* as synonyms.)

Following previous work [77–79], we denote the set of compound tasks as C and the set of primitive tasks as A. An HTN planning domain is defined as $D = (F, C, A, M)$ where, similarly to a classical planning problem, F is a set of state variables or *facts*. An HTN planning problem is defined by the tuple $p = (D, s_I, c_I)$, where $s_I \in 2^F$ is the initial state, and $c_I \in C$ is the task to be performed. The Hierarchical Task Network is defined as $tn = (T, \prec, \alpha)$ with a set of possibly empty task identifiers T, a strict order of tasks $\prec$, and $\alpha : T \rightarrow A \cup C$.

When a planner receives a compound task to be performed, it is possible that this task can be performed in different ways. In order to describe *in what way* and *under which conditions* a certain decomposition is possible, the so-called *methods* (represented by the set M) are used. A method $m \in M$ is defined by the tuple (c, pre, tn) with the compound task $c \in C$ that it decomposes, the preconditions $pre \in 2^F$ under which it is applicable, and a network of subtasks (compound or primitive) tn that it decomposes into. Multiple methods can be assigned to decompose the same compound task. Using the methods' preconditions, a planner can decide which method is applicable in its own internal simulated state.

For example, in Figure 2.3 the initial compound task c_I is *EnterRoom(agent_A, room_A)* requiring agent A to enter room A. This task can be decomposed by the methods *EnterThroughDoor* and *EnterThroughWindow*. In the first step, the planner checks the first method's precondition, which is *IsReachable(door_A, agent_A)*. Since $S0$ (the initial state in this case) contains a fact indicating that the door is reachable by the agent, this method is is selected for the decomposition of the task. It decomposes into the primitive task *GoToDoor(agent_A, door_A)* and another compound task *StepInThrough-Door(agent_A, room_A, door_A)*.

When a method decomposes a compound task into a primitive task, this primitive task is added to the final plan. The planning process runs until all subtasks are decomposed and the plan contains primitive tasks only. Similarly to actions in classical planning (see Section 2.2.1), primitive tasks are defined by the tuple (pre, add, del) – their preconditions and effects that are added or deleted to the state ($add \in 2^F$ and $del \in 2^F$ respectively). Thus, before adding a primitive task to a plan, an HTN planner checks the task's preconditions and, if the task is applicable, adds it to the plan. Afterwards, the planner updates its internal state representation with the task's effects. That way, the planner can keep track of the state changes and reason into the future.

In the above example, the planner checks the precondition of the *GoToDoor(agent_A, door_A)* task, which, in this case, is the same as the method's precondition and therefore holds in the initial state. After adding this primitive task to the final plan, the updated state $S1$ contains another fact *IsNextToDoor(agent_A, door_A)*. Using this information, the planner can further decompose the remaining compound task *StepInThrough-Door(agent_A, room_A, door_A)* following the same rules.

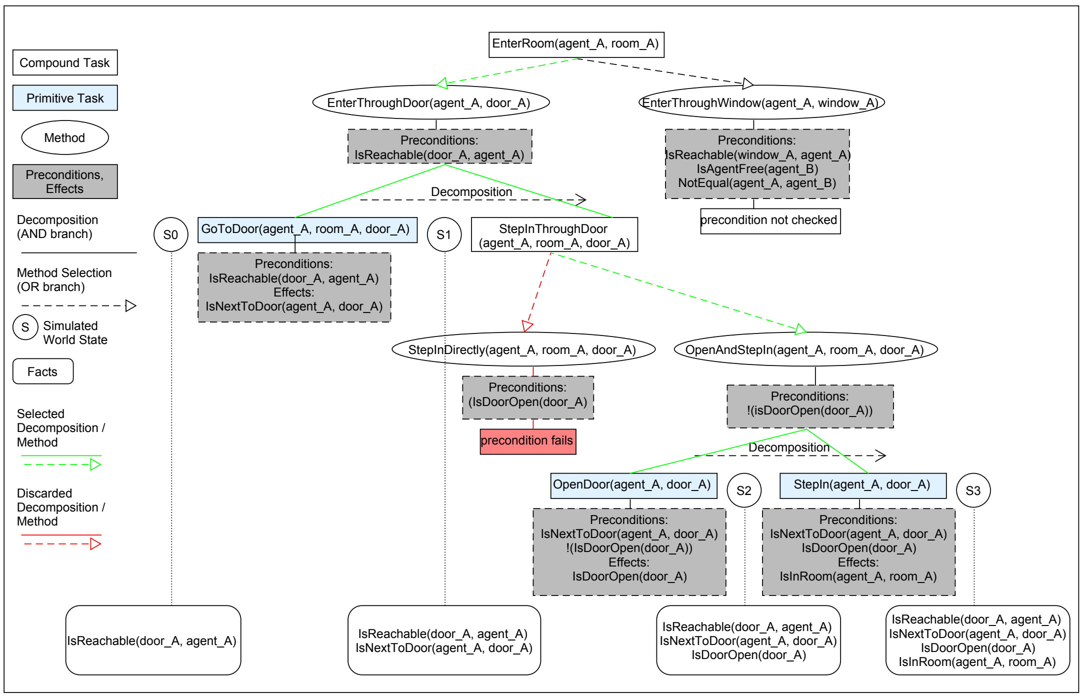

Compound Task
Primitive Task
Method
Preconditions, Effects
Decomposition (AND branch)
Method Selection (OR branch)
S Simulated World State
Facts
Selected Decomposition / Method
Discarded Decomposition / Method
EnterRoom(agent_A, room_A)
EnterThroughDoor(agent_A, door_A)
EnterThroughWindow(agent_A, window_A)
Preconditions: IsReachable(door_A, agent_A)
Preconditions: IsReachable(window_A, agent_A) IsAgentFree(agent_B) NotEqual(agent_A, agent_B)
precondition not checked
Decomposition
S0
GoToDoor(agent_A, room_A, door_A)
S1
StepInThroughDoor (agent_A, room_A, door_A)
Preconditions: IsReachable(door_A, agent_A) Effects: IsNextToDoor(agent_A, door_A)
StepInDirectly(agent_A, room_A, door_A)
OpenAndStepIn(agent_A, room_A, door_A)
Preconditions: (IsDoorOpen(door_A)
precondition fails
Preconditions: !(isDoorOpen(door_A))
Decomposition
OpenDoor(agent_A, door_A)
S2
StepIn(agent_A, door_A)
S3
Preconditions: IsNextToDoor(agent_A, door_A) !(IsDoorOpen(door_A)) Effects: IsDoorOpen(door_A)
Preconditions: IsNextToDoor(agent_A, door_A) IsDoorOpen(door_A) Effects: IsInRoom(agent_A, room_A)
IsReachable(door_A, agent_A)
IsReachable(door_A, agent_A) IsNextToDoor(agent_A, door_A)
IsReachable(door_A, agent_A) IsNextToDoor(agent_A, door_A) IsDoorOpen(door_A)
IsReachable(door_A, agent_A) IsNextToDoor(agent_A, door_A) IsDoorOpen(door_A) IsInRoom(agent_A, room_A)

Figure 2.3: Example of an HTN decomposition.

Usually, effects are reversible so that if a certain method or primitive task cannot be used in later states, the planner is able to backtrack to the previous compound task removing all effects caused by the applied method. After failing to use one method, the planner searches for a different method to decompose the compound task. If no other method is applicable or available, the planner backtracks further up in the decomposition hierarchy.

There are different possibilities for a planner to select the order of subtasks to be decomposed [80]. The most intuitive technique is the *total order* approach. Here, subtasks are decomposed in the same order in which they will be executed later on [75, 81]. In Figure 2.3 that is from left to right resulting in the following plan: *GoToDoor, OpenDoor, StepIn*. The advantage of a total-order decomposition is that the planner plans forward from the initial state and always operates on the updated state. Also, it does not require any explicit information about the tasks' dependencies.

Alternatively, subtasks can be decomposed in a *partial order* allowing subtasks of different compound tasks to be interleaved during execution [76]. In this case however, the tasks have to be either independent from each other or their dependencies have to be explicitly defined in the planning domain. An advantage of partial task decomposition is the possibility to parallelize the decomposition of multiple tasks. Furthermore, more critical or less computationally intensive precondition checks can be performed first, potentially saving some expensive computations. However, since in this case the planner is no longer planning strictly forward from the initial state, the final order of tasks in the plan requires additional checks of dependencies between preconditions and effects of the interleaved tasks.

One of the most-known HTN planners is the Simple Hierarchical Ordered Planner (SHOP) [75] and its extended version Simple Hierarchical Ordered Planner 2 (SHOP2) [76]. Both approaches implement *total-order* task decomposition. Additionally, SHOP2 allows for *partial-order* of tasks enabling interleaved sub-task. Both planners are widely used in various industrial and academic applications such as evacuation mission planning and UAV control [82].

Independently from its decomposition order, once an HTN planner is able to apply a method to decompose a task, it is not taking into consideration other methods. This way, the planner is able to prune the search space early in the search process focusing only on a small part of it. Therefore, having a well-designed HTN allows for efficient planning even within very limited computation time, which is one reason why HTN planners are used in multiple video games such as *Killzone 2*[5] [83], *Killzone 3*[6] [73], *Transformers: Fall of Cybertron*[7] [84], *Horizon Zero Dawn*[8] [85].

Another advantage of HTN planning is its similarity to the way humans think. When we are facing a task, we naturally sub-divide it into smaller tasks. For that reason, creating, reading and understanding an HTN domain is more intuitive in comparison to a classical planning domain. Additionally, the plan's execution is better understandable by a human, which is a very important aspect in video games as well as in robotics [2].

[5]Killzone 2: Developer: Guerrilla Games, Publisher: Sony Computer Entertainment, 2009
[6]Killzone 3: Developer: Guerrilla Games, Publisher: Sony Computer Entertainment, 2011
[7]Transformers: Fall of Cybertron: Developer: High Moon Studios, Publisher: Activision, 2012
[8]Horizon Zero Dawn: Developer: Guerrilla Games, Publisher: Sony Interactive Entertainment, 2017

There are some works on learning parts of an HTN planning domain [86–91]. Most learning approaches are based on explanation-based learning from expert traces. These approaches try to find causal relationships between changes in the environment and the tasks executed by a human expert and build an HTN using these relationships. Some earlier methods concentrate on learning only certain parts of the domain. For example the work presented in [89] learns preconditions of methods, [90] learns method preconditions and action models, [86, 87, 91] learn method structures, whereas [88] focuses on learning the methods that optimize the quality of plans integrating Reinforcement Learning (RL) into an HTN planner. Some other works concentrate on learning hierarchical *goal* structures (see Section 2.2.3) [92, 93].

The major question when reasoning about the domain structure is how to combine tasks into higher-level tasks. Additionally, in order to correctly generalize their preconditions and effects it requires a large amount of training data to extract knowledge from. Although existing approaches provide promising results, most of them are tested in simple static benchmark environments from planning competitions. Therefore, it is unclear whether they are scalable for more complex problems. For these reasons, learning a hierarchical planning domain for a large dynamic environment still remains a challenging problem.

There are many other aspects of HTN planning that are important to consider for certain environments. However, summarizing all of them would exceed the target of this thesis. Some survey papers can be found under [74, 94].

Hierarchical Goal Network Planning

Although Hierarchical Task Network planning formalism is quite expressive, its major flaw in comparison to classical planning is that its tasks are defined as actions that are to be performed but they have no semantic meaning. A task can be executed but there is no notion of a task being finished or achieved. Thus it is impossible to monitor their progress during execution and to check their fulfillment against the current state of the environment [95]. In the previous HTN example from Figure 2.3, the agent can *try to* open a door but the planner will not know whether the door was actually opened.

A Hierarchical Goal Network (HGN) [96,97] solves this problem by combining hierarchical decomposition with the classical planning formalism. Instead of dealing with *tasks*, an HGN deals with *goals* or goal states, which, similarly to classical planning, are described by facts. Formally, an HGN planning problem is defined as $p = (D, s_I, s_G)$ where D is an HGN planning domain, and S_I and S_G are the initial and the goal state respectively. An HGN planning domain is then represented as $D = (D', M)$ where $D' = (F, A)$ is a classical planning domain (with a set of facts F and a set of actions A) and M is a set of methods. However, in contrast to HTN methods, HGN methods are not assigned to compound tasks. Instead they only describe *which goals* they achieve. A method $m \in M$ is defined by the tuple $(pre, sub, post)$ where preconditions pre are similar to those used in HTNs, $sub =< g_1, ..., g_k >$ is a sequence of subgoals to be achieved with $g_i \in 2^F$, and $post \in 2^F$ are the postconditions of a method. If sub is non-empty, then $post = g_k$, otherwise $post = pre$. Actions are defined in the same way as in classical planning (see section 2.2.1). As shown in [96], an HGN is as expressive as an HGN.

A Goal Decomposition Planner (GDP) [96] solves an HGN planning problem by iteratively decomposing and resolving subgoals. Similarly to classical planning it tries to resolve the difference in facts between the initial state and the goal state. To achieve this, it searches for actions and methods that are relevant for the goal state, either directly adding actions to the final plan or adding subgoals by applying a relevant method. An advantage of an HGN over an HTN is that it does not require the domain to be complete in the sense that it is possible to not define all necessary methods. If no method is available to divide a goal into subgoals, the planner can apply classical planning techniques by directly searching through actions.

An extended version of GDP uses so-called *landmarks* to infer such subgoals if a top-level goal cannot be solved by any method [98]. In an HGN planning problem, landmarks are subgoals that every solution to a planning problem must satisfy. (Similarly in an HTN planning problem landmarks are tasks that have to be decomposed in every solution to the problem [99]). Additionally, it is possible to add heuristics to the classical planning search [96]. Further approaches combine HTNs and HGNs into so-called Goal-Task-Networks [97], which operate on a mixed planning domain containing elements of both network types.

2.3 Planning and Execution

A planner is usually responsible for plan generation only and does not take care of the plan execution. This is, however, problematic in real-time environments and especially in highly dynamic environments since environmental changes can invalidate the plan very quickly and a new plan can be be required within a short time period. The problem becomes even more complex when planning is performed for multiple agents. For these reasons, there are different approaches to interleaved planning and execution. The following section introduces the major problems and advantages of different architectures for interleaved planning and execution .

2.3.1 Interleaved Planning and Execution

Even though there is a lot of research being done in the area of automated planning, many works do not take into consideration the execution phase underestimating the importance of deliberation during the execution. Not only the two phases, planning and execution, are separated in time, but also are they usually performed by two different systems, a planner and an execution system. An execution system is usually responsible for environmental monitoring, simultaneous execution of actions, possible coordination, and failure management.

A related area, in which planning and execution plays an important role is spacecraft control. However, it is important to note that spacecraft and planetary robots are usually the only agents that operate in the environment and thus do not have to handle high dynamics of the environment, which are the topic of this thesis. Possible uncertainties of the environment are usually covered either through plan contingencies or by relying on on-board error detection systems [100], which are lower layers of the architecture described in the following paragraphs. The applications in this area are mostly divided between those that use model-based approaches (planning) for on-board decision-making (such as described in [101]) and those that use Markovian Decision Processes. The major advantage of model-based approaches is the generation and maintainability of their domains, which describe goals and problems logically with the help of some planning language. However, their weakness can lie in the time required to make a decision, since finding a plan happens online.

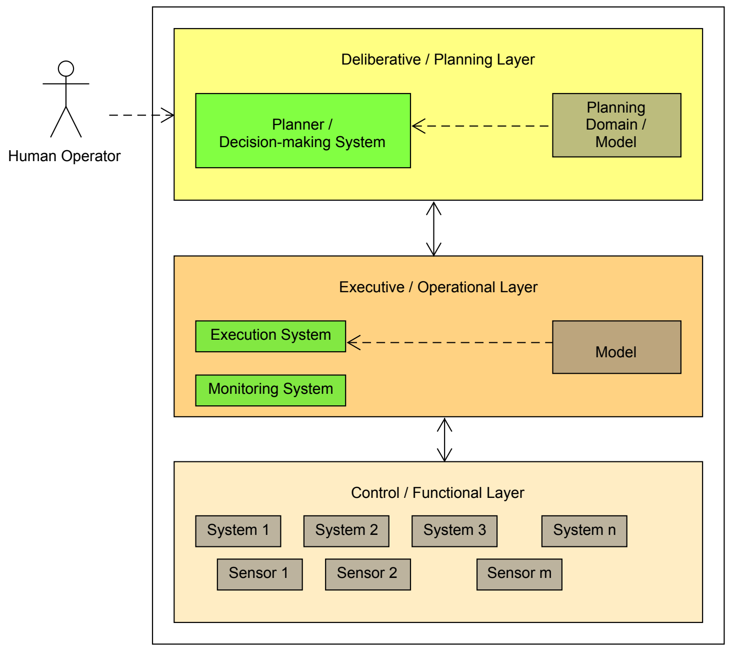


Figure 2.4: Example of a general three-layer architecture for spacecraft control.

In contrast, policies of MDPs can be generated offline but they require a sufficient amount of information on the probability distribution of state-action-transitions, which is not always available [102]. Furthermore, with an increasing size of the state-space, MDPs face the curse of dimensionality as well as the curse of modeling. Thus, even though they are used in robotics, they struggle to find application in practical cases in the area of spacecraft control [102]. Even though there are some works that combine model-based reasoning with MDPs, the solutions are still not ready to be used in real space-missions [102]. For these reasons we only concentrate on model-based approaches in the following paragraphs.

A typical approach towards full autonomy in model-based spacecraft control implements a three-layer architecture [103] shown in Figure 2.4 including a *deliberative* or *planning* layer that is responsible for decision-making (which can be either fully autonomous or incorporate human-sent objectives), an *executive* or *operational* layer that is responsible for monitoring and command execution, and a *control* or *functional* layer that reads specific sensors and controls specific subsystems on the lowest layer [102].

Depending on the underlying approach and its level of autonomy, these layers are exchanging different kinds and amounts of data, and each of these layers has a certain degree of autonomy and memory. Although high-level reasoning and planning for spacecraft control has been mostly done by humans (ground operators) so far, full autonomy is becoming increasingly important. Autonomous failure management is especially important in the time windows when communication with the ground is not possible. A detailed overview of recent space projects involving model-based mission planning and execution is provided in [102].

When it comes to *execution*, traditional spacecraft and rovers are usually controlled by a small set of high-level commands using a simple execution language. Many *execution systems* used in this sub-area do not allow for complex behaviors like loops, conditions, iterations, concurrent and sequential activities, or time- and event-driven activities. Some of such systems either use an internal planner or can be coupled with an external planner. Some approaches that are most similar to our work are presented in the following paragraph and a brief overview of further execution systems for spacecraft and robots can be found in [100].

The *Universal-Executive* system [104] represents a tree-based execution approach which is very similar to Behavior Trees (see Section 2.1.1). Here, each node is defined by pre-, post-, and invariant-conditions and can have different states such as *executing* or *finished*. Similarly to Behavior Trees, the expressiveness of the Universal-Executive system allows for concepts such as loops or concurrent activities, where internal nodes are responsible for complex compositions and leaf nodes represent actions. The input of the system is a domain description written in the *Plan Execution Interchange Language* (PLEXIL) [105], which makes the system itself independent of the domain it is being used in. The Universal-Executive system can be used as a stand-alone system as well as in combination with an external planner.

Instead of connecting to an external planner, some systems implement a hybrid approach implementing an internal planner [106]. In these cases a reactive execution system requests the internal planner to come up with a plan that achieves a certain goal chosen by the system. The planner then either generates a plan for the whole system or it is used to find optimal sequences of actions of only one subsystem in order to achieve a certain system configuration [106]. One problem with the latter approach, however, can be unpredictable behaviors caused by the concurrent handling of weakly-interacting subsystems [106].

A major problem in most systems so far remains the definition of models for reactive execution in the operational layer and their connection to the higher decision layer [102]. A planner usually operates on a *descriptive* or *declarative* action model. That means that an action representation describes *what* an action does, for example *which effects it has on the environment.* Whereas an executive system usually uses an *operational* action model, which describes *how* an action can be performed. Some approaches for integrated planning and execution try to bridge the gap between the different representations used by the planner and the execution system. For example the *Intelligent Distributed Execution Architecture* (IDEA) [107] uses declarative models on all control layers while the *Program Planning and Execution Language* (PROPEL) [108] works with operational models only.

Similarly, some works outside of the area of spacecraft control such as the works described in [109, 110] see the major problem for truly interleaved planning and execution solutions in the different ways of knowledge representation used by the planning and executive systems. In order to interleave the two systems, the work described in [110] proposes the usage of a unified language both for planning and execution. What is first described as an abstract idea, is implemented later with the help of the *Refinement Acting Engine (RAE)* [111, 112], which is responsible both for planning and execution. Using an operational action model for both phases, the engine *refines* (or decomposes) tasks into smaller sub-tasks, in a similar way to SHOP. However, since the planner now also uses operational action representations, which are programs that *execute* an action, it is required to *simulate* these programs to reason about their outcomes.

In this case, the executive part of the engine receives the high-level task to be performed and every time that the agent is making a decision on how to refine a task (thus which method to select), it calls the planner to reason about possible outcomes of the different refinement methods. The planner simulates possible sub-plans and forwards the result of its decision to the executive system for execution. For the simulation, the planner performs Monte Carlo rollouts over the different refinement methods. In its simple version, the executive part waits for the planner to complete its search (i.e. to fully refine/decompose all possible methods). However, since this can take a long time and potentially stop the execution, the later works [112, 113] experiment with different variations of the proposed solution, for example, applying a UCT-based approach for method selection instead of a planner. Still, the proposed approaches require more than 10 *seconds* for a single computation [113] and are therefore not yet applicable to highly dynamic environments.

A related approach for agent control in games is presented in [114]. Here, a planner is extended by Behavior Trees to simulate the plan's outcomes during the *planning* phase. However, this work does not provide a description of the plans' *execution.*

Another area that is closely related to our work and requires interleaved planning and execution is robotics [2]. However, similarly to spacecraft control, most earlier approaches in both areas implemented solutions for specific agents or environments - for example for a specific spacecraft or a specific robot. Even though they were very effective and successful in that implementation, they did not use standardised languages, which made their integration with other systems more difficult. An important work that proposes a modular architecture that combines two popular standards in robotics is *ROSPlan* [115].

ROSPlan provides an interface for any PDDL 2.1 based planner that can handle the syntax of the domain and Robot Operating System (ROS) [9] [116]. ROS is used in many robotic systems in the industry as well as in academia. ROSPlan has two ROS modules: a *Planning System* and a *Knowledge Base* [115]. The Knowledge Base is updated using real information such as coordinates. Additionally, it translates continuous data provided by sensors into symbolic discrete descriptions that can be used by the planner allowing for continuous monitoring through sensing actions.

The Planning System uses the information from the Knowledge Base and generates the planning problem, the initial and goal states. These are then forwarded to an external planner. Additionally, the Planning System is responsible for the generation of a so-called *planning filter* that considers static facts and the plan's actions' preconditions. This filter is then used by the Knowledge Base to be checked against the current world state. In case any inconsistency is found, the Knowledge Base notifies the Planning System, which in turn handles the replanning. Furthermore the Planning System is responsible for the translation of the high-level PDDL actions into low-level control actions (ROS messages) using pre-existent libraries for robotics domains.

Many works in the area of robotics that incorporate planning are focusing on *path planning* since the problem of navigation is still a major concern. Other works focus on *motion planning*, which is important for grasping robots. However, these types of planning are different from the long-term deliberative task planning that this thesis focuses on. The sub-areas of robotics that are most relevant for us are the so-called *social robots* [117] - robots that interact with humans and thus act in highly dynamic environments and make long-term decisions *autonomously* and *deliberatively.* Such robots are service robots and robots that work cooperatively with humans - the so-called *cobots.* To the best of our knowledge, however, interleaving deliberative task planning and execution has not been deeply studied in these sub areas since they are quite new.

[9]ROS: `https://www.ros.org`

One such work is described in [118]. It uses a three-layer architecture to control a cobot. The cobot receives multiple tasks to be performed from a human user. The highest architecture layer schedules these tasks, the middle layer controls and monitors the execution of single tasks, and the lowest layer takes care of the path planning. In addition to the typical failure detection, this work focuses on the detection of *opportunities* that can arise during the execution of a plan. For example, if the robot is given multiple high-level tasks it can interleave their subtasks in different ways. Starting with an initial execution schedule, the robot is able to detect situations in which a subtask that is scheduled for later can be executed immediately adapting the schedule. Seizing such opportunities allows to decrease the total execution time of all plans.

The approach described in [119] proposes a combination of Behavior Trees with HTN planning in a robotic scenario. This work aims to decrease the execution time of plans through parallelization of certain subtasks and elimination of unnecessary actions. It does so by using BT-like parallel nodes, and HTN-like pre- and post-condition checks during a hierarchical plan creation through task decomposition. In contrast to pure HTN planning, the resulting plans have tree structures defining sequenced and parallel execution of tasks. However, in contrast to pure BTs, the order of branches in the tree is selected dynamically during the planning phase taking into consideration the tasks' preconditions. In comparison to predefined trees, the dynamically created trees lead to shorter execution times.

As already mentioned in Section 2.2.2, an alternative to interleaved planning and execution can potentially be probabilistic or contingent planning. Probabilistic planning approaches deal with probabilistic action effects [120]. Considering these probabilities, a planner tries to find a plan with the highest success probability. Doing so, it creates multiple possible contingencies of a plan. During execution, these contingencies can be used for decision-making considering the current state of the world. In this case, replanning is not required if a fitting contingency exists. Even though the advantage of these approaches is the elimination of replanning, the major disadvantage is the increased planning time that results from the creation of all contingencies. Additionally, in highly dynamic environments that involve human actors, the approach needs to deal with a large branching factor of possible contingencies, for which it is impossible to compute the probabilities.

In order to be able to plan *for* different contingencies without actually decomposing them at plan-time, the work described in [121] postpones decision-making on conditional branches until execution-time through so-called *assertions*. Assertions are abstract actions whose purpose is the gathering of information that is unknown at *plan-time*. Similarly to other actions, they have preconditions and effects. Additionally assertions are defined by special conditions that describe when replanning - or rather plan refining - should take place. When these conditions are met at execution-time, plan refinement is triggered and the abstract action is replaced by concrete actions using the gathered information. This way, the planner is more efficient than a full contingency planner at plan-time. It can create an abstract plan even in environments with uncertainties and postpone the local decision-making to later stages.

2.3.2 Multi-Agent Planning and Execution

When speaking about multi-agent planning, there are different aspects that are important to be considered such as whether the agents are cooperating or not, whether they are communicating and whether this communication takes place only at plan-time, only at execution-time, or both. It is possible that the planner is centralized and thus no communication is required during the plan-time but the agents still *act* autonomously, i.e. there is no central coordinator of their actions during the plan execution. In this case the agents have to synchronize their actions on their own. This can be done either explicitly through communication or implicitly by observing the actions of other agents. In the latter case, the agents can be required to reason about the other agents' beliefs, goals, and plans.

Planning with beliefs is the basis for dealing with uncertainty in *epistemic planning* [122–124]. In this sub-area of planning, an agent is supposed to be able to reason about his knowledge as well as the lack of it. With this reasoning, an agent is further able to use *beliefs* about possible world states in order to plan for knowledge gathering actions and to create conditional plans. Additionally, in a multi-agent environment with decentralized planning, agents can have beliefs about other agents' capabilities and their knowledge [123]. In order to achieve coordination, agents can then reason at plan-time about possible plans of other agents and their knowledge. If an agent beliefs that another agent lacks some knowledge required to execute its (part of the) plan, the first agent can share this knowledge.

Additionally, multi-agent epistemic planning has a concept of *nested beliefs*. That means, an agent A can belief "that agent B beliefs that agent A beliefs X". Such nesting can, in theory, have an arbitrary depth. However, since this can largely increase the computational complexity of the planning algorithm, the depth can be bound [122]. Nested beliefs are usually used for *implicit coordination* between multiple agents at *execution-time*. In order to reason about possible actions of an agent B that might contribute to an agent A's plan, agent A needs to reason from agent B's perspective. This is usually done through so-called *perspective shifts* [123].

Although epistemic planning provides some interesting ideas for implicit coordination at execution-time, most experiments with it have been performed in quite simple test environments so far [125]. Thus, it is unclear whether it can be performant enough for highly dynamic environments, a high number of agents, and large search spaces. Furthermore, if *planning* is done in a decentralized manner and implicit coordination takes place at execution-time only, agents can come up with incompatible plans. This will require replanning, which can lead to livelocks [123].

With am imcreasing number of agents, communication between all agents in a tightly-coupled system becomes a combinatorial problem, whereas loosely-coupled systems require less coordination and are thus easier to plan for [126]. Some works try to save the cost of coordination *at plan-time* by using a centralized planner. However, a disadvantage of a centralized planner in a large-scale multi-agent environment in comparison to decentralized planning is the increased planning cost. In order to prevent expensive planning, many works use abstract world models on the higher levels of centralized hierarchical planning.

In order to be able to plan with simplified abstract models and not to lose important information, there are techniques for information compression. For example, the works described in [127,128] implement a centralized hierarchical planner that creates abstract high-level plans for a team of agents and enables tighter coordination of agents at the lower levels during execution through plan merging. The abstract plan contains subplans for each agent. Each subplan holds so-called *summary information* about its own and its children's conditions, which *must* or *may* hold at certain points of time during the plan's execution.

These summary information are then communicated between agents during execution and used for coordination of their subplans. The coordination of plans is done by one of the agents reasoning about all involved plans' compatibility, trying to merge them. When merging plans, the agent is taking into consideration temporal relations such as *must start before* or *must finish after* and the usage of shared resources [129]. However, the search for possible plan merging also becomes combinatorial and it is unclear how well it is suitable for high numbers of agents and long-term plans. Additionally, as pointed out in [127] the search for plan combinations does not always determine correct possibilities for overlapping. The generation of correct rules for temporal relations involves a high engineering effort, which is not feasible for certain applications, such as video games.

In the following, we concentrate on loosely-coupled multi-agent systems where as little as possible communication is required for an effective *execution* of plans that contribute to common goals. A great part of research on multi-agent coordination is working with *centralized planning* approaches and *implicit* coordination at *execution-time*. Although there are some works that consider decentralized planning[126, 130] and loosely-coupled execution.

An important subarea of research in robotics, which works with multi-agent problems quite similar to the ones explored in this work is focused around the *RoboCup* competitions[10]. RobCup represents different football leagues where teams of robots compete against each other. These competitions require competitive as well as cooperative behavior with dynamic role assignments to multiple heterogeneous agents in a highly dynamic and non-deterministic environment. Although *task* planning is less complex in such games and a greater importance is given to *navigation* planning.

Most approaches presented in RoboCup implement a hierarchical architecture that allows for a centralized abstract team coordination and reactive individual behaviors. Such architectures are similar to the three-layer architecture described in Section 2.3.1 with the difference that the highest layer is responsible for centralized decision-making for the whole team. One popular variation of such an approach is the *Skills, Tactics, and Plays (STP)* architecture [131,132]. Here, the highest level of a control hierarchy is responsible for the so-called *Plays* that define, for example, which robot plays the ball and which robot should try to receive it. On this level, the system assigns roles and objectives to all team players. Selecting Plays from a predefined pool – the so-called *Playbook* – is also adapted in further variations of hierarchical architectures such as [133].

In the next control layer – the *Tactics* layer – each robot makes decisions autonomously following its objective. This can be done through FSMs [132] or any other reactive approach. Finally, the lowest hierarchy level implements basic *Skills*, which are similar to simple actions such as moving or kicking. Depending on the robot's architecture further systems are required for navigation and motion control [131].

[10]RoboCup: `https://www.robocup.org/`

However, in contrast to our problem definition, a typical high-level RoboCup task usually does not involve long-term planning or planning at all. In most cases it is based on reactive high-level decision-making [134], for example through Play selection. In these cases, a Play is selected by considering the current world state and potential statistics about the Plays' previous success rates [131]. Once a Play is executed (or aborted), a new one is selected in a reactive manner.

More complex approaches that incorporate Playbooks outside of RoboCup are described, for example, in [135, 136]. The former work presents an early study on a user-selected Plays for teams of UAV agents in military scenarios. Mission plans are then created by an HTN planner for the whole team. This work focuses on constraint satisfaction for resource control and coordination of actions. For example, simultaneous execution of tasks of multiple agents can be defined through additional constraints on these tasks. The work mentions that in order to meet these constraints during the execution an agent would have to *wait* for another agent. However, no further description of the plan's execution is provided. The latter work describes a Playbook-based planning-and-execution approach that combines a SHOP2 planner and a reactive monitoring and execution system for multi-agent firefighting scenarios [136]. Here, the planner creates an approximate team plan reasoning with a discrete representation of the world. In the next step, this plan's parameters are translated into a continuous representation to be used during execution.

Multi-agent planning and execution in the area of video games is mostly required for coordination of *opponent* agents that are supposed to attack the player in groups. There are some well-known examples both for decentralized planning and centralized planning used in games. When using decentralized planning, coordination on abstract levels is easier to implement for loosely-coupled game scenarios. However, it becomes more complex for tightly-coupled coordination on more detailed levels. In general, there is always a trade-off between the flexibility on different coordination levels and the computational cost [127, 137].

With decentralized planning, agents usually have limited abilities to reason about the group and thus need more communication in order to achieve tightly-coupled coordination, which is costly and thus rarely implemented in games. Usually, interesting group behaviors in games are rather resulting from emergent agent behaviors than from planned coordination [138]. The best example for the use of decentralized GOAP (see Section 2.2.2) is the game *F.E.A.R* [65]. For example, what a player could perceive as a complex squad behavior in this game, was the result of emergent behaviors and vocalized dialogues between the opponent agents. However, these dialogues did not represent *actual* negotiations between agents but were rather added *after* all agents' behaviors were selected independently [64].

In contrast to decentralized planning, the use of a centralized planner allows for tighter-coupled coordination of a group members' actions on a higher level. Furthermore, hierarchical structures of agents as well as hierarchies of goals are natural for many game scenarios, such as military squads (or animal herds) with strategic as well as tactical goals. For these reasons, centralized coordination by a single *squad entity* or even multiple hierarchical layers can be used to plan group maneuvers and assaults, to send commands to group members on lower levels and to synchronize their actions [139, 140]. In this case, the central coordinating squad unit can reason with a high-level representation of the world and monitor the overall squad situation. Dealing with high-level world states and actions can decrease the computational costs on the squad level planning. Each individual can then reason about his own situation prioritizing between incoming squad commands, opportunities and their own basic goals such staying alive [140, 141].

The complexity of a central command unit can vary from simple selection of a squad mission to actual planning of a squad maneuver [142, 143]. Similarly, the individual behaviors of agents can be implemented either through a reactive approach, such as an FSM [144] or with the help of another planner that deals with a more detailed planning domain. For example, in the game *Killzone 2* the hierarchy contained 3 layers [141]. On the highest layer, a *strategy* was selected though a global designer-defined policy. On the middle layer, *squads* were build dynamically based on the availability and distance of agents. Each squad was using a multi-agent HTN planner sending high-level tasks to individual agents. Finally, on the lowest layer, every agent used its own HTN planner prioritizing and planning for either its own needs, general combat or the orders from the squad.

Since game environments are highly dynamic, generating *long-term* plans and monitoring their progress during execution is very hard. Instead of continually checking the feasibility of such plans, some approaches tend to replan in fixed time intervals ensuring that the plan is based on relatively new information, similarly to the approach described in [145]. For example, replanning in *Killzone 2* was done on at a frequency of $5Hz$ with some precautions taken to prevent unnecessary checks and oscillating behaviors [141]. The current plan was replaced by a new one if the new plan was found to be better or if the current plan became infeasible or failed to be executed. In order to interleave planning and monitoring and to prevent abrupt action cancellations, the individual task networks contained so-called *continue-branches*, which were responsible for smooth transitions between actions [141].

Some possible optimizations of multi-agent hierarchical planning in games are proposed in [143]. These include the use of heuristic-based searches such as *A** during the task decomposition. For example, the cost of a possible sub-plan can be computed as the expected execution time of the plan. When selecting a method to decompose a task, *A** can select the best sub-plan based on its estimated time. Additionally, this work proposes to decompose high-level tasks that are already fully grounded and thus do not rely on outputs of preceding tasks first, instead of using total-order decomposition. Since high-level task have a greater impact on the feasibility of the plan, recognizing failures on higher levels can prevent some unnecessary backtracking during plan creation [143].

The approach described in [146] implements a planning and reactive execution architecture for an agent acting in a Navy simulation environment. It uses an HTN planner and a so-called *state transition system* for execution of plan steps. When a plan is created, it is assigned a set of *expectations* about the world state during its execution. A monitoring systems compares the actual world state against these expectations and, if discrepancies are detected, tries to generate an explanation for the plan failure before triggering replanning. Due to the hierarchical nature of HTNs, this approach also allows for multi-agent planning, as shown within a shooter game environment in [147].

There are some academic works that combine case-based planning for high-level goal selection and BTs for reactive acting of multiple agents in RTS games [148–151]. In these cases, a goal is selected from a case base containing games played by humans. The approach described in [148, 149] uses different manager systems to control sub-areas of an agent's behavior on a tactical level. For example, one manager system is responsible for build order selection and another one for the economy. Then, either micromanagement behaviors are used for low-level control of each unit separately or squad-level behaviors are used for control of unit groups. All these types of behaviors are predefined manually and stored in a behavior library. Once a goal is selected, it is added to a so-called *Active Behavior Tree*, which is build dynamically selecting and adding the corresponding behaviors from the behavior library to the tree. This way, one single tree is used to control all sub-areas of a game and all units of a player. The work described in [151] also tries to automatically build BTs for the execution of high-level tasks provided by the case base. However, it does not provide experimental results due to the inability of the created BTs to properly perform low-level commands.

In contrast, the approach presented in [150] uses predefined BTs for the execution of tactical behaviors in an RTS game environment. Given a high-level goal selected from a case base, the tactical layer chooses among multiple Behavior Trees that are able to achieve this goal measuring the *similarity* between the current game state and the BTs' preconditions. The most similar BT is then added to a pool of running BTs for execution. This work, however, describes a theoretical approach without providing any empirical evaluation.

As already pointed out in [150], the major drawbacks of case-based approaches in general is the requirement for case bases that are large enough to cover all possible situations and the difficulty of identifying (sub-) plans and distinct actions performed by the human players in order to create corresponding low-level behaviors.

Another approach that uses HTN planning in an RTS game environment is *Adversarial Hierarchical Task Network* planning or *AHTN* [152]. This work combines planning and Minmax search to plan for a player's as well as its opponent's actions minimizing the player's loss. The main extension of the standard Minmax search here is the addition of hierarchical decomposition of nodes. This way, the tree switches between min and max nodes only when a plan is decomposed up to a primitive task, which can be directly simulated. This approach plans for all units of a player. However, in contrast to most other approaches described above, the units are not following the long-term plan that is created once. Instead, planning is performed every time that a unit can start executing a new action. This way, even though a full plan is provided to the unit, it only uses its first action and the rest of the plan is dismissed in the next replanning.

2.4 Conclusion

In this chapter, we have provided an overview of the two major focus areas of this work: reactive decision-making and planning. We have given a detailed insight into Behavior Trees and Monte Carlo Tree Search as reactive approaches and Hierarchical Task Network planners. Subsequently, we have discussed some hybrid architectures that combine both areas.

Behavior Trees provide a modular and intuitive way for human developers to manually generate reactive agent behaviors. Due to the high modularity, Behavior Trees allow for complex and yet robust behavior combinations. Over years they have been successfully used in the area of game development allowing humans to keep full control over an agent's decision-making strategies. Recently, there has been an increasing interest in Behavior Trees within the area of robotics, where first approaches to automatic generation of Behavior Trees have been introduced. An agent using a Behavior Tree is very reactive in its decision-making and execution at run-time, however it does not plan into the future and usually requires an additional system for coordination and cooperation with other agents.

On the other hand, Monte Carlo Tree Search provides a way to make reactive decisions taking into consideration possible future outcomes. Since the approach is based on action-simulation and search within a state-action space, it allows for autonomous decision-making without the need to pre-define agent behaviors. However, it requires a forward model (simulation model) of the environment that it operates in. Its extension called naïveMCTS allows for centralized *multi-agent* decision-making with durative and concurrent actions. However, with a an increasing size of the search space and a limited computation budget it suffers to simulate far enough into the future to be able to take long-term goals into consideration.

In contrast to reactive approaches, which provide the agent with a single action, planners create sequences of tasks that are supposed to lead to a long-term goal. Hierarchical planners, such as the Hierarchical Task Network planner, offer an intuitive way to generate behaviors for multiple agents. However, in highly dynamic environments, such plans are likely to fail within a short time during their *execution*. For that reason, it is necessary for real-world applications to combine planners with reactive approaches allowing to interleave decision-making on different abstraction and control levels.

In this section, we have given insights into some hybrid architectures used in environments such as spacecraft control and robotics. Most of such approaches are based on a three-layer architecture where the top-most layer is responsible for high-level long-term planning while the middle layer takes care of more detailed reactive control. The lowest layer usually represents the distinct systems responsible for perception and action execution.

These architectures offer great insights into possible problems and solutions to be considered in our work. However, most of these approaches are hand-tailored for specific environments that are less complex than those provided in video games and thus they do not fulfil all goals described in Section 1.3. Some of them neither consider multi-agent scenarios nor deal with large search spaces nor high dynamics of the environment. In the following chapters we propose different approaches that are inspired by the existing architectures while aiming to combine their advantages and avoid their disadvantages.

Chapter 3

HTN Planning in a Highly Dynamic Game

As we have seen in the previous chapter, interleaving planning, monitoring, and execution is of great importance when aiming for long-term planning in highly dynamic environments. In the following chapter, we introduce a two-layer architecture for interleaved planning and execution. This architecture is used to control a single agent within a highly dynamic adversarial video game environment. In the following sections, we describe the goals of this chapter, present the game environment and the proposed architecture, and discuss the results of the performed experiments. The ideas and results described here were previously presented in [153].

3.1 Goals

Before focusing on very complex scenarios including multiple cooperating agents and large search spaces, it is important to consider simpler *highly dynamic* environments. For that reason, we first concentrate on a single-agent scenario with the following goals.

G 1: Analysis of the agent performance using a pure planner in a highly dynamic environment. As already shown in the previous chapter, interleaving planning, monitoring, and execution is important in highly dynamic environments. Many existing works propose using a three-layer architecture that *combines* a planner and a reactive approach in order to allow for abstract long-term behaviors as well as more detailed local decision-making. However, our first goal is to evaluate whether a *pure* planning approach that is directly connected to the monitoring and execution mechanism can be sufficient for deliberate and reactive decision-making. Therefore, we aim to evaluate whether a two-layer architecture (without an additional reactive approach) can lead to a good performance of an agent.

G 2: Analysis of the advantages of long-term planning in comparison to reactive approaches when used in a highly dynamic environment. Our second goal is to analyze the advantages and disadvantages of long-term planning in a highly dynamic game environment in comparison to purely reactive approaches. In order to directly compare these approaches and to be able to measure their effects on the agent's behavior, we focus on the usage of the so-called *combos*. Combos are sequences of actions that have to be executed *in a certain order*. For that reason, we argue that the execution of such complex sequences requires *long-term plans* and cannot (or only to a limited extent) be done with a purely reactive approach. For that reason, we aim to analyze the success of the execution of such combos comparing our approach to reactive agents.

3.2 Test Environment

FightingICE is a simplified game environment built for research purposes that is provided for the *Fighting Game AI Competition*[1][154], which takes place annually since 2013. Developers and researches can submit agents that play the game and are evaluated against other competitors. The environment is very similar to most traditional fighting video games. Two opponent characters fight against each other using different *skills*. Each character starts with a certain amount of Health Points (HPs) and energy. When a character takes damage it loses HPs. Executing an attack skill can consume energy, which can be restored by hitting the opponent. A game round ends when the Health Points of a character reach zero or after 60 seconds. A full game consists of 3 rounds. The game *arena* is represented by a two-dimensional limited space.

There are three different characters in FightingICE, *ZEN, GARNET*, and *LUD*. Each character's skills are executed under different conditions and lead to different effects. Furthermore characters differ by their attributes such as jump or attack ranges. All these data are fully observable to the agent as well as to its opponent. They are static throughout the game and can be accessed through the so-called *Character Data*.

Additionally, the game environment provides the so-called *Frame Data*. A *frame* is a time interval in which the agents as well as the game can perform necessary computations. The *Frame Data* contains information about the game in a certain frame such as both characters' locations, their vertical and horizontal speed, number of their energy and health points, and information about currently executed actions. FightingICE runs at 60 frames per second, which means that an agent has approximately $16,6$ milliseconds to make a decision and forward an action to the game.

[1]Fighting Game AI Competition: `www.ice.ci.ritsumei.ac.jp/~ftgaic`

Actions are durative and, depending on the type of action, the execution of an action is divided into three different phases: *startup*, *active*, and *recovery* phase. During the first two phases of an attack skill no other action can be executed and the agent cannot change the direction or speed of its character. Depending on the skill, the third phase can be aborted by certain other skills. This means that an agent is not required to forward an action to the game in every frame but only when this is possible. Furthermore, some skills can only be executed if the character has the required amount of energy.

Actions of an artificial agent simulate an input of a human player, which can be done through the keyboard. Depending on the desired skill, a human player is required to press multiple keys simultaneously. Agents can combine the required inputs into *Key Input* data.

A major challenge presented in this game environment is an artificial delay of 15 frames with which the *Frame Data* is provided to the agents. Instead of getting the actual current game state, the agents receive the game state that existed 15 frames earlier. This is intended by the competition organizers to simulate the delay in the reaction of a human player and makes planning, monitoring, and execution even more difficult. The agent has to either deal with outdated information or try to approximate the current world state by simulating the last 15 frames while having to deal with even more uncertainty.

Each character can execute attacks that either *knock-back* or *knock-down* the opponent. In both cases the opponent character is playing the *falling* and *recovering* animations. During this time, this character is uncontrollable, which is giving its opponent a possibility to deal even more damage. A good strategy for a fighting game is executing multiple such attacks in a sequence. Such sequences can be created by a planner.

Furthermore, what makes this environment interesting for us is that it provides a way for an agent to achieve better results by executing a sequence of certain skills. Such sequences are known as *combos* and are a well-established element of fighting games. In order to count as a combo, certain attack skills have to be executed in a predefined order and every two consecutive attacks have to be performed within 30 frames. There are 14 different combos in FightingICE and each combo consists of 4 attack skills. By executing a complete combo, an agent deals more damage to its opponent than the cumulative damage dealt by the distinct skills used in the combo.

When an agent successfully executes the first skill of one of the combos, a combo counter for this agent starts. However, the opponent can abort a player's combo by executing one of the predefined counter-attacks, so called *combo breakers*, between the completion of the second and the fourth combo attack. In this case, the agent performing the combo receives extra damage instead and its combo counter is re-set to zero. Furthermore, if the agent executes an attack skill that is not part of the combo sequence or exceeds the time limit, its counter is re-set as well. Both agents have access to the counters and can react to the opponent executing an attack. However, similarly to other game information, the counter is also updated with a 15 frames delay.

There is some work that investigated the improvements of an agent's performance through combos in a different fighting game environment [155]. However, there are two major differences between the environment used in the mentioned work and FightingICE. First, combo sequences used in the mentioned work were unknown in advance. For that reason, the work focused on *finding* possible combos. Second, in contrast to the environment used in the mentioned work and some other game environments, being hit by combo attacks in FightingICE does not make the opponent character uncontrollable (also called *stunned*). In FightingICE, a character is still able to move and execute actions while receiving damage through combo skills.

In FightingICE, it can be enough to use a purely reactive approach for *aborting* the opponent's combo. For that, the agent has to monitor the opponent's combo counter and quickly react to it with a *combo-breaker* skill. However, in order to *execute* a *complete* combo it is not enough to be purely reactive. The agent needs to *plan for* a certain combo and *continue executing* the selected sequence of skills over some period of time. For that reason, the agent described in the following section incorporates a planner.

3.3 HTN Fighter

In order to use the advantages of long-term planning in a real-time environment, it requires an architecture that allows to interleave planning, monitoring, and execution. Our agent incorporates a two-layer architecture, which contains a planning layer and a monitoring and execution layer. This agent participated under the name *HTN Fighter* in the 2017 edition of the *Fighting Game AI Cometition*[2], where it scored 8th and was partially described in our previous work [153]. The distinct components of the *HTN Fighter* are described in more detail in the following sections.

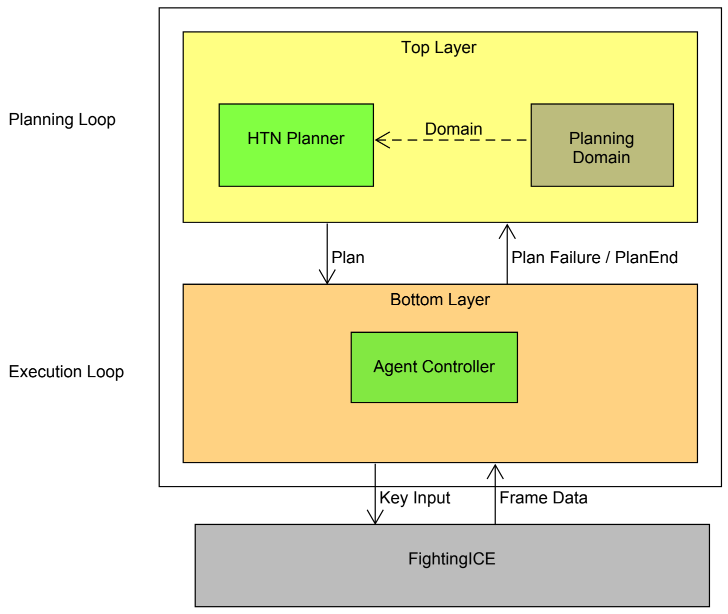


Figure 3.1: Two-layer architecture for the HTN Fighter.

3.3.1 Two-layer Architecture

The two-layer architecture of *HTN Fighter* is shown in Figure 3.1. It allows for interleaved planning and plan execution. Furthermore, it monitors the environment and the plan progression recognizing plan failures and triggering replanning. In contrast to a common three-layer architecture, there is no additional layer responsible for low-level decision-making. Instead, the planner is responsible both for high- and low-level planning and decision-making. There are two update loops connecting the two layers with each other and the game environment. These loops run at different frequencies. In the execution loop, the *Agent Controller* receives an update from the game environment in every frame, whereas the communication with the planner happens only when a new plan is required.

[2]Results of the 2017 Fighting Game AI Competition:
`http://www.ice.ci.ritsumei.ac.jp/~ftgaic/index-R17.html`

Receiving the delayed *Frame Data* in every frame, the *Agent Controller* is continuously monitoring the environment. In order to counterbalance the information delay, it is approximating the current game state by simulating the game 15 frames forward from the delayed *Frame Data*. While following a given plan, the *Agent Controller* monitors the environment for incoming attacks to which the agent potentially needs to react. Furthermore, before executing a new task, the *Agent Controller* checks the task's applicability in the approximated game state. If the task's preconditions are invalid in this state or the agent is required to react to a certain attack, the *Agent Controller* requests a new plan from the planner. Otherwise, the agent forwards the *Key Input* data that corresponds to the task to the game environment. All details of this process are described in Section 3.3.4.

3.3.2 Planning Domain

As already mentioned in Section 2.2.2, there are different ways to create a planning domain. In classical planning, a planning domain is usually defined in some planning language, such as for example PDDL (see Section 2.2.1). A *domain-independent* planner is then able to work with *any* domain that is defined in the specific planning language that the planner is using. However, since a planner itself is written in some programming language, it requires a domain compiler to translate the domain into executable code. In contrast, it is possible to define the planning domain directly in the programming language used by the planner. With such a pre-compiled domain, there is no need to translate it from the planning language into code. For that reason, we use a pre-compiled domain that is written directly in Java.

Similarly to the definitions used in Section 2.2.3, we denote an HTN planning domain by the tuple $D = (F, C, A, M)$, where C is a set of compound tasks, A is a set of primitive tasks (or actions), and M is a set of methods. Each *task* (both compound and primitive) and each *method* in the HTN of the *HTN Fighter* is represented by a corresponding class. Since the characters in FightingICE always face towards each other and each skill represents a specific *Key Input*, there is no need to parametrize the tasks in the planning domain. Each skill can be represented by a corresponding task without any ambiguities.

Furthermore, since the domain is fully pre-compiled, there is no need to define the set of facts F. Instead, it is possible to directly use the data provided by the game to represent the environment. These data include the previously mentioned *Frame Data* and the *Character Data* and represent the game arena as well as character-specific facts. In order to read some additional facts, which are not directly obtainable from the game, we have created helper functions that can be called by the planner at plan-time. For example, to receive the distance between the two game characters, the planner can call a function that computes the distance using the coordinates of the characters.

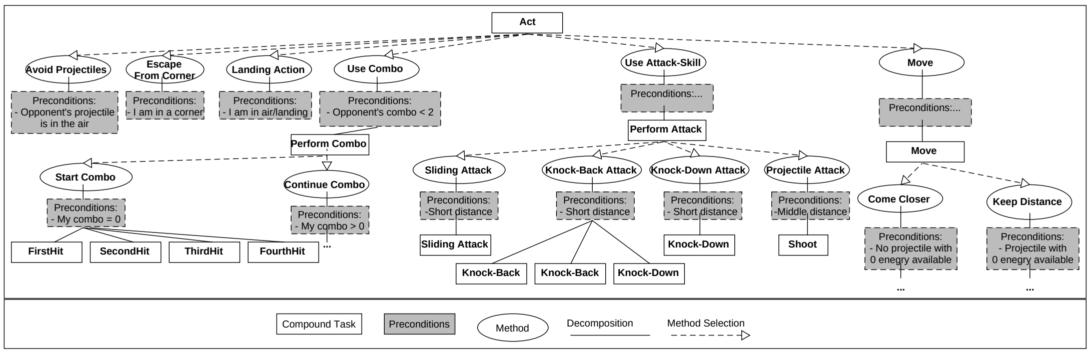

Act
Avoid Projectiles
Escape From Corner
Landing Action
Use Combo
Use Attack-Skill
Move
Preconditions: - Opponent's projectile is in the air
Preconditions: - I am in a corner
Preconditions: - I am in air/landing
Preconditions: - Opponent's combo < 2
Preconditions:...
Perform Combo
Perform Attack
Start Combo
Continue Combo
Preconditions: - My combo = 0
Preconditions: - My combo > 0
FirstHit
SecondHit
ThirdHit
FourthHit
Sliding Attack
Knock-Back Attack
Knock-Down Attack
Projectile Attack
Preconditions: -Short distance
Preconditions: - Short distance
Preconditions: -Middle distance
Knock-Back
Knock-Down
Shoot
Come Closer
Keep Distance
Preconditions: - No projectile with 0 enegry available
Preconditions: - Projectile with 0 enegry available
Compound Task
Preconditions
Method
Decomposition
Method Selection

Figure 3.2: High-level HTN for HTN Fighter [153].

Changes that are expected to happen after executing a task are usually simulated during the planning phase through *effects*. Such changes in the environment can be simulated in FightingICE directly through manipulations of the *Character Data* and *Frame Data*. However, in order to be able to backtrack during plan creation, these changes should be reversible and the planner should keep track of changes made by each task. For that reason, during the planning phase, the planner keeps a *copy* of these data before adding a new primitive task to the plan. In case the effects of this task need to be reverted, the planner can fall back on the saved copy of the data.

However, it is not necessary to pre-define task effects in the planning domain for FightingICE. Instead, actions can be directly simulated by the planner, as described in more detail in Section 3.3.3. For that reason, in contrast to the definition used in Section 2.2.3, primitive tasks in *HTN Fighter*'s domain are defined by their preconditions $pre \in 2^F$ only and do not include the effect lists add and del.

Another difference between the common representation of an HTN domain and the domain of *HTN Fighter* is the way how methods and compound tasks are related. As described in Section 2.2.3, a method is usually defined by the tuple (c, pre, tn) including c – the compound task that it decomposes, its preconditions pre, and the task network that it decomposes into. There can be multiple methods that decompose the same task. Therefore, the planner needs to search for applicable methods during plan time, or bind them to compound tasks in advance. Instead of defining methods by compound tasks, we assign applicable methods to each compound task directly. Thus, a compound task is defined by a list of methods $M' \in M$, whereas each method is now defined only by (pre, tn).

The high levels of the *HTN Fighter*'s task network are shown in Figure 3.2. However, the full network used for the *HTN fighter* contains 26 compound tasks, 104 methods, and 45 primitive tasks. Although the execution of combos is the main goal of this work (represented by the *Use Combo* task), an agent is required to act deliberately in every possible situation. That means that in addition to combos the planning domain should also include strong single attacks (*Use Attack-Skill* task), avoidance of incoming projectiles (*Avoid Projectiles* task), and actions that are can be executed when the character is landing after a jump (*Landing Action* task). In general, an agent who is trapped in a corner is usually more vulnerable to opponent's attacks since it has little chances to escape them. Therefore, the agent should also avoid *getting too close to corners* in first place (*Escape From Corner* task). Furthermore, the agent can be required to move towards or away from the opponent using the *Move* task. The highest goal is always winning the game.

When decomposing a compound task and using ordered method selection (which is described in Section 3.3.3), the order of the task's methods plays an important role. It defines which preconditions are checked first and consequently which of the possible sub-tasks are more likely to be executed. For example, in the domain in Figure 3.2, the preconditions for *Avoid Projectile* are always checked first. Therefore, whenever they hold, this method will be used and all following methods will be ignored even if their preconditions hold as well. However, the attack skills of the 3 game character in FightingICE have different character-specific characteristics (i.e. different attack ranges and damage amount) and different energy requirements. For that reason, the order of the methods that decompose the leaf tasks in Figure 3.2 is not predefined in code. Instead, the planner adds methods to these tasks dynamically at game initialization considering the character's skill specifications. Additionally, it sorts the methods by the damage amount that the corresponding skills deal. For example, the *Knock-Down* task will decompose into some methods *a, b, c, d* for the character *ZEN* and into some other methods *b, d, f, a, e* for *GARNET*. Following this approach, it is not necessary to create a static planning domain for *each* character. Instead, it is possible to create a universal domain, which is static for all characters on the *higher* strategic levels and can be dynamically adapted to each character on the *lowest* tactical levels.

There are two different high-level tasks that represent strategic behaviors and create plans that contain *more than one* action. Besides the task *Use Combo*, which can be decomposed into one of the 14 possible combos for each character, the compound task *Knock-Back Attack* leads to a sequence of 3 attack skills containing attacks that either knock back or knock down the opponent. The strategy behind this is to make the opponent uncontrollable for some time while dealing additional damage to them.

Furthermore, in order to move the character into the attack range of a certain skill, each attack method can decompose into a movement action *and* the actual attack action. This is necessary if, for example, a previous action pushes the opponent back, so that the agent cannot reach the opponent from its previous position anymore. A good plan will first move the character closer to the opponent and then trigger an attack. Therefore, at this point, a plan containing 4 combo attacks can reach a length of up to 8 actions.

3.3.3 Top Layer

Since our first goal is to investigate whether a planner can work in a specific highly dynamic environment, namely FightingICE, the planner described in this chapter is domain-specific. It is written in Java and uses the pre-compiled domain described above. Using a domain-specific planner means that the planner cannot be used for a different environment without making modifications to it. However, it allows to encode domain-specific information into the planner itself. For example, since the game environment limits the time that an agent can use for decision-making, the planner constantly checks the time passed since the beginning of the frame. If it is unable to come up with a plan within the limited time, which is 16.6 milliseconds, it aborts the planning process. In this case, the Agent Controller is responsible for assigning a default action to the character (see Section 3.3.4).

As already mentioned, it is not necessary to pre-define task *effects* in the planning domain. Instead the planner can make use of a simulator that is provided with a sample agent called *mizunoAI* [156] and simulate the task directly on copies of the *Frame Data* and the *Character Data* for both characters. Although, this way, the planner only simulates the agent's plan tasks and the action that is currently being performed by the opponent. It does not predict the opponent's future actions. For example, if the opponent was jumping when the simulation started, the simulator only predicts its landing trajectory and assumes that the opponent will stand still afterwards. There is some work that tries to predict the opponent's actions considering its previous moves [156]. This is a possible improvement for future work. However, since the focus of this work is not a general improvement of our agent's performance but the investigation of improvements through long-term strategies in the form of combos, we do not implement any prediction mechanism yet.

Using copies of these data to represent the world states between distinct tasks, the planner is able to backtrack during the planning process and to apply different decomposition methods if required.

The planner implements total-order decomposition, which means that tasks are decomposed and added to the final plan in the same order that they will be executed in. Furthermore, as already mentioned, the methods that decompose one compound task are either predefined in the domain or assigned dynamically in a certain order. As opposed to some planners that select a method randomly, we have implemented two different strategies for method selection in the planner: an *Ordered Method Selection* and a *UCB Method Selection.*

Ordered Method Selection The most simple and common way to select a method to decompose a compound task is to examine all this task's methods in their order of definition in the domain. When representing the domain as a tree as in Figure 3.2 this order is left-to-right. Following this approach, the first method whose preconditions hold is selected to decompose the compound task. Only if the decomposition of this method fails will the planner continue checking the preconditions of further methods that decompose the same compound task. Ideally, the methods should be defined in an order that prioritizes more critical tasks allowing their preconditions to be checked first. A major advantage of this selection is the avoidance of unnecessary checks of preconditions of *all* applicable methods and the early reduction of the search space. However, this selection leads to a *feasible* plan only but not necessarily to an *optimal* plan. Nevertheless, in the area of video games the improved computation time outweighs the optimality criteria.

UCB Method Selection Instead of minimizing the computational efforts by checking methods in a predefined order, the aim of the UCB1 selection approach is to find more optimal methods for a task by trying out all of them and evaluating them. This selection approach is inspired by the UCB1 selection policy applied in Monte Carlo Tree Search as described in Section 2.1.2. However, in contrast to MCTS, which gathers the required evaluation values over multiple *simulations*, we propose using this approach over a certain period of time using *actual* execution of resulting tasks.

$$m^*(c) = arg\ max_{m \in M_c} \left\{ V(c,m) + K\sqrt{\frac{ln\ N(c)}{N(c,m)}} \right\} \tag{3.1}$$

The planner can use Equation 3.1 to select the best method $m^*(c)$ for the compound task c out of all methods M_c that decompose c. This way, it will try to balance between exploitation of more promising methods (left addend) and exploration of all methods (right addend) through the value K. To evaluate the quality of a method, the value $V(c, m)$ is used. This value can be computed in different ways, depending on whether the goal is to select methods that lead to more damage, those that are successfully executed more often or any other objective. The values $N(c)$ and $N(c, m)$ represent the number of times that the compound task c was decomposed so far and the number of times that the method m was used to decompose c accordingly. By decreasing K over some period of time, for example throughout a match, the algorithm can first allow for stronger exploration at the beginning of the match and focus on more optimal methods towards the match end. Alternatively, this process can be applied over multiple games.

The numbers counting how often a method or a compound task was selected for a plan can be increased directly during the *planning* phase. However, in order to evaluate the quality $V(c, m)$ of a certain method, the *execution* of the plan that it led to needs to be observed. Similarly to back-propagation in a *MCTS* tree (as described in Section 2.1.2), the quality value obtained after the execution of a primitive task needs to be back-propagated to the tasks and methods in its hierarchy. Therefore, each primitive task in a plan saves a reference to the corresponding decomposition tree containing all linked compound tasks and methods that led to this task. Since a method can decompose into multiple tasks and therefore lead to multiple primitive tasks within a plan, all of which can back-propagate their own quality values, the quality value of the method itself ($V(c, m)$) is computed as the sum of all its sub-tasks. As described later in Section 3.3.4, these values are updated after the execution of each primitive task by the *Agent Controller* as shown in Algorithm 1 in line 27.

The proposed approach is especially interesting for cases when no expert knowledge is available to pre-define a good order of methods in the HTN. In the case of FightingICE, it can be applied if the attributes of a character's attacks are not known in advance and therefore manual ordering is not possible. For example, in the *Fighting Game AI Competition*, the skill characteristics of one of the three characters are usually hidden in advance and are only revealed during the competition games. With the proposed approach, the values for all compound tasks and methods from Equation 3.1 can be saved in a file, updated, and used throughout multiple games allowing the agent performance to improve over time.

3.3.4 Bottom Layer

The *Agent Controller* is responsible for plan execution, environmental monitoring, and recognizing and reacting to task failures. The communication between the *Agent Controller* and the game environment follows the protocol provided by the environment. The controller receives the *Frame Data* from the game environment in every frame, runs its update logic and returns a *Key Input* data to the environment. The pseudocode describing the complete decision-making logic of the Agent Controller is shown in Algorithm 1. Due to the fixed game frame rate of 60 frames per second, the computation time of the agent is limited to 16.6 milliseconds. This time budget is checked in line 16 of the algorithm. If an agent exceeds this time, its input will only be applied in the next frame and the previous input will be repeated in the current frame. To prevent this, in case the planner does not manage to create a plan within the given time, the *Agent Controller* triggers a default attack skill, which is set in line 5. A distinct default skill is predefined for each character.

As already mentioned, after receiving the delayed *Frame Data*, the *Agent Controller* uses a simulator to approximate the current game state by simulating the last 15 frames, as shown in line 3. In order to perform further checks on the updated data fd_{new}, the controller forwards this data to the *Command Center* in line 8.

Command Center is a class that is provided by FightingICE and is responsible for starting and processing skills (or *commands*). For example, a skill can require the following list of keys to be pressed : 6 2 3_*B*. Where the first two keys should be pressed in a sequence and the last two keys should be pressed simultaneously. There is no need for the agent itself to keep track of the exact timing of the distinct *Key Inputs*. Instead, the corresponding skill can be forwarded once to the *Command Center*, which will convert it into the corresponding *Key Input* data and take care of the sequential input. For the time that a command is being processed by the *Command Center* no other command can be accepted. The *Agent Controller* checks in line 9 whether the *Command Center* is still executing a skill. In case it is still processing the input queue of a previous skill, the controller just returns the corresponding *Key Input*.

What makes planning in FightingICE even more challenging is the aforementioned delay of 15 frames. This is less problematic for purely-reactive approaches since they do not rely on a correct task execution order and therefore do not need to take into consideration the progress of any previous or future actions. For a successful plan execution, however, the order as well as the timing of tasks plays an important role. As already mentioned, when performing a combo, the agent is required to execute all 4 combo skills in the predefined order and with a maximum of 30 frames between two consecutive skills. If either of these conditions is violated, the combo is aborted.

Algorithm 1 Processing

Input: Frame Data

Output: Key Input, which corresponds to the action to be executed

```
 1: $cc \leftarrow Command\ Center$
 2: $fd \leftarrow Frame\ Data$
 3: $fd_{new} \leftarrow$ simulate($fd, 15$)
 4: $\pi \leftarrow current\ plan$
 5: $a \leftarrow default\ character\ action$
 6: $k \leftarrow a.Key$
 7: UpdateUCBWeight() {used only in the UCB approach}
 8: $cc$.setFrameData($fd_{new}$)
 9: if $cc.getSkillFlag$ then
10:   $k \leftarrow cc.inputKey$ {continue executing a skill}
11:   return $k$
12: end if
13: $decisionMade \leftarrow$ false
14: $replan \leftarrow$ false
15: if $canPerformNextAction$ then
16:   while $decisionMade =$ false and $belowTimeBudget$ do
17:     if $opponentProjectileInAir$ or $prevActionFailed$ then
18:       $replan \leftarrow$ true
19:     end if
20:     if $\pi = nil$ or $replan =$ true then
21:       $\pi \leftarrow CreatePlan()$ {create a new plan}
22:       if $\pi = nil$ then
23:         break
24:       end if
25:     end if
26:     $t_{prev} \leftarrow t$
27:     UpdateUCBStatistics($t_{prev}, fd_{new}$) {used only in the UCB approach}
28:     $t \leftarrow$ next task in $\pi$ {get next plan task}
29:     if $t$ valid in $fd_{new}$ then
30:       $decisionMade \leftarrow$ true
31:       $k \leftarrow t.action.Key$
32:     else {$t$ invalid}
33:       $\pi \leftarrow nil$ {continue}
34:     end if
35:   end while
36: end if
37: return $k$
```

Furthermore, every attack in FightingICE is durative and cannot always be aborted. Therefore, in the next step in line 15 the controller checks whether the character is able to start the next action. This includes checking both whether the previous skill has been finished and whether the *Command Center* shows the character as controllable (see Figure 3.3). A character can be uncontrollable even if it is not actively executing any skill, for example when it is falling down or recovering from a hit.

Due to the delay, it is not possible to know whether a forwarded command was actually executed. Therefore, *CanPerformNextAction* first checks whether more than 15 frames passed since the previous skill was forwarded to the *Command Center*. In this case, the start of the skill can actually be observed in the delayed *Frame Data* and this information can be used to check whether the skill was actually triggered. Furthermore, its start time and its length can be used to calculate the end time of the skill. Also, in the following frames these data can be used to ensure that the skill was not aborted by an opponent's hit as described below and shown in line 17.

Waiting 15 frames for the information provided by the *Command Center*, however, is only feasible for *attack* skills because all of them last longer than 15 frames. In contrast, simple *movement* actions are executed as long as the corresponding *Key Input* is forwarded to the game. Therefore, if the character is moving to the right by "pressing" the right arrow key for only 5 frames this will be only shown by the *Command Center* 10 frames after he *stops*. If the agent was waiting for this information before executing the next skill, it would be standing still for 10 frames. For that reason, whenever an action that is shorter than 15 frames is executed, the *Agent Controller assumes* that the action was successfully executed and starts the next action without waiting for the *Command Center*. If, according to the checks performed in line 15, a new action can be started, the algorithm tries to execute the next task within the *while loop* in line 16. Before executing a task, however, it performs further checks, which can lead to replanning. This is done until it either succeeds to execute a task or exceeds the total time budget in line 16.

An important requirement for an agent in such a dynamic game is staying reactive even when using a long-term planner. Since projectiles deal a lot of damage, an agent should always try to avoid being hit by them. Therefore, even if the *HTN Fighter* is executing a combo, the plan containing combo actions should be dismissed in favor of avoiding the projectile first and continuing the combo or starting a new one afterwards. In Fighting-ICE, a projectile can be avoided by moving away or jumping over it. In order to detect projectiles and react to them, the agent controller checks for *opponentProjectileInAir* in line 17 and triggers replanning if needed. It is noteworthy that the projectile data is also delayed by 15 frames.

Monitoring the execution success of previous actions is another important objective of a long-term planning agent since it relies on the correct execution order of action *sequences*. For that reason, the algorithm checks for *prevActionFailed* in line 17 deciding whether to continue executing the current plan or to replan. Again, due to the artificial delay in the *Frame Data*, the real value of *prevActionFailed* is only known if the length of the previous action is higher than 15 frames and at least 15 frames have passed since the action was triggered. In this case, it is possible to directly check the action in the delayed *Frame Data*. Before this time or in case the action itself is shorter than 15 frames the *Agent Controller* assumes that the action was successfully executed. A possible future improvement at this point is to learn the action success rates from past experiences.

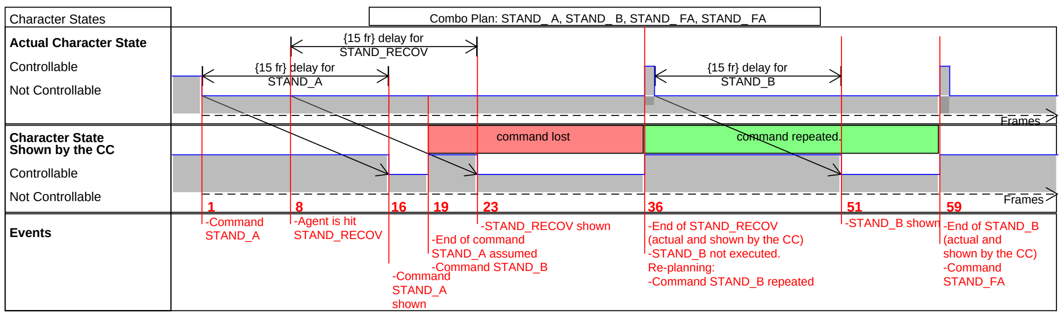

Character States
Combo Plan: STAND_ A, STAND_ B, STAND_ FA, STAND_ FA
Actual Character State
Controllable
Not Controllable
{15 fr} delay for STAND_RECOV
{15 fr} delay for STAND_A
{15 fr} delay for STAND_B
Frames
Character State Shown by the CC
Controllable
Not Controllable
command lost
command repeated.
Frames
Events
1
-Command STAND_A
8
-Agent is hit STAND_RECOV
16
-Command STAND_A shown
19
-End of command STAND_A assumed
-Command STAND_B
23
-STAND_RECOV shown
36
-End of STAND_RECOV (actual and shown by the CC)
-STAND_B not executed.
Re-planning:
-Command STAND_B repeated
51
-STAND_B shown
59
-End of STAND_B (actual and shown by the CC)
-Command STAND_FA

Figure 3.3: Timeline with the actual character state (in terms of controllability) and the state shown by the *Command Center* to the *Agent Controller* [153].

If either one of the previous checks requires replanning or the previous plan was successfully completed, a new plan is requested from the planner in line 21. If the planner is not able to create a plan within the given time, the main loop is exited in line 23 and the default *Key Input* is forwarded to the game in line 37.

However, if a valid plan π exists, the *Agent Controller* selects the task to be executed next (line 28). Before executing this task, the controller checks whether its preconditions still hold in the approximated current game state using the simulated *Frame Data* fd_{new} in line 29. This is especially important for tasks that are scheduled later in the plan since the game state might have changed a lot since the time that the plan was created. Depending on the validity of the task's preconditions either its *Key Input* is assigned to be forwarded to the game (line 31) or the plan is dismissed (line 33). This causes the while-loop to repeat the process and to request a new plan.

If the planner uses the UCB1 method selection described in Section 3.3.3, the weight balancing between exploration and exploitation is updated in each frame in line 7. Furthermore, once the *Agent Controller* ensures that the previous action completed, the corresponding UCB1 statistics are updated in line 27. During the ordered method selection these lines are ignored.

The timeline presented in Figure 3.3 [153] shows the progress of a plan execution and the differences between the character's actual state and its state as shown by the *Command Center*, which is delayed by 15 frames. In this case, the agent is supposed to execute the plan containing the following actions: $STAND\_A$, $STAND\_B$, $STAND\_FA$, $STAND\_FA$. In frame 1, the agent triggers the first command $STAND\_A$. It starts executing it and therefore becomes uncontrollable for the next frames (actual state is uncontrollable). However, the *Command Center* only recognizes this 15 frames later and therefore shows the character as controllable until frame 16. In frame 16, the *Agent Controller* uses the information provided by the $CommandCenter$ knowing that $STAND\_A$ was successfully triggered. Using its skill length of 18 frames, the *Agent Controller* assumes that the skill will end in frame 19.

However, while the character is executing $STAND\_A$, it is hit by its opponent in frame 8. This causes the character to play the $STAND\_RECOVER$ animation recovering from the hit. This animations lasts 28 frames and therefore the character remains uncontrollable until frame 36. It cannot accept any new commands during this time. The information about the hit is not available to the *Agent Controller* in frame 19 and therefore, assuming that the character is controllable, it triggers the second skill in the plan $STAND\_B$. Only in frame 23 (15 frames after the hit) does the *Agent Controller* recognize the hit and can deduct that the character is uncontrollable and that command $STAND\_B$ could not be accepted in the meantime.

This failure recognition is what allows the agent to repeat the same command later on in frame 36 instead of continuing with the next skill. After a successful execution of $STAND_B$, the agent can trigger the following command $STAND\_FA$ in frame 59 ensuring the correct execution of the plan. Although it is still possible that commands that are shorter than 15 frames are lost, for attack skills this strategy allows a safe execution without unnecessary plan re-generation.

3.4 Evaluation

The experiments described in this chapter were performed focusing on the two goals defined in Section 3.1. The first goal was to evaluate whether the proposed two-layer planning approach can lead to a strong performance of our agent. The second goal was the evaluation of the ability to follow long-term strategies while using a planner in comparison to purely reactive approaches. For the evaluation of the first goal, we measure the win rate of the *HTN Fighter* while it is playing against multiple reactive opponents. The success of the second goal is measured by the number of executed combos of each agent during these games.

The opponent agents used in all experiments described below were *Thunder01*, *Ranezi*, *MrAsh*, and a sample *MCTS* agent that is provided with the game environment. The first three opponents were the top agents submitted to the 2016 edition of the *Fighting Game AI Competition*[3]. Each of these agents is based on *MCTS* extended by additional rules.

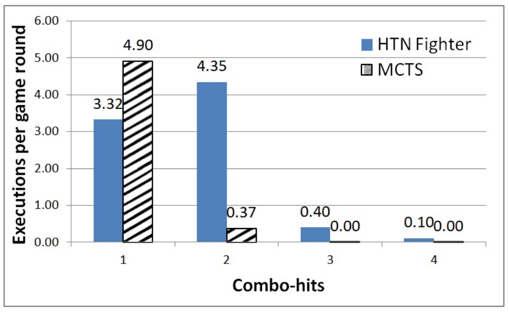
We have performed two different experiments with the two method selection approaches described in Section 3.3.3. Having good knowledge about the game environment and the skills of the character *ZEN*, we have manually defined its planning domain and used the common ordered method selection in order to evaluate the success of the goals described above. Afterwards, we have used the UCB method selection for the character *LUD*, whose skills' characteristics differ from *ZEN's* and therefore the character can benefit from a different order of methods. Both experiments are described in the following.

3.4.1 Ordered Method Selection

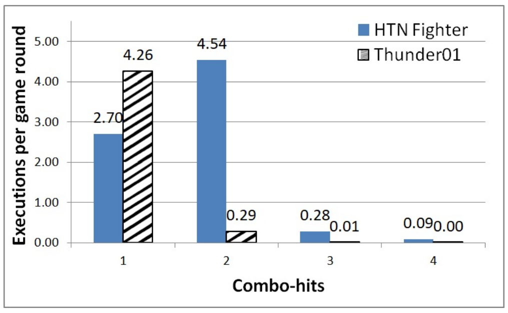
The description and results of the experiments described in this section were published in our previous work [153]. For these experiments, the planner used ordered method selection. Thus, when it was selecting a method to decompose a compound task, it checked (the preconditions of) the methods assigned to this task in the order that they were defined in the code. As already mentioned, skills in FightingICE are different for each character in terms of the required energy, their range, and the damage they deal. A skill that is very powerful for example for the character *ZEN* can be a worse choice for other characters. For that reason, the initial domain described in Section 3.3.2 was designed according to our knowledge of the game and some prior test runs for the character *ZEN* specifically. This domain was used in the experiments described in this section.

In order to evaluate both, the overall performance of the agent and the frequency and length of the executed combos, *HTN Fighter* played 100 games against each of the 4 opponents with 50 games on each played side. A game consists of 3 rounds meaning that in total 300 games were played between *HTN Fighter* and each opponent. Following the rules of the FightingICE competition, each agent started a game round with 400 Health Points (HPs) and the round lasted for maximum 60 seconds. If the HPs of one of the agents reached 0, the round ended with the other agent winning the game. If none of the agents was able to win the game within 60 seconds, the agent with the higher number of HPs at the end of the round was considered winner. Figure 3.5 shows the number of games won by *HTN Fighter* against each opponent. Our agent was able to outperform all opponents and after closer observation of the games we could see that the usage of combos contributed to these results as described below.

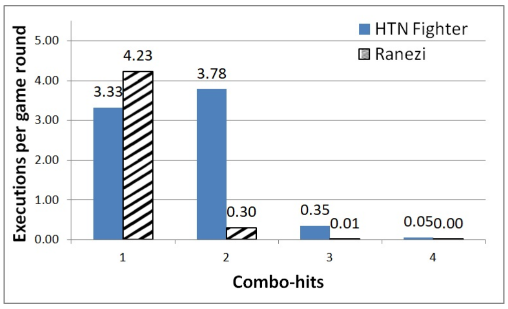
[3]Results of the 2016 Fighting Game AI Competition:
`http://www.ice.ci.ritsumei.ac.jp/~ftgaic/index-R16.html`



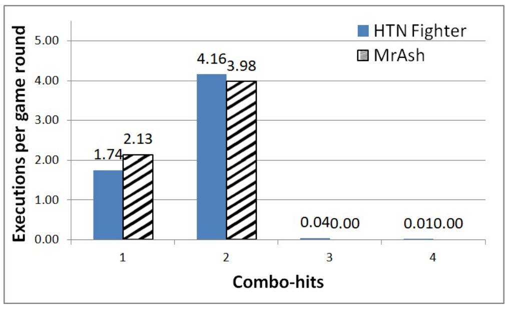
(a) HTN Fighter vs. MCTS



(b) HTN Fighter vs. Thunder01



(c) HTN Fighter vs. Ranezi



(d) HTN Fighter vs. MrAsh

Figure 3.4: The average number of successfully performed chains of combo-hits of the length 1 – 4 for each agent pair playing with the character ZEN. HTN Fighter uses ordered method selection [153].

As already mentioned, a full combo in FightingICE consists of 4 attack actions that have to be executed within a limited time between each two consecutive attacks and there are 14 different combos in total. An agent's combo can be aborted by its opponent through one of the combo-breaker attacks performed after the second combo-attack and before the last combo attack. However, it is not aborted between the first two attacks.

For each agent pair, the average number of consecutively executed combo-attacks was recorded during the 300 game rounds. These numbers are shown in Figure 3.4. The value in each column means that an agent hit its opponent exactly this many times before the combo ended. With 4 consecutive hits a combo was successfully completed, whereas an end after only 1 to 3 hits means that the combo was aborted either by a combo-breaker or because the agent did not execute any further combo-attacks within the time limit.

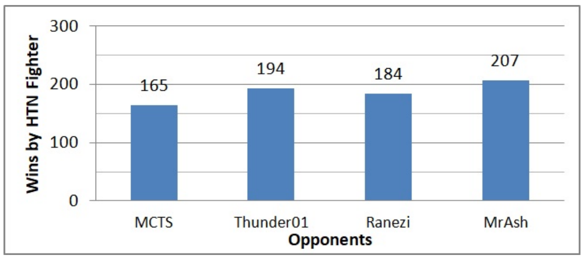


Figure 3.5: Number of games won by HTN Fighter playing as character ZEN using ordered method selection in 100 games (300 rounds) against opponent agents [153].

Many combos start with an attack skill that does not require much energy and can be executed very quickly. This skill on its own is very popular amongst the majority of agents. Therefore, having a high value of only 1 combo hits (shown in the first column of each diagram) does not necessarily imply that an agent intentionally started a combo. It can mean that the agent executed this skill for some other reason. However, having a high value in the second column indicates a higher possibility that the agent intended to execute a combo. In Figure 3.4, it is clearly visible that *HTN Fighter* started a combo much more often than most of its opponents. Sequences of 2 combo-hits were executed only by *MrAsh*.

Since a combo can be aborted after the second hit only, it is not surprising that the number of 3 consecutive combo hits drops in comparison to 2-hit sequences. Still, *HTN Fighter* was able to execute longer sequences and successfully complete some combos, which was not achieved by any of the opponents.

Another possible reason for the high difference between sequences of 2 and 3 combo-attacks performed by *HTN Fighter* is the fact that the second attack usually pushed the opponent away. Consequently, the preconditions of the planned third attack did not hold anymore. In this case, following the logic described in Section 3.3.4, the *Agent Controller* recognized a plan failure and triggered replanning. In many cases, we observed that the following plan involved a *sliding* action, which moved the agent quickly towards the opponent and dealt high damage. Followed by a sequence of *Knock-Back Attacks*, the opponent was uncontrollable for some time and received more damage from *HTN Fighter*. Such sequences of close-range skills explain the overall good performance of *HTN Fighter* against three opponents and the slightly worse performance against the pure *MCTS* agent as shown in Figure 3.5.

Due to the limited computation time in FightingICE, pure *MCTS* is able to perform rolls-out up to a certain depth only. That means that if its opponent was far away, the *MCTS* agent would need to simulate movement actions far enough into the future to actually reach its opponent and perform a close-range attack. However, due to time limits, the *MCTS* agent was not able to simulate that far ahead and was therefore not approaching its opponent. Instead it was rather firing range attacks. From the observations of the games against the *MCTS* agent, we could see that such situations were problematic to handle for *HTN Fighter* since it did not have a specific strategy for ranged fights. This can be the major reason for the worse performance against the *MCTS* agent in comparison to other agents.

As already mentioned, the other three agents were using *MCTS* in combination with some enhancements, which allowed them to approach the opponent and have close-range fights rather then staying in distance. In these cases, however, the *HTN Fighter* could apply the strong sequences of attacks described above, which explains the higher win numbers against other opponents shown in Figure 3.5.

The fact that *HTN Fighter* was able to execute some longer sequences of attacks shows that the use of a planner in a highly dynamic fighting game is possible in general. Furthermore, the recognition of plan failures through the monitoring *Agent Controller* shows that the architecture is suitable for reactive planning. However, especially cases where small changes caused replanning show that an additional low-level decision-making mechanism can be beneficial to deal with such situations without triggering global replanning. For example, when the opponent moved out of the next attack's range, the *global* plan would be still valid if our agent moved slightly closer to the opponent. In this case, replanning could be prevented by some additional movement actions added at *execution* time.

3.4.2 UCB Method Selection

As already mentioned, the planning domain described in Section 3.3.2 was created for the character *ZEN* specifically. The order of methods used in this domain was selected to the best of our knowledge of the game and was used with ordered method selection in the first experiments described above. Other characters' skills, however, have different attributes and therefore a different order of methods can be more optimal for them. In order to evaluate the effects of *ordered* method selection and *UCB* method selection, we have tested both selection approaches for different characters in further experiments described in this section. First, the experiments described above were repeated with UCB method selection for the character *ZEN*. Afterwards, they were repeated with both ordered and UCB method selection for the character *LUD* while using *ZEN's* planning domain.

For both characters, we have first used a learning phase to collect the UCB values that were used later in test games following Equation 3.2. Each character was trained independently while playing against the same character (i.e. *ZEN* versus *ZEN* and *LUD* versus *LUD*) controlled by each of the 4 opponents from the previous section: *Thunder01*, *Ranezi*, *MrAsh*, and *MCTS*. The learning games consisted of 300 rounds against each opponent (with 100 games against each opponent, a game consisting of 3 rounds and 50 games played on each player side). During the training games, UCB did not use any *exploitation*. Instead it was *exploring* all applicable methods (methods whose preconditions held in the current situation) with a uniform selection probability. Throughout all games, all UCB values were recorded. These values were the number of times that a compound task c was decomposed so far ($N(c)$), the number of times that the method m was used to decompose c ($N(c, m)$), and the quality of method m.

$$m^*_{balanced}(c) = arg\ max_{m \in M_c} \left\{ RelDamageDealt + K\sqrt{\frac{ln\ N(c)}{N(c,m)}} \right\} \tag{3.2}$$

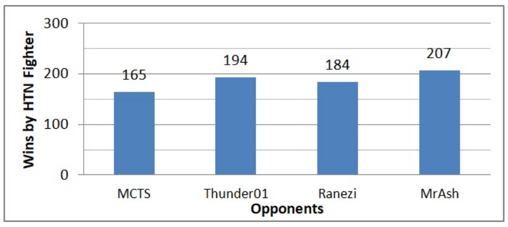
Following a strategy that balanced between offense and defense, the quality of a method was computed as the *relative damage* that was dealt to the opponent by the tasks that resulted form a method. As shown in Equation 3.3, this was computed as the difference between the damage *dealt* by *HTN Fighter* and the damage *received* from its opponent. These values were computed at *execution* time after finishing the execution of a task that resulted from method m.

$$RelDamageDealt = \frac{DamageDealt - DamageReceived}{DamageDealt + DamageReceived + \epsilon} \tag{3.3}$$

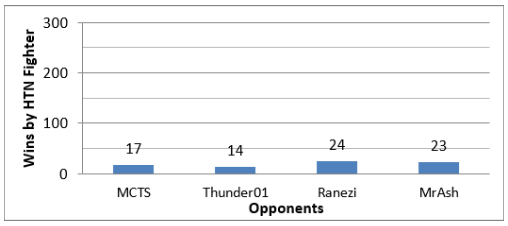
The dealt damage, again, was computed as the difference between the opponent's HPs after the end of the task and before the start of the task as shown in Equation 3.4. Similarly, the received damage was computed taking into consideration *HTN Fighter's* HPs as shown in Equation 3.5. Following these computations, HTN Fighter was supposed to select a strategy that allowed to deal more damage to the opponent than to get hit.

$$DamageDealt = oppHP_{current} - oppHP_{previous} \tag{3.4}$$

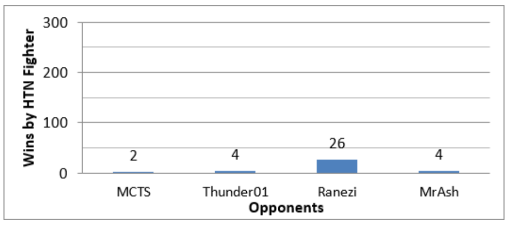
$$DamageReceived = myHP_{current} - myHP_{previous} \tag{3.5}$$



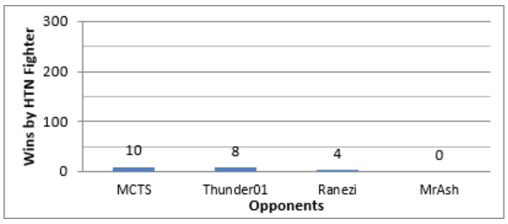
(a) *Ordered* method selection - *ZEN*.



(b) *Ordered* method selection - *LUD*.



(c) *UCB* method selection - *ZEN*.



(d) *UCB* method selection - *LUD*.

Figure 3.6: Number of games won by HTN Fighter playing as character *ZEN* and *LUD* using different method selection approaches. The agents played 100 games (300 rounds) in each setup against 4 opponent agents.

After the training phase against all 4 opponents, the collected values for each character were used by UCB during further 300 test rounds against each opponent according to Equation 3.2. Similarly to the first experiment, we have measured the general performance of our agent by the number of games it won against each opponent. Furthermore, we have measured the average number of combos performed by each agent during the 300 rounds.

Figure 3.6 shows the general performance of *HTN Fighter* against the 4 opponents. For the purpose of a better comparison, we have added the results already shown in Figure 3.5 as Figure 3.6a. It becomes very obvious that *HTN Fighter's* performance was much worse in the 3 additional experiments. In these cases, the agent was not able to win most games against all opponents. When looking at the games played by *ZEN*, we assume that the performance drop with UCB selection appears due to the fact that we pre-tweaked the original planning domain in such a way that the strongest strategies were considered first when using *ordered* method selection. Therefore, adding more variation through UCB meant potentially selecting weaker strategies or combinations of actions that are less optimal in their combination. In the games with UCB selection, the agent's behavior seemed more chaotic and oscillating than with ordered method selection.

Another possible reason is the way that the quality of a method was measured during the training phase. A damage dealt by a task was measured directly after finishing the task (and the additional delay of 15 frames.) However, it was not possible to correctly register the damage dealt by projectile attacks since a projectile can hit the opponent many frames after the character finishes executing the attack. Furthermore, there can be multiple projectiles in the air at the same time. Therefore, it requires a complex system in order to track all projectiles and to connect their damage with the respective previously executed tasks. For that reason, UCB did not learn the correct values for projectiles, which might have led to a decreased usage of projectiles and an even worse performance of the agent both as *ZEN* and *LUD*.

The major insight from *LUD's* results is that while using *ZEN's* planning domain, *LUD* performs much worse against all opponents with both ordered and UCB method selection. This was expected since all skills in the game are character-specific and therefore each character requires a specific high-level strategy. For example, we have noticed that while playing as *LUD*, our agent often did not move at all when its opponent was out of hit range. If played as *ZEN*, the agent usually used a projectile attack in this case. However, this attack required a certain amount of energy from *LUD*, whereas it did not require *any* energy from *ZEN*. Therefore, *LUD* did not have the option for a ranged attack and remained passive until the opponent came closer. This problem was not even solved by a different method order through the UCB selection since *approaching the opponent* was not regarded as rewarding according to the UCB computation.

Similarly to the weak general performance of the agent, the numbers of combos performed while playing as *LUD* or as *ZEN* with UCB selection are also much lower than those achieved while playing as *ZEN* using ordered selection (shown in Figure 3.4). These numbers for all 3 additional experiment setups are shown in Appendix A. For all setups, these numbers indicate that the agent was not even able to execute sequences of 2 consecutive combo attacks, which can also explain the worsened performance. Furthermore, there is no obvious difference between the numbers of combos achieved with the two different method selection mechanisms for *LUD*. This, again, can result from the usage of high-level strategies that were not designed for *LUD*.

In general we can conclude that the usage of UCB for method selection did not provide the expected advantage of finding strategies that were more optimal than those defined manually. Instead, it led to more chaotic behaviors and a worsened performance. Furthermore, it diminished the main advantage of an HTN, namely the early decrease of the search space size through the avoidance of unnecessary precondition checks. With ordered method selection, a planner only checks a task's methods' preconditions until it finds an applicable method, which, in the best case, is only one method. With UCB selection, however, the planner was checking the preconditions of *all* methods of a compound task, which often led to increased planning times.

3.5 Limitations of Pure HTN Planning

The experiments of this chapter have shown that the usage of a planner allows an agent to execute more advanced and long-term behaviors than with purely reactive approaches. Our agent was able to execute a higher number of combos than its opponents and achieved a higher win rate through further sequences of specific attacks.

Furthermore, using the proposed architecture allowed the agent to monitor the progress of a plan, compare it against the world state, and trigger replanning, if required. As intended, replanning was triggered whenever the preconditions of a task could no longer be met or if the goal of the agent changed. The former happened very often if the agent was no longer withing the attack range and therefore was not able to hit its opponent. The latter happened if, for example, the agent was required to abort its current plan in order to avoid being hit by a projectile.

Although in these situations a full replanning led to better behaviors than if the agent continued to follow the old plan, we see the high frequency of such global replanning as problematic for multi-agent environments. Assuming that in such cases a centralized planner creates plans for all agents, every time that an agent is out of the attack range the planner will replan for *all* agents globally. That, again, means that every other agent will potentially change its behavior every time, which can be especially problematic if it happens with a high frequency (for example, every few milliseconds). Alternatively, if every agent has its own (decentralized) planner, replanning of one agent will require it to communicate its new plan to all other agents, all agents to synchronize their plans, and potentially to adapt their plans through replanning.

Naturally, global replanning cannot and should not be avoided completely. However, we argue that in many cases seen in this chapter the execution of the plan could be continued if the agent reacted to the situation locally by, for example, slightly adjusting its position. This leads us to the conclusion that using a high-level planner *only* is not enough for reactive execution. It requires some low-level mechanism in order to stay reactive while following long-term plans without every minor local change causing a global replanning. In the following chapters, we propose some solutions that tackle this problem.

3.6 Conclusion

In this chapter, we have introduced a first approach to planning and execution to be used in a highly dynamic environment. The proposed approach is based on a two-layer architecture. Here, the top layer consists of an HTN planner and is responsible both for strategic (high-level) and tactical (low-level) planning. The bottom layer is represented by an *Agent Controller*, which takes over the monitoring of the environment, the monitoring of the plan's execution, and the forwarding of commands to the game environment in order for the agent to start their execution. By checking the planned tasks against the environment, the *Agent Controller* is able to recognize plan failures as well as important changes in the world and trigger replanning when needed. For this, it requests a new plan from the top layer.

The approach was tested in a typical fighting game environment, where an artificial agent can control a game character fighting against other artificial players. The main goal of proposed approach was the evaluation of the ability of a long-term planner to stay reactive in a highly dynamic environment while following a long-term strategy. For that purpose, we have created a planning domain for one of the available game characters and performed experiments letting our agent play against multiple strong opponents. In order to be able to measure the success of *long-term* strategies, we have focused on so-called *combos*, sequences of certain attacks that have to be executed consecutively without any interruption. During the experiments, we have tracked the number of successfully executed combos for each character, as well the number of only partially executed combo sequences. Additionally, we have measured the overall performance of every agent tracking their win rates. The experiments have shown that the use of a planner greatly contributes to the successful execution of combos as well as further sequences of actions. Our agent tried to execute a combo more often than its opponents and was able to execute longer combos. Through further sequences of strong attacks, it was able to outperform all opponents.

This, however, was only true for the one character for which the planning domain was specifically created. When the same planning domain was applied to a different character whose actions have different requirements and effects, the agent was very ineffective in both the overall performance and the execution of combos. This shows the importance and effects of a well-defined planning domain. In addition to a common planning technique where a planner selects the *first* applicable method to decompose a compound task and dismisses the other possibilities, we have evaluated the effects of method selection based on Upper Confidence Bound (UCB1). Here, the planner was supposed to balance between exploration and exploitation of all applicable methods. This approach, however, could not outperform the manually designed order of methods and worsened the agent's performance resulting in more chaotic behaviors. Furthermore, since applying the UCB1 selection possibly means checking the preconditions of *all* methods, this approach dismisses the main advantage of an HTN planner, namely the early decrease of the search space. For that reason, we do not recommend applying this selection in highly dynamic environment where the planner's computation time should be decreased.

Although the first experiments have shown good results in terms of strategic long-term behaviors, we see the major problem in the two-layer approach in the fact that every minor change in the environment can trigger a replanning. In many cases, the change can be so small that a local adjustment of the behavior could allow the agent to continue executing its existing plan. However, without an intermediate decision system, the agent is required to request a decision from the planner, which unnecessarily increases the replanning rate. We regard this fact as especially problematic in multi-agent environments where global replanning can be caused even more frequently by every agent, which, again can lead to further problems. For that reason, we conclude this chapter with the suggestion to add an intermediate layer to the proposed architecture that should be responsible for local reactive decision-making following the examples of three-layer architectures described in Section2.3.1.

Chapter 4

Hybrid Approach : General Idea

In the previous chapter have pointed out some drawbacks that can arise when using a two-layer architecture in a highly dynamic environment. Seeing a necessity for a third layer with the goal to take over some low-level decision-making, we describe the general idea of a three-layer architecture in this chapter.

4.1 Goals

One major problem observed in Chapter 3 is that the replanning frequency can increase if failure management and decision-making are fully performed by a planner. Furthermore, with the planner dealing with both high-level and low-level decisions, the planning time can unnecessarily increase. In addition, the last chapter dealt with only a single agent in an environment which was modified by only one other actor (the opponent player). With an increasing number of factors that influence the environment and with multiple agents being controlled by our approach, the mentioned problems become even more complex and challenging.

G 1: Introduction of the general idea of a three-layer architecture The major goal of this chapter is to describe the general idea of a domain-independent hybrid planning-and-execution approach that is based on a three-layer architecture. The proposed approach is designed to be used in different highly dynamic environments and, if necessary, adapted to their needs while still following the general idea described in this chapter.

G 2: Separation of the decision-making levels The goal of the hybrid three-layer approach is the separation of the decision-making into different granularity levels, the reduction of decision complexities on the different levels, and the distribution of the responsibilities between the higher layers of the architecture. The intention behind this separation is the reduction of the planning time and the planning frequency as well as the improvement of the execution.

4.2 Three-Layer Architecture

Inspired by the three-layer architectures that are common in robotics and spacecraft control (as described in Section 2.3.1), we propose a three-layer architecture to be used in highly dynamic environments such as video games. The proposed architecture allows for strategic multi-agent planning, environmental monitoring, and reactive decision-making and execution.

The two higher layers of the architecture are responsible for decision-making on different abstraction levels and in regards to different time spans for which the decisions are made. The top layer is supposed to reason far into the future and make long-term plans. However, since no details about the future can be known in advance, we propose abstracting the representation of the environment on which the top layer operates. We expect these measures to decrease the planning complexity and the planning time of the top layer.

Instead of creating a fully detailed plan or being prepared for its contingencies, the top layer forwards the refinement of the abstract plans to the middle layer. Receiving an abstract plan, the middle layer is responsible for finding a way to execute each plan task in the most appropriate (feasible or optimal) way. In order to do so, it requires information about the properties and boundaries of each task. Therefore, we propose an extension to a common planning domain, which is described in more details in the following section. Using this information and monitoring the environment, the middle layer is able to make reactive decisions that allow an agent (or multiple agents) to show robust behaviors while proceeding with the long-term plan.

When refining and executing a certain plan task, the middle layer is responsible for reasoning about this task only, which decreases its own decision complexity and prevents it from exceeding computation time budgets and delaying its action execution. Furthermore, the use of the more reactive middle layer allows for additional security measures such as fall-back behaviors, which can be executed in extreme situations such as unplanned delays or minor failures. With a long-term planner only, such situations would require full replanning. However, since the top layer generates only abstract plans, we expect them to fail only in the case of severe environmental changes but not due to small local changes. We assume that, in most cases, small changes can be handled locally by the middle layer without requiring global replanning.

During the execution of a plan task, the middle layer is also monitoring the environment and validating the task against the current state of the environment. For that purpose, it is receiving sensory information from the lowest layer of the architecture. Additionally, the lowest layer is responsible for the actual updates of different systems. This layer is very specific to an agent's general architecture and is, therefore, not within the focus of this chapter.

4.3 Top Layer

The top architecture layer is responsible for strategic long-term planning. Since we are considering multi-agent planning problems, this layer can incorporate either one central planner or multiple decentralized planners assigned to each agent. In this work, we focus on coordinating and cooperating agents and therefore coordination is an important aspect to consider when selecting a planning mechanism. Decentralized planning requires coordination between agents at *plan-time*, which can lead to high communication and coordination costs and longer planning times of the involved planners. This, in turn, can delay the *execution* and negatively affect the agents' behaviors. As we have mentioned in Chapter 2, such delays are critical in highly dynamic environments such as video games.

On the other hand, coordination costs at plan-time can be omitted when using a centralized planner, which can access all agents' knowledge. A single planner creating plans for all agents allows for tightly-coupled coordination on higher hierarchy levels. However, the planning problem becomes combinatorial and, if these plans are very detailed, the planning time can exceed all acceptable limits. As already described in Section 2.3.2, in order to decrease the size of the search space and the planning costs of a centralized planner, it is common to use abstract world models on higher levels of a task hierarchy.

We propose using a centralized HTN planner on the top layer of the architecture. The purpose of the planer is to generate long-term plans while deciding on high-level strategies and operating on an abstract world representation. The planner is not be responsible for low-level detailed decision-making and therefore only replans when the high-level plan fails or a new strategy is required. Furthermore, it can generate common plans for multiple agents, if required. As already mentioned in Section 2.3.2, in many cases, hierarchical planners are used for coordinated multi-agent planning since hierarchical structures of agents and goals allow for intuitive strategy generation.

4.4 Planning Domain

A major challenge when interleaving a planner with a reactive approach that is responsible for task refinement is the representation of plan tasks in a form that is readable by the reactive approach. For that purpose, we introduce the *Hierarchical Task Network with Postconditions (HTN_p)* extending the standard notion of HTNs by an element that allows for *refinement* of high-level plans during *execution*. To the best of our knowledge, such an extension has not been made so far.

The definition of the HTN_p's planning problem is similar to the standard notion of HTNs described in Section 2.2.3: $p = (D, s_I, c_I)$, with $s_I \in 2^F$ being the initial state, $c_I \in C$ the compound task to be decomposed, and $D = (F, C, A, M)$ the planning domain. The domain consists of a set of facts F, a set of compound tasks C, a set of methods M, and a set of primitive tasks A. However, in contrast to the common definition, primitive tasks are defined not only by their preconditions but also by postconditions. That is, a primitive task $t \in A$ is defined by the tuple $(pre, add, del, post)$, their preconditions, effects, and postconditions with $add \in 2^F$, $del \in 2^F$, and $post \in 2^F$.

The purpose of postconditions is the definition of a primitive task's end. Combining notions of *tasks* (as used in HTNs) and *goals* (as used in HGNs), we assume that the primitive tasks used by the abstract high-level planner are durative and can be *finished* or *achieved* through different influences, not only by (an) agent(s) but also through some external effects. For example, a task to *collect a barrel of water* can be performed by an agent bringing multiple buckets of water. But if it starts to rain into the barrel this task can be finished or achieved[1] earlier through the external influence.

The knowledge about changes caused by the rain, however, is only available once the agent starts *executing* the task. Therefore, the main difference between effects and postconditions, is that effects are used by the *planner* during the *planning phase*, whereas postconditions are used by the *executing* system during *execution.* This allows for such complex behavioral structures as iterations, loops, concurrent and sequential activities, or time- and event-driven activities at *execution-time.* The importance of such structures has been addressed in multiple planning-and-execution approaches used in spacecraft control as described in Section 2.3.1.

Postconditions do not affect the search and decomposition process of the planner in any way. The decomposition is still done using preconditions and effects only. That way, any HTN planner can be used at the top layer of the architecture ignoring the postconditions of the tasks. For example, after adding the task of *collecting a barrel of water* to the plan, the planner will add the effect of *having a barrel of water* to its internal world state representation. At this point, the planner is not concerned with *how* the task will be achieved later on, the *effect* is expected to be the same. During execution, however, the middle layer can implement different ways to *refine* the task, such as *bringing multiple buckets of water* or *waiting for the rain to fill the bucket*, all of which aim to meet the same *postcondition* of *having a barrel of water.*

[1]For simplicity, we use the notion of *achieving a task* in this work.

Although, intuitively the effects and postconditions of most of the tasks will be the same, they can as well differ in some cases. For example, postconditions can include facts that have to hold in the environment without a direct influence of any agent. This kind of postconditions replace certain temporal conditions such as *the task has to be achieved at daytime.* Furthermore, postconditions can be used for synchronization of tasks of multiple agents. For example, one agent can be required to *keep the barrel open* for as long as another agent is *filling water into it.* An effect of the first agent's task is *barrel is open* while its postcondition is *barrel is filled.*

Using the postconditions, the middle layer can monitor the progression of the currently executed high-level task and decide when to proceed from one task to the next one. This way, it is also possible that a task can be achieved due to external events and not only through the agent's actions only. For example, using the postcondition of the first agent (*barrel is filled*), the middle layer will recognize at execution time when the task is achieved and will proceed to the next task.

4.5 Middle Layer

On the middle layer, we propose using one of the common reactive approaches described in Section 2.1. By using a standardized and widely used approach, we aim to make the use of the architecture more intuitive. The purpose of this layer is to allow (an) agent(s) some local autonomy while following a (common) long-term plan provided by the top-level planner.

In particular, this layer is responsible for refining a high-level plan. During the refinement, it is responsible for the monitoring of the plan progress and for plan validation against the current environment state. In case of plan failures, it requests a new plan. Communicating with the lowest layer, it is responsible for a translation of plan tasks into actions that can be directly executed by the low-level control systems, while interpreting the information provided by the lowest layer and additional systems.

Since this layer is responsible for more detailed decision-making at execution time, it uses a more detailed world representation than the top layer. Therefore, there are two major aspects that need to be considered when using different world models on different architecture levels, namely knowledge representation and action representation. When abstracting the world representation, it is important to not lose information and to be able to translate between different knowledge representations used by the different architecture layers. While the highest layer operates on abstract information, the lowest layer can use detailed sensory information perceived by dedicated systems. This data can be filtered and combined with real data (such as world coordinates) in the middle layer before being further abstracted and represented as *facts* to the top layer planner. Therefore, it requires some binding between the different data representations. The details of this binding are presented for each proposed approach in the following chapters.

Another important aspect is the translation of high-level tasks into low-level behaviors and actions. According to most of the three-layer architectures described in previous works, the middle layer usually deals with *operational* action representations, as opposed to the *declarative* representation used by a planner. Although there have been some approaches that used either operational or declarative representations on *both* layers, these are very uncommon in practice, which makes their maintainability more difficult and less intuitive. Our goal, however, is the creation of an approach that is easy to use and maintain. For that reason, we propose using combinations of standardized and common methodologies. Therefore, the proposed architecture includes a *declarative* planning layer and *operational* low-level decision-making and execution layers.

Since reactivity and fast decision-making is very important at the middle layer, we avoid explicit communication between all agents at execution-time to save the costs of communication. Instead of explicitly communicating with *every* other agent, we propose coordinating individual plans either through specific synchronization signals and synchronization actions between the *affected* agents, similarly to the idea described in [127] (as used in Chapter 5), or through fully implicit communication (as described in Chapter 6).

4.6 Bottom Layer

Similarly to most of the three-layer architectures, the lowest layer is responsible for communication to concrete systems of an agent retrieving information from sensors or perception systems and forwarding the low-level actions provided by the middle layer to systems like the navigation and locomotion systems. The details of the lowest layer depend mostly on the environment, the available systems, and possibly the agent's hardware. In most cases of *virtual* environments, the agents are required to implement certain predefined protocols and software interfaces. For that reason, this layer is not standardized in general. In the following sections, more details are given on how this layer is implemented for two concrete environments.

Chapter 5

Hybrid Approach *I* : HTN + BT

In this chapter, we introduce a novel hybrid planning-and-execution approach that implements a three-layer architecture based on the idea described in Chapter 4. This approach uses Behavior Trees as the low-level decision-making and executing mechanism on the middle architecture layer, which allows for local autonomy of each agent in a multi-agent scenario. In the following, we describe the goals, the use-case environment, and the details of the hybrid architecture, followed by the description of the evaluation. The ideas and experiments described in the following sections were published in a previous work [157].

5.1 Goals

As we have seen in Chapter 3, in most cases, a plan failure happens due to small changes in the environment such as the opponent not being close enough for the agent to execute its next attack directly. However, these failures can usually be managed locally. They do not invalidate the overall plan and only require some adjustments on the lowest levels of a task hierarchy. This, however, cannot be done with a two-layer architecture where a planner is responsible for both high-level and low-level decision-making. Furthermore, if a planner is responsible for very detailed planning, its planning time, even for one agent, can be unacceptably high. Therefore, the following section discusses how to tackle these problems and further challenges described in Section 2.1 and Section 2.3.1.

The introduced approach is intended to be used for a manageable number of agents that build *teams* of up to 10 agents, as opposed to a *crowd* or a *swarm* of agents, which can exceed this number by far. Furthermore, as a team, the agents are following a common goal while each individual agent's goals and actions remain important. In contrast, for example, in a swarm, only the group's goal are important to every agent. Furthermore, the proposed approach should allow for very precise action execution in a *continuous* environment even though planning is performed on an abstract *discretized* world model. A scenario including a higher group size with a higher focus on the group's goal as well as execution in a discrete environment will be dealt with in Chapter 6. The goals of this chapter are the following:

G 1: Reduction of the global replanning frequency. The main goal of this chapter is to reduce the global high-level replanning. The applied solution tries to hand over the management of small local changes to individual agents that use a reactive approach. To achieve this, it requires a hybrid planning-and-execution approach that allows for strategic multi-agent planning with common goals and autonomous execution by each agent separately. The replanning frequency at the group level should be reduced through failure recognition and autonomous failure management on the individuals' level.

G 2: Improvement of the reactive execution in terms of execution time and execution success rate. The purpose of the middle layer in a three-layer architecture is the autonomous and deliberative refinement of high-level tasks into reactive behaviors with local failure management. In particular, the use of BTs in the following approach allows for creation of complex behaviors through constructs such as loops, conditional execution, sequential and parallel execution, which are not possible with other reactive approaches like, for example, FSMs. The importance of such constructs was already pointed out [100] and addressed in Section 2.3.1. The second goal of this chapter is to investigate whether, and to what extent, the use of BTs and such structures affects the execution times and the success rate of multi-agent plan execution in comparison to an architecture that uses only a planner.

G 3: Intuitive use and maintenance. Finally, we aim to introduce a framework that allows for a combination of *wide-spread* and *well-known* approaches that are commonly used in multiple areas of research and in multiple industries. The advantage of using such approaches is the available expertise in them, which allows for intuitive and easy creation and maintenance of individual agent behaviors and common high-level strategies. Similarly to *ROSPlan* [115], which combines a planner with the Robot Operating System that is very popular in robotics, the hybrid approach introduced in the following sections combines a planner with Behavior Trees, which are similarly popular in video game development and are gaining more attention in other industries. They are well-readable by human users and usually come with specialized editing and debugging tools.

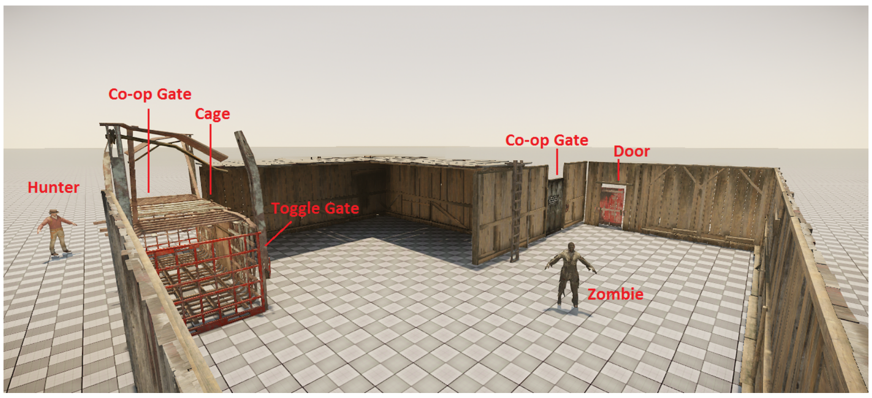


Figure 5.1: An example room including all elements and actors used in the test environment.

5.2 Test Environment

With the goal to test the proposed approach in a complex and highly dynamic environment, we take a realistic continuous simulation environment that requires fluid movement and precise execution of actions. Such environments are usually given in realistic $3D$ video games. However, there are very few available benchmark environments that are representative enough to show the full complexity and high dynamics of a realistic game.

For that reason, we have created an example environment using *CryEngine*[1], a game engine that allows to build custom environments (called *levels*) with realistic physics simulations. Furthermore, it offers multiple components and systems that are required to build agents that can act in the created environments, either autonomously (AI characters) or controlled by a user (player characters). Similarly to agents acting in other simulation environments, AI characters can use multiple sensory systems to simulate perception. They can access a path-planning and a navigation system, and use an animation system to visually represent a character's movements. Additionally, CryEngine provides a Behavior Tree system, which can be directly integrated into the proposed hybrid approach (as described in more details in Section 5.3). Most of these AI systems and comparable level editors can be found in other game engines as well, so that similar experiments can be re-created in another engine.

In the creation process of the test environment and the problems to be addressed by the hybrid approach, the following aspects were considered as important: the simulation had to offer *a*) a *multi-agent environment* with *b*) a *common long-term goal* that required *c*) *coordinated actions* while dealing with *d*) *uncertainty* and *e*) *high dynamics* of the environment. Both *d* and *e* can be caused by other actors operating in the environment.

[1]CryEngine: `www.cryengine.com`

With these aspects in mind, we have built the following experiment scenario: there are multiple *rooms* that are all accessible from an open space between them. One such room is shown in Figure 5.1. Each room can be entered either through a normal *door*, which can be opened by any agent, or through a gate that slides upwards. In order to open the gate it requires one agent to crank a handle and hold this handle to keep the gate open. During this time, another agent can pass through the gate. Therefore, it requires 2 agents to coordinate their actions. We refer to this kind of gates as *co-op gates.*

In another corner of the room, there is a metal *cage.* The cage can be entered from inside the room through a *toggle gate* that, similarly to a door, remains in its state (opened or closed) until changed by an agent. Opening this gate takes a few seconds but it can be closed very quickly. The other side of the cage has another *co-op gate* and leads into the open space between the rooms. Therefore, a room can be exited by going through the cage.

In every room, there is a *zombie* wandering around. If the zombie sees an agent, who is referred to as a *hunter*, the zombie becomes aggressive and chases the hunter while trying to attack him. A hunter has 100 *health points* and with each hit by the zombie the hunter's health points are reduced by 15. With no health points left, a hunter dies. The hunter is 0.5 meters per second faster than the zombie.

There are multiple hunters in the level. Initially, all of them are located in the open space between the rooms. Their common high-level goal is to catch every zombie inside the cage of the corresponding room. This means that each zombie should be inside a cage and both gates of the cage should be closed. To do so, a hunter needs to enter the room, catch the zombie's attention, and then quickly run into the cage through the toggle gate making sure the zombie follows him inside the cage. Next, the hunter has to exit the cage through the co-op gate leading outside of the room. For this to happen, another hunter needs to hold the co-op gate of the cage open at the right moment, while making sure it will be closed again to prevent the zombie from exiting. Then, a hunter needs to enter the room (again) and close the entry toggle gate of the cage making sure that the zombie is trapped inside.

In order to accomplish this task for one room, it requires at least 2 hunters, since the co-op gate of the cage can only be opened by a second hunter. The hunter closing the entry toggle gate can be one of the first two hunters or a third one. Therefore, at most 3 hunters are required to *clear* a room. Since there will be multiple such rooms, depending on the total number of hunters, the task of clearing *all* rooms can be achieved either by working in all rooms *in parallel* or *in a sequence.*

The agents can have two different roles. A *runner* is the hunter that goes into the room and lures the zombie into the cage. The *gate keeper* opens the co-op gate to let the runner out of the cage.

The high level goal is considered as achieved when all zombies are caught in cages. An additional constraint is that all hunters have to survive in order for the common goal to be achieved. Therefore, if a hunter dies, the global goal is considered as failed.

With the presented scenario, the environment is highly dynamic because of the zombies that change their location and chase the hunters. Additionally, the hunters themselves can trigger changes in the environment such as changing the state of the doors, occupying crank handles, and changing their own locations. Also, the fact that the zombies are behaving autonomously and are not controlled by our approach adds uncertainty to the environment. Certain information such as, for example, the actual locations of the zombies cannot be known at plan-time and has to be dealt with at execution-time. The goal of catching *all* zombies requires long-term planning.

While hunters are following their common long-term goal, they are also required to react to each other's and the zombie's actions and adapt their behaviors. For example, two hunters can be in each other's ways when going to a door and should figure out at execution-time how to behave in that situation without triggering global replanning. Furthermore, the usage of co-op gates requires a certain degree of coordination between the hunters at execution-time.

5.3 Hybrid Approach

Following the general idea of a three-layer architecture described in Chapter 4 and keeping the goals described in Section 5.1 in mind, the following section describes the major elements of two approaches. A so-called *hybrid approach* combines the advantages of an HTN planner and Behavior Trees. Additionally, a *pure approach* that is based on an HTN planner only is described in the following section and compared against the hybrid approach.

5.3.1 Three-layer Architecture

The hybrid approach implements a three-layer architecture as shown in Figure 5.2, which includes a centralized HTN planner on the top layer. The planner operates on a Hierarchical Task Network with Postconditions. The planner decomposes a common task of multiple agents into abstract plans for each agent and forwards these plans to the middle layer.

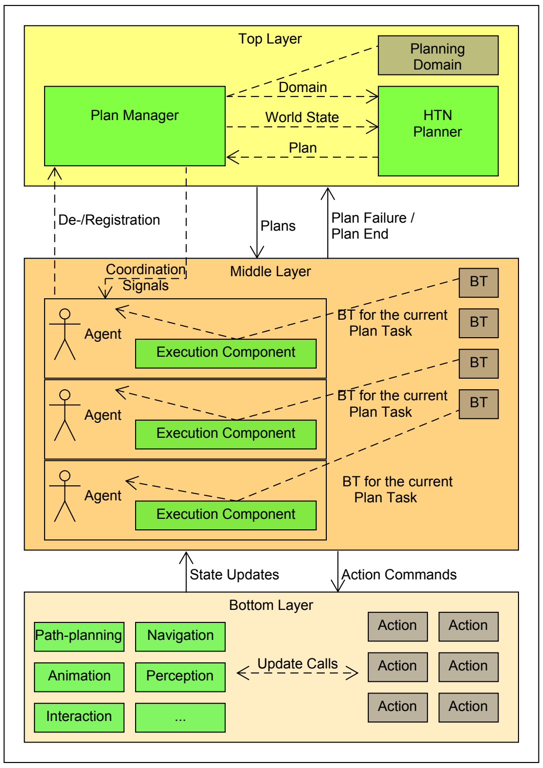


Figure 5.2: Three-layer architecture for the hybrid approach including a centralized HTN planner on the top layer and decentralized agent control with the help of Behavior Trees on the middle layer.

On the middle layer, the architecture is distributed among the agents. Each agent receives its own abstract plan and refines it using Behavior Trees. Here, each primitive task from the planning domain is represented as a separate BT that can be executed under the task's preconditions, is responsible for producing the task's effects and ensuring the achievement of the task's postconditions (more details in Section 5.3.4). Due to the modularity of BTs, it is possible to refine a task in different ways combining conditional, sequential, and parallel behaviors, which is not possible with, for example, FSMs as used in [144]. A BT allows for constant monitoring of the environment and reacting to signals coming from other agents as well as an agent's own low-level systems.

Comparable to the idea described in [121] and some prior works described in Section 2.3.1, this allows the planner to operate with incomplete information at plan-time leaving primitive tasks abstract. When the information becomes known to a BT at execution-time, the BT will try to refine and execute the task in the best way dealing locally with unexpected situations. Only if the corresponding BT is not able to execute the task in *any* way, the failure is reported back to the top layer and global replanning is triggered.

With a centralized HTN planner and decentralized BTs for each agent, this approach allows for tight high-level coordination at plan-time and implicit coordination at execution-time.

Since the environment is build with CryEngine, all parts of the architecture are written in C++. The details of the planning domain, each architecture layer, and the communication between them are described in the following sections.

5.3.2 Planning Domain

In Chapter 3, the planning domain was written in Java and a domain-specific planner was used in order to avoid the effort of writing a domain compiler that translates the domain from a planning language into compiled code. However, for the approach described in the current chapter, we have extended the open source planner *derPlanner*[2]. It provides a compiler that translates a domain written in a custom planning language into executable C++ code. It is possible to write the planning domain in one text file that follows a clear structure. Afterwards, the compiler can generate all required C++ methods and data structures to be used by the planner. The use of such a compiler allows for quick and easy modifications of the planning domain in the text file without the need to modify program code directly.

The goal of this chapter is reducing the replanning frequency on the high level. In order to be able to evaluate the success of our approach, we have created two planning domains with different levels of details. The first domain allows for planning up to the lowest actions that can be directly executed by an agent. An example sequence of such actions is *aligning to a door, opening a door, stepping through a door, aligning to a door*, and *closing a door*. Each of these low-level actions is represented by a primitive task and the sequence of them can be combined into the compound task *going through a closed door*.

The first domain with this level of details allows the planner to generate exhaustive plans that do not require refinement at execution-time. However, local changes in the environment can lead to failures in the plan and trigger global replanning. For example, if the agent is not close enough to the door or there is an obstacle between the agent and the door, the agent will be unable to open the door and will have to replan in order to move into a better position. This domain is presented in Appendix B.1 and contains 9 primitive tasks and 30 compound tasks. Since its primitive tasks do not require any task refinement, there is no need to use a reactive approach in addition to the planner. For that reason, this domain is used in the *pure HTN approach*.

[2]derPlanner: `www.github.com/alexshafranov/derplanner`

The second planning domain is more abstract and can be regarded as a condensed version of the first domain that is *cut off* at some hierarchy level. For example, the mentioned task *go through closed door* is now a primitive task with the effects and postconditions that the agent is on the opposite site of the door and the door is closed. This domain now depends on local refinement of its tasks at execution-time in order to obtain the same results. For example, an agent that is trying to go through a door, is now able to recognize the obstacle being in its way and adjust its position without triggering global replanning. This refinement is done by Behavior Trees and used in the *hybrid approach.* The full domain for the hybrid approach can be found in Appendix B.2 and contains 7 primitive tasks and 13 compound tasks in total.

In the following, we briefly describe all elements of a planning domain using the syntax of the planning language used to define the domains for this work (see Appendix B). While most of the components described below are available in the original version of derPlanner, the syntax of some elements has been extended for this work. In particular, primitive tasks have been extended with agent IDs in order to allow for multi-agent planning. As shown in Planning Domain Element 5.1, the first argument of every primitive task is the ID of the agent that this task is assigned to. The planner is keeping each agent's plan in a separate list. When a compound task is decomposed into primitive tasks, the planner adds each primitive task into the list of the corresponding agent. This way, each agent can independently proceed with its own tasks during execution.

```
prim
{
  GoTo!( id32, vec3 )
}
```

Planning Domain Element 5.1: Primitive Task

For the synchronization of plans, we use so-called *wait* tasks where an agent is supposed to wait for a certain event to happen before proceeding with its plan. The details of this way of implicit coordination are described in Section 5.3.4.

Similarly, each fact describing an entity (zombie, hunter, room or an object) includes an ID as its first variable ensuring a unique identification of that entity. In general, all literals of a fact as well as all parameters of a primitive task are defined by their types. For example, the fact *agent(id32, vec3)* shown in Planning Domain Element 5.2 shows that an agent is represented by its ID (which is a 32-bit integer) and its location (which is a vector containing the coordinates on 3 axes). The use of unique IDs ensures that the same fact is only saved once for each entity. For example the agent with the $ID = 1$ cannot be in two places at the same time.

However, some facts can be intentionally saved multiple times. For example the fact *connectsRooms(2, 3, 4)* means that a door with $ID = 2$ connects two rooms with their respective IDs 3 and 4. Thus, this fact represents a *relation* between the rooms, which is symmetric. Therefore, the fact *connectsRooms(2, 4, 3)* is allowed as well. The possibility to have multiple entries of a fact with the same ID is shown by the ˆ-symbol (caret) in front of the fact name *connectsRooms* in Planning Domain Element 5.2.

```
fact
{
   agent(id32, vec3)
   ˆconnectsRooms(id32, id32, id32)
}
```

Planning Domain Element 5.2: Facts

Another addition to the original version of the planner and its domain are effects. The original version of derPlanner did not include effect propagation during the planning phase. Therefore, the planner was not changing its internal world state and was very limited when it came to long-term planning. For that reason, a planning domain of derPlanner did not have a concept of effects at all. For this work, the concept of effects and their propagation during planning has been added. Following the common representation (such as described in Section 2.2.3), effects can be defined as facts that are *added* to or *deleted* from a world state.This is shown in Planning Domain Element 5.3 by the respective effects. The parameter of an effect is the table name of the fact to be added or deleted. For example, the task of opening a door has the effect *delete(closed(door*(2)*))*

```
effect
{
   add(table)
   delete(table)
   change(table, table)
}
```

Planning Domain Element 5.3: Effects

Additionally, we have introduced the notion of effects that can *change* a fact f_1 into a fact f_2 as shown in Planning Domain Element 5.3. A change is done by first deleting f_1 from a world state and then adding f_2 to it. The purpose of these effects was a more intuitive change of certain attributes of *the same* object. For example, changing the position of an agent a from x to y can be done by using the effect *change(agent(a,x), agent(a,y))*. Therefore, in order to make sure that only the attributes of the same entity are changed, the table names (here *agent*) as well as the IDs of the entity (here ID $= a$) are required to be equal.

Following the standard notion of derPlanner, different methods of a compound task are represented as distinct *case* branches as shown in Planning Domain Element 5.4. They are checked in the order of definition. This way, the planner follows ordered method selection. For example, the task of opening a co-op object o can be performed by an agent a in two different cases. If it is an object that is opened by using a crank handle h that is close to the object, the agent first needs to *go to* the handle, align to it (or turn into its direction), and than interact with it (which is represented by the *Hold* task). Otherwise, if it is one of the gates that is opened by pushing the gate upwards (either the co-op gate of the cage or the toggle gate), the agent can directly interact with it. Respectively, preconditions of a method are written before the $->$ symbol (arrow) and the sequence of tasks that the method decomposes into is written thereafter. In this case, the *GoTo* and the *Hold* tasks are compound tasks and *AlignTo* is a primitive task, which is shown by the *!* symbol (exclamation mark) behind its name.

```
task OpenCoopObject(a, o)
{
   case ( crank_object(o, y) & crank_handle(h,z) & handleForObject(h,o))
      -> [GoTo(a,z), AlignTo!(a,h), Hold(a,h)]

   case ( (coop_gate(o, y) | toggle_gate(o, y)))
      -> [Hold(a,o)]
}
```

Planning Domain Element 5.4: Compound Task

The last element of a domain used by derPlanner are the so-called *macros*. They can be regarded as shortcuts or simplifications of complex combinations of precondition checks. For example, the macro *handleForObject(h, o)* shown in Planning Domain Element 5.5 checks whether the passed objects h and o represent a handle (*crank_handle*) and an object that is opened by the crank handle (*crank_object*), and whether their locations x and y are closer than 3 meters to each other.

```
macro
{
   handleForObject(h, o) =
      crank_handle(h, x) & crank_object(o, y) & dist(x, y) < 3
}
```

Planning Domain Element 5.5: Macro

Here, *dist* is an external function that is defined separately in C++ code. This allows to call functions provided by the agent's low-level systems, by the environment, or other interfaces. Furthermore, it allows for maintainability of these functions without affecting the planning domain. For example, if developers decide to change the distance calculation from Manhattan Distance to actual walking distance on a path, this change will not require any adjustments in the planning domain.

With the elements described above elements, the planning domains for both approaches (hybrid and pure) are defined by $D = (F, C, A, M)$ with a set of facts F, compound tasks C, primitive tasks A, and methods M. The postconditions of an Hierarchical Task Network with Postconditions are not defined in the planning domain directly. Since they are not required at planning time but at execution time only, they are part of the BTs in the hybrid approach (as described in more details in Section 5.3.4). Whereas the pure approach simply uses the preconditions of the *following* task as postconditions for the current task.

5.3.3 Top Layer

The top layer of the architecture consists of a centralized coordinating *plan manager* system and an external *planner*. As already mentioned, the planner used in this section is an extended version of the open source HTN planner *derPlanner*.

The Planner

The planner is a domain-independent total-order HTN planner, which means that it decomposes tasks in the order that they will be executed on. It is a multi-agent planner and therefore each agent's plan (the sequence of actions that should be executed by the same agent) is saved separately. After successfully decomposing the initial (common) compound task, the planner provides the plan manager with a collection of plans mapped to corresponding agent IDs.

Due to the added concept of effects, the extended version of the planner allows for effect propagation. This, again, allows for longer plans causing more changes in the simulated world. Furthermore, the planner saves the history of applied methods and effects caused by them allowing for backtracking in case of an unsuccessful decomposition (see Section 2.2.3 for backtracking).

Additionally, the bindings of arguments of each primitive task as well as the bindings of literals used in the preconditions of the task are provided with the task. This allows to *a*) use them as arguments during execution and *b*) check the validity of the task at execution-time as described in Section 5.3.4.

Although derPlanner allows to time-slice the planning procedure over multiple frames, this feature was not used in this work.

Plan Manager

The centralized plan manager acts as an interface between the planner and the lower layers of the architecture. At the beginning of a simulation (experiment run), each agent registers with the plan manager providing its ID and location. If an agent dies during the experiment, it is de-registered and the manager deletes its plan and further related data. In theory, agents can register to and de-register from the manager at any point of time. This allows for dynamic group creation and adaptation in more complex scenarios. However, it is not required in the given test scenario.

Additionally, the manager collects all information about relevant objects, zombies, and rooms. Currently, the manager has full observability over the environment. However, this information can as well be added and deleted at any point of time. This way, with partial observability, the manager can receive these data from agents perceiving new information at execution time.

Since the planner is domain-independent, the pre-compiled planning domain is saved separately. During the initialization phase, the plan manager provides the domain to the planner. Afterwards, binding the actual environment information to the planning domain's facts, the plan manager generates the initial world state for the planner. As the plan manager gets notified about changes in the environment during the execution, it updates the corresponding facts forwarding the most recent data to the planner.

As already mentioned, the agents in the considered scenario can have different roles of either opening co-op doors or running into the room. In general, the problem of assigning agents to roles is combinatorial and can be solved by the planner itself or by any external solver. In this work, the plan manager is responsible for role assignment as it has the required information about all agents. Since all agents used here are homogeneous and therefore can perform each role, the manager simply assigns 2 agents to each room in the order that the agents are registered. If more agents are available or the number of agents is odd, it adds a third agent to a room.

In more complex scenario, heterogeneous agents can be used as well. In this case, the manager will require additional information about the skills and attributes of all agents. Furthermore, the assignment of agents to roles can be optimized in the future through the use of heuristics such as distance of agents to rooms, their availability or the applicability of their skills to the tasks, as it was done for the game *Horizon Zero Down* [85].

Once the planner's state is initialized and the agents have been assigned to their roles, the plan manager requests the planner to decompose the goal task described in Section 5.2 given the planning domain. The planner decomposes the common task and returns plans for all involved agents mapping each agent's plan to the corresponding agent ID. All plans are stored in the manager and, before triggering replanning, the manager clears all agent's plans. After receiving the plans from the planner, the manager forwards each plan to the middle layer triggering the start of their execution, which is described in the following Section.

Furthermore, the manager is responsible for the communication between agents and for the notification of agents about relevant environmental changes. The manager keeps track of all objects that are relevant to the planning domain and it has information about all agent's task. Whenever an object's or another agent's state changes, the manager identifies the agents whose current task involves the former object or agent and notifies the latter agent about the change. That way, only implicit coordination, which is guided by the plan manager, happens between the agents at execution-time, as described in more details in the following section.

5.3.4 Middle Layer

As already mentioned, there are two different approaches used in this chapter, a hybrid approach that involves plan refinements and a pure HTN approach that generates very detailed plans. The difference between these approaches lies in the middle architecture layer. The middle layer of the hybrid approach is described in the following subsection, followed by the description of the middle layer of the pure approach.

Middle Layer for the Hybrid Approach

Following the general idea of the three-layer architecture described in Chapter 4, the purpose of the middle layer in the hybrid approach is the interpretation of the high-level plan tasks and their refinement into low-level actions. The layer is organized in a decentralized manner. This way, each agent is responsible for its own plan progression and local decision-making. In the process of task refinement, each agent is allowed some local autonomy. On this layer, each agent contains a general execution component and a running Behavior Tree. The execution component is responsible for plan progression and communication with the central plan manager on the top layer, while the BT refines the current plan task. The Behavior Trees used in this work are the so-called *Modular Behavior Trees* provided with CryEngine[3].

[3]Documentation on the Modular Behavior Trees of CryEngine:
`https://docs.cryengine.com/display/SDKDOC4/Modular+Behavior+Tree`

Once the agent receives its plan from the top layer, it starts executing its tasks in their order. Similarly to the approach used in Chapter 3 and described in detail in Section 3.3.4, before starting the execution of a task, the agent needs to check its validity in the current world state. To do so, it checks the preconditions of the task using the variables bound to its arguments during planning.

The automatic creation of C++ code that allows such checks was added together with the expansions described in Section 5.3.2 to the original compiler of derPlanner. Now, when the planning domain is automatically translated from text into program code, the compiler generates not only functions for plan-time checks of preconditions but also functions for execution-time checks. The difference between those is that, when *planning*, the planner *searches for variables* whose values allow a precondition to become true. Once such a variable is found, it *is bound*, potentially used in further decomposition, and used to update the planner's world state. During the *execution* an agent is not searching for a new variable. Instead, it uses the bound variable and checks whether its current value (which might have changed since planning) still allows the precondition to be true.

For example, at plan-time the planner assumes that the zombie with $ID = 1$ will be inside the cage, when the agent wants to exit the cage. Therefore, it binds the ID of the zombie to the argument of the precondition. Then, at execution-time, before leaving the cage, the agent checks the actual position of the zombie with $ID = 1$ and finds out that it did not enter the cage yet. If the preconditions of an agent's task do not hold in the current world state, the agent notifies the plan manager and a global replanning is triggered on the top layer, dismissing the remaining plans for all agents and creating new ones.

When checking certain preconditions against the *actual* world state, an agent can use the fact tables stored in the top-layer plan manager. For example, the state of a gate (opened or closed) does not change very frequently and is updated instantly in the fact table whenever it changes. Therefore, its state can be checked at execution-time through the table. However, since the planning domain is kept abstract, there are certain preconditions, that can be checked more precisely by calling external functions. For that purpose, some facts are bound to external functions. That way, the *planner* can use the fact table during the *planning* process, while the *agents* can call the corresponding external function during the *execution*. For example, during planning, the planner can assume that, if a door is open, there is a path through that door. However, at execution-time, there can be some other agent or an obstacle blocking that path. In order to find out whether a path actually exists, the agent can query the path-finding system directly at execution-time.

If the preconditions of the current task still hold, the agent selects the Behavior Tree corresponding to the task to be executed. All BTs used in the hybrid approach are presented in Appendix C. For this work, the mapping between the primitive tasks and Behavior Trees is simply done through the usage of identical names. In the future, however, more efficient data structures can be used for more efficient mapping. Once a BT is selected, the agent sends a signal with the BT name to switch the running behavior tree of the agent. Since Behavior Trees in CryEngine are not parametrized, the parameters of the current task are saved for each agent in its execution component and can be queried from within the running BT. Parametrized BT, however, could take task parameters directly.

A Behavior Tree is the main system that controls an agent. It is responsible for reacting to certain events, decision-making, and action execution. Therefore, without a running BT, an agent is not able to do any of this. In our scenario, every agent starts with a default BT simply waiting for commands to come from the top layer. In more complex scenarios, however, the default BT can implement some basic behaviors, for example, ensuring the safety of an agent in any situation.

Once the agent receives a signal to start a certain task, it switches to the corresponding BT. Depending on the task, it is then listening for other signals relevant to the task. Such signals can result from other agent's actions and can be used for the coordination of multiple agents' plans. For that purpose, inspired by the work described in [127], instead of letting an agent communicate with *all* other agents, we define what signals an agent should wait for.

For example, assume that an agent is supposed to hold a co-op gate until another agent passes through the gate. In the BT of the first agent, the action of holding the gate is executed until the signal of the second agent having passed is received. This is done by having the *hold* action and the *WaitForEvent* node running in parallel[4]. In this case, the time that the first agent waits is intentionally not limited because there is no danger to the first agent and the task of waiting for the other agent has a high priority. That means that *this* agent's BT cannot fail at this point. However, the second agent might never arrive at the co-op gate, which can leave the first agent waiting endlessly.

In such situations where multiple agents are required to coordinate their actions and depend on each other, we distinguish between an *active* agent, the one *going* through the door, and a *passive* agent that is *waiting* for a signal. In order to prevent the passive agent from infinitely executing an action, the active agent is supposed to notify the plan manager about it passing through the gate or failing its task. Since the manager triggers global replanning for *all* agents if *one* agent fails to execute its task, the passive agent will be notified about the other agent's failure.

[4]A tutorial on the creation of a *Modular Behavior Tree* including a *WaitForEvent* node in CryEngine: `https://docs.cryengine.com/display/CEMANUAL/Tutorial+-+Getting+Started+With+The+Behavior+Tree+Editor`

Similarly, when the active agent goes through the gate, it will notify the manager as well. The manager, in turn, will use the knowledge about each agent's currently executed task and its parameters. Using this information, it will notify all agents, whose task's parameters include either the gate or the active agent's ID. This way, only affected agents will receive the information relevant to them and will be able to react to it. That means, the agent holding the gate will know that it is no longer required to hold it. There will be no unnecessary communication between *all* agents.

In theory, the relation between active and passive agents may be an $n - to - m$ relation with multiple agents waiting for one agent to send a signal or one agent awaiting multiple signals to arrive. In this case, it is important to prevent deadlocks with multiple agents waiting for each other. Since the situation in which multiple agents need to coordinate their actions is represented by a common compound task *in the planning domain*, the check for possible deadlocks can be performed for each common task on some hierarchy level of the HTN. These kind or relations, however, are not present in this work and show a potential way for improvement for the future. For the current scenario, only $1 - to - 1$ relations are required and deadlocks are prevented by manually ensuring that no dependencies exist for every pair of cooperating agents. Alternatively, additional types of BT nodes can be implemented in order to synchronize BTs of cooperating agents as proposed in [17].

Besides validating preconditions, Behavior Trees are also responsible for checking for the fulfillment of *postconditions* of the task that they represent. Therefore, in this hybrid approach, postconditions of tasks are not defined in the planning domain but within each BT. The check for postconditions is done in parallel to the action sequence executed by the agent. Only when all postconditions become true *and* the agent finishes the execution of its own actions, the task can be regarded as achieved. In this case, the Behavior Tree will succeed and send a *success* signal to the execution component. If the agent has more tasks assigned to it, the next task in the plan will be validated following the same procedure as described above.

After replanning, the manager provides each agent with a new plan and each agent sends a signal to switch to a potentially different BT. In this work, switching between BTs is done instantly. However, in environments where an agent's behavior should always be robust, instant abortions of actions can be undesired and even dangerous. This can be solved on the side of the running BT by deciding how to handle incoming signals. For example, there can be special behavior branches for finishing an action in a safe way and only then switching to a new tree. Alternatively, such safety measures can be implemented on the lower architecture layer allowing certain systems to *queue* any changes until the end of a current action instead of applying them immediately. This is commonly done in the Animation System of game characters. Aborting some animations can lead to unnatural visual representations of the characters, for example if a *sitting* character starts *running* without first playing a *stand up* animation. Therefore, animation switches are queued, potentially playing transitioning animations in-between.

With the procedures described above, global replanning is triggered in two different cases. The first case is if the preconditions of an agent's task do not hold at execution time. This means that the plan does not hold anymore and a new strategy should be selected. The second case for global replanning is an agent not being able to execute its task even though its preconditions still hold. This means that the agent is not able to find a local solution and needs guidance through new commands from the central planner. In this case, the planner's world state will be updated with the most recent knowledge and therefore the agent can rely on the planner to recognize the local changes and generate a global plan that is different from the previous one. In more complex scenarios, however, more attention should be given to forwarding the *exact reason* for the plan failure to the planner. This will prevent the planner from creating the same plan over and over again and can be done by blocking the variables that were bound to the task in the failed plan. This way, the planner will not be able to bind these variables in the next planning cycle.

Since BTs *refine* high-level tasks, they are, in general, more detailed, more complex, and more expressive than the plan tasks. They allow to represent different ways of achieving a task through alternative and parallel branches, including additional run-time checks, loops, and signal receivers, which are not present in HTNs. For that reason, they cannot be *directly* generated from an HTN task. Therefore, it is the developer's task to create each BT in such a way that it leads to the desired effects, includes listeners for the required signals, and checks for the desired postconditions. For the approach described in this chapter, all BTs were created *manually* under consideration of the respective HTN tasks and the simulation environment.

Middle Layer for the Pure Approach

In general, the architecture of the pure approach looks very similar to the one of the hybrid approach. However, since the planner uses a very detailed planning domain, all plans are granular and do not need further refinement. Here, primitive tasks represent actions that can be directly executed by an agent. For that reason, the middle layer of the pure approach does not include Behavior Trees. Instead, each agent simply receives its plan, checks for the preconditions of a task in the same manner as described above and directly triggers its execution.

Similarly to the hybrid approach, in case the preconditions of a task do not hold, the agent forwards the failure to the top layer, where the plan manager triggers replanning. A major difference to the hybrid approach is, however, is the handling of task execution failures. Since plans do not include any logic for execution failure and the agents do not incorporate any additional mechanisms for local decision-making, they do not try to solve such failures locally. Therefore, if an action fails to be executed, the agent also forwards this failure back to the manager, which leads to a global replanning for all agents. However, if an action is successfully executed, the agent proceeds with the next plan task.

One special case represent the *wait* tasks in this architecture. Coordinating two agents is done in the same manner as in the hybrid approach by having an active agent and a passive agent waiting for a signal to proceed with or abort its plan. In the hybrid approach, such *wait* tasks are part of BTs and are not present in the planning domain In contrast, in the pure approach, all granular tasks are encoded in the planning domain, including the *wait* tasks. However, the time that an agent will need to wait during the *execution*, is not known at *plan-time*. Therefore, at least *some* autonomy is required at execution-time from the agents to decide how long to wait.

As already mentioned, in the hybrid approach, an agent is *waiting* for a signal while executing its own actions. This is done with the help of the *WaitForEvent* BT node, which waits for a predefined signal, and a *Parallel* node. However, neither signal receivers nor parallel branches are provided by a planner. For that reason, in the pure approach, the agent's execution component itself includes an artificial loop for the execution of the *wait* task. By default, a *wait* task in the pure approach lasts a certain amount of time. In our case 1 second. Once the task finishes, the agent validates that the preconditions of the *next* task hold. If this is not the case, the *wait* task is repeated. This procedure is repeated until either the preconditions of the next task become true or the agent's plan is aborted. Similarly to the hybrid approach, the passive agent relies on the active agent succeeding or failing with its task and that way affecting the passive agent's plan execution. Additionally, the preconditions of the next task in the plan replace the postconditions of the *wait* task. Therefore, when creating the pure domain, it is the developer's duty to define correct preconditions for the task that *follows* a *wait* task.

5.3.5 Bottom Layer

On the most granular level, each action, regardless of whether executed directly by the agent or from a Behavior Tree, can be represented as an FSM. Typical states of such an FSM include a beginning and an end of an action. These states can be used for the initialization and proper release of variables, which are used by further low-level systems. Depending on the action, further states can be included between the beginning and the end of an action, in which the agent can trigger and react to changes from the low-level systems.

For example, the action *GoTo* will initialize the goal position and forward it to the *Path-planning* system. Once this systems returns either a failure or a valid path, the state of the *GoTo*-FSM will change to fail the action or to start the navigation respectively. When the agent starts navigating, both the *Navigation* and the *Animation* system run corresponding functions.

The details of how exactly these systems communicate with each other depends on the underlying architecture of the (simulation) environment and the agent and are not relevant for this work. Depending on whether the navigation succeeds or fails, the final state of the FSM releases the variable with the goal position and forwards the success or failure signal of the *GoTo* action to the middle layer. Depending on the approach used in the middle layer, this signal is then handled either in the task-BT or by the agent's execution component directly.

5.4 Evaluation

The experiments of this chapter were performed with the intention to measure the success of the goals defined in Section 5.1. In particular, goal $G1$ was the reduction of the global replanning frequency through local autonomy and failure management. The second goal $G2$ was a faster and more successful execution through reactive behaviors provided by the hybrid approach than achieved by a pure planning approach, which was addressed by the use of BTs on the middle architecture layer. The experiments and results described here were previously published in a prior work [157].

Given the environment described in Section 5.2 and the common task of catching *all* zombies into cages, the experiment scenario has a clearly defined end. Therefore, the evaluation measurements for each experiment run can be taken between the start of a simulation run and the point of time when all zombies are caught in cages. In order to evaluate the success of the first goal, we measure the average number of times that a global replanning is triggered before achieving the common task. Assuming that the BTs of the hybrid approach will manage some local failures, the expected result is a lower replanning frequency with the hybrid approach than with the pure planning approach.

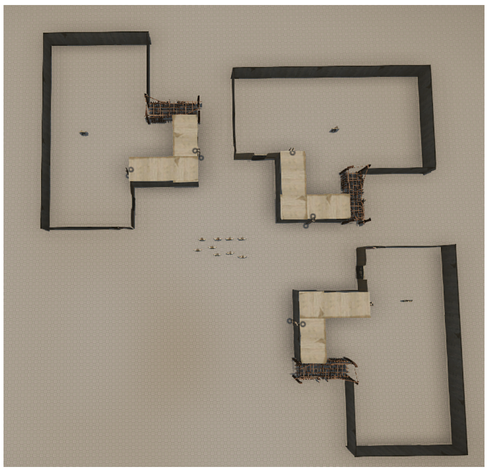


Figure 5.3: Top-down view of the experiment setup with 3 rooms (1 zombie in each room) and a group of 9 hunters in the middle.

For the evaluation of the second goal, we measure the average time required to achieve the common task. The idea behind this measurement is that when simply following the plan without any local autonomy, the agent might perform some unnecessary actions taking more time than required. Since the plan is based on the information that was available to the planner at plan-time, it is possible that the environment has changed by the time that the agent executes a task. For example, the agent will run to an outdated location to catch the zombie's attention, even if the zombie is much closer at this point of time. With the hybrid approach, however, the agent will recognize that the zombie's position has changed when executing this task through a Behavior Tree. This way, the agent can switch to the next task in its plan earlier.

Additionally, considering that the experiment fails if one of the agents dies, we assume that with the hybrid approach such failures should happen less often than with the pure planner approach. Similarly to the situation described above, reactive behaviors of the BTs should protect the agents from being hit by a zombie too frequently. For that reason, we measure the failure rate of the experiments performed with both approaches.

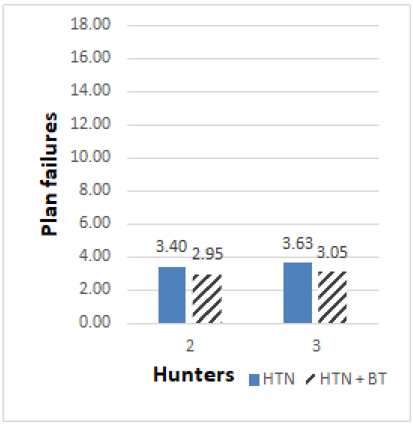
In order to compare the performance of both approaches under different complexities of the problem, the experiments were performed with the following setups. First, the number of rooms to be cleared from a zombie varied between 1 and 3. Second, different numbers of agents were tested in order to generate situations where multiple rooms could be cleared simultaneously by different agents as well as situations with sequential clearing of multiple rooms by the same agents. As already described in Section 5.2, the minimum number of agents required to clear *one* room is 2 and the maximum number is 3. For that reason, the total number of agents used in the experiments varied between 2 and $3 \times n$, with n representing the current number of rooms used in the respective experiment setup. That means, for 2 rooms, the maximum number of agents was 6 and for 3 rooms 9. All 3 rooms (with one zombie in each room) and 9 agents standing in the open space between the rooms are shown in the top-down view in Figure 5.3. The positions of the agents and zombies in Figure 5.3 represent their locations at the start of each simulation run. Additionally, selected experiment runs with different setups and different approaches are shown in the uploaded videos[5].

Each experiment setup was first tested 20 times measuring the experiment failure rate. In some cases the common task was not achieved due to an agent's death. Out of these 20 tries, only successfully finished experiments were used for the other two evaluations, the time that the agents required to achieve the common task and the replanning frequency. For that reason, more experiments were performed afterwards to measure the time and the frequency until 20 successful attempts were achieved for each setup.

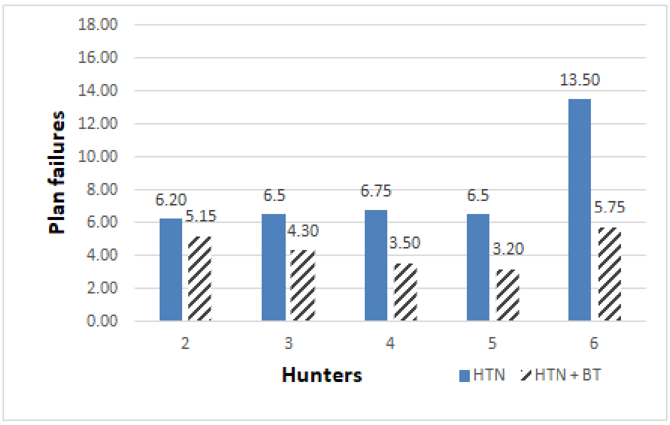
The average number of occurrences of global replanning are shown in Figure 5.4 for all experiment setups. The results give two insights. First, the replanning numbers of the pure approach, which uses an HTN planner only, are higher than the ones of the hybrid approach for all scenarios. Second, there is a (more) obvious increase of the replanning numbers with an increasing number of agents when using the pure approach in comparison to the hybrid approach.

The first point confirms our expectation in regards of goal $G1$ that, with Behavior Trees on the middle layer, the agents are able to react to environmental changes and solve some problems locally preventing global replanning. In some experiments we could observe situations where agents with the *hybrid* approach managed to avoid an obstacle (represented by another agent) or moved to a zombie's actual position without triggering global replanning. However, if an agent was not able to move because of an obstacle or did not find the zombie at its *planned* position, the *pure* approach triggered global replanning immediately.

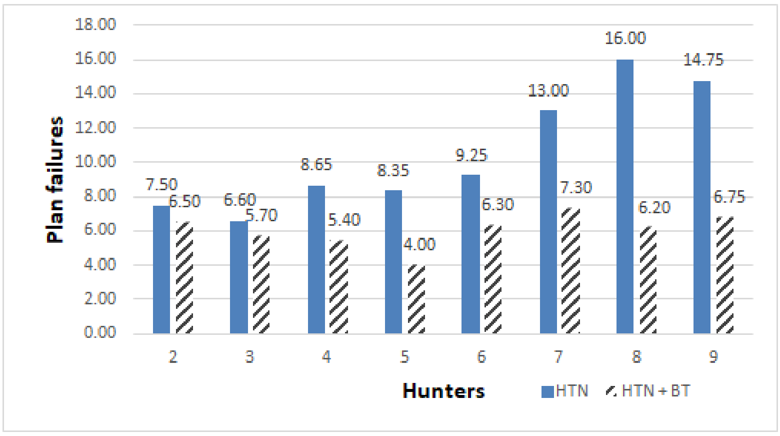
[5]Experiment videos: `https://bit.ly/2MW1OdY`



(a) 1 room with 1 zombie.



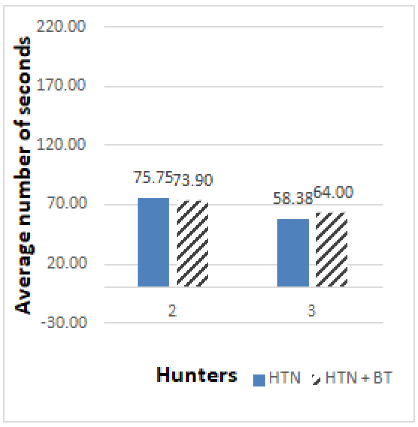
(b) 2 rooms with 1 zombie each.



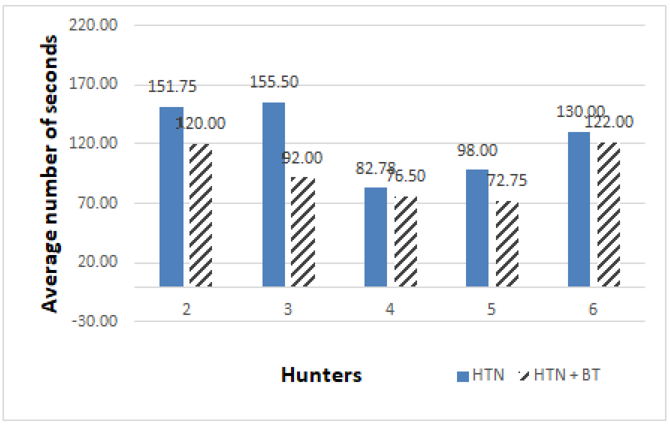
(c) 3 rooms with 1 zombie each.

Figure 5.4: The average number of plan failures that triggered global replanning before the goal was reached. Measured with 1, 2, and 3 zombies and a varying number of hunters using a pure HTN planner and the combination of an HTN planner with Behavior Trees [157].

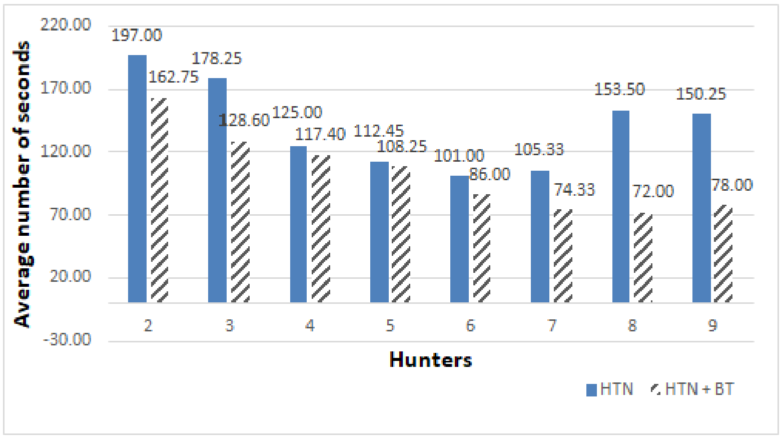
With an increasing number of agents, more coordination was required at execution-time. However, the higher number of actors (agents and zombies) caused even higher dynamics in the environment. Looking at Figure 5.4c, we can see that these changes affected the pure approach more than the hybrid approach. This, again, confirms that the environmental changes were small enough for the agents with the *hybrid* approach to handle them locally. However, the agents with the *pure* approach required a new global plan in case of such changes. For that reason, only the failure numbers of the *pure* approach grew with a higher complexity of the environment.



(a) 1 room with 1 zombie.



(b) 2 rooms with 1 zombie each.

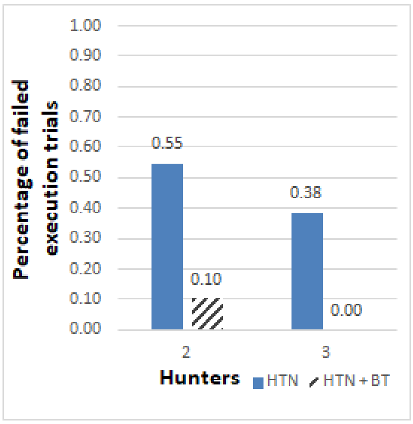


(c) 3 rooms with 1 zombie each.

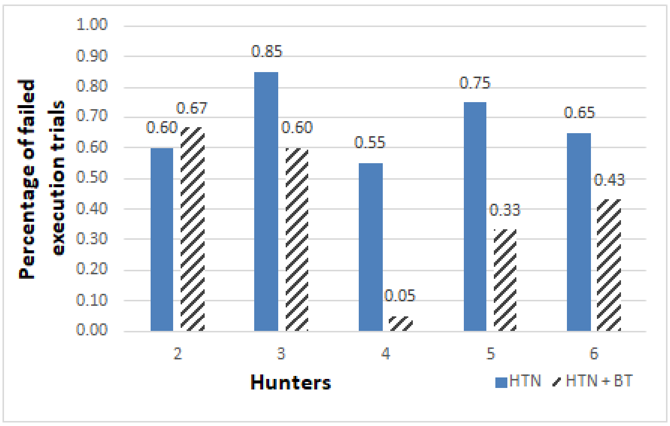
Figure 5.5: The average time in seconds required to reach the goal. Measured with 1,2, and 3 zombies and a varying number of hunters using a pure HTN planner and the combination of an HTN planner with Behavior Trees [157].

The higher reactivity of agents with the hybrid approach can also be observed in the average time required to achieve the group task, which is shown in Figure 5.5. In most scenarios the agents with the hybrid approach were able to clear all rooms faster than the agents with the pure approach. The difference, however, is not as high as expected. This can be explained by the fact that, even though the agents with BTs could perceive local changes and react to local failures faster, they sometimes required multiple attempts to solve these failures, which resulted in slightly longer execution times.

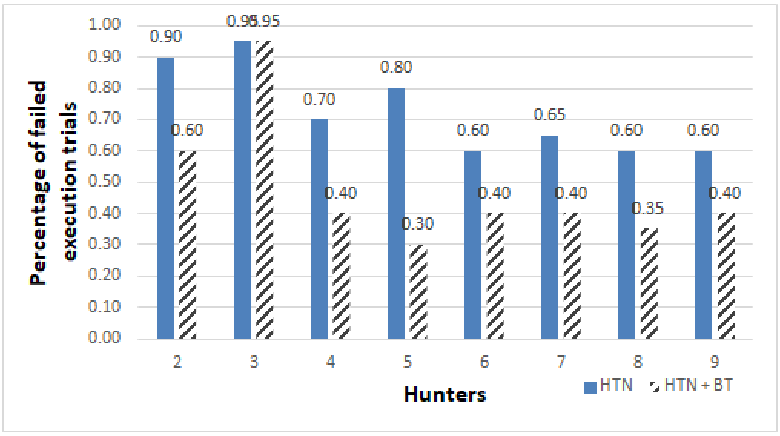
For example, as mentioned in Section 5.3.4, if an agent encountered an obstacle between itself and the object it was required to interact with (a door), its BT tried to make the agent step aside and try to interact again. This solution worked if the obstacle was static. However, if the obstacle itself was moving (if it was another agent) or if the first agent step further *away* from the interactable object, its following attempts failed as well. This behavior, however, results from a bad design choice made during the BT creation, which can be solved by either adding more run-time checks in the corresponding branch or by solving this problem in a different way than letting the agent step aside. For that reason, we assume that an even faster execution can be achieved with the hybrid approach.



(a) 1 room with 1 zombie.



(b) 2 rooms with 1 zombie each.



(c) 3 rooms with 1 zombie each.

Figure 5.6: The rate of failed execution attempts, which occurred due to a hunter's death trying to catch 1,2, and 3 zombies by varying numbers of hunters using a pure HTN planner and the combination of an HTN planner with Behavior Trees [157]

Lastly, the rate of failed experiments is shown in Figure 5.6. Out of 20 experiment runs for each scenario, these were the ones where an agent died and the common task could not be achieved. As the results shows, the failure rate was high for both approaches for scenarios where only 2 or 3 agents were assigned to clear more than one room, which is reflected in the columns for 2 and 3 hunters in Figures 5.6b and 5.6c. Even with BTs, the agents got hit by a zombie in rare cases and, with more then one room to clear, the agent that ran into the rooms had a higher chance to be hit by each zombie. Therefore, even with the hybrid approach, the agents died quite frequently in these scenarios.

However, in cases where enough agents were assigned to execute the task, the failure rate of the *pure* approach was significantly higher. This confirms the expectation that the use of BTs allows the agents to be more reactive and execute the tasks in a safer and more successful way. Even though the agents still got hit, they were able to evade enough hits to finish the task. The results from both Figures 5.5 and 5.6 confirm the expectations regarding goal $G2$.

Since both approaches share the same high-level HTN and the task roles are assigned in the same way, we assume that all results can be improved for both approaches in general. However, the experiments performed here were intended to show the differences between a pure planning approach and an approach with interleaved high-level planning and reactive low-level decision-making and execution. These differences can be observed in the presented results. Therefore, even with an improved planning domain for both approaches, we expect that the hybrid approach will outperform the pure HTN approach.

As for the goal $G3$ of providing an approach that allows for an intuitive and easy creation and maintenance, we have shown that it is possible to create the domains for the given scenario with the use of existent systems. Due to the human-readable syntax of the planning language used by derPlanner the pure planning domain could easily be transformed into the hybrid domain by simply removing some primitive tasks and transforming former compound tasks into primitive tasks. Through the provided compiler, there was no need to modify the program code when changing the domain. Additionally, reactive behaviors of the hybrid approach could be achieved with the help of Behavior Trees provided by the CryEngine without any modifications to the trees. The combination of standard nodes in the trees and the possibility to execute low-level actions allowed for improved local behaviors, even with simple BTs, which are shown in Appendix C. The resulting BTs were created with the help of CryEngine's BT editor[6], which allows for intuitive BT generation through a graphical interface. The possibility to add nodes through drop-down lists ensured the syntactic as well as semantic correctness of the BT. Furthermore, the provided BT debugging interface allowed to observe the behaviors of the agents at execution-time and recognize potential opportunities for improvement. An additional advantage of the proposed combination was that each BT could be changed without affecting the planning domain or requiring any changes in the program code.

Although manual generation of BTs means a higher engineering effort, it is not necessarily undesired. It can even be required for some environments where full human control over the agents' behaviors is necessary in order to ensure robustness and coverage of all potential situations. In this case, BTs are an expressive tool that is easily understandable and maintainable by human users. On the other hand, an interesting direction for future work includes research on automatically learning such BTs when high-level tasks are provided, potentially using one of the approaches described in Section 2.1.1.

5.5 Conclusion

In this chapter, we have introduced a hybrid planning-and-execution approach that combines a Hierarchical Task Network with Postconditions and Behavior Trees. Combining these two decision-making techniques, the hybrid approach follows the general idea proposed in Chapter 4. The proposed approach is intended to be used in multi-agent scenarios where the agents are required to follow common goals as a team while each agent's well-being and its goals remain important. In order to achieve the common goals, the agents need to work cooperatively and coordinate their actions in certain situations.

[6]Documentation on the Behavior Tree editor of CryEngine:
`https://docs.cryengine.com/display/CEMANUAL/Behavior+Tree+Editor+Window`

In the proposed architecture, the top layer consists of a centralized HTN_p planner that ensures a tightly-coupled agent coordination on the high behavioral level. The planner works with an abstract world model and generates a high-level plan for each agent. Such plans do not contain any details that can be easily invalidated at execution-time. Instead, the architecture relies on the middle layer to refine every plan step according to the actual world state at its execution-time.

For that purpose, each primitive task is represented by a distinct Behavior Tree on the middle layer. Here, the architecture is decentralized with each agent following its own plan. When executing a certain plan task, an agent selects the corresponding Behavior Tree, which, in turn, tries to refine the plan task taking into consideration run-time information, which was not available to the planner at plan-time. Due to its modularity, a Behavior Tree allows for complex and yet flexible behaviors that can run sequentially or in parallel. This way, it also enables agents to wait for specific coordination signals ensuring coordinated execution of certain tasks. The lowest architecture layer is responsible for actual execution of primitive actions and knowledge gaining through sensory systems.

The main goal of this chapter were the evaluation of hypothesis that such separation between high-level long-term planning and low-level reactive decision-making can reduce the global replanning rate. The idea behind this hypothesis was that most plan failures happen locally can be managed by an agent without invalidating the overall plan and triggering replanning for all agents. Another goal was the evaluation whether such three-layer architecture can improve the execution of plans in terms of time and success rate. For that purpose, the proposed approach was compared against a pure planning approach in a multi-agent scenario. The pure planning approach consisted of a planner only, which was responsible for detailed plan creation and did not require plan refinement. The observed scenarios included $2 - 9$ agents where pairs of agents were required to coordinate their actions at different points of time.

The experiments have shown that the hybrid approach does reduce the global replanning frequency compared to the pure planning approach. Furthermore the results have shown that due to the higher flexibility in the agent behaviors when using the Behavior Trees, the agents have a higher rate of successfully achieving the common goal and can do this in a shorter time period. Although the time difference is smaller than initially expected.

Another important insight of this chapter is that the creation of the hybrid approach is possible with only minor program changes to the HTN planner and no changes to Behavior Trees. This allows for a wider use of the hybrid approach due to the existing expertise on these systems. Since postconditions are encoded in the structure of the Behavior Trees and not in the planning domain, any planner and any Behavior Tree system can be used for this purpose. The main addition is the architecture around these approaches enabling the connection and the communication between the three layers. This includes the translation of plan steps into Behavior Trees and the possibility of the Behavior Trees to check pre- and postconditions of a plan step. Another advantage is that both approaches allows for a well-readable domain definition and maintenance. At this point, the transformation was done manually. On the one hand it means additional effort for a human developer. On the other hand, it allows developers to keep full control over the agents' behaviors and ensure their robustness. However, if full control is not a required criterion, Behavior Trees can be automatically generated as described in Section 2.1.1.

Chapter 6

Hybrid Approach II : HTN + MCTS

The following chapter introduces a hybrid planning-and-execution approach that, similarly to the approach described in Chapter 5, is based on the general idea of a three-layer architecture described in Chapter 4. The focus of this chapter is on large search spaces with a large number of actors with common goals where the achievement of the common group goals is more important than each individual's goals. Due to the large number of actors, the considered environments are expected to have even more dynamics. The proposed solution combines a planner that operates on a Hierarchical Task Network with Postconditions (HTN_p), which is used for strategic long-term planning and Monte Carlo Tree Search (MCTS) for tactical short-term decision-making. Due to the use of Monte Carlo Tree Search, the major requirement for this approach is the availability of a Forward Model (or simulation model) of the environment.

The hybrid approach is tested in an adversarial real-time game environment. The following sections describe the goals, the test environment, and the details of the hybrid approach. Afterwards, intermediate experiment setups and their results are discussed. Next, an extension to the approach is proposed that allows for an improved achievement of the high-level tasks through MCTS. The chapter concludes with an evaluation of the extended approach. The presented solutions and their evaluation have been published in previous works [79, 158].

6.1 Goals

The intention of this chapter is the introduction of an approach that can scale in terms of the number of agents that it controls. In order to allow for fast planning and execution for a high number of agents, the approach should allow for both the planning and the decision-making at execution time being performed on discretized world models. Concrete goals of this work are described in the following.

G 1: Introduction of an architecture that allows for a combination of strategic and tactical behaviors in a multi-agent environment that can work in large search spaces. The architecture should allow for long-term strategic planning, which can be refined into short-term tactical decision-making at execution-time while monitoring the environment and adapting the tactical behaviors according to the environmental changes. As an extension to the scenarios regarded in the previous chapters, the focus of this chapter are multi-agent scenarios with even larger search spaces through a) a higher number of agents, and b) heterogeneous agents. Furthermore, this chapter's scenarios deal with additional uncertainty through an adversarial setting and a high number of opponent agents.

G 2: No explicit/manual generation of tactical behaviors. Since this chapter deals with even more complex problems than those regarded in the previous chapters, typical execution systems that are used, for example, for spacecraft control (see Section 2.3.1) and rely on fully predefined domains are not suitable because it is not possible to pre-define the combinations of low-level behaviors that cover all edge-cases for all agents. For that reason, another goal of this chapter is the introduction of an approach that does not require manual generation of such behaviors but instead allows for run-time decision-making based on simulations of the environment.

G 3.1: Guidance of the mid-level reactive approach by the high-level planner. The proposed approach is based on the three-layer architecture with the top layer consisting of a planner and the middle layer consisting of MCTS. Depending on the high-level strategic plan tasks, the tactical behaviors should be distinguishable from each other and each of them should aim to achieve the corresponding high-level task. However, assuming that the tactical behaviors are not manually created (goal $G2$), the next goal of this chapter is to provide a way for the planner to guide the reactive approach (in this case MCTS) in its search for the correct tactical actions. In particular, this requires an interface between the two top layers that will allow for *implicit* guidance.

G 3.2: More efficient execution of tactical behaviors through an improved guidance from the planner. Once it is possible to guide the middle layer approach by the planner (Goal G 3.1), it is important to optimize the execution of tactical behaviors. This should, in turn, lead to a better performance in regards to the strategic high-level plan execution.

6.2 Test Environment

In order to evaluate the hybrid approach proposed in this chapter, we have selected a highly dynamic adversarial research environment called microRTS[1]. This environment represents a simplified version of a typical Real-Time Strategy (RTS) game. Its major differences to most commercial RTS games are the simple graphics used to represent the environment and the discrete world, in which the actors move on a $2D$ grid. Nevertheless, it offers the major mechanics of a typical RTS game such as collecting resources, building different types of buildings, and creating different types of agents. Additionally, in contrast to commercial games, microRTS offers a Forward Model, which can be accessed by an agent playing the game. For these reasons, it has been successfully used as a benchmark environment for artificial agents in the microRTS AI Competitions since 2017 [159]. The environment is written in Java and therefore artificial agents playing the game have to be written in Java as well.

In microRTS, two *players* play against each other. A player can be a human player or an artificial agent. Each player can control multiple *units* at the same time. Units are static buildings – either a *base* or a *barrack*, as well as dynamic units – *workers* and *light, range,* and *heavy* units. The last three units are military units and can only move and attack the opponent units, whereas a worker can also gather resources, bring them to the base, and build further buildings. A base, in turn, can produce more workers, whereas a barrack can produce military units. In contrast to other more complex RTS games, there is only one type of resources in microRTS, different amounts of which are used for the creation of both static and dynamic units. Each unit has a predefined number of Health Points (HPs). Additionally, each type of dynamic unit has a different amount of damage that it deals and a different movement speed. For example, light units move fast but deal little damage, whereas heavy units are slow but deal more damage. Additionally to resources and player units, there are non-destructible *walls* in some maps of microRTS. All units are the size of one grid cell on the map.

There are multiple maps of different sizes provided with microRTS, all of which are symmetric. The players start a match in the opposite sides of a map and have an equal number of units present in the beginning. They are required to provide a so-called *player action* to the environment in every frame. A player action represents the *combination* of actions for all of the player's units that can start a new action. Similarly to the environment described in Chapter 3, actions in microRTS are durative and therefore a unit is only able to start a new action after finishing the previous one. The frame time that a player can use to make a decision is limited to 100 milliseconds. Exceeding this limit multiple times leads to a disqualification of the player from the competition.

[1]microRTS: `https://sites.google.com/site/microRTSaicompetition`

A typical game will usually progress as follows: a player will first start gathering resources to be able to build barracks and increase the size of their army. Once the army has the desired size, they will start an attack on the opponent player. Therefore, a high-level strategy will usually consist of a *Game Opening*, a *Middle Game*, and an *End Game*. Depending on the map (size) and the start configuration of the players' units, the details of these parts can vary and require different tactics on the lower levels. For example, on very small maps it is not feasible to build more buildings. Instead a so-called *rush* is a better strategy to quickly attack the opponent with available units. On the other hand, if the game starts with players having enough resources, they can begin the middle game immediately.

Since there are some benchmark agents provided with microRTS and all agents from previous competitions are available, it is possible to evaluate an agent's performance against existing opponents.

In its default version, microRTS offers full observability and deterministic actions. Although, it is possible to enforce partial observability by limiting each unit's sight. Additionally, there is an option to introduce non-determinism to actions, which, for example, results in non-deterministic damage dealt. In this work, however, we use the default version of the environment with full observability and determinism.

Nevertheless, due to its adversarial nature and a possibly large number of units on both sides, which can select from multiple different actions, and a possibly big size of the map, the size of the search space is large. For example, for a map of size 12×12 cells the branching factor can exceed 10^{17} [42]. The main source of uncertainty in this environment are the opponent's actions. Additionally, more uncertainty is added if multiple units try to step onto the same grid cell at the same time. In the non-deterministic version of the game, one of the units is selected randomly and is allowed to do its move while all other units' actions are cancelled. However, in the default version used in this work, all units' actions are cancelled in such a case.

6.3 Hybrid Approach

Following the general idea of a planning-and-execution system as described in Chapter 4, the following section describes the implementation of a hybrid approach that includes an HTN_p planner and MCTS and is used for centralized control of multiple units in a real-time environment. Based on the goals defined in Section 6.1, the architecture aims to let the planner provide guidance to MCTS through an interface allowing for strategic and tactical behaviors.

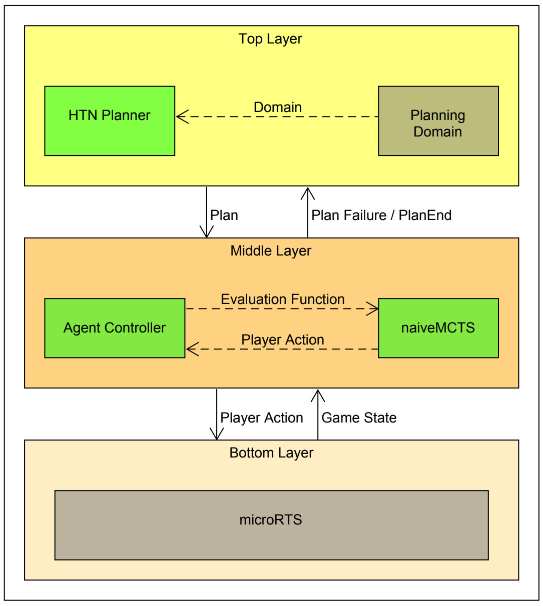


Figure 6.1: Three-layer architecture for the hybrid approach including a centralized HTN planner on the top layer and a centralized Agent Controller with naïveMCTS on the middle layer.

6.3.1 Three-layer Architecture

The hybrid approach presented in this chapter is built as a three-layer architecture that is shown in Figure 6.1. With the goal to introduce an architecture that allows both for strategic and tactical behaviors, which in their nature can be easily represented by hierarchies, a centralized HTN_p planner is used in this chapter for strategic long-term planning. As we have already seen in Chapter 5 such a centralized planner allows for co-ordination of multiple units on a high level without requiring inter-unit communication.

The high-level strategies are refined on the middle layer. This layer uses a centralized MCTS algorithm to select combinations of actions for the agent's units. While making decisions about the actions, it takes into consideration the high-level task provided by the planner. The task itself, is represented by an Evaluation Function that can be used by MCTS for Game State evaluation. That way, a strategic task implicitly guides the selection of tactical behaviors during execution. In addition to tactical decision-making, the middle layer is responsible for environmental monitoring and recognition of certain events and plan failures. Similarly to the architecture described in Chapter 5, the middle layer notifies the top layer about plan failures and requests new plans.

Given the current environment, the lowest layer of the architecture is not part of our work. Similarly to the architectures used in previous chapters, the lowest layer is responsible for information gathering and the actual execution of each unit's action. However, microRTS does not allow the agent to control each unit separately. Instead, a *Player Action* that includes the combination of all unit's actions is passed to the game and its corresponding systems take care of the low-level execution. Similarly, the game provides an aggregated representation of the game state, which includes information about all units.

6.3.2 Planning Domain

Since the planner is responsible for high-level strategies, dealing with low-level details during the planning phase given the large search space is very inefficient. For that reason, the planning domain is kept very abstract. The use of an abstract world model as well as abstract actions is very common in RTS game environments and has been mentioned in multiple previous works [36, 37, 46, 160].

Furthermore, due to the high dynamics of the environment, keeping track of the absolute positions of every *dynamic* unit in the planning domain can result in a large overhead. Therefore, in addition to abstract high-level tasks and an abstract world representation, the domain only includes facts that describe the absolute positions of *static* units such as buildings and resources, whereas information about only the *most important dynamic* units is represented in the planning domain. This includes, for example, the *distance* of the *closest* opponent unit to a friendly building but not its exact position or the positions of any other units. Further facts include abstract information such as the closest resource point for every friendly base and the current number of friendly and opponent moving units.

The test environment used here is written in Java and therefore all agents have to be implemented in Java as well. Therefore, a planner very similar to the one used in Chapter 3 is used in this chapter's hybrid approach. The planner is, again, domain-specific and therefore the planning domain is written as Java code directly. Similarly to the definition used in Section 3.3.2, we define an HTN_p planning domain as $D = (F, C, A, M)$, consisting of sets of facts F, compound tasks C, primitive tasks A, and methods M respectively. Each task and each method is represented by a separate Java class.

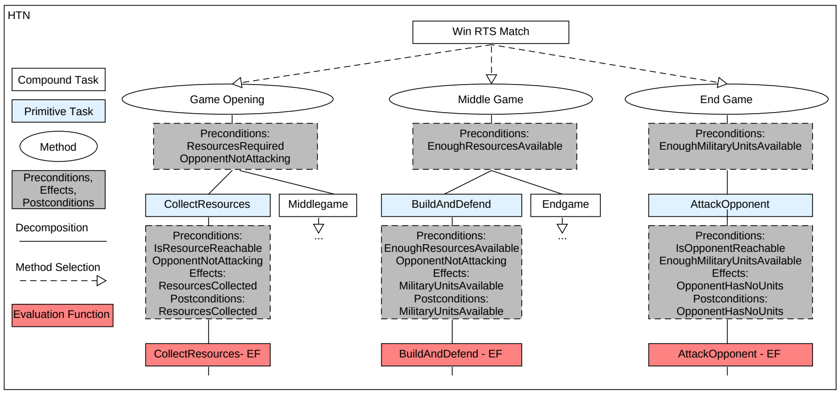

HTN
Win RTS Match
Compound Task
Primitive Task
Method
Preconditions, Effects, Postconditions
Decomposition
Method Selection
Evaluation Function
Game Opening
Middle Game
End Game
Preconditions: ResourcesRequired OpponentNotAttacking
Preconditions: EnoughResourcesAvailable
Preconditions: EnoughMilitaryUnitsAvailable
CollectResources
Middlegame
BuildAndDefend
Endgame
AttackOpponent
Preconditions: IsResourceReachable OpponentNotAttacking Effects: ResourcesCollected Postconditions: ResourcesCollected
Preconditions: EnoughResourcesAvailable OpponentNotAttacking Effects: MilitaryUnitsAvailable Postconditions: MilitaryUnitsAvailable
Preconditions: IsOpponentReachable EnoughMilitaryUnitsAvailable Effects: OpponentHasNoUnits Postconditions: OpponentHasNoUnits
CollectResources- EF
BuildAndDefend - EF
AttackOpponent - EF

Figure 6.2: Abstract HTN_p with 3 Evaluation Functions (EFs) used by the hybrid approach II in microRTS.

However, there are two major differences between the planning domain in Chapter 3 and the one used in this chapter. Since the planner in Chapter 3 was the only decision-making mechanism in the agent architecture, it was planning on actual low-level actions that could be directly executed by the agent. For that reason, it was not necessary to define effects of primitive tasks. Instead, tasks could be directly simulated during the planning phase using the game's simulation model. Here, the environment also provides a simulation model. However, this chapter's planner deals with abstract high-level actions, which cannot be directly simulated by the simulation model of microRTS. For that reason, it is necessary to define effects of the abstract primitive tasks used in this domain. Therefore, in addition to preconditions, a primitive task is also defined by its effects.

The second difference to the domain from Chapter 3 are postconditions of primitive tasks. The hybrid approach described in this chapter is based on the general idea of task refinement at execution-time from Chapter 4. Therefore, primitive tasks of this chapter's planning domain are also defined by their postconditions that are used at execution-time by the middle layer. Postconditions were not present in the domain in Chapter 3 since there was no task refinement at execution-time and a task's end was equivalent to the end of an action. Furthermore, the architecture used in Chapter 3 was designed for a single agent who was the only agent responsible for achieving a task. Here, however, the architecture is designed for multiple agents and a more dynamic environment. Therefore, an abstract task can be achieved not only through the agents' actions but also as a result of environmental changes that are not controlled by the hybrid approach (such as the water filling a barrel example from Section 4.4.) Therefore, here, a primitive task is also defined by its postconditions and the planner is working on a Hierarchical Task Network with Postconditions.

As already mentioned in Section 6.2, an RTS game typically consists of a *Game Opening*, a *Middle Game* and an *End Game*. These abstract compound tasks are shown in Figure 6.2 as the highest level of the HTN_p used in this chapter. In practice, each of these 3 tasks can be decomposed by different methods that depend on the map size and the start configuration of the game. Nevertheless, independent from the decomposition path, there are only the following 4 primitive tasks that can be used as part of a plan: *CollectResources*, *BuildAndDefend*, *AttackOpponent*, and *PreventAttack*. The first three tasks are used in this order in a typical game scenario. The task *PreventAttack* is usually not part of an initial plan as its purpose is to defend the friendly base from an attack, which usually does not happen right at the start of a game. However, it can become part of the *Game Opening* or the *Middle Game* in case the opponent attacks first. In this situation, the agent will recognize the attack (by monitoring the environment on the middle layer) and trigger a replanning. The new plan will consist of the *PreventAttack* task followed by other tasks according to the situation.

Although typically an agent will proceed with the tasks in the order *CollectResources*, *BuildAndDefend*, *AttackOpponent*, it can be required to return to a previous task if the preconditions for the current task no longer hold. This, however, can result in oscillating behaviors. The fact that *postconditions* of the *current* task (that are used at execution-time to recognize a task's end) can be different from the *preconditions* of the *following* task (that are also used at execution-time to check the next task's validity) allows for fuzzy rules for switching back and forth between tasks. They can help preventing oscillating behaviors. For example, postconditions of the *BuildBarracks* task can require to have 1 barrack and the number of friendly military units has to be $m_o + 3$, where m_o is the number of the opponent's military units. This way, once the number of the agent's military units outweighs by 3, the building task can be considered as achieved and the agent can switch to the attack task. Consequently, all its military units will stop protecting the friendly base and will start moving towards the opponent's base.

Assume that now the opponent reacts to the attack and destroys one of the agent's units on their way. If the preconditions for the *AttackOpponent* task also require $m_o +$ 3 military units, the task will be invalidated and the agent will switch back to the *BuildBarracks* task. This will force all units to return back to their own base. And if one unit is produced again very quickly, the movement will result in an oscillation. However, if the preconditions of the *AttackOpponent* task are set to lower numbers than the ones used for postconditions of the preceding *BuildBarracks* task, the generated hysteresis buffer will allow for faster switches to the *next* plan tasks and delayed switching to the *preceding* task are possible.

With the planner being able to plan on abstract high-level tasks, it is the responsibility of MCTS to refine these tasks during the execution. In order to allow for such refinements, it requires a way to translate the descriptive representation of an HTN_p task into a mathematical representation that can be used by MCTS. For that purpose, we propose assigning Evaluation Functions (EFs) to primitive tasks. These EF represent the most important extension to primitive tasks in the planning domain of this chapter.

As already mentioned in Section 2.1.2, MCTS samples different action sequences trying to find the optimal action to be executed in the next step. In order to decide what action is the most *optimal* one, the game state resulting from each action sequence is evaluated with the help of an EF. Depending on what behaviors and what outcomes are desired, such an EF can consist of multiple objectives. For example, if an agent in microRTS is supposed to *CollectResources*, it has to minimize the *distances* of its worker units to the resources, maximize the *number of resources* that are carried by each worker while at the same time trying to avoid being hit by opponent units and therefore maximizing the *number of* all its units' *HPs.* Whereas for the *AttackOpponent* task, it does not care about the resources but tries to maximize the friendly units' *HPs* while minimizing the *distance* to the opponent's base and minimizing the opponents' units' *HPs.*

As shown in Equation 6.1, we represent such a multi-objective Evaluation Function E as a weighted sum over Evaluation Functions e_j that optimize the distinct objectives x_j. Depending on the task that E is representing, the number K of its objectives and the values of the distinct weights w_j can vary. Respectively, we define a primitive task $t \in A$ by the tuple $(pre, add, del, post, E)$ – its preconditions, effects, postconditions, and its Evaluation Function.

$$E = \sum_{j=1}^{K} w_j e_j(x_j) \tag{6.1}$$

The possibility to combine multiple objectives in an EF also allows for implicit parametrization of each unit's actions. For that purpose, instead of using explicit parameters in plan tasks, the planning domain contains facts that can be used within objectives of an EF. Such a fact, for example, can be the ID of the enemy unit that is closest to the agent's base. If such an enemy unit comes too close to the base, the planner will schedule the *PreventAttack* task. During the execution of this task, the knowledge about the closest (and therefore most dangerous) enemy unit can be used to minimize the distances between every agent's unit and the enemy unit, allowing for a focused and quick elimination of the enemy unit. If multiple bases are available, it is possible to track the closest enemy unit for *each* base and assign friendly units to defend different bases. In case of an attack, MCTS will then minimize the distances between each friendly unit and the attacker of the base that the friendly unit is assigned to.

With the use of EFs, we provide a possibility for the planner to guide the decision process of MCTS without explicitly pre-defining each unit's behaviors for every possible situation. That way, we aim for the goals described in Section 6.1. More details on how these EFs are forwarded to and used by MCTS are given in Section 6.3.4.

6.3.3 Top Layer

The top-most layer of the proposed architecture consists of a centralized planner and its planning domain. Similarly to the previous approaches, the planner used here is a total-order HTN planner that uses ordered method selection. It is domain specific as it uses the Java planning domain described in Section 6.3.2, which includes the postconditions of primitive tasks and the newly introduced Evaluation Functions for MCTS.

Although the planner generates common strategies for multiple units, it is different from the multi-agent planner described in Chapter 5. Here, the planner does not distinguish between the units and does not assign sub-plans to each unit. Instead, it generates a *single* strategy for *all* units. This way, the result of the planning procedure is one single plan. The assignment of units actions is done later on the middle layer at execution-time.

Not dealing with distinct unit plans and operating on an abstract model offers the planner a major advantage in terms of computational time, which can otherwise become a problem with high numbers of units that are present in an RTS game. Furthermore, this way, the combinatorial problem is forwarded to MCTS, which is better suitable for dealing with such problems, as described in more details in the next section.

In the future, however, the planner can be adapted to be able to deal with hierarchies of units. For example, units can be assigned to squads according to some metric, as mentioned in some prior work in Section 2.3.2. This way, the planner can handle a more detailed model and assign more detailed sub-plans to each squad. This can be especially useful for larger environments, where different groups of units are required to execute different tasks. For example, in the more complex game StarCraft division of units are more desired than on the relatively small maps of microRTS.

6.3.4 Middle Layer

The design of this chapter's middle layer is, to some extent, enforced by the design of the environment and by the interface that the agent can use in order to operate in the environment. For that reason, the layer consists of a central controller, which according to the requirements, extends the *AI* Java class provided by microRTS and, for reasons described below, directly extends the provided naïveMCTS class.

In contrast to the environment described in Chapter 5, the level of detail of the action execution is less granular here. In Chapter 5 the agents were supposed to execute their actions in a very precise manner since they were acting in a continuous environment. There, each agent should be in the correct location and have the required rotation in order to be able to interact with an object. This was especially important when two agents were required to cooperate. In contrast, in microRTS, the units are moving on a $2D$ grid and only have 4 different (movement/rotation) directions. Therefore, the level of detail in this discrete environment is much lower.

Additionally, in Chapter 5 the number of agents was relatively low and we focused rather on the local effects of cooperation between two agents. Here, however, friendly units do not explicitly cooperate with each other. Instead, they can be regarded as a swarm of units, which is required to execute a common task with implicit coordination only. For these reasons, the focus of this chapter's middle layer is on the achievement of the common high-level task rather than on the detailed actions of each unit.

Nevertheless, the agent is controlling each unit and should return the combination of all units' actions to the game environment. Therefore, its aim is to find a combination of unit actions that contributes the most to the common goal. Searching for an optimal solution through a full tree search over all possible action combinations, however, is not feasible if either the computational budget is limited or if a decision needs to be made at run-time, both of which apply to microRTS.

However, as already pointed out in Section 2.1.2, such a problem can be regarded as a Combinatorial Multi-Armed Bandit (CMAB) problem over a set of variables $X = \{X_1, ..., X_n\}$, where each *unit* is represented by a variable $X_i \in X$ and the unit's *actions* are represented by the values of X_i. Regarding the search problem as a CMAB allows naïveMCTS to perform a search over combinations of unit actions. As already mentioned in Section 2.1.2, naïveMCTS is a version of Monte Carlo Tree Search, which on its own is particularly known for good performance in large search spaces. Furthermore, naïveMCTS has been successfully used for agent control in microRTS previously [34, 42]. This version is both adapted to perform simulations on durative actions and to approximate the joint reward of an action combination as the sum of rewards over the distinct units' actions. The fact that microRTS provides a forward model, in general, allows for the use of simulation-based methods such as MCTS.

The goal of naïveMCTS in the scenario presented here is the maximization of the total reward gained from selecting a combination of unit action. The reward is usually computed with the help of an Evaluation Function (EF) that evaluates the game state at the end of a simulation (more details on MCTS and naïveMCTS are given is Section 2.1.2). In the architecture proposed here, this EF is provided by the top-layer planner and differs depending on the high-level task that is currently executed.

That way, whenever the agent is required to provide a Player Action to the game, it forwards the EF of the current plan task to naïveMCTS. Similarly to the game environment described in Chapter 3, this is done in every frame. Therefore, in contrast to some other works, which regard MCTS as a *planning* mechanism [110,111], we use it for *reactive* decision-making in every frame. It is reactive since it makes a decision based on the most-current data and therefore can react to any events. Furthermore, even though MCTS internally simulates *sequences* of actions, which can be regarded as plans, it only returns *one* single action to be performed in the current step. It does not provide a full plan.

Although very similar on the first look, there is a major difference between our work and the approach described in [111,112]. First, those works use a pure *Monte Carlo* method, which in contrast to MCTS does not include a tree search. Second, the time and the direction of the communication and control between the planner and the corresponding *Monte Carlo*-based technique are different. In those works *Monte Carlo* rollouts are performed on applicable planning methods during task decomposition, while in our case a *planner guides* MCTS during the execution phase, which simulates actual low-level actions and not planning methods. A major disadvantage of the former work, however, is that during plan decomposition, the planner relies on the *Monte Carlo* approach to finish its simulation. That means, it has to simulate until a full decomposition of a compound task has been achieved. Only that way primitive tasks can be added to a plan for execution. This, however, can take a long time and delay the execution as pointed out in [112].

In contrast, in our hybrid approach the abstract plan is generated in an efficient way before the actual execution. If, for some reason, the planner does take more time for planning than the environment allows, the middle layer ensures that the agent's behavior stays robust. In this case, naïveMCTS simply uses either a default EF or the EF used in the previous cycle. Since MCTS can make a decision at any depth of the search tree, it can provide an action at any point of time, even if its computational budget is limited. That way, the agent is always able to make a decision through naïveMCTS even if the planner is unable to come up with a plan within the given time.

The complete workflow of the agent's *GetAction* function, which is called by microRTS in every frame is shown in Algorithm 2. Since in this chapter the planner is operating on an abstract world model and it is not required to generate a new plan in every step, its world state is not continuously updated. Instead, in order to provide more time for decision-making as described below, the planner's world state is updated when all units are currently executing an action and consequently no new action can be started and therefore no decision has to be made in this frame. Additionally, in order to ensure that the planner's world state does not become too outdated, it is updated at certain intervals, in our case every 10 frames (lines 5 – 10).

Afterwards, the general workflow is quite similar to the one described in Chapter 3. If the agent is able to assign a new action to at least one of its units, it first checks whether the currently executed task t has finished (line 14) using its postconditions. If it has not finished yet, the algorithm checks whether the task is still valid. To check the task's validity, the agent checks its preconditions against the current world state. In case the task was achieved, the agent needs to make a new decision before proceeding with the next task in the current plan. If, however, the task was not finished yet but became invalid, the agent has to replan and make a new decision given the new plan.

Making a new decision, in this case, means finding the next valid task – whether in an existent or in a new plan – and forwarding this task's Evaluation Function to naïveMCTS. This is done within the while loop in lines 22 – 37. As already mentioned, the time budget that is provided by microRTS to an agent is limited to 100 milliseconds. A *maximum* of 80 milliseconds of this time is allocated to the planner while the remaining time can be fully used by MCTS. The planner is allowed to use more time since replanning will presumably happen on rare occasions compared to the reactive decision-making of MCTS. As long as the agent is within the time budget, it can try to make a decision.

If replanning is required, a new plan is requested from the HTN planner in line 25. If a plan is available, the agent steps to the next (or first) task in this plan in line 30. Before starting to execute the task its validity is checked in line 31. If the task is valid in the current game state, its Evaluation Function E is selected in line 33. Otherwise another replanning is triggered in line 35. Note that in case the loop exceeds the time budget before having found a valid task, the previous Evaluation Function will stay selected. In the beginning of a game, a default EF is assigned to ensure a robust behavior of the agent. Finally, the current EF E is forwarded to naïveMCTS in line 38. This, in turn, checks internally for the remaining time budget and returns a Player Action trying to best balance between optimizing E and exploring all action combinations. The exact parameters of naïveMCTS used in this work's experiment are presented in Section 6.4. The Player Action provided by naïveMCTS is then forwarded to the game environment microRTS.

Algorithm 2 GetAction

Input: Current game state

Output: PlayerAction, which represents the combination of actions of all units of the playing agent

```
1: $s \leftarrow current\ game\ state$
2: $\pi \leftarrow current\ plan$
3: $t \leftarrow current\ task$
4: $E \leftarrow current\ evaluation\ function$
5: if cantExecuteAnyUnitAction or isTimeToUpdate then
6:    UpdatePlannerWorldState()
7:    if cantExecuteAnyUnitAction then
8:       return noAction
9:    end if
10: end if
11: $decisionMade \leftarrow$ false
12: $replan \leftarrow$ false
13: if $t \neq$ nil then
14:    if $t$ finished then
15:       {continue in line 22}
16:    else if $t$ running and $t$ valid then
17:       $decisionMade \leftarrow$ true {proceed with naiveMCTS}
18:    else {$t$ invalid}
19:       $replan \leftarrow$ true
20:    end if
21: end if
22: while $decisionMade =$ false and belowTimeBudget do
23:    if $\pi = nil$ or $replan =$ true then
24:       {generate a new plan}
25:       $\pi \leftarrow GeneratePlan()$
26:       if $\pi = nil$ then
27:          continue
28:       end if
29:    end if
       {get next plan task}
30:    $t \leftarrow next\ task\ in\ \pi$
31:    if $t$ valid then
32:       $decisionMade \leftarrow$ true {proceed with naiveMCTS}
33:       $E \leftarrow E_t$ {$evaluation\ function\ of\ t$}
34:    else {$t$ invalid}
35:       $\pi \leftarrow nil$ {continue}
36:    end if
37: end while
38: return naiveMCTS($E$)
```

6.3.5 Bottom Layer

The Player Action provided by the middle layer is forwarded to the lowest layer of the architecture. A Player Action contains an action for each free unit. On the lowest layer, these actions are executed by microRTS itself and therefore the lowest layer is not an actual part of our architecture. Here, microRTS updates the units' health points, their locations and rotations, and, if the visual display is turned on, draws them. Additionally, it handles local conflicts if, for example, two units want to move into the same cell simultaneously. Depending on the game mode, this is solved either by updating the units in a predefined order or in a non-deterministic way (in the non-deterministic game mode). Finally, microRTS forwards the updated environmental data to every player. This corresponds to sensory perception and is handled by the agent's middle layer.

6.4 First Evaluation

The experiments described in this chapter were performed in order to evaluate the achievement of the goals that were set in Section 6.1 and their results have been published in [79]. In particular, the focus of the initial experiments was on $G1$, which addressed the possibility to combine strategic (long-term) and tactical (short-term) behaviors in a large search space.

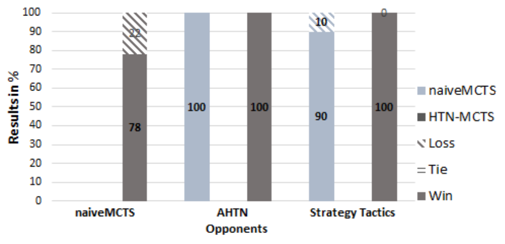
For strategic behaviors, we have used the planning domain described in Section 6.3.2. This domain included 4 primitive tasks *CollectResources, BuildAndDefend, AttackOpponent, PreventAttack* and the 4 corresponding Evaluation Functions *CollectEF, BuildEF, AttackEF* and *PreventEF*. Inspired by EFs used by benchmark agents that are provided with microRTS, the initial evaluation functions that were used in the experiments described in this section considered different variables. For example, all EFs except for the *CollectEF* considered the number of all units' HPs (friendly and opponent units separately) and the distance to their targets. *CollectEF* considered only the number of HPs of worker units and bases. Additionally, *CollectEF* and *BuildEF* considered the number of resources carried by workers and the number of resources available for use. The selection of features to be considered in each EF as well as their weights in the weighted sums were selected manually according to our knowledge of the game and tweaked in some prior experiments.

We assume that the different strategic behaviors executed by our agent should be especially visible when comparing them against an agent that follows a *single* strategy throughout the whole game. For that reason, we have compared our agent against a pure naïveMCTS agent [34], which the middle layer of our approach is based on. The naïveMCTS agent was following the same strategy throughout all stages of a game using one single Evaluation Function – the *SqrtEF* function [34]. This function is trying to maximize the relative strength of an agent compared to the total number of friendly and opponent units on the map. As shown in Equation 6.2, it sums the cost (resources spent) of its own units U weighted by the square root of their remaining HPs, then subtracting the same sum for the opponent's units O:

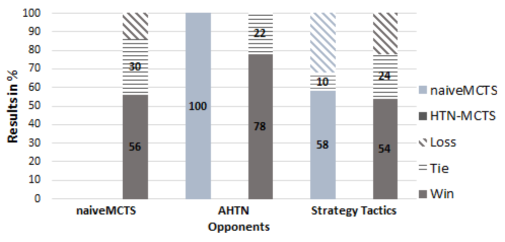
$$SqrtEF = (\sum_{u=1}^{U} cost_u * \sqrt{\frac{HP_{u-left}}{HP_{u-max}}}) - (\sum_{o=1}^{O} cost_o * \sqrt{\frac{HP_{o-left}}{HP_{o-max}}}) \tag{6.2}$$

All other MCTS parameters have been set to identical values for our hybrid agent and the pure naïveMCTS agent. These parameters were pre-set to the default values of the naïveMCTS agent distributed with microRTS. We follow the definitions of parameters as described in Section 2.1.2. All policies π_0, π_l, and π_g were ϵ-greedy policies with $\epsilon_0 = 0.4$, $\epsilon_l = 0.3$ and $\epsilon_g = 0$ respectively. The maximum simulation time of an action was set to 100 frames. The maximum tree depth was 10 and the default policy was *RandomBiasedAI*. This policy gives attack, collect, and return (a resource) actions a higher chance to be selected than movement actions.

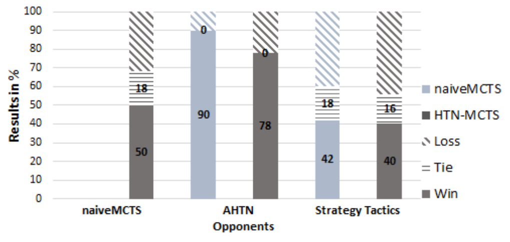
For a direct comparison, we have let these two agents play against each other and for a comparison of their performance in general, we have let both of them play against further opponents. Both agents were tested against a pure planning agent that uses an HTN planner. For that purpose, the Adversarial Hierarchical Task Network (AHTN) agent [152] was selected. To the best of our knowledge, this is the only available agent for microRTS that is using a planner. However, it is noteworthy that, in contrast to our agent, which is creating abstract high-level tasks, this agent is operating on low-level unit actions. For that reason, the generated plans are very short and very detailed. Additionally, due to the high dynamics of the environment, most plans cannot be carried over to the next frame, since a plan failure is very likely to happen. For that reason, AHTN generates a new plan in every frame. Therefore, when comparing against our agent, we did not expect from the AHTN agent to show any long-term strategies.



(a) FourBasesWorkers-8x8



(b) NoWhereToRun-9x8

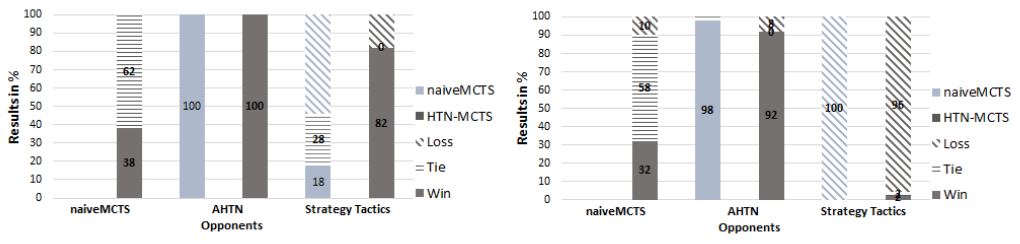


(c) basesWorkers-8x8A

Figure 6.3: Results of 50 matches played by our agent (HTN-MCTS) and naïveMCTS against 3 (2) opponents on small maps [79].

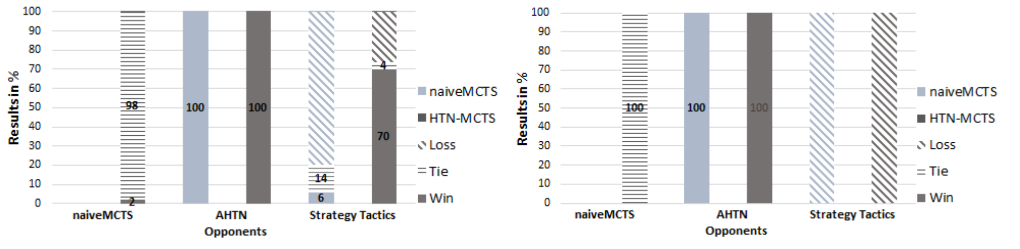
In addition to the two pure approaches mentioned above, we have compared our agent and naïveMCTS against another agent that combines strategic and tactical behaviors. The agent *StrategyTactics* [161] uses tree search for low-level tactical behaviors and predefined scripts (programs) for strategic behaviors. First used in the *Puppet Search* agent [160] in StarCraft such scripts were responsible for forwarding the game state by more than just one action and allow for look-ahead search over longer time spans. For *StrategyTactics*, the approach was extended by a convolutional neural network to select the scripts. This agent won the 2017 edition of the microRTS competition.

In order to test their performances and to observe their behaviors, our agent and the naïveMCTS agent were tested against the three (two in naïveMCTS' case) mentioned opponents on 7 different maps of microRTS, which are shown in Appendix D. These maps belonged to 3 different size categories: 3 *small* maps (8×8 cells), 2 *mid-size* maps (16×16 cells) and 2 *big* maps (24×24 cells). Following the competition rules [162], the maximum length of each game was limited according to the size category of the map to 3000 frames, 4000 frames, and 5000 frames respectively. An agent won a game if its opponent had no units left on the map. If there was no winner after the maximum number of frames, the game result was a tie. Each agent's computational time was limited to 100 milliseconds. Each agent combination played 50 games on each map. Out of the 50 games, each agent played 25 games on each player side.



(a) basesWorkers-16x16A (b) TwoBasesBarracks-16x16

Figure 6.4: Results of 50 matches played by our agent (HTN-MCTS) and naïveMCTS against 3 (2) opponents on mid-size maps [79].



(a) basesWorkers-24x24A (b) DoubleGame-24x24

Figure 6.5: Results of 50 matches played by our agent (HTN-MCTS) and naïveMCTS against 3 (2) opponents on bigger maps [79].

The results of the experiments are shown in Figures 6.3 – 6.5. In General, the experiments have shown that the weights have a large effect on the agent's behaviors and weights that lead to good performance on maps of a certain size or type are not good enough for other maps. For example, military units do not play an important role on small maps since the agents start the game so close to each other that the best strategy is to attack quickly with worker units only, instead of building an army. This, however, is a weak strategy for big maps where assaults performed by military units lead to a higher chance of success.

Additionally, we assume that the manually predefined weights were not balanced well enough between the different unit types. Furthermore, the general importance of HPs in comparison to distance minimization was not proportional to the map sizes. This way, on bigger maps, a higher importance was given to keeping friendly military units alive than to minimizing the distance to the opponent base and destroying the opponent units. For that reason, our agent's military units stayed close to the friendly base trying to defend themselves and the base rather than starting a determined attack on bigger maps. That way, most of the games of our agent against the naïveMCTS agent on the bigger maps resulted in a tie. On small maps, however, all distances were so short, that it was more optimal for our agent to reach the opponent and destroy its units.

The defensive behavior of our agent was especially effective against AHTN and StrategyTactics on the mid-size maps and the big map with only one base (*basesWorkers*$24 \times 24A$ in Figure 6.5). In contrast to the naïveMCTS agent, these agents are aggressive and start an attack quickly. Therefore, in most games against these opponents, our agent's *BuildAndDefend* task was interrupted by their attack and the *PreventAttack* task was executed instead. During this task, collecting resources and creating new units was not important. Instead, all units tried to defend their base by counter-attacking the incoming opponent units. The *PreventAttack* task was executed until no more attacking units were in close range to the friendly base.

As can be seen in the example video[2] of a game against StrategyTactics on the map *basesWorkers*$24 \times 24A$, after successfully preventing an attack, our agent returned to the *BuildAndDefend* task. The ability to quickly switch between different behaviors showed that the proposed architecture enabled the agent to stay reactive while still following a long-term strategy. Throughout one game, the agent could switch multiple times between the building and the attack-preventing task before finally starting its own attack. Although, in some cases our agent was not able to handle the persistent stream of incoming StrategyTactics units and was defeated.

In contrast to StrategyTactics, who always kept a few worker units at its base during its attack to ensure that further resources were collected and new units were created, AHTN focused *all* its units on attacking the opponent. Therefore, no new units were created and the stream of attackers ended at some point. This allowed our agent to first destroy all mobile AHTN units without having to cross the map and to attack the opponent base later on. That way, our agent won most of the games against AHTN. Compared to the performance of the naïveMCTS agent against these two agents, we can see that the strongly defensive behavior of our agent allowed for a higher win rate especially on mid-size and big maps.

However, there are two special maps, which are different from the other maps used in these experiments. These maps are the second mid-size map and the second big map, whose results are shown on the right side of Figure 6.4 (TwoBasesBarracks16×16) and Figure 6.5 (DoubleGame24×24) respectively. In both maps, each agent started with 2 bases instead of one. Additionally, in the mid-size map, each agent had initially 2 barracks pre-built. The big map was separated by a wall of resources so that one base of each agent was on each side of the wall and the player could, in theory, play two simultaneous games on each side.

[2]Experiment video: `https://youtu.be/Eox_ab836tM`

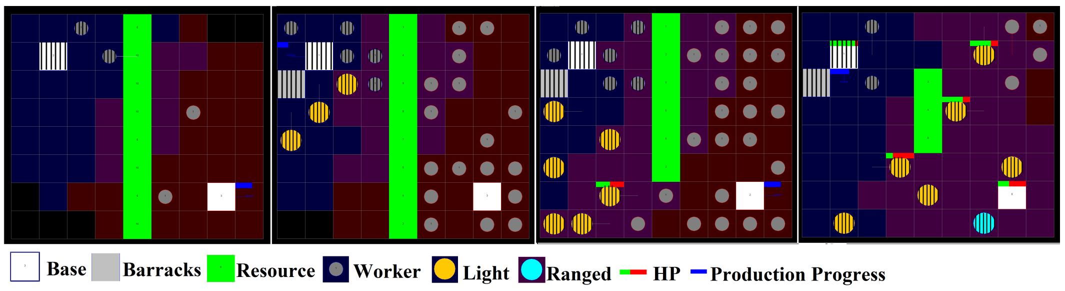


Figure 6.6: Progress of a match between our agent (striped units) and naiveMCTS [79].

Since StrategyTactics is able to build an army and simultaneously attack the opponent, it was very strong on the big map with 2 barracks. Additionally, because our agent stopped the building task when preventing an attack, it quickly lost the games against StrategyTactics on the mid-size map, which is reflected in the 4% win rate in Figure 6.4. On the big map, the fact that the agent was supposed to play different strategies on the different sides of the wall, was the reason for a weak performance of our agent. As already mentioned, StrategyTactics allowed distinct agents to follow separate strategies. However, our agent was designed in such a way that a single strategic task was selected for *all* units. For that reason, if the planner made a high-level decision based on the event on one side of the map, this decision affected the units on the other side as well. This, again, resulted in less optimal behaviors and led to a high failure rate. This is shown in the results against StrategyTactics on the *DoubleGame* map in Figure 6.5. We assume that changes in the planning domain as well as a possibility for the planner to separate units into groups will solve this problem.

A possible reason for a generally weak performance of naïveMCTS on bigger maps is the limited simulation depth, which did not allow the agent to look ahead far enough to find an optimal solution. Therefore, it was mostly acting locally and spreading its units very slowly. In contrast to that, our agent, was showing more deliberate behaviors depending on its current EF. Since our agent always tried to minimize a distance (either to resources or to the opponent), it was able to move more consciously towards its target than naïveMCTS, who was always trying to optimize its strength in relation to the opponent. These deliberate behaviors were very distinguishable in many cases, as can be seen in the video mentioned above. An additional example showing the progress of a strategic long-term plan is shown in Figure 6.6.

This scenario shows a game between our agent (on the left side of the map) and the naïveMCTS agent (on the right side) on the small map *NoWhereToRun* 9 × 8. (The blue, red, and violet background colors indicate the visibility of a cell to different players in a partially observable mode. However, these are not relevant for our experiments, which are performed with full observability.) In the first part of Figure 6.6, the beginning of the game is shown. Both players start with a base and a worker. They cannot reach each other because of the resource wall in the middle. Therefore, first, both players start gathering resources. In this case, our hybrid agent executes the task *CollectResources*. The corresponding EF is maximizing the number of resources and minimizing the workers' distances to resources (if not carrying a resource) or to the base (if carrying a resource). For that reason the workers stay within a narrow range to the base.

Since the naïveMCTS agent is always maximizing the number of its units, it immediately starts creating more workers and therefore spends all its resources. The creation of worker units requires the least amount of resources and therefore Equation 6.2 is maximized in this case. The second part of Figure 6.6 shows that naïveMCTS has created many workers, which spread all around its side of the map. In contrast, once our agent has collected enough resources, it switches to the *BuildAndDefend* task and builds a barrack. As we can see on the second picture, some workers still keep collecting resources staying close to the base. In the meantime, 3 *light* units were created by the barrack. These units' distances to the base are also minimized since they are supposed to defend the base from attacks. At this point in time, the wall is still closed and there is no way to the opponent's base. Therefore our agent is not able to start an attack and keeps creating more military units.

Once the wall is opened up in the third part of Figure 6.6, our agent does not immediately start an attack since the opponent is outnumbering and therefore the attack task's preconditions still do not hold. However, the opponent's units start spreading onto our agent's side of the map. At this point, the agent detects an attack and triggers a replanning. The task *PreventAttack* starts and the light units move towards attacking opponent workers that are closest to the base (in this case the opponent workers on the bottom of the map).

Afterwards, our agent keeps on building up its army and defending the base until, finally, the preconditions for an assault are met. The last part of Figure 6.6, shows that its units are now deliberately moving towards the opponent's base and destroying opponent units.

The results of the experiments described here and the observations made throughout multiple game rounds offer multiple insights. First, in regards to goal $G1$, the proposed hybrid architecture clearly allows for a combination of strategic and tactical behaviors interleaving high-level planning and low-level decision-making for a high number of units in a large search space. Based on the efficient sampling of the naïveMCTS agent, our agent is showing promising results even in the large search space of microRTS. Although, an improvement of the strategic behaviors is required for the future.

Regarding goal $G2$, the agent is able to make deliberate tactical decisions at run-time without being provided any explicit commands. With the use of MCTS on its middle layer, the agent is capable of finding combinations of unit actions that implicitly lead to tactical behaviors, which are very close to the expected ones.

Furthermore, the sequencing of the distinguishable tactical behaviors leads to a recognizable player strategy, which is provided by the planner. Therefore, the experiments have shown that it is possible to guide the search of MCTS through a planner, as defined by goal $G3.1$. By constantly monitoring the environment and the progress of the high-level plan, the agent is able to recognize when certain adjustments to the plan are required from the planner in order to change the reactive behavior of the agent.

In general, the results show that a good balance between the weights used in the tasks' Evaluation Functions can have a big impact on the progress and the outcome of the game. As we have seen on certain map types and on maps of bigger sizes, the EFs used in these experiments are not optimal and should be constructed in a more granular way. It is not enough to use one single set of EFs for all types and sizes of maps. Instead, there remains a requirement to better adapt them to some specific features of certain groups of maps. Furthermore, we assume that general adjustments to the weights can lead to better results. However, balancing the weights *manually* is a major difficulty of this hybrid approach. For that reason, we regard goal $G3.2$ as not accomplished and propose further improvements in the following sections.

6.5 Evolution of Evaluation Functions

The focus of the following sections is on goal $G3.2$ defined in Section 6.1, namely to optimize the execution of tactical behaviors through an *improved* guidance from the planner. Since the guidance is happening through the tasks' Evaluation Functions, we assume that adjusting the weights used in these functions will lead to optimized tactical behaviors.

In order to improve the weighting of different features used in the tasks' Evaluation Functions, we propose two steps. First, the balance between the weights of EFs needs be different for different types of maps. For example, in bigger maps, the weight of distance optimization should be higher in relation to other features than it is on small maps, where units can easily reach their targets. Second, we propose replacing the manual configuration and adjustment of weights by an optimization algorithm. In particular, we propose using a Genetic Algorithm (GA) for this purpose. Although, we assume that alternative Machine Learning algorithms can be used in a similar manner at this point. The ideas and results described in the following sections have been previously published in [158].

The idea behind using a GA for the optimization of weights is the following: each EF is using a predefined number of features and corresponding weights. This means that one EF can be represented as a chromosome, where each gene represents a certain weight.

As we have seen in the initial experiments, changing these weights, directly affects the tactical (low-level) behavior of the agent and depending on the quality of its tactical behaviors, the agent proceeds better or worse with its overall strategy (high-level behavior). Therefore, the weights do not only affect the local behaviors of units but also the global plan. For that reason, the quality of an EF, that leads to a certain tactical behavior, can be measured by the time required to finish the plan task that this EF is representing. This way, an EF with well-selected weights is expected to lead to a deliberate and fast achievement of the corresponding task, whereas poorly balanced weights will lead to undesired behaviors. For that reason, the *success* of a task execution and the *required time* can be used as a fitness measurement of the EFs or rather the chromosomes that represent their weights.

Furthermore, as we have noticed in previous experiments, some weights can be interrelated. This means that changing complete tuples of weights can entail certain behavioral patterns. Such changes of tuples can be represented by applying a crossover operator to two chromosomes and exchanging multiple genes (weight values) between them. For the reasons mentioned above, a Genetic Algorithm offers a good possibility for weight optimization in an Evaluation Function.

Previously, different Evolutionary Algorithms (EAs) have been successfully applied to different parts of MCTS. For example, [49] used an EA to evolve the default policy of MCTS instead of using a predefined one. Changing Evaluation Functions in order to guide the search of MCTS was also done with the help of EAs in previous works. For example, [51] fully evolved EFs through a Genetic Algorithm (GA), which did not only adjust the weights of its sub-functions but changed the structure of the EFs. This way, it also evolved the operators used in the functions.

In a way that is more similar to our approach, [50] evolved weights of EFs used by MCTS. The major difference to our work is, however, that evolution in [50] was done *online*, meaning that one generation of individuals was evaluated during one single iteration of MCTS. Both approaches ([51] and [50]) have shown that evolving EF can lead to great improvements of MCTS.

6.5.1 Application of the Genetic Algorithm

In the following section, we describe the elements of the proposed application of a Genetic Algorithm using the notion of [158].

Each EF that represents a plan task τ is a sum of weighted features (objectives) that should either be minimized (for example distance or the opponent's Health Points (HPs)) or maximized (for example resources or HPs of friendly units). For the Genetic Algorithm, each EF is represented as a *chromosome* of the length equal to the number of the EF's features. Each *gene* g_j of a chromosome represents the weight $w_j \in [0.0, 1.0]$ assigned to the corresponding feature. We refer to an instance of a chromosome as an *individual* i. The initial *population* of individuals is generated by assigning random values between 0 and 1 to each gene of each individual.

We apply two genetic operators to every individual. First, a *uniform crossover* between two neighboring individuals is applied by switching the values of each pair of genes with a 50% chance. This allows certain genes to be carried over into the next *generation*. Additionally, we apply a *mutation* operator, which changes the value of each gene to a new value between 0 and 1. For every gene, the mutation is applied with a certain probability. In the experiments described in the next section, we apply two different mutation probabilities of 50% and 20%.

As already mentioned, the goal of applying a GA to the tasks' EFs is optimizing their weights in such a way that the resulting EFs enable MCTS to achieve the corresponding task *as fast and efficient* as possible. For that reason, we use the *time required to successfully finish a task* as a measurement of an individual's fitness $F(i)$, where *time* corresponds to the *number of frames*. Since the time should be *minimized*, we are also minimizing the fitness values as opposed to the usual fitness maximization.

Each individual's fitness is evaluated after the agent plays a game using the current individual for the initialization of the weights in the corresponding task EF. Equation 6.3 shows the calculation of the individual's fitness $F(i)$ after one game. As already mentioned, each game in microRTS is limited to a maximum number of frames t_{max}. The time t_i, in which a task τ is executed represents the difference between the task's end and its beginning. A task can end either being achieved or by failing (i.e. the agent triggers replanning).

$$F(i) = \begin{cases} -(t_{max} - t_i) & \text{if task } \tau \text{ achieved} \\ (t_{max} - t_i) & \text{if task } \tau \text{ failed} \end{cases} \tag{6.3}$$

Since the fitness should be minimized, we are considering the *negative* difference between maximum time t_{max} and the required time t_i for successfully finished tasks. For the failed cases, we assume that if the agent needed *more* time before the task failed as a *better* option than if it failed after a very *short* time. The latter case would indicate that the agent's behavior was very wrong. However, *failing* a task is not desired in general. For that reason, we consider the positive difference between the maximum time and the task's time for the fitness calculation.

Since a game progress does not only depend on our agent's behavior but also on the opponent's actions, it is not enough to evaluate each individual in one single game. In order to better approximate the fitness, it requires multiple games with the agent playing on both sides of the map. Furthermore, in order to evolve EFs that generalize well against different kinds of opponents and different situations, each individual should be evaluated by playing multiple games against different opponents on multiple maps. For these reasons, the total fitness of an individual $F_T(i)$ is computed as shown in Equation 6.4.

$$F_T(i) = t_{MAX} - \sum_{match=1}^{M} F(i) \tag{6.4}$$

The total fitness considers the fitness values $F(i)$ gained in M games (or matches), which are computed according to Equation 6.3. The M games are played multiple times against multiple opponents on multiple maps. The exact numbers and types of opponents and maps used in this work, are described in Section 6.5.2. The final fitness is the difference between the total maximum time of all M games (t_{MAX}) and the sum of all games' fitness values. This way, the total value is minimized, while varying between the total maximum time and 0.

Once all individuals of one generation are evaluated, we use a $\mu + \lambda$ with elitism *selection strategy*. Here, μ is the number of the individuals in the parent population and λ the number of individuals in the offspring population. For this work, we use $\mu = \lambda$ and select the μ best individuals among parents and the offspring for the next generation. Furthermore, we always keep the μ best individuals found so far in a so called *archive*.

6.5.2 Second Evaluation

The experiments of this section describe the evolution of the EFs' weights and their effects on the agent's behaviors. These experiments and their results have been published in [158]. With the aim to improve the previous behaviors, we have used the same planning domain as used in the first experiments and described in Section 6.3.2. However, the results of the initial experiments have indicated that the balance of weights should vary according to the map size that the agent is playing on. For that reason, in the following experiments, the agent was still using the 4 previously described plan tasks *CollectResources, BuildAndDefend, AttackOpponent, PreventAttack* with the corresponding EFs *CollectEF, BuildEF, AttackEF* and *PreventEF*. However, during a game, the configuration of each EF was selected according to the current size of the map. Here, maps were divided into 3 sets: *small* maps (8×8 cells), *mid-size* maps (16×16 cells), and *big* maps (24×24 cells and bigger). This way, given the 3 map sizes and 4 tasks, in total, 12 configurations of EFs were evolved.

In addition to the 7 maps used in the first experiments, we have used an even larger map of the size of 32×32 cells, which is also shown in Appendix D. The major difficulty of this map is that the resources are at a long distance from both players' bases. Therefore, in order to reach them, the players are required to use some form of deliberate navigation and long-term planning. It can be problematic for simple search-based agents (including MCTS) to reach these resources due to a limited search depth. Similarly to the first experiments, the maximum number of frames was limited to 3000 frames and 4000 frames for small and mid-size maps respectively and 5000 frames for big maps with a 24 cell width and 6000 frames for the biggest map. As before, a frame was limited to 100 milliseconds.

As mentioned in Section 6.5.1, an individual's fitness is measured while the agent is playing a certain number of games against each of the opponents from the training set (described below) on each of the maps of the corresponding size. The fitness values of each game are then summed into the total fitness of an individual according to Equation 6.4. A fitness of an individual after *one* game depends on the time that was required to finish the task that is represented by the EF, whose values are currently evolved. Therefore, when the weights of, for example, the *AttackEF* are evolved, the evaluation starts only when the agent starts executing the task *AttackOpponent*. However, it can happen that the agent does not get a chance to start this task throughout a game because, for example, its opponent is very strong and the agent is required to only defend its base. In such cases, the game round cannot be evaluated and is *repeated* until the agent executes the attack task.

Whether the agent gets a chance to execute a task depends to a big extent on the agent's previous behavior. For example, if the agent did not collect enough resources or did not create enough military units, it cannot attack. For that reason, the probability of executing the *AttackOpponent* task depends on a successful and efficient execution of the tasks *CollectResources* and *BuildAndDefend*. This efficiency cannot be guaranteed using manually balanced EFs for these tasks. For that reason, we propose evolving the EFs in the order, in which the tasks will usually be scheduled providing a more optimal game flow to the agent. For each map size, the EFs are evolved in the following order: *CollectEF, BuildEF, PreventEF* and *AttackEF*. This way, we first try to optimize *CollectEF* and only after running the GA for a certain number of generations on this EF do we switch to evolving the next EF.

However, since some maps present special conditions, through which the planner will never schedule certain tasks, these maps were left out during the evolution of such tasks. For example, on very small maps, a good strategy is attacking the opponent with worker units only. For that reason, an agent switches from collecting resources directly to attacking the opponent, without ever starting the *BuildAndDefend* task. On some other maps, both players already start with barracks built and some resources available. Here, the agent will usually not go back to the task *CollectResources*.

The experiments of this section were performed in order to evaluate goal $G3.2$ from Section 6.1, which was a more efficient execution of tactical behaviors. For that reason, a direct comparison of the performance of the hybrid agent that uses the *evolved* EFs with the agent that uses the *manually balanced* EFs was required. Therefore, we have tested the agent with all configurations of the EFs against the previous 3 opponents (naïveMCTS [34], AHTN [152], and StrategyTactics [161]). Both, the naïveMCTS agent and our agent used the same parameters as described in Section 6.4, namely π_0, π_l, and π_g ϵ-greedy policies with with $\epsilon_0 = 0.4$, $\epsilon_l = 0.3$ and $\epsilon_g = 0$ respectively, a maximum tree depth of 10, a maximum simulation time of 100 frames, and *RandomBiasedAI* as the default policy.

Training set	Test set
Tiamat	Tiamat
SCVPlus	SCVPlus
NaiveMCTS	NaiveMCTS
	Capivara
	AHTN
	StrategyTactics

Table 6.1: Agents used in the training set and in the test set.

Additionally, we used further strong agents from the 2018 edition of the microRTS competition[3]: *Tiamat* [163, 164], *Capivara* [165], and *SCVPlus* [166]. Similarly to the idea behind StrategyTactics, Tiamat [163] – the winner of the competition – defines unit behaviors through scripts. These scripts are assigned per unit type, which allows groups of different unit types to follow different strategies. Later, [164] scripts with different parameters are used to represent different strategies. An Evolutionary Algorithm is then used to find optimal strategies. Also extending the idea behind [163], Capivara [165] combines unit-type specific scripts with naïveMCTS. While running naïveMCTS, the scripts restrict the actions for certain unit types according to the current situation enforcing certain behaviors. SCVPlus [166] is also based on scripting. Here, a voting system is used to generate new scripts from existing ones.

First, we have evolved the EFs against the strongest of the 6 opponents – Tiamat. However, after the first results, which are described below, we have divided the agents into a training set and a test set as shown in Table 6.1 aiming to train the agent to play equally well against different types of agents, strong and weak. The training set was used during the evolution and included the agents Tiamat, SCVPlus, and naïveMCTS, which are sorted from the strongest to the weakest agent. After the evolution process ran for a certain number of generations, our agent was tested in games against all 6 opponents from the test set.

Table 6.2 shows the parameters of the GA used in the three experiments. The first two runs were done using Tiamat only. The major difference between the first experiments is the changed mutation rate of 50% and 20%. Since playing a high number of games on multiple maps takes a long time given a population of 10 individuals, we have further varied the number of games played in each configuration (6 and 2 respectively) and the total number of generations (20 and 30).

[3]Results of the 2018 microRTS competition: `https://sites.google.com/site/micrortsaicompetition/competition-results/2018-cig-results`

Parameter	Tiamat only (T20)	Tiamat only (T30)	Full training set (F)
Number of opponents	1	1	3
Mutation probability	50%	20%	50%
Crossover probability	50%	50%	50%
Population size $= \mu = \lambda$	10	10	10
Number of generations	20	30	10
Number of games per opponent per map	6	2	6

Table 6.2: Parameters of the Genetic Algorithm used in three separate evolution processes, using Tiamat only in the first two processes and all 3 opponents from the training set in the last process.

After finishing the evolution of all EFs, the best 10 individuals of each EF were saved in the corresponding archive. Out of each archive, we have selected the best individual to be tested against the opponents in the test set. The agents using combinations of these individuals are shown as I_{T20} and I_{T30} (for the experiments over 20 and 30 generations respectively) in Table 6.3.

In order to *test* the evolved agents, every agent played 25 games against each opponent from the test set on each player side (50 in total), on each map. The resulting winning percentages shown in Table 6.3 were computed as the sum of the number of games that our agent won and half of the number of draws divided by the total number of games and multiplied by 100. Additionally, in order to evaluate the performance *gain* of the evolved agents, the *initial* agent that was using the manually balanced weights (described in Section 6.4) played the same number of games. This agent is represented as the *Init.* agent in Table 6.3.

The major disadvantage of training only against Tiamat was the strength of this opponent. As already mentioned, the probability of the *AttackOpponent* task depends on the progress of the game. With a very strong opponent, our agent sometimes was not able to start this task, which resulted in multiple repetitions of a game. Furthermore, even if the agent was able to *start* executing the task, especially on mid-size and big maps, it was not able to successfully achieve it. Instead, it lost the game. In these cases, when computing the individual's fitness, Equation 6.3 only tried to maximize the time that the agent required to *fail* the task. There was no regularization through positive examples and consequently the agent was rather forced to survive for as long as possible by avoiding the opponent instead of attacking it.

For that reason, in the third experiment, we have used the full training set to play against during the evolution. By introducing weaker agents, we ensured that our agent had better chances to *start* all tasks and to successfully *finish* them. As shown in Table 6.2, we have used the same parameters as in the first experiment. However, playing 6 games against 3 opponents on 2 (or 3, depending on the map size) maps with a population of 10 individuals resulted in at least 360 (or 540) games per population, with possibly many repetitions of some games. Considering that a maximum game length varied between 3000 and 6000 frames depending on the map size and a frame lasts 100 milliseconds, the evaluation of one population could take up to multiple days. For that reason, we have limited the total number of generations to 10.

After the evolution of all EFs over 10 generations each, the final archives contained 10 best individuals for each EF. Similarly to the first experiments, we have selected the best individuals of each archive and tested the agent with this combination against all 6 opponents from the test set. The results of this configuration are shown as I_{F1} in Table 6.3 with the best results against each opponent (max. of each column) marked green.

As we can see, in most cases, this combination has shown better results than the agents that used the EFs trained against Tiamat only (I_{T20} and I_{T30}). For that reason, we have further tested combinations of individuals from the last experiment. For that purpose, in each archive, we have identified groups of individuals that had the highest number of equal genes. These were *related* individuals, who were most likely derived from the same ancestors and consequently were carrying the same genetic information. Our goal was to test different kinds of behaviors. Assuming that groups of *similar weight values* (related individuals) will lead to *similar behaviors*, we have selected the best individual (the one with the lowest fitness values) from 2 groups that were *different* from the group that I_{F1} belonged to. This way, these individuals were unrelated or only weakly related with each other. The agents with the combinations of these individuals are shown as I_{F2} and I_{F3}.

Table 6.3 shows that in most cases the agent with the best individuals – I_{F1} – shows the highest performance gain in comparison to the initial agent with manually balanced weights. Interestingly, the agents I_{T20} and I_{T30}, which were trained against Tiamat, did not outperform I_{F1} *against Tiamat* on any map. This indicates that, in comparison to using a single opponent during *training*, using multiple opponents improves the agent's performance not only against *other* opponents but also against the one concrete opponent (in this case Tiamat).

FourBasesWorkers8x8	Tia-mat	SCV-plus	Naive-MC-TS	Capi-vara	A-HTN	ST
Init.	85	96	90	80	**100**	98
I_{T20}	91	86	78	70	**100**	84
I_{T30}	80	91	63	60	**100**	86
I_{F1}	96	**100**	**96**	**98**	**100**	**100**
I_{F2}	**97**	**100**	**96**	**98**	**100**	**100**
I_{F3}	86	94	82	82	**100**	98

basesWorkers8x8A	Tia-mat	SCV-plus	Naive-MC-TS	Capi-vara	A-HTN	ST
Init.	51	**100**	51	17	91	51
I_{T20}	38	94	57	16	85	72
I_{T30}	39	100	16	25	65	19
I_{F1}	**74**	100	**74**	**29**	85	77
I_{F2}	62	**100**	73	26	77	**79**
I_{F3}	46	99	57	10	**96**	53

NoWhereToRun9x8	Tia-mat	SCV-plus	Naive-MC-TS	Capi-vara	A-HTN	ST
Init.	0	98	78	14	98	58
I_{T20}	0	95	80	27	94	86
I_{T30}	0	**100**	99	38	**100**	94
I_{F1}	**10**	99	**100**	**46**	**100**	93
I_{F2}	6	99	99	32	**100**	**94**
I_{F3}	0	84	32	27	92	41

basesWorkers16x16A	Tia-mat	SCV-plus	Naive-MC-TS	Capi-vara	A-HTN	ST
Init.	0	97	64	0	85	37
I_{T20}	2	68	54	0	57	20
I_{T30}	**10**	98	82	0	64	58
I_{F1}	**10**	**100**	**96**	0	88	63
I_{F2}	2	**100**	93	0	**98**	58
I_{F3}	4	**94**	96	**2**	75	**84**

TwoBasesBarracks16x16	Tia-mat	SCV-plus	Naive-MC-TS	Capi-vara	A-HTN	ST
Init.	0	83	57	0	96	2
I_{T20}	0	59	50	0	69	2
I_{T30}	**2**	94	77	0	**100**	13
I_{F1}	**2**	96	73	**2**	**100**	**16**
I_{F2}	0	93	**85**	0	96	8
I_{F3}	0	**100**	**85**	0	98	12

basesWorkers24x24A	Tia-mat	SCV-plus	Naive-MC-TS	Capi-vara	A-HTN	ST
Init.	0	**81**	**50**	0	80	47
I_{T20}	0	52	**50**	0	100	**48**
I_{T30}	0	48	**50**	0	98	43
I_{F1}	0	72	**50**	0	**100**	32
I_{F2}	0	42	**50**	0	60	10
I_{F3}	0	54	**50**	0	94	44

DoubleGame24x24	Tia-mat	SCV-plus	Naive-MC-TS	Capi-vara	A-HTN	ST
Init.	43	69	**50**	6	**99**	0
I_{T20}	**50**	**72**	**50**	**8**	68	**8**
I_{T30}	**50**	70	**50**	0	40	**8**
I_{F1}	**50**	69	**50**	6	**99**	0
I_{F2}	**50**	70	**50**	6	96	0
I_{F3}	48	68	**50**	4	80	2

BWDistantResources32x32	Tia-mat	SCV-plus	Naive-MC-TS	Capi-vara	A-HTN	ST
Init.	0	0	50	0	10	3
I_{T20}	0	0	52	0	14	**18**
I_{T30}	0	0	**58**	**5**	**28**	6
I_{F1}	0	0	50	0	0	14
I_{F2}	0	0	52	0	16	10
I_{F3}	0	0	50	0	8	4

Table 6.3: Winning percentage of the row player against the column player. Computed as the sum of victories of the row-player against the column-player and half of the number of draws, divided by the number of matches and multiplied by 100 [158]. Best results of each column are marked green.

In some cases, the agents I_{F2} and I_{F3} were able to outperform the agents trained against Tiamat and to perform comparably to I_{F1}. Although, in most cases I_{F1} was stronger. This indicates that selecting the best individual for each EF separately also leads to a better performance of the agent in general. Consequently, improving the execution of distinct tactical behaviors also improves the execution of a complete strategy.

The differences between I_{F1}, I_{F2}, and I_{F3} were mainly dependent on the opponents. For example, we can see that on the midsize map $basesWorkers16 \times 16A$ I_{F3} performed best against StrategyTactics (ST) but was weaker than I_{F2} against AHTN. This indicates that the different (unrelated) individuals in the final archives can represent different player profiles that are differently suited to play against certain opponents. This means that combining different kinds of individuals of the distinct EFs can lead to more aggressive or more defensive agents. This opens an interesting outlook for future work, where evolving EFs can be used for procedural generation of game-playing agents.

In general, the results show that evolving EF weights through a Genetic Algorithm can improve the efficiency of a task execution done by MCTS. Although the agent does not outperform the strongest opponents, the evolutionary approach shows promising results. The agents using the evolved individuals achieve a winning percentage that is similar to or higher than the one of the initial agent with the manually defined weights, even against the strongest opponents. Consequently, the plan tasks can be improved in such a way that they provide a more efficient guidance for the execution of tactical behaviors as aimed for by the goal $G3.2$ in Section 6.1.

Furthermore, the high-level tasks act as a guidance for the Genetic Algorithm during the evolution process. An individual's fitness is computed as the time required to finish a task. Knowing when a plan task is finished is possible due to the use of the proposed postconditions. During the training phase they are the key indicators of a task's end. By constantly monitoring the game state and comparing it to a task's pre- and postconditions, the agent is able to track the time and the success of the task's execution and to forward this information to the Genetic Algorithm.

The major difficulty of using the proposed approach, however, are the very long training times. On the one hand, they result due to the way of how an individual's fitness is measured: by playing multiple games in real-time. On the other hand, the training time can be drastically extended by failed game trials. That is, whenever the task, whose EF is currently evolved does not start throughout a whole game match, this match has to be repeated. This can happen in multiple consecutive games and therefore delay the evaluation of an individual a lot.

A possible reason for a task not being started in a game is that its preconditions are never met and a reason for that can be a non-efficient execution of preceding tasks. For example, an attack cannot be started if the agent did not create an army. This is especially the case for tasks that are usually scheduled later within a strategic plan. Their success depends a lot on the performance of the preceding tasks. In order to minimize such situations, we have proposed to evolve the tasks in the order that they are usually scheduled in. However, even after evolving a task's EF over a certain number of generations, there is no guarantee that the agent will execute this task in an optimal way. Therefore, alternatively to evolving EFs in a certain order, a parallelized evolution or co-evolution of EFs can decrease the training time. Shorter training times will, in turn, allow for a higher number of generations to be evolved in the same time.

Additionally, with an increasing map size the size of the search space handled by MCTS also increases. Consequently, the agent can execute a larger variety of action sequences. That way, with even slightly less optimally balanced weight values it becomes less likely for the agent to achieve any task at all. For example, we could see that in many cases on the bigger maps the agent collected a resource and approached its base in a very targeted manner. However, it stopped a few cells in front of the base and did not drop the resource into the base. This happened because the balance between the *distance to the base* weight and the *resources in the base* weight were balanced in a very fine-tuned way.

As long as the agent was holding the resource, it was only minimizing the distance. If it dropped the resource, the resources value would increase (the objective is to increase this value). However, the agent's target would now become another resource, which was far away. Therefore, the distance value would suddenly increase (the objective is to *decrease* this value). This would affect the total EF value in a negative way a lot more than the benefit from the resource. For that reason, MCTS did not select this action. At this point, the weights of these two conflicting objectives required a better balancing. However, this could not be achieved with the number of generations used in our experiments. We assume that a longer evolution process will lead to better results. However, as already mentioned, with the ordered evolution of EFs, this will also increase the training time by days.

6.6 Conclusion

In this chapter, we have proposed a hybrid approach that allows for strategic long-term planning, tactical short-term decision-making, and reactive execution. The focus of this chapter were large-scale environments with a large number of agents that are controlled by the hybrid approach as well as a large number of other actors that operate in the same environment. For such environments, we have focused on scenarios where the common group goals are more important than an agent's own goals. Such scenarios are well represented in Real-Time Strategy (RTS) games, which provide highly dynamic large-scale adversarial environments. One such game environment was used in this chapter's experiments.

Similarly to the approach described in Chapter 5, the approach if this chapter follows the general idea of Chapter 4 and is built as a three-layer architecture. On the top-most layer, it uses a central Hierarchical Task Network with Postconditions (HTN_p) planner that operates on an abstract world model and is responsible for strategic planning. It generates a common plan for all units (agents) that it controls and forwards the plan to the middle layer.

In order to be able to make deliberate decisions for a large number of units, the middle layer requires a mechanism that can operate in large search spaces. At this point, manual generation of agent behaviors (as it was done in Chapter 5) becomes infeasible due to the size of the search space and should be avoided. For that reason, we have proposed using a mechanism that is able to perform a search in a large search space and to find solutions within a limited time without relying on manually created behaviors but using a simulation model of the environment.

As already described in Section 2.1.2, Monte Carlo Tree Search (MCTS) has proven to perform well under such circumstances. For that reason, the middle layer of the hybrid approach uses MCTS. In particular it uses naïveMCTS, which allows for a search of actions for multiple agents and can operate on durative actions. Using naïveMCTS, the middle layer decides on tactical behaviors for all units while following a strategic high-level task provided by the planner. At this point, the achievement of the common plan task by the combination of all units' actions is more important than any single unit's action.

For a translation between descriptive HTN_p tasks and the operational search performed by MCTS, it requires some connection between the two techniques. The main contribution of this chapter is the introduction of so-called Evaluation Functions (EFs) as part of primitive tasks of the HTN_p. Similarly to the approach described in Chapter 5 where each primitive task was represented by a distinct Behavior Tree, in this chapter, each primitive task is described by a distinct EF. Such an EF represents a weighted sum of objectives that are to be optimized in order to achieve the corresponding plan task. Such objectives can be, for example, minimization of distances and maximization of resources.

When an agent using this hybrid approach starts executing a certain plan task, the middle architecture layer forwards the task's EF to naïveMCTS. This, in turn, uses the provided EF to evaluate the world states resulting after its simulations and to make decisions based on the EF values (more details on MCTS are provided in Section 2.1.2). This way, we allow the planner to guide the search process of naïveMCTS towards actions that are supposed to lead to the achievement of a plan task. Similarly to the hybrid approach in Chapter 5, the middle layer uses post-conditions of a task in order to recognize a task's end. The lowest layer of the architecture, which is responsible for actual action execution, is implemented as part of the game environment.

With the goals to provide an architecture that allows for strategic and tactical decision-making in large search-spaces without relying on manual behaviors generation and with the possibility for the planner to guide the reactive approach on the middle layer, we have tested the proposed approach in some initial experiments in the microRTS game environment. Here, we have defined the EFs of all tasks to the best of our knowledge of the game environment. The hybrid agent was tested against multiple benchmark agents in multiple games. In addition to the agent's overall performance against other agents, we have compared its performance and its behaviors to an agent that used naïveMCTS with a single EF throughout all games. The initial experiments have shown that the proposed architecture allows for visibly distinct and different tactical behaviors when executing different strategic tasks even without explicitly pre-defining tactical behaviors for single units. In comparison to the pure naïveMCTS agent, we could observe that the hybrid agent was acting more goal-oriented in different stages of a game and therefore performed similarly well or better than the pure agent.

Nevertheless, the major disadvantage of the initial approach was the strong impact of the weights used in EFs on the agent's behaviors and the difficulty to tweak these weights manually. Therefore, another goal and an important contribution of this chapter was the proposal of an automatic way to improve EFs. We have proposed using a Genetic Algorithm (GA) to evolve the weights used in the weighted sums of EFs with the goal to find combinations of weights that allow for a faster and a more successful execution of plan tasks. In order to provide a more optimal game flow during the evolution process, we have proposed to evolve the EFs of tasks in the order that the tasks are usually scheduled in.

After evolving the EFs on different map sizes and testing them against multiple benchmark agents and directly comparing the behaviors resulting from the evolved Evaluation Functions to the ones resulting from the manually created Evaluation Functions, we have observed the following results. The major insight was that the evolution of EFs does, indeed, improve the behaviors of the agent. However, *consecutive* evolution of EFs is very time-consuming, which becomes very challenging with an increasing map size, a larger search space, and a lower chance for an individual to succeed in earlier tasks and to start later tasks. For that reason, we regard automatic improvement of EFs in general as an important addition to the proposed hybrid approach. However, we believe that it can be improved in terms of training time by either simultaneous evolution (or co-evolution) of all tasks' EFs or by the usage of a learning algorithm other than a GA.

As future research opportunities the weighted sums used in EFs can be replaced by other approaches for multi-objective optimization. For example, instead of using the sum, MCTS can compare the Hyper Volumes of alternative solutions for the evaluation of the resulting game states [167]. Another option is the evolution of the complete structure of an EF instead of simply adjusting the weights. Alternatively, MCTS can use a trained neural network, a so-called value network [168], in place of the EFs.

Chapter 7

Conclusion

This chapter summarizes the ideas and findings of this work. Afterwards it discusses possibilities and limitations of the proposed approaches in general and specifically in regards to the goals defined in Chapter 1. Finally, it concludes outlining ideas for future work.

7.1 Summary

This work deals with deliberate behaviors of artificial agents in highly dynamic environments. In such environments, small local changes can happen at a very high, possibly continuous, frequency. Additionally, severe global changes that affect long-term goals can happen every few seconds to minutes. The main motivation of this work is the requirement for a decision-making approach that allows for long-term (strategic) planning and reactive (tactical) execution while handling different types of environmental changes without delaying the action execution of an agent or leading to unnatural or not robust behaviors.

Major problems in highly dynamic environments are uncertainties and large search spaces, which become even more challenging with an increasing number of actors that operate in the environment and the number of agents, which possibly have to cooperate and to coordinate their actions. This work focuses on video games as highly dynamic multi-agent environments, which allow for high complexities comparable to those found in many real-world scenarios.

The proposal of a framework that allows for long-term planning and reactive execution is defined as the main goal in Chapter 1. With this goal, this thesis focuses on two major areas of decision-making, *planning* and *reactive decision-making approaches*. Basic concepts of both areas are described in Chapter 2, which gives detailed descriptions of Behavior Trees and Monte Carlo Tree Search as reactive approaches, and Hierarchical Task Networks as a possibility for multi-agent long-term planning. Additionally, this chapter covers the importance of *interleaved* planning and execution.

Looking at environments such as video games, robotics, and spacecraft control, Chapter 2 outlines major questions and solutions that exist in these areas. Many of these solutions propose hybrid and multi-layer approaches, most of which consist of three layers with the top-most layer being responsible for abstract high-level decision-making (possibly provided by human operators) and the middle layer dealing with low-level decisions. The lowest layer is usually responsible for action execution and sensory perception. Although most of the existing solutions are used in either less dynamic environments or in single-agent scenarios, they are the main inspiration for the approaches proposed in this work.

Focusing on interleaved planning and execution, Chapter 3 explores the use of an HTN planner in a highly dynamic adversarial video game environment. As a start, we observe a single-agent scenario before diving deeper into multi-agent settings in the following chapters. The environment represents a typical fighting game where two agents play against each other controlling one game character each. The major goal of this study is the analysis of the performance of an agent that uses a long-term planner but no additional reactive decision-making mechanism in comparison to purely reactive agents.

This chapter proposes a two-layer architecture where the top layer HTN planner directly communicates with the bottom-layer system that monitors the environment and the plan progress, recognizes plan failures, and executes single actions. In this case, the planner operates on a detailed representation of the world and generates detailed plans, which consist of distinct actions that can be directly executed by the agent. Consequently, there is no abstraction involved in this scenario.

In order to evaluate the advantages of a planner in comparison to reactive approaches, this chapter focuses on so-called *combos*. A *combo* is a predefined sequence of 4 actions, which have to be executed within a limited time frame without being interrupted by an opponent attack. Therefore, it requires long-term planning in order to generate a plan consisting of all 4 actions and a deliberate system to execute these actions consecutively. The performed experiments of Chapter 3 show that the use of a long-term planner greatly increases the number of such combos and improves the success of their execution when compared to multiple reactive agents. This indicates that it is, in general, possible to achieve long-term goals in such highly dynamic environments even with a two-layer architecture.

In terms of the general performance in comparison to its opponents, the agent shows good results. However, due to the high dynamics and the fact that the planner is responsible both for strategic and tactical planning, we observe very high replanning rates caused by minor environmental changes. This indicates a serious problem for multi-agent environments with even higher dynamics and even larger search spaces. Therefore, assuming that most minor changes do not invalidate the overall long-term plan and can be handled locally, and inspired by the existing three-layer solutions, Chapter 4 proposes to add an intermediate architecture layer responsible for reactive low-level decision-making.

When combining two different decision layers, there are certain aspects that are very important to consider. Chapter 4 summarizes these aspects and describes the general idea of a hybrid approach that is implemented in the following chapters. The idea consists of separating the decision-making responsibilities between a long-term planner and a reactive approach. In this case, the planner operates on an abstract world representation and generates abstract plans. During the execution of an abstract plan task, the reactive approach monitors the environment and refines the task under the consideration of the current, more detailed representation of the world state.

The major novelty of the proposed approach are *postconditions* that extend the common definition of HTN plan tasks (see Section 4.4). Postconditions are conditions that have to hold at the end of a task. They are used during the execution by the reactive approach in order to recognize a task's end. Using postconditions of the current task and preconditions of the following task, the reactive approach is able to validate each task and to proceed with the long-term plan. Furthermore, postconditions can be used to encode certain temporal conditions and provide a possibility for implicit coordination of multiple agents. In case of the recognition of a global plan failure, or if the reactive approach is not able to refine a task, the middle layer is responsible for requesting a new plan. That way, replanning is still possible but the replanning frequency is decreased through local failure management by the reactive approach.

Another important aspect of the hybrid approach is the link between declarative plan tasks, and an operational reactive approach. The following chapters propose different solutions to that challenge. Based on the general idea of Chapter 4, Chapters 5 and 6 discuss the implementation of two different hybrid approaches that are used in other highly dynamic game environments. First, in order to evaluate the effects of adding a reactive layer to the architecture, Chapter 5 implements two architectures, which operate on two planning domains of different detail levels and compares them in the same environment. For this, it observes multi-agent scenarios with a varying number of agents and further non-controllable actors existing in the environment. A two-layer architecture resembles the one from Chapter 3 with detailed planning and no additional reactive layer. A three-layer architecture operates on an abstract high-level Hierarchical Task Network with Postconditions combined with detailed Behavior Trees that refine plan tasks at execution time.

For the three-layer approach, each high-level task is represented by a single Behavior Tree, which refines the task in the best possible way. For that purpose, it monitors the environment and makes local decisions for a single agent considering information that was not available during plan-time. The beginning and the end of a high-level task are represented to the Behavior Tree through preconditions and the newly introduced post-conditions respectively. Due to features of Behavior Trees such as event-driven and looping behaviors and parallel branches, which are not available for planners, the agents are able to perform additional checks at execution time and to execute high level tasks in a more flexible and efficient way.

The advantages of the three-layer architecture can be seen in decreased global replanning frequencies, decreased overall execution times, and increased execution success when compared to the two-layer architecture. An additional advantage of Behavior Trees is the possibility to coordinate actions of multiple agents by using *parallel* branches (see Section 2.1.1) to *wait for* other agents while executing a task. Another important characteristic of Behavior Trees that Chapters 5 points out is the possibility to keep full control over an agent's behavior at any point of time using manually predefined Behavior Trees. Keeping full control over an agent can be desired in different environments and for different reasons. For example, in dangerous real-world situations, full control can be required for security and robustness reasons, whereas in video games reproducibility and predictability of an agent's behavior play a very important role. Considering the fact that the hybrid approach does not require any changes to common Behavior Tree systems, it grants great maintainability to experts of existing systems.

Having studied a multi-agent scenario with a relatively small number of homogeneous agents (max. 9 agents) in Chapter 5, Chapter 6 focuses on environments with even larger numbers of heterogeneous agents (max. 42 agents) and non-controllable actors and consequently even larger search spaces. Considering the large numbers of agents, the large search space, and very high environment dynamics, manually pre-defining the agent's behaviors for all possible situations is an unmanageable task. Therefore, the proposed approach grants more autonomy to the agents on the tactical decision level and only uses predefined strategic behaviors on the highest level. For that purpose it is using Monte Carlo Tree Search for reactive decision-making on the middle architecture layer. Similarly to the previous chapter, the top-layer planner operates on an abstract Hierarchical Task Network with Postconditions. The usage of a multi-agent version of MCTS allows for centralized decision-making on both upper layers of the proposed approach.

The middle-layer MCTS is responsible for task refinements at execution-time. However, in contrast to Behavior Trees, which explicitly execute predefined behaviors and therefore can be clearly assigned to corresponding tasks, MCTS combines tree search and Monte Carlo simulations (see Section 2.1.2). Therefore, the major challenge here is the connection between high-level plan tasks and the search process of MCTS. As a solution to this challenge, we propose changing the Evaluation Functions which are used by MCTS to evaluate each branch of the search tree, in such a way that MCTS selects actions that facilitate a quick accomplishment of the current plan task. By providing such an Evaluation Function with every task, the planner guides the search process of MCTS at execution time. As a result, the agents show emergent behaviours, which clearly differ from each other depending on the current task. These behaviors are very close to the expected ones, although the agents are not provided with any explicit low-level commands.

Similarly to the previous chapter, the progress of the long-term plan is possible due to the preconditions and postconditions of plan tasks. These indicate the beginning and the end of each task and are used by the middle layer at execution time to recognize global plan failures and to decide when to proceed to the next task in the plan. Furthermore, postconditions play an important role in the next step of Chapter 6.

In order to further optimize tactical decision-making of MCTS, we propose to automatically improve the mentioned Evaluation Functions. For that purpose, each Evaluation Function is evolved by a Genetic Algorithm. Here, the average execution time of a plan task is minimized in the fitness function of the Genetic Algorithm during the evolution. That way, the algorithm prefers Evaluation Functions that lead to *fast* achievements of a task and improves the overall execution times. In order to measure the execution time of a task, the algorithm computes the time difference between the start of a task, which is defined by its preconditions, and the time point when its postconditions are satisfied and the task is regarded as achieved.

As discussed in Chapter 6, the proposed approach shows that the evolution of Evaluation Functions improves the search of MCTS in general, allowing it to achieve plan tasks faster than with manually defined Evaluation Functions. However, the proposed process of *consecutive* evolution of Evaluation Functions is very time-consuming due to the inter-dependencies of different plan tasks and can be improved by simultaneously learning all tasks' Evaluation Functions.

7.2 Discussion

The major goal of this work was the proposal of an approach that allows for combined long-term planning and reactive execution in highly dynamic environments with a focus on multi-agent scenarios. In conclusion, we discuss the results of this work under the consideration of the sub-goals stated in Chapter 1 and propose possibilities for future work.

Separation of decision levels and interleaved decision-making An important goal of this work was the *separation* of the responsibilities between a planning approach and a reactive approach while allowing to *combine* (or interleave) decision-making of both approaches. The experiments of Chapter 3 have shown that although a pure planning approach can be used in highly dynamic single-agent environments, it needs to deal with both minor and major environmental changes. This can lead to high replanning frequencies and long planning times, which confirms the requirement for a hybrid approach.

Inspired by different three-layer architectures found in literature, we have proposed a hybrid approach that transfers the responsibility of local decision-making to a reactive approach and leaves abstract long-term reasoning and planning to a planner. A major contribution of this work that allows for an *interleaved* decision-making and acting of a planner and a reactive approach are *postconditions*, which extend a common planning domain. In addition to the usage of the preconditions by the planner at plan-time, both preconditions and postconditions are used by the reactive approach at execution-time. This allows to combine reactive behaviors with deliberate execution of long-term plans as well as for a recognition of local *and* global plan failures by the reactive approach.

What remains an open question here is how to find the optimal level of the decision hierarchy at which to separate the responsibilities between the two decision-making mechanisms. Future research can examine the effects of giving more control to one of the approaches.

Reduction of the planning complexity and the replanning rate In order to reduce the planning complexity, the planning time, and the replanning rate, a major step of this work was the abstraction of the planning domain. The effects of the abstraction have been shown in Chapter 5 in a direct comparison between an approach that uses a detailed planning domain and one that uses an abstract domain and hands over its refinement to a reactive approach. By reducing the detail level of the world representation and the action representation that the planner operates on, its search space is reduced, which decreases the planning time. At the same time, the lack of detailed information in a plan reduces the probability of a plan failure caused by a *minor* environmental change at execution time and therefore decreases the replanning frequency.

With regard to the previous open question, another question that can be explored in future research is how to find a good balance between decreasing the size of the search space through abstraction and not loosing too much important information required for long-term reasoning.

Improvement of the execution In order to ensure robust behaviors of agents, the execution of a plan cannot be delayed by long planning times. For that reason, an agent is required to always be able to quickly make local decisions and act in accordance to the current situation. This can be done by transferring some responsibility to the reactive approach. Due to the fact that the reactive approach is now responsible for only local decision-making, the size of its search space is also decreased and therefore its own computations can be done very quickly. In the case of Behavior Trees, this can be done instantaneously by simply checking the tree in the predefined order of branches, whereas the search process of MCTS can be aborted at any time delivering the best solution found so far. That way, even if the planner does require a long time to generate a plan, the reactive approach is able to find a local solution and the agent can continue with its execution.

Furthermore, the extension of a planner by a reactive approach also decreases the overall execution time of a whole plan. As we have shown in Chapter 5, when using a hybrid approach, agents can achieve their common plans faster and with a higher success probability than with a pure planning approach. This is possible due to the modular structures of Behavior Trees that allow for advanced behaviors such as loops and sequential and parallel branches with additional condition checks that allow an agent to quicker recognize failures and possibilities and execute a plan task in the most efficient way.

However, since Behavior Trees are usually hand-crafted, the actual improvement of the execution time depends to a high extent on the expert knowledge of the designer of the respective Behavior Tree. For that reason, an interesting outlook for future work is research for an automatic way to either improve hand-crafted Behavior Trees or to generate them from scratch aiming for an even more efficient execution of a plan. This corresponds to the goal of *automatic improvement of the guidance* described below.

Guidance of the reactive decision-making from the planner In order to allow an agent to keep its long-term goals in mind while acting in a reactive way, a very important sub-goal of this work was the proposal of a possibility to represent plan tasks in a way that can be understood by a reactive approach. For that purpose, we have proposed two different alternatives that depend on the underlying reactive approach. In the hybrid approach of Chapter 5, each plan task is represented by a single Behavior Tree that is explicitly designed in such a way that it achieves the corresponding plan task. By switching between Behavior Trees, an agent can proceed with its overall plan. This solution can be used with other reactive approaches that explicitly define behaviors and can be clearly assigned to a certain task.

In contrast to that, search mechanisms which do not rely on predefined rules but perform a search over a state-action space are more difficult to be guided by a planner. As a solution to this challenge, we have proposed to use Evaluation Functions to guide the search process of MCTS. By encoding a plan task as a mathematical function, the algorithm can prefer actions that lead to states with more optimal values of the Evaluation Function and allow for a faster achievement of the corresponding task. Since most search mechanisms use some kind of Evaluation Function, this solution can be used with other search-based approaches as well.

In this work, we have encoded the Evaluation Functions as weighted sums of multiple objectives that are to be optimized for a plan task. Future work can examine multi-objective optimization approaches other than the weighted sum approach and study their effects on the execution of long-term plans.

Automatic improvement of the guidance With the described possibilities to guide reactive decision-making according to a plan, the next goal was the improvement of this guidance aiming for a more efficient achievement of plan tasks. The proposed solution consisted of an evolution of the aforementioned Evaluation Functions for MCTS. The experiment results of Chapter 6 have shown that it is, in general, possible to improve the reactive behaviors for distinct high-level tasks. The provided access of the reactive approach to the tasks' pre- and postconditions did not only allow it to track the task's progress at execution time, but also allowed for the computation of a task's execution duration during the evolution.

Although the proposed evolution is very time consuming, the fact that a reactive behavior can be improved by considering the the end and the beginning of a high-level task opens great possibilities for future research. As proposed in Chapter 6, the training can be done in the form of co-evolution of multiple tasks' functions. Alternatively, a machine learning approach other than a Genetic Algorithm can lead to different results.

Another interesting possibility opens up for the improvement of reactive decision-making with Behavior Trees. As described in Section 2.1.1, there are some works on automatic generation and improvement of Behavior Trees. Future work can examine the applicability of the proposed Machine Learning approach to Behavior Trees given the tasks that they represent. This way, a Behavior Tree can be automatically generated (or improved) considering the preconditions under which the corresponding task can start and the postconditions that have to hold at its end.

Intuitive use and maintenance A side-goal for the concrete implementations of the proposed solution was intuitive use of the new approaches and the maintenance of the available expertise in existing solutions. For that purpose, we have used existing and well-known techniques with a minimum amount of adjustments made to them.

Behavior Trees are widely-spread in the area of video game development and gain more interest in the area of robotics. MCTS is a well-known technique in research areas such as computational intelligence in games. For both hybrid solutions, no changes to the underlying reactive approaches were required. Therefore, both reactive approaches can be intuitively used by many experts in the corresponding fields. Similarly, the only extension to a typical HTN planning domain introduced in this work are postconditions. Due to the descriptive nature of common planning languages, the design of postconditions is very intuitive and resembles the creation of preconditions and effects (see Section 4.4). Furthermore, since postconditions are not used during the planning phase but during the execution only, they do not require any changes to the planner itself.

The major novelty of the proposed solutions lays in the architecture around the existent techniques. An important question here is how to combine a long-term planner with a reactive approach. This work has proposed two solutions to this question, which can be adapted to further environments and techniques in accordance to the general idea described in Chapter 4.

7.3 Limitations and Future Work

In addition to the open questions mentioned in the previous section, there remain some major limitations of the proposed solutions, which open up more possibilities for future work. One such limitation in regards to the Hybrid Approach *II* described in Chapter 6 is the requirement for a Forward Model for MCTS. Currently, the proposed solution cannot be used in environments for which such a model does not exist. Future work can study the question whether approximating the Forward Model [169] can solve this problem and facilitate interleaved planning in any way.

Considering the planning itself, there are multiple options for an extension of this work. For example, in order to allow for time-dependent assumptions or reasoning with nested beliefs, the planner can be replaced or extended by temporal or epistemic planning respectively (see Sections 2.2.2 and 2.3.2). The effects of such enhancements are especially interesting in regards to coordinated behaviors.

Another interesting possibility for future work is the applicability of the proposed solution in partially observable environments. In such cases, high-level plans can contain tasks for the agent to actively gather certain information. The requirement for this information can be encoded in the postconditions of the corresponding tasks. Using these postconditions, the reactive approach will try to refine the task until either obtaining the required knowledge or failing the task.

This work has proposed a way to automatically improve reactive behaviors under the consideration of high-level tasks. This approach can be extended to a fully-automatic generation of such behaviors. However, a major limitation of the proposed solution is that, so far, it requires the high-level planning domain to be manually designed. As such, every solution is very specific and cannot be transferred to other environments. A possibility to automatically generate both the planning domain and the reactive behaviors can extend the proposed idea to a general-purpose solution and open up its use for further industries and research areas.

Bibliography

[1] J. Branke, *Evolutionary optimization in dynamic environments.* Springer Science & Business Media, 2012, vol. 3.

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

[3] X. Neufeld, S. Mostaghim, D. Sancho-Pradel, and S. Brand, "Building a planner: A survey of planning systems used in commercial video games," in *IEEE Transactions on Games*, vol. 11, no. 2. IEEE, 2019, pp. 91–108.

[4] D. Isla, "Managing complexity in the halo 2 AI system," in *Game Developers Conference*, 2005.

[5] C. Côté, "Reactivity and deliberation in decision-making systems," in *Game AI Pro: Collected Wisdom of Game AI Professionals.* CRC Press, 2013, pp. 137–147.

[6] S. Rabin, "#define GAME_AI," in *Game Developers Conference*, 2009, last accessed on August 4th, 2020. [Online]. Available: http://gdcvault.com/play/1366/(307)-define-GAME

[7] A. Champandard, "Behavior trees: Three ways of cultivating strong AI," in *Game Developers Conference*, 2010, last accessed on August 4th, 2020. [Online]. Available: https://www.gdcvault.com/play/1012416/Behavior-Trees-Three-Ways-of

[8] A. Champandard and P. Dunstan, "The behavior tree starter kit," in *Game AI Pro: Collected Wisdom of Game AI Professionals.* CRC Press, 2013, pp. 73–91.

[9] J. Gillberg, "AI behavior editing and debugging in 'tom clancy's the division'," in *Game Developers Conference*, 2016, last accessed on August 4th, 2020. [Online]. Available: https://www.gdcvault.com/play/1023382/AI-Behavior-Editing-and-Debugging

[10] M. Colledanchise and P. Ögren, "How behavior trees modularize hybrid control systems and generalize sequential behavior compositions, the subsumption architecture, and decision trees," in *IEEE Transactions on robotics*, vol. 33, no. 2. IEEE, 2016, pp. 372–389.

[11] P. Ogren, "Increasing modularity of UAV control systems using computer game behavior trees," in *AIAA Guidance, Navigation, and Control Conference*, 2012, p. 4458.

[12] C. I. Sprague, Ö. Özkahraman, A. Munafo, R. Marlow, A. Phillips, and P. Ögren, "Improving the modularity of auv control systems using behaviour trees," in *IEEE OES Autonomous Underwater Vehicle Workshop.* IEEE, 2018, pp. 1–6.

[13] M. Colledanchise and P. Ögren, *Behavior Trees in Robotics and Al: An Introduction.* CRC Press, 2018.

[14] D. Hu, Y. Gong, B. Hannaford, and E. J. Seibel, "Semi-autonomous simulated brain tumor ablation with ravenii surgical robot using behavior tree," in *IEEE International Conference on Robotics and Automation.* IEEE, 2015, pp. 3868–3875.

[15] C. Paxton, A. Hundt, F. Jonathan, K. Guerin, and G. D. Hager, "Costar: Instructing collaborative robots with behavior trees and vision," in *IEEE International Conference on Robotics and Automation.* IEEE, 2017, pp. 564–571.

[16] B. Hannaford, D. Hu, D. Zhang, and Y. Li, "Simulation results on selector adaptation in behavior trees," *arXiv preprint arXiv:1606.09219*, 2016.

[17] A. Marzinotto, M. Colledanchise, C. Smith, and P. Ögren, "Towards a unified behavior trees framework for robot control," in *IEEE International Conference on Robotics and Automation.* IEEE, 2014, pp. 5420–5427.

[18] C. I. Sprague and P. Ögren, "Adding neural network controllers to behavior trees without destroying performance guarantees," *arXiv preprint arXiv:1809.10283*, 2018.

[19] C.-U. Lim, R. Baumgarten, and S. Colton, "Evolving behaviour trees for the commercial game defcon," in *European Conference on the Applications of Evolutionary Computation.* Springer, 2010, pp. 100–110.

[20] D. Perez-Liebana and M. Nicolau, "Evolving behaviour tree structures using grammatical evolution," in *Handbook of Grammatical Evolution.* Springer, 2018, pp. 433–460.

[21] M. Nicolau, D. Perez-Liebana, M. O'Neill, and A. Brabazon, "Evolutionary behavior tree approaches for navigating platform games," in *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 9, no. 3. IEEE, 2016, pp. 227–238.

[22] M. Colledanchise, R. N. Parasuraman, and P. Ogren, "Learning of behavior trees for autonomous agents," *IEEE Transactions on Games*, vol. 11, no. 2, pp. 183–189, 2019.

[23] Q. Zhang, J. Yao, Q. Yin, and Y. Zha, "Learning behavior trees for autonomous agents with hybrid constraints evolution," in *Applied Sciences*, vol. 8, no. 7. Multidisciplinary Digital Publishing Institute, 2018, pp. 17–38.

[24] S. Jones, M. Studley, S. Hauert, and A. Winfield, "Evolving behaviour trees for swarm robotics," in *Distributed Autonomous Robotic Systems.* Springer, 2018, pp. 487–501.

[25] K. Y. Scheper, S. Tijmons, C. C. de Visser, and G. C. de Croon, "Behavior trees for evolutionary robotics," *Artificial life*, vol. 22, no. 1, pp. 23–48, 2016.

[26] M. Colledanchise, D. Almeida, and P. Ögren, "Towards blended reactive planning and acting using behavior trees," in *International Conference on Robotics and Automation*. IEEE, 2019, pp. 8839–8845.

[27] M. KARTAŠEV, "Integrating reinforcement learning into behavior trees by hierarchical composition."

[28] R. d. P. Pereira and P. M. Engel, "A framework for constrained and adaptive behavior-based agents," *arXiv preprint arXiv:1506.02312*, 2015.

[29] R. Dey and C. Child, "Ql-bt: Enhancing behaviour tree design and implementation with q-learning," in *IEEE Conference on Computational Intelligence in Games*. IEEE, 2013, pp. 275–282.

[30] L. Kocsis and C. Szepesvári, "Bandit based monte-carlo planning," in *European Conference on Machine Learning*. Springer, 2006, pp. 282–293.

[31] G. Chaslot, S. Bakkes, I. Szita, and P. Spronck, "Monte-carlo tree search: A new framework for game ai." in *Fourth AAAI Artificial Intelligence and Interactive Digital Entertainment Conference*, 2008, pp. 216–217.

[32] M. Chung, M. Buro, and J. Schaeffer, "Monte carlo planning in RTS games." in *IEEE Symposium on Computational Intelligence and Games*, 2005, pp. 117–124.

[33] R.-K. Balla and A. Fern, "UCT for tactical assault planning in real-time strategy games." in *International Joint Conference on Artificial Intelligence*, 2009, pp. 40–45.

[34] S. Ontañón, "The combinatorial multi-armed bandit problem and its application to real-time strategy games," in *AAAI Artificial Intelligence and Interactive Digital Entertainment Conference*, 2013, pp. 58–64.

[35] D. Soemers, "Tactical planning using MCTS in the game of starcraft," in *Bachelor's thesis*. Department of Knowledge Engineering, Maastricht University, 2014.

[36] D. Churchill and M. Buro, "Portfolio greedy search and simulation for large-scale combat in starcraft," in *IEEE Conference on Computational Intelligence in Games*. IEEE, 2013, pp. 1–8.

[37] N. Justesen, B. Tillman, J. Togelius, and S. Risi, "Script-and cluster-based UCT for starcraft." in *IEEE Conference on Computational Intelligence in Games*. IEEE, 2014.

[38] C. F. Sironi, J. Liu, D. Perez-Liebana, R. D. Gaina, I. Bravi, S. M. Lucas, and M. H. Winands, "Self-adaptive MCTS for general video game playing," in *International Conference on the Applications of Evolutionary Computation*. Springer, 2018, pp. 358–375.

[39] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez-Liebana, S. Samothrakis, and S. Colton, "A survey of monte carlo tree search methods," in *IEEE Transactions on Computational Intelligence and AI in games*, vol. 4, no. 1. IEEE, 2012, pp. 1–43.

[40] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," in *Machine learning*, vol. 47, no. 2-3. Springer, 2002, pp. 235–256.

[41] Y. Gai, B. Krishnamachari, and R. Jain, "Learning multiuser channel allocations in cognitive radio networks: A combinatorial multi-armed bandit formulation," in *IEEE Symposium on New Frontiers in Dynamic Spectrum.* IEEE, 2010, pp. 1–9.

[42] S. Ontañón, "Combinatorial multi-armed bandits for real-time strategy games," in *Journal of Artificial Intelligence Research*, vol. 58, 2017, pp. 665–702.

[43] A. Shleyfman, A. Komenda, and C. Domshlak, "On combinatorial actions and cmabs with linear side information," in *European Conference on Artificial Intelligence*, 2014, pp. 825–830.

[44] D. Churchill, A. Saffidine, and M. Buro, "Fast heuristic search for rts game combat scenarios," in *AAAI Artificial Intelligence and Interactive Digital Entertainment Conference*, 2012, pp. 112–117.

[45] S. Ontañón, "Informed monte carlo tree search for real-time strategy games," in *IEEE Conference on Computational Intelligence and Games.* IEEE, 2016, pp. 1–8.

[46] R. O. Moraes, J. R. Mariño, L. H. Lelis, and M. A. Nascimento, "Action abstractions for combinatorial multi-armed bandit tree search," in *AAAI Artificial Intelligence and Interactive Digital Entertainment Conference*, 2018, pp. 74–80.

[47] R. O. Moraes and L. H. Lelis, "Asymmetric action abstractions for multi-unit control in adversarial real-time games," in *AAAI Conference on Artificial Intelligence*, 2018, pp. 876–883.

[48] A. Uriarte and S. Ontañón, "Game-tree search over high-level game states in RTS games," in *AAAI Artificial Intelligence and Interactive Digital Entertainment Conference*, 2014, pp. 73–79.

[49] A. M. Alhejali and S. M. Lucas, "Using genetic programming to evolve heuristics for a monte carlo tree search ms pac-man agent," in *IEEE Conference on Computational Inteligence in Games.* IEEE, 2013, pp. 65–72.

[50] S. M. Lucas, S. Samothrakis, and D. Perez, "Fast evolutionary adaptation for monte carlo tree search," in *European Conference on the Applications of Evolutionary Computation.* Springer, 2014, pp. 349–360.

[51] A. Benbassat and M. Sipper, "EvoMCTS: Enhancing MCTS-based players through genetic programming," in *IEEE Conference on Computational Inteligence in Games*. IEEE, 2013, pp. 57–64.

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

[53] D. M. McDermott, "The 1998 ai planning systems competition," in *AI magazine*, vol. 21, no. 2. AAAI, 2000, pp. 35–55.

[54] M. Fox and D. Long, "Pddl2. 1: An extension to pddl for expressing temporal planning domains," in *Journal of artificial intelligence research*, vol. 20, 2003, pp. 61–124.

[55] A. Gerevini and D. Long, "Preferences and soft constraints in pddl3," in *International Conference on Automated Planning and Scheduling. Workshop on Planning With Preferences and Soft Constraints*, 2006, pp. 46–53.

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

[57] D. S. Nau, "Current trends in automated planning," in *AI magazine*, vol. 28, no. 4, 2007, pp. 43–58.

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

[59] R. Dearden, N. Meuleau, S. Ramakrishnan, D. E. Smith, and R. Washington, "Incremental contingency planning," in *International Conference on Automated Planning and Scheduling. Workshop on Planning Under Uncertainty and Incomplete Information*, 2003, pp. 38–47.

[60] M. Horstmann and S. Zilberstein, "Automated generation of understandable contingency plans," in *International Conference on Automated Planning and Scheduling. Workshop on Planning Under Uncertainty and Incomplete Information*, 2003, pp. 57–63.

[61] P. Bertoli, A. Cimatti, M. Roveri, and P. Traverso, "Planning in nondeterministic domains under partial observability via symbolic model checking," in *International Joint Conferences on Artificial Intelligence*, vol. 2001, 2001, pp. 473–478.

[62] D. Hyde, "Using bayesian networks to reason about uncertainty," in *AI Game Programming Wisdom*, vol. 4, 2008, pp. 429–442.

[63] M. Fox, A. Gerevini, D. Long, and I. Serina, "Plan stability: Replanning versus plan repair." in *International Conference on Automated Planning and Scheduling*, vol. 6, 2006, pp. 212–221.

[64] J. Orkin, "Three states and a plan: the ai of fear," in *Game Developers Conference*, 2006, pp. 1–18.

[65] ——, “Three states and a plan: the ai of fear,” 2006, last accessed on August 4th, 2020. [Online]. Available: https://www.gdcvault.com/play/1013459/Three-States-and-a-Plan

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

[67] ——, “Agent architecture considerations for real-time planning in games.” in *AAAI Artificial Intelligence and Interactive Digital Entertainment Conference*, 2005, pp. 105–110.

[68] P. Higley, “Goal-oriented action planning: Ten years old and no fear!” 2015, last accessed on August 4th, 2020. [Online]. Available: http://www.gdcvault.com/play/1022019/Goal-Oriented-Action-Planning-Ten

[69] M. Ghallab, D. Nau, and P. Traverso, *Automated planning: theory & practice.* Elsevier, 2004, ch. 13 Time for Planning.

[70] L. A. Castillo, J. Fernández-Olivares, O. Garcia-Perez, and F. Palao, “Efficiently handling temporal knowledge in an htn planner.” in *International Conference on Automated Planning and Scheduling*, 2006, pp. 63–72.

[71] L. Castillo, J. Fdez-Olivares, Ó. García-Pérez, and F. Palao, “Temporal enhancements of an htn planner,” in *Conference of the Spanish Association for Artificial Intelligence.* Springer, 2005, pp. 429–438.

[72] M. Li, H. Wang, C. Qi, and C. Zhou, “Handling temporal constraints with preferences in htn planning for emergency decision-making,” in *Journal of Intelligent & Fuzzy Systems*, vol. 30, no. 4. IOS Press, 2016, pp. 1881–1891.

[73] R. Straatman, T. Verweij, A. Champandard, R. Morcus, and H. Kleve, “Hierarchical AI for multiplayer bots in killzone 3,” in *Game AI Pro: Collected Wisdom of Game AI Professionals.* CRC Press, 2013, pp. 377–390.

[74] I. Georgievski and M. Aiello, “Htn planning: Overview, comparison, and beyond,” in *Artificial Intelligence*, vol. 222. Elsevier, 2015, pp. 124–156.

[75] D. Nau, Y. Cao, A. Lotem, and H. Muñoz-Avila, “SHOP: Simple hierarchical ordered planner,” in *International Joint Conference on Artificial Intelligence.* Morgan Kaufmann Publishers Inc., 1999, pp. 968–973.

[76] D. S. Nau, T.-C. Au, O. Ilghami, U. Kuter, J. W. Murdock, D. Wu, and F. Yaman, “SHOP2: An htn planning system,” in *Journal of artificial intelligence research*, vol. 20, 2003, pp. 379–404.

[77] D. Höller, G. Behnke, P. Bercher, and S. Biundo, “Plan and goal recognition as HTN planning,” in *International Conference on Tools with Artificial Intelligence.* IEEE, 2018, pp. 466–473.

[78] D. Höller, P. Bercher, G. Behnke, and S. Biundo, “HTN plan repair using unmodified planning systems,” in *International Conference on Automated Planning and Scheduling. Workshop on Hierarchical Planning*, 2018, pp. 26–30.

[79] X. Neufeld, S. Mostaghim, and D. Perez-Liebana, "A hybrid planning and execution approach through HTN and MCTS," in *International Conference on Automated Planning and Scheduling. Workshop on Integrating Planning, Acting, and Execution.*, 2019, pp. 37–45.

[80] D. S. Nau, S. J. Smith, K. Erol *et al.*, "Control strategies in htn planning: Theory versus practice," in *AAAI/IAAI*, 1998, pp. 1127–1133.

[81] A. Menif, E. Jacopin, and T. Cazenave, "SHPE: HTN planning for video games," in *Workshop on Computer Games, European Conference on Artificial Intelligence.* Springer, 2014, pp. 119–132.

[82] D. Nau, T.-C. Au, O. Ilghami, U. Kuter, D. Wu, F. Yaman, H. Muñoz-Avila, and J. W. Murdock, "Applications of SHOP and SHOP2," in *IEEE Intelligent Systems*, vol. 20, no. 2. IEEE, 2005, pp. 34–41.

[83] A. Champandard, T. Verweij, and R. Straatman, "Killzone 2 multiplayer bots," in *Game AI Conference*, 2009, last accessed on August 4th, 2020. [Online]. Available: https://www.guerrilla-games.com/read/killzone-2-multiplayer-bots

[84] T. Thompson, "HTN planning in transformers: Fall of cybertron," in *AI & Games*, last accessed on August 4th, 2020. [Online]. Available: https://www.youtube.com/watch?v=kXm467TFTcY

[85] A. Beij, "The AI of horizon zero down," in *Game AI North*, 2017, last accessed on August 4th, 2020. [Online]. Available: http://eej.dk/gain/2017/slides/THE_AI_OF_HORIZON_ZERO_DAWN.pdf

[86] N. Nejati, P. Langley, and T. Konik, "Learning hierarchical task networks by observation," in *International Conference on Machine learning.* ACM, 2006, pp. 665–672.

[87] C. Hogg, H. Muñoz-Avila, and U. Kuter, "Htn-maker: Learning htns with minimal additional knowledge engineering required." in *AAAI*, 2008, pp. 950–956.

[88] C. Hogg, U. Kuter, and H. Muñoz-Avila, "Learning methods to generate good plans: integrating htn learning and reinforcement learning," in *AAAI Conference on Artificial Intelligence.* AAAI, 2010, pp. 1530–1535.

[89] O. Ilghami and D. S. Nau, "Camel: Learning method preconditions for htn planning," in *International Conference on Artificial Intelligence Planning Systems*, 2002, pp. 131–141.

[90] H. H. Zhuo, D. H. Hu, C. Hogg, Q. Yang, and H. Muñoz-Avila, "Learning htn method preconditions and action models from partial observations," in *International jont conference on Artifical intelligence.* Morgan Kaufmann Publishers Inc., 2009, pp. 1804–1809.

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

[92] T. Könik and J. E. Laird, "Learning goal hierarchies from structured observations and expert annotations," in *Machine Learning*, vol. 64, no. 1. Springer, 2006, pp. 263–287.

[93] M. Fine-Morris and H. Muñoz-Avila, "Learning domain structure in hgns for non-deterministic planning," in *International Conference on Automated Planning and Scheduling. Workshop on Hierarchical Planning*, 2019, pp. 22–30.

[94] P. Bercher, R. Alford, and D. Höller, "A survey on hierarchical planning–one abstract idea, many concrete realizations," in *International Joint Conference on Artificial Intelligence*, 2019, pp. 6267–6275.

[95] D. Dvorak, A. Amador, and T. Starbird, "Comparison of goal-based operations and command sequencing," in *SpaceOps 2008 Conference*, 2008, p. 3335.

[96] V. Shivashankar, U. Kuter, D. S. Nau, and R. Alford, "A hierarchical goal-based formalism and algorithm for single-agent planning," in *International Conference on Autonomous Agents and Multiagent Systems*, 2012, pp. 981–988.

[97] R. Alford, V. Shivashankar, M. Roberts, J. Frank, and D. W. Aha, "Hierarchical planning: Relating task and goal decomposition with task sharing." in *International Joint Conference on Artificial Intelligence*, 2016, pp. 3022–3028.

[98] V. Shivashankar, R. Alford, U. Kuter, and D. Nau, "The godel planning system: a more perfect union of domain-independent and hierarchical planning," in *International Joint Conference on Artificial Intelligence*, 2013, pp. 2380–2386.

[99] M. Elkawkagy, P. Bercher, B. Schattenberg, and S. Biundo, "Improving hierarchical planning performance by the use of landmarks," in *AAAI Conference on Artificial Intelligence*, 2012, pp. 1763–1769.

[100] V. Verma, A. Jónsson, R. Simmons, T. Estlin, and R. Levinson, "Survey of command execution systems for nasa spacecraft and robots," in *International Conference on Automated Planning and Scheduling. Workshop on Plan Execution: A Reality Check*, 2005, pp. 92–99.

[101] M. Bozzano, A. Cimatti, A. Guiotto, A. Martelli, M. Roveri, A. Tchaltsev, and Y. Yushtein, "On-board autonomy via symbolic model-based reasoning," in *ESA Workshop on Advanced Space Technologies for Robotics and Automation*, 2008, pp. 1–8.

[102] M. Tipaldi and L. Glielmo, "A survey on model-based mission planning and execution for autonomous spacecraft," in *IEEE Systems Journal*, vol. 12, no. 4. IEEE, 2017, pp. 3893–3905.

[103] E. Gat, R. P. Bonnasso, R. Murphy *et al.*, "On three-layer architectures," in *Artificial intelligence and mobile robots.* AAAI Press, 1998, pp. 195–210.

[104] V. Verma, A. Jónsson, C. Pasareanu, and M. Iatauro, "Universal-executive and plexil: engine and language for robust spacecraft control and operations," in *AIAA Space*, 2006.

[105] V. Verma, T. Estlin, A. Jónsson, C. Pasareanu, R. Simmons, and K. Tso, "Plan execution interchange language (plexil) for executable plans and command sequences," in *International symposium on artificial intelligence, robotics and automation in space*, 2005.

[106] B. Pell, E. B. Gamble, E. Gat, R. Keesing, J. Kurien, W. Millar, C. Plaunt, and B. C. Williams, "A hybrid procedural/deductive executive for autonomous spacecraft," in *Autonomous Agents and Multi-Agent Systems*, vol. 2, no. 1. Springer, 1999, pp. 7–22.

[107] N. Muscettola, G. A. Dorais, C. Fry, R. Levinson, C. Plaunt, and D. Clancy, "IDEA: Planning at the core of autonomous reactive agents," in *International NASA Workshop on Planning and Scheduling for Space.* AAAI, 2002.

[108] R. Levinson, "Unified planning and execution for autonomous software repair," in *International Conference on Automated Planning and Scheduling. Workshop on Plan Execution: A Reality Check*, 2005, pp. 55–62.

[109] M. Ghallab, D. Nau, and P. Traverso, "The actor's view of automated planning and acting: A position paper," in *Artificial Intelligence*, vol. 208. Elsevier, 2014, pp. 1–17.

[110] D. S. Nau, M. Ghallab, and P. Traverso, "Blended planning and acting: Preliminary approach, research challenges." in *AAAI*, 2015, pp. 4047–4051.

[111] S. Patra, M. Ghallab, D. Nau, and P. Traverso, "Acting and planning using operational models," in *AAAI Conference on Artificial Intelligence*, 2019.

[112] ——, "Interleaving acting and planning using operational models," in *International Conference on Automated Planning and Scheduling. Workshop on Integrating Planning, Acting, and Execution.*, 2019, pp. 46–54.

[113] S. Patra, J. Mason, A. Kumar, M. Ghallab, P. Traverso, and D. Nau, "Integrating acting, planning, and learning in hierarchical operational models," in *International Conference on Automated Planning and Scheduling*, 2020, pp. 478–487.

[114] D. Hilburn, "Simulating behavior trees: A behavior tree/planner hybrid approach," in *Game AI Pro: Collected Wisdom of Game AI Professionals.* CRC Press, 2013, pp. 99–111.

[115] M. Cashmore, M. Fox, D. Long, D. Magazzeni, B. Ridder, A. Carrera, N. Palomeras, N. Hurtos, and M. Carreras, "Rosplan: Planning in the robot operating system," in *International Conference on Automated Planning and Scheduling*, 2015, pp. 333–341.

[116] "Robot operating system (ROS)," last accessed on August 4th, 2020. [Online]. Available: https://www.ros.org/

[117] J. C. González, F. Fernández, A. Garcıa-Olaya, and R. Fuentetaja, "On the application of classical planning to real social robotic tasks," in *International Conference on Automated Planning and Scheduling. Workshop on Planning and Robotics*, 2017, pp. 38–47.

[118] J. C. González Dorado, M. Veloso, F. Fernández, and A. García-Olaya, "Task monitoring and rescheduling for opportunity and failure management." in *Proceedings of the Twenty-Eighth International Conference on Automated Planning and Scheduling. Workshop on Integrated Planning, Acting and Execution. Association for the Advancement of Artificial Intelligence (AAAI)*, 2018, pp. 24–31.

[119] F. Rovida, B. Grossmann, and V. Krüger, "Extended behavior trees for quick definition of flexible robotic tasks," in *IEEE/RSJ International Conference on Intelligent Robots and Systems.* IEEE, 2017, pp. 6793–6800.

[120] I. Little, S. Thiebaux *et al.*, "Probabilistic planning vs. replanning," in *ICAPS Workshop on IPC: Past, Present and Future*, 2007.

[121] M. Brenner and B. Nebel, "Continual planning and acting in dynamic multiagent environments," in *Autonomous Agents and Multi-Agent Systems*, vol. 19, no. 3. Springer, 2009, pp. 297–331.

[122] C. J. Muise, V. Belle, P. Felli, S. A. McIlraith, T. Miller, A. R. Pearce, and L. Sonenberg, "Planning over multi-agent epistemic states: A classical planning approach." in *AAAI Conference on Artificial Intelligence*, 2015, pp. 3327–3334.

[123] T. Engesser, T. Bolander, R. Mattmüller, and B. Nebel, "Cooperative epistemic multi-agent planning with implicit coordination," in *International Conference on Automated Planning and Scheduling. Workshop on Distributed and Multi-Agent Planning*, 2015, pp. 68–76.

[124] T. Bolander, "A gentle introduction to epistemic planning: The del approach," in *9th Workshop on Methods for Modalities*, vol. 243, 2017, pp. 1–22.

[125] A. Alshehri, T. Miller, and L. Sonenberg, "Improving performance of multiagent cooperation using epistemic planning," in *arXiv preprint arXiv:1910.02607*, 2019.

[126] R. I. Brafman and C. Domshlak, "From one to many: Planning for loosely coupled multi-agent systems." in *International Conference on Automated Planning and Scheduling.*, 2008, pp. 28–35.

[127] B. J. Clement and E. H. Durfee, "Top-down search for coordinating the hierarchical plans of multiple agents," in *Proceedings of the third annual conference on Autonomous Agents.* ACM, 1999, pp. 252–259.

[128] "Abstract reasoning for planning and coordination," in *Clement, Bradley J and Durfee, Edmund H and Barrett, Anthony C*, vol. 28, 2007, pp. 453–515.

[129] B. J. Clement, A. C. Barrett, G. R. Rabideau, and E. H. Durfee, “Using abstraction in planning and scheduling,” in *Sixth European Conference on Planning*, 2014, pp. 35–42.

[130] U. Kuter, R. P. Goldman, and J. Hamell, “Assumption-based decentralized htn planning,” in *International Conference on Automated Planning and Scheduling. Workshop on Hierarchical Planning*, 2018, pp. 9–16.

[131] B. Browning, J. Bruce, M. Bowling, and M. Veloso, “Stp: Skills, tactics, and plays for multi-robot control in adversarial environments,” in *Proceedings of the Institution of Mechanical Engineers, Part I: Journal of Systems and Control Engineering*, vol. 219, no. 1. SAGE Publications, 2005, pp. 33–52.

[132] P. Cooksey and M. Veloso, “Intra-robot replanning to enable team plan conditions,” in *IEEE/RSJ International Conference on Intelligent Robots and Systems.* IEEE, 2017, pp. 1113–1118.

[133] R. D’Andrea, T. Kalmar-Nagy, P. Ganguly, and M. Babish, “The cornell robocup team,” in *Robot Soccer World Cup.* Springer, 2000, pp. 41–51.

[134] S. Behnke and R. Rojas, “A hierarchy of reactive behaviors handles complexity,” in *Workshop on Balancing Reactivity and Social Deliberation in Multi-Agent Systems.* Springer, 2000, pp. 125–136.

[135] R. P. Goldman, K. Z. Haigh, D. J. Musliner, and M. J. Pelican, “Macbeth: a multi-agent constraint-based planner [autonomous agent tactical planner],” in *Digital Avionics Systems Conference, 2002. Proceedings. The 21st*, vol. 2. IEEE, 2002, pp. 7E3–1.

[136] R. P. Goldman, D. Bryce, M. J. Pelican, D. J. Musliner, and K. Bae, “An architecture for hybrid planning and execution,” in *The Workshops of the Thirtieth AAAI Conference on Artificial Intelligence Planning for Hybrid Systems*, 2016, pp. 592–600.

[137] N. Wallace, “Hierarchical planning in dynamic worlds,” in *AI Game Programing Wisdom*, vol. 2, 2004, pp. 229–236.

[138] W. Van Der Sterren, “Squad tactics: Team ai and emergent maneuvers,” in *AI game programming wisdom*, 2002, pp. 233–246.

[139] J. Reynolds, “Tactical team ai using a command hierarchy,” in *AI Game Programming Wisdom*, vol. 1, 2002, pp. 260–271.

[140] W. Van Der Sterren, “Squad tactics: Planned maneuvers,” in *AI Game Programming Wisdom*, 2002, pp. 247–259.

[141] T. Champandard, Alex J.and Verweij and R. Straatman, “Killzone 2 multiplayer bots,” in *Paris Game AI Conference*, 2009, last accessed on August 4th, 2020. [Online]. Available: https://www.guerrilla-games.com/read/killzone-2-multiplayer-bots

[142] W. Van Der Sterren, “Multi-unit planning with htn and a*,” in *Paris Game AI Conference*, 2009, last accessed on August 4th, 2020. [Online]. Available: https://www.cgf-ai.com/docs/multi_unit_htn_with_astar.pdf

[143] W. van der Sterren, “Hierarchical plan-space planning for multi-unit combat maneuvers,” in *Game AI Pro: Collected Wisdom of Game AI Professionals.* CRC Press, 2013, pp. 169–183.

[144] H. Muñoz-Avila and H. Hoang, “Coordinating teams of bots with hierarchical task network planning,” in *AI Game Programing Wisdom*, vol. 3, 2006, pp. 417–427.

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

[146] M. Molineaux, M. Klenk, and D. Aha, “Goal-driven autonomy in a navy strategy simulation,” in *AAAI Conference on Artificial Intelligence*, 2010, pp. 1548–1554.

[147] H. Muñoz-Avila, D. W. Aha, U. Jaidee, M. Klenk, and M. Molineaux, “Applying goal driven autonomy to a team shooter game,” in *International Florida Artificial Intelligence Research Society Conference*, 2010, pp. 465–470.

[148] B. G. Weber, P. Mawhorter, M. Mateas, and A. Jhala, “Reactive planning idioms for multi-scale game AI,” in *IEEE Symposium on Computational Intelligence and Games.* IEEE, 2010, pp. 115–122.

[149] B. G. Weber, M. Mateas, and A. Jhala, “Building human-level AI for real-time strategy games.” in *AAAI Fall Symposium: Advances in Cognitive Systems*, vol. 11, 2011, pp. 329 – 336.

[150] R. Palma, P. A. González-Calero, M. A. Gómez-Martín, and P. P. Gómez-Martín, “Extending case-based planning with behavior trees,” in *International Florida Artificial Intelligence Research Society Conference*, 2011, pp. 407 – 412.

[151] G. Robertson and I. Watson, “Building behavior trees from observations in real-time strategy games,” in *International Symposium on Innovations in Intelligent SysTems and Applications.* IEEE, 2015, pp. 1–7.

[152] S. Ontañón and M. Buro, “Adversarial hierarchical-task network planning for complex real-time games,” in *International Conference on Artificial Intelligence.* AAAI Press, 2015, pp. 1652–1658.

[153] X. Neufeld, S. Mostaghim, and D. Perez-Liebana, “Htn fighter: Planning in a highly-dynamic game,” in *IEEE International Computer Science and Electronic Engineering Conference.* IEEE, 2017, pp. 189–194.

[154] F. Lu, K. Yamamoto, L. H. Nomura, S. Mizuno, Y. Lee, and R. Thawonmas, “Fighting game artificial intelligence competition platform,” in *Global Conference on Consumer Electronics.* IEEE, 2013, pp. 320–323.

[155] G. L. Zuin, Y. Macedo, L. Chaimowicz, and G. L. Pappa, “Discovering combos in fighting games with evolutionary algorithms,” in *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference.* ACM, 2016, pp. 277–284.

[156] K. Yamamoto, S. Mizuno, C. Y. Chu, and R. Thawonmas, “Deduction of fighting-game countermeasures using the k-nearest neighbor algorithm and a game simulator,” in *IEEE Conference on Computational Intelligence and Games.* IEEE, 2014, pp. 437–441.

[157] X. Neufeld, S. Mostaghim, and S. Brand, “A hybrid approach to planning and execution in dynamic environments through hierarchical task networks and behavior trees,” in *AAAI Artificial Intelligence and Interactive Digital Entertainment Conference.* AAAI, 2018.

[158] X. Neufeld, S. Mostaghim, and D. Perez-Liebana, “Evolving game state evaluation functions for a hybrid planning approach,” in *IEEE Conference on Games.* IEEE, 2019.

[159] S. Ontañón, N. A. Barriga, C. R. Silva, R. O. Moraes, and L. H. Lelis, “The first microRTS artificial intelligence competition.” in *AI Magazine*, vol. 39, no. 1, 2018.

[160] N. A. Barriga, M. Stanescu, and M. Buro, “Puppet search: Enhancing scripted behavior by look-ahead search with applications to real-time strategy games,” in *AAAI Artificial Intelligence and Interactive Digital Entertainment Conference*, 2015, pp. 9–15.

[161] ——, “Combining strategic learning with tactical search in real-time strategy games,” in *AAAI Artificial Intelligence and Interactive Digital Entertainment Conference.* AAAI, 2017, pp. 9–15.

[162] S. Ontañón, “Microrts AI competition,” 2017, last accessed on August 4th, 2020. [Online]. Available: https://sites.google.com/site/micrortsaicompetition/

[163] L. H. S. Lelis, “Stratified strategy selection for unit control in real-time strategy games,” in *International Joint Conference on Artificial Intelligence*, 2017, pp. 3735–3741.

[164] J. R. H. Mariño, R. O. Moraes, C. F. M. Toledo, and L. H. S. Lelis, “Evolving action abstractions for real-time planning in extensive-form games,” in *AAAI Conference on Artificial Intelligence*, vol. 33. AAAI, 2019, pp. 2330–2337.

[165] R. O. Moraes, J. R. H. Mariño, L. H. S. Lelis, and M. A. Nascimento, “Action abstractions for combinatorial multi-armed bandit tree search,” in *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment.* AAAI, 2018, pp. 74–80.

[166] C. R. Silva, R. O. Moraes, L. H. Lelis, and K. Gal, “Strategy generation for multi-unit real-time games via voting,” in *IEEE Transactions on Games*, vol. 11, no. 4. IEEE, 2018, pp. 426–435.

[167] D. Perez, S. Mostaghim, S. Samothrakis, and S. M. Lucas, “Multiobjective monte carlo tree search for real-time games,” in *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 7, no. 4. IEEE, 2014, pp. 347–360.

[168] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, “Mastering the game of go with deep neural networks and tree search,” in *nature*, vol. 529, no. 7587. Nature Publishing Group, 2016, p. 484.

[169] S. M. Lucas, A. Dockhorn, V. Volz, C. Bamford, R. D. Gaina, I. Bravi, D. Perez-Liebana, S. Mostaghim, and R. Kruse, “A local approach to forward model learning: Results on the game of life game,” in *IEEE Conference on Games*. IEEE, 2019, pp. 1–8.

List of Figures

List of Tables

Acronyms

HTN_p Hierarchical Task Network with Postconditions. 23, 75, 76, 83, 89, 104, 105, 107, 110–115, 140, 146, 147, 167

ABCD α-β Considering Durations. 22

AHTN Adversarial Hierarchical Task Network. 122, 125, 133, 138

BT Behavior Tree. 3, 12–18, 35, 36, 38, 44, 45, 79–81, 83–86, 89, 91, 93–99, 101–106, 140, 143, 146, 147, 149–151, 166

CMAB Combinatorial Multi-Armed Bandit. 21, 22, 117

EA Evolutionary Algorithm. 17, 18, 129, 134

EF Evaluation Function. 111, 113, 115–119, 121, 122, 126–142, 147, 148, 150, 167

FSM Finite State Machine. 13, 41, 43, 80, 84, 97

GA Genetic Algorithm. 129, 130, 133–135, 138, 141, 142, 147, 150, 168

GDP Goal Decomposition Planner. 32

GE Grammatical Evolution. 17

GOAP Goal Oriented Action Planner. 26, 42

HGN Hierarchical Goal Network. 32, 76

HPs Health Points. 48, 63, 67, 109, 115, 121, 122, 124, 130

HTN Hierarchical Task Network. 27, 28, 30–32, 38, 42–46, 52, 53, 57, 69, 71, 72, 75, 76, 83–85, 89, 91, 94, 95, 99, 103, 106, 111, 116, 119, 122, 143–145, 151, 166, 167, 171

LSI Linear Side Information. 22

MAB Multi-Armed Bandit. 20, 21

MCTS Monte Carlo Tree Search. 3, 12, 13, 18–20, 22, 23, 45, 46, 56, 107, 108, 110, 111, 115–119, 122, 128–130, 132, 138–143, 147–151

MDP Markovian Decision Process. 13, 19, 33, 34

MLPS Matching Learning with Polynomial Storage. 22

NPC Non-Player Character. 14

PDDL Planning Domain Definition Language. 24, 27, 37, 52

RAE Refinement Acting Engine. 36

RL Reinforcement Learning. 31

ROS Robot Operating System. 37, 80

RTS Real-Time Strategy. 8, 9, 17, 20–22, 44, 45, 109, 112, 114, 116, 140

SHOP Simple Hierarchical Ordered Planner. 30, 36

SHOP2 Simple Hierarchical Ordered Planner 2. 30, 42

STP Skills, Tactics, and Plays. 41

STRIPS Stanford Research Institute Problem Solver. 24–26

UAV Unpiloted Aerial Vehicle. 30, 42

UCB1 Upper Confidence Bound. 20, 56, 61, 72

UCT Upper Confidence Tree. 20, 22, 36

UCTCD Upper Confidence Tree Considering Durations. 23

Glossary

actor throughout this work, we use the word *actor* for artificial or natural entities acting in an environment. 4, 143

agent throughout this work, we use the word *agent* for artificial agents. 3, 143

FightingICE Fighting Game AI Competition: `http://www.ice.ci.ritsumei.ac.jp/~ftgaic/` . 48–50, 52–55, 57–60, 63, 64, 66

Forward Model a simulation model of the environment that allows to apply actions to a world state and roll the state forward. 107, 109, 151

microRTS microRTS: `https://sites.google.com/site/microRTSaicompetition`. 22, 23, 109, 110, 112, 113, 115–119, 121–123, 128, 131, 134, 141, 167

planning domain an HTN planning domain is defined as follows: $D = (F, C, A, M)$ with a set of facts F, a set of compound tasks C, a set of primitive tasks A, and a set of methods M. 3, 23–28, 30–32, 43, 52–55, 63, 66, 68, 69, 71, 72, 74, 76, 84–87, 89–92, 94, 96, 103, 104, 106, 112, 113, 115, 116, 121, 126, 132, 146, 148, 149, 151, 152, 171

planning problem an HTN planning problem is defined as follows: $p = (D, s_I, c_I)$, with $s_I \in 2^F$ being the initial state, $c_I \in C$ the compound task to be decomposed, and $D = (F, C, A, M)$ the planning domain. 24, 26–28, 32, 37, 75, 76

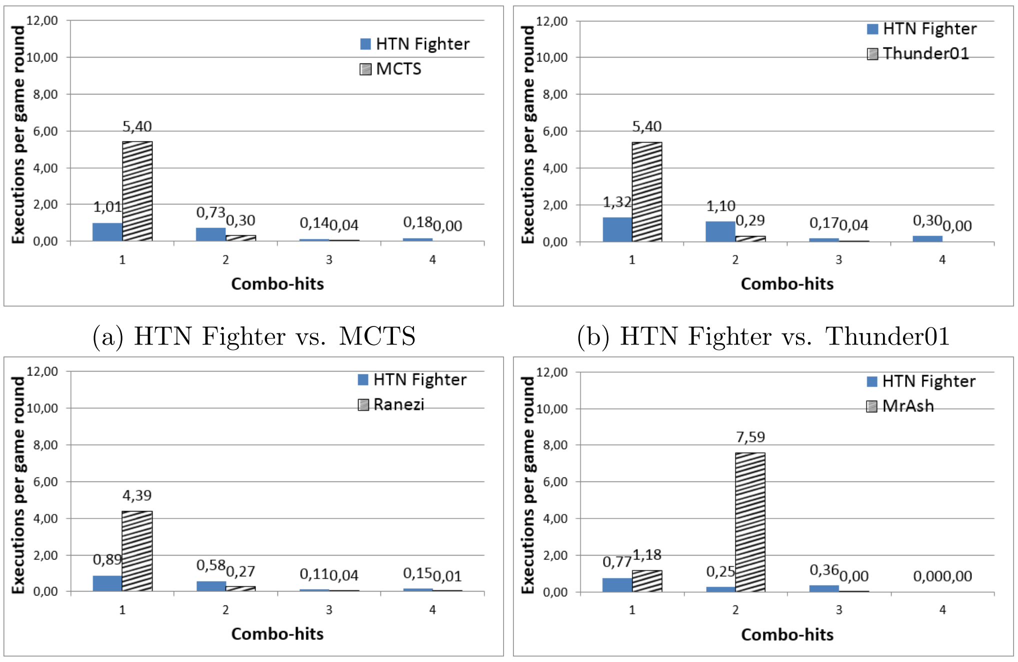
Player Action In microRTS: a combination of all players' units' actions. 118, 119, 121

StarCraft StarCraft AI Competition: `https://www.cs.mun.ca/~dchurchill/starcraftaicomp/`. 23, 116, 123

unit In microRTS: an entity controlled by a player. Following types of units are available: worker, light, ranged, heavy, base, barrack. 20, 23, 109

Appendix A

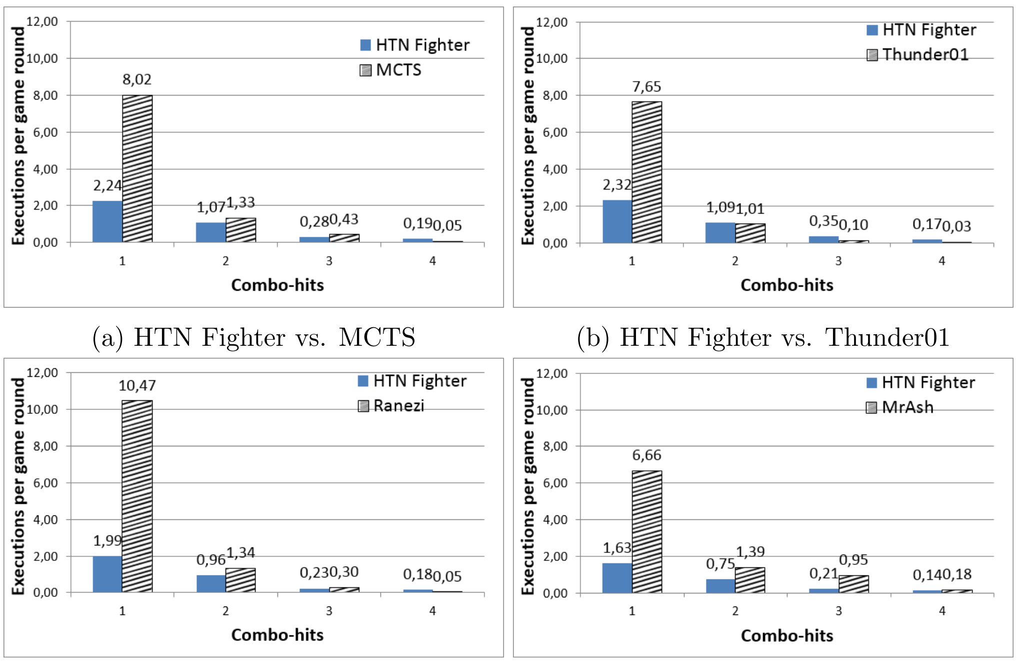
Combo Results of the FightingIce Experiments with UCB Method Selection



(a) HTN Fighter vs. MCTS (b) HTN Fighter vs. Thunder01

(c) HTN Fighter vs. Ranezi (d) HTN Fighter vs. MrAsh

Figure A.1: The average number of successfully performed chains of combo-hits of the length 1 – 4 for each agent pair playing with the character ZEN. HTN Fighter uses *UCB* method selection.



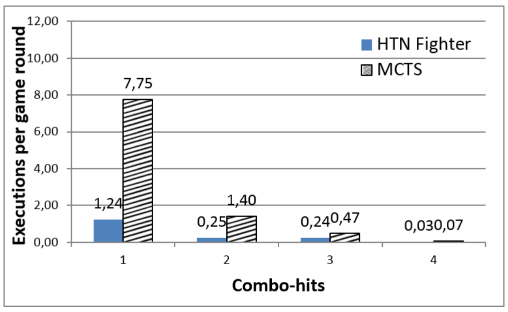
(a) HTN Fighter vs. MCTS

(b) HTN Fighter vs. Thunder01

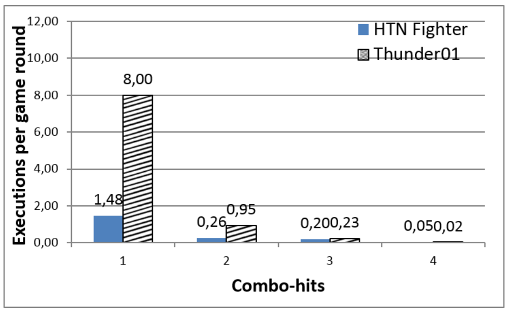
(c) HTN Fighter vs. Ranezi

(d) HTN Fighter vs. MrAsh

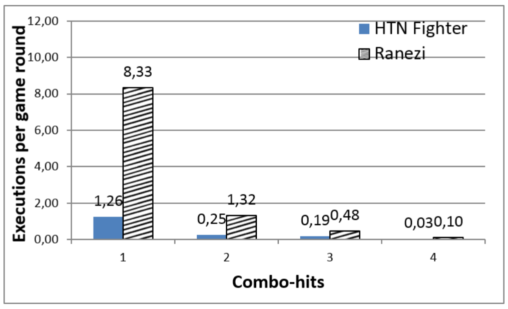
Figure A.2: The average number of successfully performed chains of combo-hits of the length 1 – 4 for each agent pair playing with the character LUD. HTN Fighter uses *ordered* method selection.



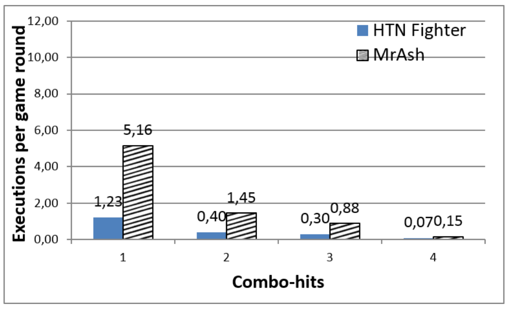
(a) HTN Fighter vs. MCTS



(b) HTN Fighter vs. Thunder01



(c) HTN Fighter vs. Ranezi



(d) HTN Fighter vs. MrAsh

Figure A.3: The average number of successfully performed chains of combo-hits of the length 1 – 4 for each agent pair playing with the character LUD. HTN Fighter uses *UCB* method selection.

Appendix B

HTN Domains for the Hybrid Approach *I*

B.1 Pure HTN Domain

```
domain pure_domain
{
   fact
   {
      agent(id32, vec3)
      zombie(id32, vec3)
      room(id32, vec3)
      ^connects(id32, vec3, vec3)
      crank_handle(id32, vec3)
      door(id32, vec3)
      crank_object(id32, vec3)
      coop_gate(id32, vec3)
      toggle_gate(id32, vec3)
      closed(id32)
      holding(id32, id32)
      isInRoom(id32, id32)
      runner(id32)
      gateKeeper(id32)
      zombieRoom(id32)
      cageRoom(id32)
      outsideRoom(id32)
      handleForObject(id32, id32)
      ^roleAssigned(id32, id32)
      ^connectsRooms(id32, id32, id32)
   }

   prim
   {
      GoTo!(id32, vec3)
      AlignTo!(id32, id32)
      InteractUse! (id32, id32)
      InteractHold!(id32, id32)
      StopAction!(id32, id32)
      Wait!(id32, int8)
      Crouch!(id32)
      StandUp!(id32)
      Run!(id32)
}

  effect
  {
      add(table)
      delete(table)
      change(table, table)
  }


macro
{
  object_far_from_point(x ,y) = dist(x, y)>=3
}

macro
{
  object_close_to_point(x ,y) = dist(x, y)<3
}

macro
{
  closerToFirstPos(p, x, y) = dist(p, x) <= dist(p, y)
}

macro
{
  notClosed(o) = isntInTable(o, 12)
}

macro
{
  isRoomInbetween(a, r, s, q) = room(r, m) & room (s, l) & room(q, n) &
      connects(o, i, j) & path(a, i, l) & path(a, j, n) & connects(p, x, z
      ) & path(a, x, n) & path(a, z, m)
}

macro
{
  handleForObject(h, o) = crank_handle(h, x) & crank_object(o, y) & dist(
      x, y) < 3
}

task root()
{
  each (zombie(g,y)) -> [AssignRoles(g)]
```

```
}
task Wait(a, t)
{
  case() -> [Wait!(a,t)]
}

task GoTo(a, p)
{
  case(agent(a,x) & holding(a,o)) -> [StopInteracting(a,o), GoTo(a, p)]
  case(agent(a, x)) -> [Run!(a), GoTo!(a, p), change(agent(a,x), agent(a,
      p))]
}

task Hold(a, o)
{
  case(agent(a, x) & toggle_gate (o, y) & isntInTable(a,17) &
      object_close_to_point(x, y)) -> [AlignTo!(a, o), InteractHold!(a, o)
      , delete(closed(o)), add(holding(a, o)), Wait(a, 10)]
  case(agent(a, x) & coop_gate (o, y) & isntInTable(a,17) &
      object_close_to_point(x, y)) -> [AlignTo!(a, o), InteractHold!(a, o)
      , delete(closed(o)), add(holding(a, o)), Wait(a, 10)]
  case(agent(a, x) & crank_handle(o,y) & crank_object(h, z) &
      object_close_to_point(z, y) & isntInTable(a,17) &
      object_close_to_point(x, y)) -> [AlignTo!(a, o), InteractHold!(a, o)
      , delete(closed(h)), add(holding(a, o)), Wait(a, 10)]
  case(agent(a, x) & toggle_gate (o, y) & holding(a, o)) -> [Wait(a, 10),
       delete(closed(o)), add(holding(a, o))]
  case(agent(a, x) & coop_gate (o, y) & holding(a, o)) -> [Wait(a, 10),
      delete(closed(o)), add(holding(a, o))]
  case(agent(a, x) & crank_handle(o,y) & crank_object(h, z) &
      object_close_to_point(z, y) & holding(a, o)) -> [Wait(a, 10), delete
      (closed(h)), add(holding(a, o))]
}

task Use(a, o)
{
  case(agent(a,x) & holding(a,h)) -> [StopInteracting(a,h), Use(a, o)]
  case(agent(a, x) & toggle_gate (o, y) & object_close_to_point(x, y)) ->
       [AlignTo!(a, o), InteractUse!(a, o)]
}

task StopInteracting(a, o)
{
  case(agent(a,x) & holding(a,o)) ->[StopAction!(a, o),delete(holding(a,o
      ))]
}

task GoToObject(a, o)
{
  case (agent(a,x) & connects(o,m,l) & path(a,x,m) & dist(x,m) < 1) -> []
  case (agent(a,x) & connects(o,m,l) & path(a,x,m) & closerToFirstPos(x,
      m, l)) -> [GoTo(a, m)]
  case (agent(a,x) & room(r,y) & isInRoom(a,r) & connects(o,m,l) & path(a
      ,x,m) & closerToFirstPos(y,m,l)) -> [GoTo(a,m)] }
task OpenDoorObject(a, o)
{
  case (door(o, y) & agent(a,x) & object_close_to_point(x, y) & closed(o)
      ) -> [AlignTo!(a, o), InteractUse!(a, o), delete(closed(o))]
  case (door(o, y) & agent(a,x) & object_close_to_point(x, y)) -> []
}

task CloseSingleAgentObject(a, o)
{
  case (door(o, y) & agent(a,x) & object_close_to_point(x, y) & notClosed
      (o)) -> [AlignTo!(a, o), InteractUse!(a, o), add(closed(o))]
  case (toggle_gate(o, y) & agent(a,x) & object_close_to_point(x, y) &
      notClosed(o)) -> [AlignTo!(a, o), InteractUse!(a, o), add(closed(o))
      ]
}

task TraverseThroughDoor(a, o)
{
  case (door(o, y) & agent(a,x) & connects(o, m, l) & closerToFirstPos(x,
       l, m) & notClosed(o)) -> [GoTo!(a, m), change(agent(a,x), agent(a,m
      ))]
}

task TraverseThroughToggleGate(a, o)
{
  case (toggle_gate(o, y) & agent(a,x) & notClosed(o) & connects(o, m, l)
       & closerToFirstPos(x, l, m)) -> [GoTo!(a, m), change(agent(a,x),
      agent(a,m))]
  case (toggle_gate(o, y) & agent(a,x) & notClosed(o) & connects(o, m, l)
       & closerToFirstPos(x, m, l)) -> [GoTo!(a, l), change(agent(a,x),
      agent(a,l))]
  case (toggle_gate(o, y) & agent(a,x) & closed(o) & connects(o, m, l) &
      closerToFirstPos(x, l, m)) -> [Hold(a, o) ,
      TraverseThroughToggleGate(a, o)]
  case (toggle_gate(o, y) & agent(a,x) & closed(o) & connects(o, m, l) &
      closerToFirstPos(x, m, l)) -> [Hold(a, o) ,
      TraverseThroughToggleGate(a, o)]
}

task PrepareAndGoThroughObjectSingleAgent(a, o)
{
  case (door(o, y)) -> [OpenDoorObject(a, o), Wait(a, 3),
      TraverseThroughDoor(a, o)]
  case (toggle_gate(o, y)) -> [TraverseThroughToggleGate(a, o)]
}

task GoThoughCoopObject (a, o)
{
  case (agent(a, x) & notClosed(o) & connects(o, m, l) & closerToFirstPos
      (x, m, l)) -> [GoTo!(a, l), change(agent(a,x), agent(a,l))]
```

```
}



task OpenCoopObject(a, o)
{
  case (crank_object(o, y) & crank_handle(h,z) & object_close_to_point(z, y)) -> [GoTo(a,z), Hold(a,h)]
  case ((coop_gate(o, y) | toggle_gate(o, y))) -> [Hold(a,o)]
}

task EnterRoom(a, s, g)
{
  case (isInRoom(a,r) & outsideRoom(r) & crank_object(o, p) & connectsRooms(o, s, r) & agent(b, y) & a~=b & roleAssigned(g,b) & isntInTable(b, 21) ) -> [GoToObject(a, o), OpenCoopObject(b, o), Wait(a,10), GoThoughCoopObject(a, o), change(isInRoom(a,r), isInRoom(a,s)), Wait(b, 10), StopInteractingAfterwards(b, o, a, s)]

  case (isInRoom(a,r) & door(o, p) & connectsRooms(o, r, s)) -> [GoToObject(a, o), PrepareAndGoThroughObjectSingleAgent(a, o), change(isInRoom(a,r), isInRoom(a,s))]

  case (isInRoom(a,r) & toggle_gate(o, p) & connectsRooms(o, r, s)) -> [GoToObject(a, o), PrepareAndGoThroughObjectSingleAgent(a, o), change(isInRoom(a,r), isInRoom(a,s))]
}

task StopInteractingAfterwards(b, o, a, c)
{
  case (agent(a, x) & runner(a) & isInRoom(a,r) & outsideRoom(r) & agent(b, y) & holding(b, o) & coop_gate(o, g) & dist(x,y)<2 & dist(x, g)>0.75) -> [StopAction!(b, o), delete(holding(b,o)), add(closed(o))]

  case (agent(a, x) & agent(b, y) & zombieRoom(c) & isInRoom(a,c) & crank_object(o,g) & holding(b, z) & crank_handle(z, p)) -> [StopAction!(b, z), delete(holding(b,z)), add(closed(o))]
}

task TryToEnterCage(a, g, r, c, t)
{
  case (agent(a,x) & isInRoom(a,r) & connects(t, m, l) & closerToFirstPos(x, m, l) & isInRoom(g,h)) -> [GoTo(a,m), PrepareAndGoThroughObjectSingleAgent(a, t), change(isInRoom(a,r), isInRoom(a,c)) , change(isInRoom(g,h), isInRoom(g,c))]
}

task GetGruntsAttention(a, g, q)
{
  case (isInRoom(g, q)) -> []
  case (zombie(g,y) & isInRoom(g, r) & agent(a,x) & isInRoom(a,h)) -> [GoTo(a, y), change(isInRoom(a,h), isInRoom(a,r))]
```

```
}



task TryToExitCage(a, r, o, t, g, q)
{
  case(isInRoom(a, q) & outsideRoom(j) & connects(o, y, z) & isInRoom(g,q
      )) -> [GoToObject(a, o), Crouch!(a), GoThoughCoopObject(a, o),
      change(isInRoom(a,q), isInRoom(a,j))]
  case(isInRoom(a, r) & outsideRoom(j) & notClosed(t) & isInRoom(g,l)) ->
       [GoToObject(a, o), Crouch!(a), GoThoughCoopObject(a, o), change(
      isInRoom(a,r), isInRoom(a,j)), change(isInRoom(g,l), isInRoom(g,q))]
  case(isInRoom(a, r)) -> [TryToEnterCage(a, g, r, q, t), TryToExitCage(a
      , r, o, t, g, q)]
}

task ReplaceRunner(a)
{
  case(empty(runner)) -> [add(runner(a))]
  case(agent(b,x) & runner(b)) -> [delete(runner(b)), add(runner(a))]
}

task ReplaceGateKeeper(a)
{
  case(empty(gateKeeper)) -> [add(gateKeeper(a))]
  case(agent(b,x)&gateKeeper(b))->[delete(gateKeeper(b)),add(gateKeeper(a
      ))]
}

task DoRunnerJob(a, r, q, o, t, g)
{
  //0 zombie in cage, runner outside
  case(agent(a,x) & zombie(g,y) & isInRoom(g,q) & outsideRoom(t) &
      isInRoom(a,t)) -> [FinishRunnerJob(a, g, q)]

  //1 both in cage
  case(agent(a,x) & isInRoom(a,q) & zombie(g,y) & isInRoom(g,q)) -> [
      TryToExitCage(a, r, o, t, g, q), FinishRunnerJob(a, g, q)]

  //2 runner in cage,zombie inzombieroom
  case(agent(a,x) & isInRoom(a,q) & zombie(g,y) & isInRoom(g,r)) -> [
      GetGruntsAttention(a, g, q), TryToExitCage(a, r, o, t, g, q),
      FinishRunnerJob(a, g, q)]

  //3 runner inzombieroom,zombie in cage
  case(agent(a,x) & isInRoom(a,r) & zombie(g,y) & isInRoom(g,q)) -> [
      GetGruntsAttention(a, g, q), TryToExitCage(a, r, o, t, g, q),
      FinishRunnerJob(a, g, q)]

  //4 runner,zombie inzombieroom
  case(agent(a,x) & isInRoom(a,r) & zombie(g,y) & isInRoom(g,r)) -> [
      GetGruntsAttention(a, g, q), TryToExitCage(a, r, o, t, g, q),
      FinishRunnerJob(a, g, q)]

  //5 runner outside
  case(agent(a,x) & zombie(g,y) & isInRoom(g,r)) -> [EnterRoom(a, r, g),
      DoRunnerJob(a, r, q, o, t, g)]  }
task FinishRunnerJob(a, g, q)
{
  case(agent(a, x) & isInRoom(g, q) & outsideRoom(t) & isInRoom(a, t)) ->
       [Run!(a)]
}

task CloseCageDoor(r,o, g, c)
{
  case(closed(o)) ->[]

  // zombie in cage, helper in room, runner outside
  case(agent(a,x) & roleAssigned(g,a) & runner(b) & a˜=b & isInRoom(b,t)
      & outsideRoom(t) & isInRoom(a,r) & isInRoom(g,c)) -> [RunToCage(a,o,
      g), CloseCage(a,o, g, c), EnterRoom(a,t, g)]

  // zombie in cage, helper and runner outside
  case(agent(a,x) & roleAssigned(g,a) & runner(b) & a˜=b & isInRoom(b,t)
      & outsideRoom(t) & isInRoom(a,t) & isInRoom(g,c)) -> [EnterRoom(a,r,
       g), CloseCageDoor(r,o, g, c)]
}

task RunToCage(a,o,g)
{
  case(notClosed(o)) ->[GoToObject(a, o)]
}

task CloseCage(a,o, g, c)
{
  case(isInRoom(g,c) & notClosed(o)) ->[AlignTo!(a, o), InteractUse!(a, o
      ), add(closed(o))]
}

task SolveCoop(g, r, q, t, o, a, b)
{
  //0 both in cage
  case(isInRoom(b, s) & outsideRoom(s) & isInRoom(g,q) & isInRoom(a,q))
      -> [ReplaceRunner(a), ReplaceGateKeeper(b), GoToObject(b, o),
      OpenCoopObject(b, o), DoRunnerJob(a, r, q, o,t, g),
      StopInteractingAfterwards(b, o, a, r), CloseCageDoor(r, t, g, q),
      delete(runner(a)), delete(gateKeeper(b))] //& isInRoom(a,r)

  //1 zombie in the cage
  case(isInRoom(b, s) & outsideRoom(s) & isInRoom(g,q) & isInRoom(a,r))
      -> [ReplaceRunner(a), ReplaceGateKeeper(b), GoToObject(b, o),
      OpenCoopObject(b, o), DoRunnerJob(a, r, q, o,t, g),
      StopInteractingAfterwards(b, o, a, r), CloseCageDoor(r, t, g, q),
      delete(runner(a)), delete(gateKeeper(b))] //& isInRoom(a,r)

  //2 zombie in cage and runner outside
  case(isInRoom(b, s) & outsideRoom(s) & isInRoom(g,q)) -> [ReplaceRunner
      (a), ReplaceGateKeeper(b), CloseCageDoor(r,t, g, q), delete(runner(a
      )), delete(gateKeeper(b))]

  //3
  case(isInRoom(b, s) & outsideRoom(s) & isInRoom(g,r) & isInRoom(a,q))
      -> [ReplaceRunner(a), ReplaceGateKeeper(b), GoToObject(b, o),
      OpenCoopObject(b, o), DoRunnerJob(a, r, q, o, t, g),
      StopInteractingAfterwards(b, o, a, r), CloseCageDoor(r,t, g, q),
      delete(runner(a)), delete(gateKeeper(b))] //& isInRoom(a,r)

  //4
  case(isInRoom(b, s) & outsideRoom(s) & isInRoom(g,r) & isInRoom(a, r))
      -> [ReplaceRunner(a), ReplaceGateKeeper(b), GoToObject(b, o),
      OpenCoopObject(b, o), DoRunnerJob(a, r, q, o,t, g),
      StopInteractingAfterwards(b, o, a, r), CloseCageDoor(r,t, g, q),
      delete(runner(a)), delete(gateKeeper(b))] //& isInRoom(a,r)

  //5
  case(isInRoom(b, s) & outsideRoom(s) & isInRoom(g,r)) -> [ReplaceRunner
      (a), ReplaceGateKeeper(b), GoToObject(b, o), OpenCoopObject(b, o),
      DoRunnerJob(a, r, q, o, t, g), StopInteractingAfterwards(b, o, a, r)
      , CloseCageDoor(r,t, g, q), delete(runner(a)), delete(gateKeeper(b))
      ] //& isInRoom(a,r)
}

task AssignRoles(g)
{
  // finished - free
  case (agent(a,y) & roleAssigned(g,a) & zombie(g,x) & cageRoom(r) &
      isInRoom(g,r) & isInRoom(a,s) & r~=s & toggle_gate(t,m) &
      connectsRooms(t,r, p) & closed(t)) -> []

  // assigned - continue
  case (roleAssigned(g,a) & roleAssigned(g,b) & a~=b & zombie(g,x) &
      zombieRoom(r) & isInRoom(g,r) & toggle_gate(t,m) & agent(a,y) &
      connectsRooms(t,r,c) & coop_gate(o, p) & connectsRooms(o, c, d)) ->
      [SolveCoop(g, r, c, t, o, a, b)]
  case (roleAssigned(g,a) & roleAssigned(g,b) & a~=b & zombie(g,x) &
      cageRoom(c) & isInRoom(g,c) & toggle_gate(t,m) & agent(a,y) &
      connectsRooms(t,r,c) & coop_gate(o, p) & connectsRooms(o, c, d)) ->
      [SolveCoop(g, r, c, t, o, a, b)]

  // not enough agents - wait
  case() -> []
```

```

    // assigning done in code!
  }
}
```

B.2 Hybrid HTN Domain

```
 domain hybrid_domain
{
   fact
   {
      agent(id32, vec3)
      zombie(id32, vec3)
      room(id32, vec3)
      ^connects(id32, vec3, vec3)
      crank_handle(id32, vec3)
      door(id32, vec3)
      crank_object(id32, vec3)
      coop_gate(id32, vec3)
      toggle_gate(id32, vec3)
      closed(id32)
      holding(id32, id32)
      isInRoom(id32, id32)
      runner(id32)
      gateKeeper(id32)
      zombieRoom(id32)
      cageRoom(id32)
      outsideRoom(id32)
      handleForObject(id32, id32)
      ^roleAssigned(id32, id32)
      ^connectsRooms(id32, id32, id32)
   }

   prim
   {
      OpenCoopObject!(id32, id32)
      GoThoughCoopObject!(id32, id32)
      GetGruntsAttention!(id32, id32)
      TryToEnterCage!(id32, id32)
      TryToExitCage!(id32, id32)
      TraverseThroughDoor!(id32, id32)
      CloseCage!(id32, id32, id32)
   }

  effect
  {
    add(table)
    delete(table)
    change(table, table)
  }


  macro
  {
    object_far_from_point(x ,y)  = dist(x, y)>=3
  }
```

```


macro
{
 object_close_to_point(x ,y) = dist(x, y)<3
}

macro
{
 closerToFirstPos(p, x, y) = dist(p, x) <= dist(p, y)
}

macro
{
 notClosed(o) = isntInTable(o, 12)
}

macro
{
 isRoomInbetween(a, r, s, q) = room(r, m) & room (s, l) & room(q, n) &
     connects(o, i, j) & path(a, i, l) & path(a, j, n) & connects(p, x, z
     ) & path(a, x, n) & path(a, z, m)
}

 task root()
 {
  each (zombie(g,y)) -> [AssignRoles(g)]
  }

  task OpenCoopObject(a, o)
  {
     case (crank_object(o,y)&crank_handle(h,z) & object_close_to_point(z,
         y)) -> [OpenCoopObject!(a,h) ,delete(closed(o)),add(holding(a,h))
         ]
     case ((coop_gate(o, y) | toggle_gate(o, y))) -> [OpenCoopObject!(a,
         o), delete(closed(o)), add(holding(a, o))]
  }

task EnterRoom(a, s, g)
{
  case (isInRoom(a,r) & outsideRoom(r) & crank_object(o, p) &
      connectsRooms(o, s, r) & agent(b,y) & a~=b & roleAssigned(g,b) &
      isntInTable(b, 21)) -> [OpenCoopObject(b, o), GoThoughCoopObject!(a,
       o), change(isInRoom(a,r), isInRoom(a,s))]

  case (isInRoom(a,r) & door(o, p) & connectsRooms(o, r, s)) -> [
      TraverseThroughDoor!(a, o), change(isInRoom(a,r), isInRoom(a,s))]
}

task TryToEnterCage(a, g, r, c, t)
{
  case (isInRoom(a,r) & isInRoom(g,h) & agent(a,x) &zombie(g,y) & room(c,
     z)) -> [TryToEnterCage!(a, t), delete(closed(t)), change(agent(a,x),
      agent(a,z)), change(zombie(g,y),zombie(a,z)), change(isInRoom(a,r),
      isInRoom(a,c)) , change(isInRoom(g,h), isInRoom(g,c))]  }
task GetGruntsAttention(a, g, q)
{
  case (isInRoom(g, q)) -> []
  case (zombie(g,y) & isInRoom(g, r) & agent(a,x) & isInRoom(a,h)) -> [
     GetGruntsAttention!(a, g), change(isInRoom(a,h), isInRoom(a,r))]
}

task TryToExitCage(a, r, o, t, g, q)
{
  case(isInRoom(a, q) & outsideRoom(j) & connects(o, y, z) & isInRoom(g,q
     )) -> [TryToExitCage!(a, o), change(isInRoom(a,q), isInRoom(a,j))]
  case(agent(a,x) &zombie (g,y) & dist(x,y)< 3 & outsideRoom(j) &
     notClosed(t) & isInRoom(g,l)) -> [TryToExitCage!(a, o), change(
     isInRoom(a,r), isInRoom(a,j)), change(isInRoom(g,l), isInRoom(g,q))]
  case(isInRoom(a, r)) -> [TryToEnterCage(a, g, r, q, t), TryToExitCage(a
     , r, o, t, g, q)]
}

task ReplaceRunner(a)
{
  case(empty(runner)) -> [add(runner(a))]
  case(agent(b,x) & runner(b)) -> [delete(runner(b)), add(runner(a))]
}

task ReplaceGateKeeper(a)
{
  case(empty(gateKeeper)) -> [add(gateKeeper(a))]
  case(agent(b,x) & gateKeeper(b)) -> [delete(gateKeeper(b)), add(
     gateKeeper(a))]
}

task DoRunnerJob(a, r, q, o, t, g)
{
  //0 zombie in cage, runner outside
  case(agent(a,x) &zombie(g,y) & isInRoom(g,q) & outsideRoom(t) &
     isInRoom(a,t)) -> []

  //1 both in cage
  case(agent(a,x) & isInRoom(a,q) &zombie(g,y) & isInRoom(g,q)) -> [
     TryToExitCage(a, r, o, t, g, q)]

  //2 runner in cage,zombie inzombieroom
  case(agent(a,x) & isInRoom(a,q) &zombie(g,y) & isInRoom(g,r)) -> [
     GetGruntsAttention(a, g, q), TryToExitCage(a, r, o, t, g, q)]

  //3 runner in zombie room,zombie in cage
  case(agent(a,x) & isInRoom(a,r) &zombie(g,y) & isInRoom(g,q)) -> [
     GetGruntsAttention(a, g, q), TryToExitCage(a, r, o, t, g, q)]

  //4 runner,zombie in zombie room
  case(agent(a,x) & isInRoom(a,r) &zombie(g,y) & isInRoom(g,r)) -> [
      GetGruntsAttention(a, g, q), TryToExitCage(a, r, o, t, g, q)]

  //5 runner outside
  case(agent(a,x) &zombie(g,y) & isInRoom(g,r)) -> [EnterRoom(a, r, g),
      DoRunnerJob(a, r, q, o, t, g)]
}

task CloseCage(a,o,g, r, c)
{
  case(isInRoom(a,r) & isInRoom(g,c)) -> [CloseCage!(a,o, g)]
}

task CloseCageDoor(r,o, g, c)
{
  case(closed(o)) ->[]

  // zombie in cage, helper in room, runner outside
  case(agent(a,x) & roleAssigned(g,a) & runner(b) & a˜=b & isInRoom(b,t)
      & outsideRoom(t) & isInRoom(a,r) & isInRoom(g,c)) -> [CloseCage(a,o,
       g, r, c), EnterRoom(a,t, g)]

  // zombie in cage, helper and runner outside
  case(agent(a,x) & roleAssigned(g,a) & runner(b) & a˜=b & isInRoom(b,t)
      & outsideRoom(t) & isInRoom(a,t) & isInRoom(g,c)) -> [EnterRoom(a,r,
       g), CloseCageDoor(r,o, g, c)]
}

task SolveCoop(g, r, q, t, o, a, b)
{
  //0 both in cage
  case(isInRoom(b, s) & outsideRoom(s) & isInRoom(g,q) & isInRoom(a,q))
      -> [ReplaceRunner(a), ReplaceGateKeeper(b), OpenCoopObject(b, o),
      DoRunnerJob(a, r, q, o,t, g), CloseCageDoor(r, t, g, q), delete(
      runner(a)), delete(gateKeeper(b))]

  //1 zombie in the cage
  case(isInRoom(b, s) & outsideRoom(s) & isInRoom(g,q) & isInRoom(a,r))
      -> [ReplaceRunner(a), ReplaceGateKeeper(b), OpenCoopObject(b, o),
      DoRunnerJob(a, r, q, o,t, g), CloseCageDoor(r, t, g, q), delete(
      runner(a)), delete(gateKeeper(b))]

  //2 zombie in cage and runner outside
  case(isInRoom(b, s) & outsideRoom(s) & isInRoom(g,q)) -> [ReplaceRunner
      (a), ReplaceGateKeeper(b), CloseCageDoor(r,t, g, q), delete(runner(a
      )), delete(gateKeeper(b))]

  //3
  case(isInRoom(b, s) & outsideRoom(s) & isInRoom(g,r) & isInRoom(a,q))
      -> [ReplaceRunner(a), ReplaceGateKeeper(b), OpenCoopObject(b, o),
      DoRunnerJob(a, r, q, o, t, g), CloseCageDoor(r,t, g, q), delete(
      runner(a)), delete(gateKeeper(b))]

  //4
  case(isInRoom(b, s) & outsideRoom(s) & isInRoom(g,r) & isInRoom(a, r))
      -> [ReplaceRunner(a), ReplaceGateKeeper(b), OpenCoopObject(b, o),
      DoRunnerJob(a, r, q, o,t, g), CloseCageDoor(r,t, g, q), delete(
      runner(a)), delete(gateKeeper(b))]

  //5
  case(isInRoom(b, s) & outsideRoom(s) & isInRoom(g,r)) -> [ReplaceRunner
      (a), ReplaceGateKeeper(b), OpenCoopObject(b, o), DoRunnerJob(a, r, q
      , o, t, g), CloseCageDoor(r,t, g, q), delete(runner(a)), delete(
      gateKeeper(b))]
 }

 task AssignRoles(g)
 {
  //finished - free
  case (agent(a,y) & roleAssigned(g,a) & zombie(g,x) & cageRoom(r) &
      isInRoom(g,r) & isInRoom(a,s) & r~=s & toggle_gate(t,m) &
      connectsRooms(t,r,p) & closed(t)) -> []

  //assigned - continue
  case (roleAssigned(g,a) & roleAssigned(g,b) & a~=b & zombie(g,x) &
      zombieRoom(r) & isInRoom(g,r) & toggle_gate(t,m) & agent(a,y) &
      connectsRooms(t,r,c) & coop_gate(o, p) & connectsRooms(o, c, d)) ->
      [SolveCoop(g, r, c, t, o, a, b)]
  case (roleAssigned(g,a) & roleAssigned(g,b) & a~=b & zombie(g,x) &
      cageRoom(c) & isInRoom(g,c) & toggle_gate(t,m) & agent(a,y) &
      connectsRooms(t,r,c) & coop_gate(o, p) & connectsRooms(o, c, d)) ->
      [SolveCoop(g, r, c, t, o, a, b)]

  //not enough agents - wait
  case() -> []

  //assigning done in code!
 }
}
```

Appendix C

Behavior Trees for the Hybrid Approach *I*

The documentation for standard nodes of the Modular Behavior Trees in CryEngine can be found under: `https://docs.cryengine.com/display/SDKDOC4/Modular+Behavior+Tree+Nodes`

Default Behavior Tree

```
<BehaviorTree>
 <Root>
<StateMachine>
   <State name="DefaultBT">
    <Transitions>
     <Transition to="CloseCage" onEvent="CloseCage!"/>
     <Transition to="GoThoughCoopObject" onEvent="GoThoughCoopObject!"/>
     <Transition to="OpenCoopObject" onEvent="OpenCoopObject!"/>
     <Transition to="StopInteractingAfterwards" onEvent="
        StopInteractingAfterwards!"/>
     <Transition to="TraverseThroughDoor" onEvent="TraverseThroughDoor!"/>
     <Transition to="GetGruntsAttention" onEvent="GetGruntsAttention!"/>
     <Transition to="TryToEnterCage" onEvent="TryToEnterCage!"/>
     <Transition to="TryToExitCage" onEvent="TryToExitCage!"/>
     <Transition to="StopPlan" onEvent="StopPlan!"/>
    </Transitions>
    <BehaviorTree>
     <Loop>
      <Log message="Waiting for commands"/>
     </Loop>
    </BehaviorTree>
   </State>
  </StateMachine>
 </Root>
</BehaviorTree>
```

"CloseCage" Behavior Tree

```
<BehaviorTree>
 <Root>
<StateMachine>
<State name="CloseCage">
   <Transitions>
    <Transition to="CloseCage" onEvent="CloseCage!"/>
    <Transition to="GoThoughCoopObject" onEvent="GoThoughCoopObject!"/>
    <Transition to="OpenCoopObject" onEvent="OpenCoopObject!"/>
    <Transition to="StopInteractingAfterwards" onEvent="
        StopInteractingAfterwards!"/>
    <Transition to="TraverseThroughDoor" onEvent="TraverseThroughDoor!"/>
    <Transition to="GetGruntsAttention" onEvent="GetGruntsAttention!"/>
    <Transition to="TryToEnterCage" onEvent="TryToEnterCage!"/>
    <Transition to="TryToExitCage" onEvent="TryToExitCage!"/>
    <Transition to="StopPlan" onEvent="StopPlan!"/>
   </Transitions>
   <BehaviorTree>
    <Loop>
     <Selector>
      <Parallel successMode="any">
       <Sequence>
        <WaitForEvent name="GruntLeftRoom"/>
        <X-functionName="FailBT"/>
        <Halt />
       </Sequence>
       <LoopUntilSuccess attemptCount="5">
        <Sequence>
         <Selector>
          <Sequence>
                    <Case functionName="PreconditionsHold"
                        returnValueOutputParam="True" />
                    <Log message="Preconditions hold"/>
          </Sequence>
          <Sequence>
                    <Case functionName="PreconditionsHold"
                        returnValueOutputParam="False"/>
                    <Log message="Preconditions do not hold"/>
                    <Action functionName="FailBT"/>
                    <Halt />
          </Sequence>
         </Selector>
         <Log message="cage open?"/>
         <Action name="Run"/>
         <Selector>
          <Sequence>
                    <Check functionName="IsClosed" returnValueOutputParam="
                        True"/>
                    <Action functionName="SucceedBT"/>
                    <Halt />
          </Sequence>
```

```xml
           <Sequence>
                        <Action name="StopAction"/>
                        <Parallel successMode="any">
                          <Selector>
                            <Action name="NavigateTo"/>
                            <Sequence>
                              <Action name="StepAside"/>
                              <Action name="NavigateTo"/>
                            </Sequence>
                          </Selector>
                          <LoopUntilSuccess>
                            <Check functionName="DistReached"
                                 returnValueOutputParam="True"/>
                          </LoopUntilSuccess>
                        </Parallel>
                        <Action name="AlignTo"/>
                        <Action name="InteractUse"/>
           </Sequence>
          </Selector>
         </Sequence>
        </LoopUntilSuccess>
        <Sequence>
         <WaitForEvent name="ToggleGateClosed"/>
         <Action name="SucceedBT"/>
         <Halt />
        </Sequence>
       </Parallel>
       <Sequence>
         <Action functionName="FailBT"/>
        <Halt />
       </Sequence>
      </Selector>
     </Loop>
    </BehaviorTree>
   </State>
  </StateMachine>
 </Root>
</BehaviorTree>
```

”TraverseThroughDoor” Behavior Tree

```
<BehaviorTree>
 <Root>
<StateMachine>
<State name="TraverseThroughDoor">
   <Transitions>
    <Transition to="CloseCage" onEvent="CloseCage!"/>
    <Transition to="GoThoughCoopObject" onEvent="GoThoughCoopObject!"/>
    <Transition to="OpenCoopObject" onEvent="OpenCoopObject!"/>
    <Transition to="StopInteractingAfterwards" onEvent="
        StopInteractingAfterwards!"/>
    <Transition to="TraverseThroughDoor" onEvent="TraverseThroughDoor!"/>
    <Transition to="GetGruntsAttention" onEvent="GetGruntsAttention!"/>
    <Transition to="TryToEnterCage" onEvent="TryToEnterCage!"/>
    <Transition to="TryToExitCage" onEvent="TryToExitCage!"/>
    <Transition to="StopPlan" onEvent="StopPlan!"/>
   </Transitions>
   <BehaviorTree>
    <Loop>
     <Selector>
      <Parallel>
       <Sequence>
        <WaitForEvent name="AgentEnteredRoom"/>
        <WaitForEvent name="DoorClosed"/>
        <functionName="SucceedBT"/>
        <Halt />
       </Sequence>
       <Sequence>
        <Selector>
         <Sequence>
          <Case functionName="PreconditionsHold" returnValueOutputParam="
              True"/>
          <Log message="Preconditions hold"/>
         </Sequence>
         <Sequence>
          <Case functionName="PreconditionsHold" returnValueOutputParam="
              False"/>
          <Log message="Preconditions do not hold"/>
          <Action name="FailBT"/>
          <Halt />
         </Sequence>
        </Selector>
        <Action name="Run"/>
        <LoopUntilSuccess attemptCount="5">
         <Parallel successMode="any">
          <Selector>
                   <Action name="NavigateTo"/>
                   <Sequence>
                     <Action name="StepAside"/>
                     <Action name="NavigateTo"/>
                   </Sequence>
          </Selector>
          <LoopUntilSuccess>
                      <Case functionName="MinDistReached"
                          returnValueOutputParam="True" />
          </LoopUntilSuccess>
         </Parallel>
        </LoopUntilSuccess>
        <LoopUntilSuccess attemptCount="5">
         <Selector>
          <Sequence>
                      <Case functionName="IsClosed" returnValueOutputParam="
                          True" />
                      <Action name="AlignTo"/>
                      <Action name="InteractUse"/>
                      <WaitForEvent name="DoorOpened"/>
                      <Action name="StepThrough"/>
          </Sequence>
          <Sequence>
                      <<Case functionName="IsClosed" returnValueOutputParam="
                          False"/>
                      <Action name="StepThrough"/>
          </Sequence>
         </Selector>
        </LoopUntilSuccess>
        <LoopUntilSuccess attemptCount="5">
         <Sequence>
          <<Case functionName="IsClosed" returnValueOutputParam="False"/>
          <Action name="AlignTo"/>
          <Action name="InteractUse"/>
          <WaitForEvent name="DoorClosed"/>
          <Action name="SucceedBT"/>
         </Sequence>
        </LoopUntilSuccess>
       </Sequence>
      </Parallel>
      <Sequence>
       <Action name="FailBT"/>
       <Halt />
      </Sequence>
     </Selector>
    </Loop>
   </BehaviorTree>
  </State>
    </StateMachine>
 </Root>
</BehaviorTree>
```

”GoThoughCoopObject” Behavior Tree

```
<BehaviorTree>
 <Root>
<StateMachine>
<State name="GoThoughCoopObject">
   <Transitions>
    <Transition to="CloseCage" onEvent="CloseCage!"/>
    <Transition to="GoThoughCoopObject" onEvent="GoThoughCoopObject!"/>
    <Transition to="OpenCoopObject" onEvent="OpenCoopObject!"/>
    <Transition to="StopInteractingAfterwards" onEvent="
        StopInteractingAfterwards!"/>
    <Transition to="TraverseThroughDoor" onEvent="TraverseThroughDoor!"/>
    <Transition to="GetGruntsAttention" onEvent="GetGruntsAttention!"/>
    <Transition to="TryToEnterCage" onEvent="TryToEnterCage!"/>
    <Transition to="TryToExitCage" onEvent="TryToExitCage!"/>
    <Transition to="StopPlan" onEvent="StopPlan!"/>
   </Transitions>
   <BehaviorTree>
    <Loop>
     <Selector>
      <Parallel>
       <Sequence>
        <Action name="StopAction"/>
        <WaitForEvent name="AgentEnteredRoom"/>
        <Action name="SucceedBT"/>
        <Halt />
       </Sequence>
       <LoopUntilSuccess attemptCount="5">
        <Sequence>
         <Selector>
          <Sequence>
                        <Case functionName="PreconditionsHold"
                            returnValueOutputParam="True"/>
                        <Log message="Preconditions hold"/>
          </Sequence>
          <Sequence>
                        <Case functionName="PreconditionsHold"
                            returnValueOutputParam="False"/>
                        <Log message="Preconditions do not hold"/>
                        <Action name="FailBT"/>
                        <Halt />
          </Sequence>
         </Selector>
         <Action name="Run"/>
         <Parallel successMode="any">
          <Selector>
                        <Action name="NavigateTo"/>
                        <Sequence>
                          <Action name="StepAside"/>
                          <Action name="NavigateTo"/>
                        </Sequence>
          </Selector>
          <LoopUntilSuccess>
                    <Case functionName="MinDistReached" returnValueOutputParam="True"/>
          </LoopUntilSuccess>
         </Parallel>
         <Selector>
          <Sequence>
                    <Case functionName="IsClosed" returnValueOutputParam="True"/>
                    <WaitForEvent name="CoopOpened"/>
                    <LoopUntilSuccess attemptCount="5">
                      <Action name="StepThrough"/>
                    </LoopUntilSuccess>
                    <Sequence>
                      <WaitForEvent name="AgentEnteredRoom"/>
                      <Action name="SucceedBT"/>
                    </Sequence>
          </Sequence>
          <Sequence>
                    <Case functionName="IsClosed" returnValueOutputParam="False"/>
                    <Action name="StepThrough"/>
                    <Sequence>
                      <WaitForEvent name="AgentEnteredRoom"/>
                      <Action name="SucceedBT"/>
                    </Sequence>
          </Sequence>
         </Selector>
        </Sequence>
       </LoopUntilSuccess>
      </Parallel>
      <Sequence>
       <Action name="FailBT"/>
       <Halt />
      </Sequence>
     </Selector>
    </Loop>
   </BehaviorTree>
  </State>
       </StateMachine>
 </Root>
</BehaviorTree>
```

”OpenCoopObject” Behavior Tree

```
<BehaviorTree>
 <Root>
<StateMachine>
<State name="OpenCoopObject">
   <Transitions>
    <Transition to="CloseCage" onEvent="CloseCage!"/>
    <Transition to="GoThoughCoopObject" onEvent="GoThoughCoopObject!"/>
    <Transition to="OpenCoopObject" onEvent="OpenCoopObject!"/>
    <Transition to="StopInteractingAfterwards" onEvent="
        StopInteractingAfterwards!"/>
    <Transition to="TraverseThroughDoor" onEvent="TraverseThroughDoor!"/>
    <Transition to="GetGruntsAttention" onEvent="GetGruntsAttention!"/>
    <Transition to="TryToEnterCage" onEvent="TryToEnterCage!"/>
    <Transition to="TryToExitCage" onEvent="TryToExitCage!"/>
    <Transition to="StopPlan" onEvent="StopPlan!"/>
   </Transitions>
   <BehaviorTree>
    <Loop>
     <Selector>
      <Parallel>
       <Sequence>
        <WaitForEvent name="AgentEnteredRoom"/>
        <Action name="StopAction"/>
        <Action name="StepAside"/>
        <Action name="SucceedBT"/>
        <Halt />
       </Sequence>
       <Sequence>
        <Selector>
         <Sequence>
          <Case functionName="PreconditionsHold" returnValueOutputParam="
              True"/>
          <Log message="Preconditions hold"/>
         </Sequence>
         <Sequence>
          <Case functionName="PreconditionsHold" returnValueOutputParam="
              False"/>
          <Log message="Preconditions do not hold"/>
          <Action name="FailBT"/>
          <Halt />
         </Sequence>
        </Selector>
        <Selector>
         <Case functionName="MinDistReached" returnValueOutputParam="True"/>
         <Sequence>
          <Action name="Run"/>
          <Action name="StopAction"/>
          <LoopUntilSuccess attemptCount="5">
                    <Parallel successMode="any">
                     <Selector>
```

```
                      <Action name="NavigateTo"/>
                      <Sequence>
                        <Action name="StepAside"/>
                        <Action name="NavigateTo"/>
                      </Sequence>
                    </Selector>
                    <LoopUntilSuccess>
                      <Case functionName="MinDistReached"
                           returnValueOutputParam="True"/>
                    </LoopUntilSuccess>
                  </Parallel>
         </LoopUntilSuccess>
        </Sequence>
       </Selector>
       <LoopUntilSuccess attemptCount="5">
        <Selector>
         <Sequence>
                  <Case functionName="IsClosed" returnValueOutputParam="
                       True"/>
                  <Action name="AlignTo"/>
                  <Action name="StopAction"/>
                  <Action name="InteractHold"/>
                  <WaitForEvent name="CoopOpened"/>
                  <WaitForEvent name="AgentEnteredRoom"/>
         </Sequence>
         <Sequence>
                  <Case functionName="IsClosed" returnValueOutputParam="
                       False"/>
                  <Action name="AlignTo"/>
                  <Action name="StopAction"/>
                  <Action name="InteractHold"/>
                  <WaitForEvent name="AgentEnteredRoom"/>
         </Sequence>
        </Selector>
       </LoopUntilSuccess>
      </Sequence>
     </Parallel>
     <Sequence>
      <Action name="FailBT"/>
      <Halt />
     </Sequence>
    </Selector>
   </Loop>
  </BehaviorTree>
 </State>
      </StateMachine>
 </Root>
</BehaviorTree>
```

"GetGruntsAttention" Behavior Tree

```
<BehaviorTree>
 <Root>
<StateMachine>
<State name="GetGruntsAttention">
   <Transitions>
    <Transition to="CloseCage" onEvent="CloseCage!"/>
    <Transition to="GoThoughCoopObject" onEvent="GoThoughCoopObject!"/>
    <Transition to="OpenCoopObject" onEvent="OpenCoopObject!"/>
    <Transition to="StopInteractingAfterwards" onEvent="StopInteractingAfterwards!"/>
    <Transition to="TraverseThroughDoor" onEvent="TraverseThroughDoor!"/>

    <Transition to="GetGruntsAttention" onEvent="GetGruntsAttention!"/>
    <Transition to="TryToEnterCage" onEvent="TryToEnterCage!"/>
    <Transition to="TryToExitCage" onEvent="TryToExitCage!"/>
    <Transition to="StopPlan" onEvent="StopPlan!"/>
   </Transitions>
   <BehaviorTree>
    <Loop>
     <Selector>
      <Parallel successMode="any">
       <Sequence>
        <WaitForEvent name="TargetSawMe"/>
        <Log message="target saw me"/>
        <Action name="SucceedBT"/>
       </Sequence>
       <LoopUntilSuccess attemptCount="10">
        <Sequence>
         <Selector>
          <Sequence>
                    <Case functionName="PreconditionsHold" returnValueOutputParam="True"/>
                    <Log message="Preconditions hold"/>
          </Sequence>
          <Sequence>
                    <Case functionName="PreconditionsHold" returnValueOutputParam="False"/>
                    <Log message="Preconditions do not hold"/>
                    <Action name="FailBT"/>
                    <Halt />
          </Sequence>
         </Selector>
         <Action name="Run"/>
         <Selector>
          <Sequence>
                    <Case functionName="IAmVisible" returnValueOutputParam="True"/>
                    <Log message="I am visible"/>
                    <Action name="SucceedBT"/>
                    <Halt />
          </Sequence>
          <Sequence>
                      <Log message="not visible"/>
                      <Action name="NavigateTo"/>
                      <Case functionName="IAmVisible" returnValueOutputParam="
                          True"/>
          </Sequence>
         </Selector>
        </Sequence>
       </LoopUntilSuccess>
      </Parallel>
      <Sequence>
       <Action name="FailBT"/>
       <Halt />
      </Sequence>
     </Selector>
    </Loop>
   </BehaviorTree>
  </State>
       </StateMachine>
 </Root>
</BehaviorTree>
```

”TryToEnterCage” Behavior Tree

```
<BehaviorTree>
 <Root>
<StateMachine>
  <State name="TryToEnterCage">
    <Transitions>
     <Transition to="CloseCage" onEvent="CloseCage!"/>
     <Transition to="GoThoughCoopObject" onEvent="GoThoughCoopObject!"/>
     <Transition to="OpenCoopObject" onEvent="OpenCoopObject!"/>
     <Transition to="StopInteractingAfterwards" onEvent="
         StopInteractingAfterwards!"/>
     <Transition to="TraverseThroughDoor" onEvent="TraverseThroughDoor!"/>
     <Transition to="GetGruntsAttention" onEvent="GetGruntsAttention!"/>
     <Transition to="TryToEnterCage" onEvent="TryToEnterCage!"/>
     <Transition to="TryToExitCage" onEvent="TryToExitCage!"/>
     <Transition to="StopPlan" onEvent="StopPlan!"/>
    </Transitions>
    <BehaviorTree>
     <Loop>
      <Selector>
       <Parallel>
        <Sequence>
         <WaitForEvent name="AgentEnteredRoom"/>
         <Action name="SucceedBT"/>
         <Halt />
        </Sequence>
        <LoopUntilSuccess attemptCount="5">
         <Sequence>
          <Selector>
           <Sequence>
                     <Case functionName="PreconditionsHold"
                         returnValueOutputParam="True"/>
                     <Log message="Preconditions hold"/>
           </Sequence>
           <Sequence>
                     <Case functionName="PreconditionsHold"
                         returnValueOutputParam="False"/>
                     <Log message="Preconditions do not hold"/>
                     <Action name="FailBT"/>
                     <Halt />
           </Sequence>
          </Selector>
          <Action name="Run"/>
          <Action name="NavigateTo"/>
          <Log message="toggle gate Open?"/>
          <Selector>
           <Sequence>
                     <Case functionName="IsClosed" returnValueOutputParam="
                         True"/>
                     <Action name="AlignTo"/>
                     <Action name="InteractHold"/>
```

```
                    <WaitForEvent name="ToggleGateOpened"/>
                    <Action name="StepThrough"/>
          </Sequence>
          <Sequence>
                    <Case functionName="IsClosed" returnValueOutputParam="
                        False"/>
                    <Action name="StepThrough"/>
          </Sequence>
         </Selector>
        </Sequence>
       </LoopUntilSuccess>
      </Parallel>
      <Sequence>
       <Action name="FailBT"/>
       <Halt />
      </Sequence>
     </Selector>
    </Loop>
   </BehaviorTree>
  </State>
       </StateMachine>
 </Root>
</BehaviorTree>
```

”TryToExitCage” Behavior Tree

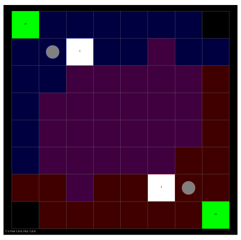
```
<BehaviorTree>
 <Root>
<StateMachine>
  <State name="TryToExitCage">
    <Transitions>
     <Transition to="CloseCage" onEvent="CloseCage!"/>
     <Transition to="GoThoughCoopObject" onEvent="GoThoughCoopObject!"/>
     <Transition to="OpenCoopObject" onEvent="OpenCoopObject!"/>
     <Transition to="StopInteractingAfterwards" onEvent="
         StopInteractingAfterwards!"/>
     <Transition to="TraverseThroughDoor" onEvent="TraverseThroughDoor!"/>
     <Transition to="GetGruntsAttention" onEvent="GetGruntsAttention!"/>
     <Transition to="TryToEnterCage" onEvent="TryToEnterCage!"/>
     <Transition to="TryToExitCage" onEvent="TryToExitCage!"/>
     <Transition to="StopPlan" onEvent="StopPlan!"/>
    </Transitions>
    <BehaviorTree>
     <Loop>
      <Selector>
       <Parallel>
        <Sequence>
         <WaitForEvent name="AgentEnteredRoom"/>
         <Action name="Run"/>
         <WaitForEvent name="CoopClosed"/>
         <Action name="SucceedBT"/>
         <Halt />
        </Sequence>
        <LoopUntilSuccess attemptCount="5">
         <Sequence>
          <Selector>
           <Sequence>
                    <Case functionName="PreconditionsHold"
                        returnValueOutputParam="True"/>
                    <Log message="Preconditions hold"/>
           </Sequence>
           <Sequence>
                    <Case functionName="PreconditionsHold"
                        returnValueOutputParam="False"/>
                    <Log message="Preconditions do not hold"/>
                    <Action name="FailBT"/>
                    <Halt />
           </Sequence>
          </Selector>
          <Parallel successMode="any">
           <Action name="NavigateTo"/>
           <LoopUntilSuccess>
                    <Case functionName="MinDistReached"
                        returnValueOutputParam="True"/>
           </LoopUntilSuccess>
          </Parallel>
        <Action name="Crouch"/>
        <Log message="doorOpen?"/>
        <Selector>
         <Sequence>
                    <Case functionName="IsClosed" returnValueOutputParam="
                        True"/>
                    <WaitForEvent name="CoopOpened"/>
                    <Action name="StepThrough"/>
                    <Sequence>
                      <WaitForEvent name="AgentEnteredRoom"/>
                      <Action name="SucceedBT"/>
                    </Sequence>
         </Sequence>
         <Sequence>
                    <Case functionName="IsClosed" returnValueOutputParam="
                        False"/>
                    <Action name="StepThrough"/>
                    <Sequence>
                      <WaitForEvent name="AgentEnteredRoom"/>
                      <Action name="SucceedBT"/>
                    </Sequence>
         </Sequence>
        </Selector>
       </Sequence>
      </LoopUntilSuccess>
     </Parallel>
     <Sequence>
      <Action name="FailBT"/>
      <Halt />
     </Sequence>
    </Selector>
   </Loop>
  </BehaviorTree>
 </State>
      </StateMachine>
 </Root>
</BehaviorTree>
```

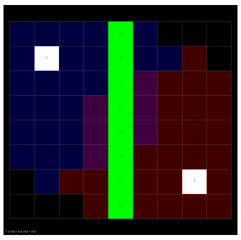
”StopPlan” Behavior Tree

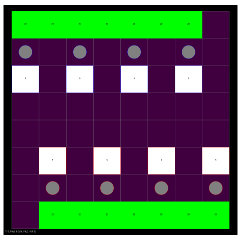
```
<BehaviorTree>
 <Root>
<StateMachine>
<State name="StopPlan">
   <Transitions>
    <Transition to="CloseCage" onEvent="CloseCage!"/>
    <Transition to="GoThoughCoopObject" onEvent="GoThoughCoopObject!"/>
    <Transition to="OpenCoopObject" onEvent="OpenCoopObject!"/>
    <Transition to="StopInteractingAfterwards" onEvent="
        StopInteractingAfterwards!"/>
    <Transition to="TraverseThroughDoor" onEvent="TraverseThroughDoor!"/>
    <Transition to="GetGruntsAttention" onEvent="GetGruntsAttention!"/>
    <Transition to="TryToEnterCage" onEvent="TryToEnterCage!"/>
    <Transition to="TryToExitCage" onEvent="TryToExitCage!"/>
    <Transition to="StopPlan" onEvent="StopPlan!"/>
   </Transitions>
   <BehaviorTree>
    <Halt />
   </BehaviorTree>
  </State>
      </StateMachine>
 </Root>
</BehaviorTree>
```

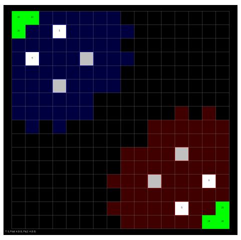
Appendix D

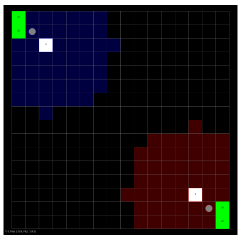
Game Maps for the Hybrid Approach *II*

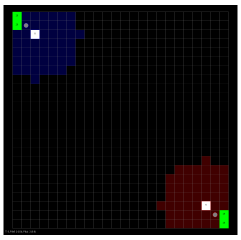

(a) basesWorkers-8x8A

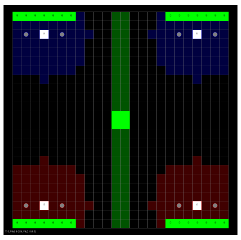

(b) NoWhereToRun-9x8

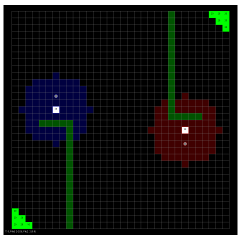

(c) FourBasesWorkers-8x8


(d) TwoBasesBarracks-16x16


(e) basesWorkers-16x16A


(f) basesWorkers-24x24A


(g) DoubleGame-24x24


(h) BWDistantResources-32x32

□ **Base** ▨ **Barracks** 🟩 **Resource** ● **Worker** 🟩 **Wall**

Figure D.1: Start configurations of the game maps used in the experiments of Chapter 6. Player 1 starts on the top/left side, player 2 on the bottom/right side of each map. The blue, red, and violet background colors indicate the visibility of a cell to player 1, 2, or both respectively in a partially observable mode. However, these are not relevant for our experiments, which are performed with full observability.

E h r e n e r k l ä r u n g

Ich versichere hiermit, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; verwendete fremde und eigene Quellen sind als solche kenntlich gemacht. Insbesondere habe ich nicht die Hilfe eines kommerziellen Promotionsberaters in Anspruch genommen. Dritte haben von mir weder unmittelbar noch mittelbar geldwerte Leistungen für Arbeiten erhalten, die im Zusammenhang mit dem Inhalt der vorgelegten Dissertation stehen.

Ich habe insbesondere nicht wissentlich:

- Ergebnisse erfunden oder widersprüchliche Ergebnisse verschwiegen,
- statistische Verfahren absichtlich missbraucht, um Daten in ungerechtfertigter Weise zu interpretieren,
- fremde Ergebnisse oder Veröffentlichungen plagiiert,
- fremde Forschungsergebnisse verzerrt wiedergegeben.

Mir ist bekannt, dass Verstöße gegen das Urheberrecht Unterlassungs- und Schadensersatzansprüche des Urhebers sowie eine strafrechtliche Ahndung durch die Strafverfolgungsbehörden begründen kann. Die Arbeit wurde bisher weder im Inland noch im Ausland in gleicher oder ähnlicher Form als Dissertation eingereicht und ist als Ganzes auch noch nicht veröffentlicht.

Magdeburg, den 17.08.2020

Xenija Neufeld