Automatic hierarchical task network planning system for the Unity game engine

Boris Krylov[a], Maxim Abramov[a,b]

[a]St. Petersburg State University, St. Petersburg, Russia
[b]St. Petersburg Institute for Informatics and Automation of the Russian Academy of Sciences, St. Petersburg, Russia

Abstract

The article relates to the automatic planning system integrated into the Unity game engine environment, which controls an agent behavior. A graphical notation for domain knowledge definition was developed. Goals reached in this paper will allow quick complex behaviour pattern implementation for non-player characters within the game environment.

1. Introduction

Social engineering attacks are different sets of applied psychological and analytical methods and techniques used by malefactors to trick the users of public or corporate network into violating established rules and information security policies, aiming to steal data, personal information, money and even identity [19, 2]. These methods vary from simple use of victim's carelessness to sophisticated fraudulent schemes. In the most cases, it's easier and cheaper for scammers to get what they want from the users themselves, than try to find vulnerabilities in the system [19].

The problem of social engineering attacks has been relevant for a long time. This fact is confirmed by both the statistics of news agencies [7] and the research of well-known cybersecurity companies [20]. Based on the laboratory of theoretical and interdisciplinary problems of computer science studies on the topic of automated assessing the level of user protection via artificial intelligence methods such as Bayes networks [9, 10] and how to increase it are being conducted [1]. Particularly, the level of protection from social engineering attacks of information systems users can be increased through a digital educational game. Despite the relevance of the task, confirmed by the demand for training, surveys, etc. from organizations, such games have not yet been presented. The general direction of the research branch is to solve the designated problem — the development of an educational game that helps to increase user awareness of social engineering attacks and, consequently, security. As a part of this development, it was necessary to propose a game artificial intelligence system, i.e. a component of the game that serves to control non-player characters (NPC), creating the illusion of their

Russian Advances in Artificial Intelligence: selected contributions to the Russian Conference on Artificial intelligence (RCAI 2020), October 10-16, 2020, Moscow, Russia
krylovbs2301@gmai.com (B. Krylov); mva@dscs.pro (M. Abramov)

meaningful activities. NPC, in turn, act as adversaries for the player and the crowd [4, 12] in the game environment.

Typically, the game artificial intelligence system is used to solve following problems:

- Movement and pathfinding. This problem concerns the character moves within the game environment taking into account surroundings, obstacles and environmental features.
- Interaction with the environment and simple behavioural reactions. To simulate different activities NPC need to be able to interact with their surroundings and change their behaviour according the changes of the environment.
- Decision-making. In other words, the choice between several different sequences of elementary actions, the NPC can take to achieve the given task[4].

The work proposes an automatic hierarchical task network planning system, implemented as a part of the development of educational game about social engineering attacks. This planning system is out to generate sequences of NPC's actions based on the current state of environment. This will solve the problem of simulating the decision-making process and behavioural reactions. Creating instruments to encode and execute different planning problems allows to set the multitude of behaviour patterns for non-player characters.

The main reasoning for choosing hierarchical task network planning is its expressiveness. Some approaches to behaviour modeling became classic in video game industry, such as finite-state machines (FSM) or behaviour trees[5, 4]. However, FSM approach limits the number of different situations the system can process and the transition conditions [4]. The more nuanced and complex behaviour it is required to model, the more compound and extensive state diagram of the system becomes. On the other hand, automated planners are better suited for the problems with multiple goals.

Making a believable simulation of human behaviour, composed of purposeful activities and lots of varying reactions to the external stimuli, is in fact a problem with multiple goals. Therefore, it is appropriate to use automated planning. There are two types of automated planners that have been used in games: Stanford Institute Research Problem Solver (STRIPS) and Hierarchial Task Network planning (HTN planning). But HTN is more expressive than STRIPS [6] and allows more intuitive and easier approach to designing NPCs' behaviour.

2. Related works

The development of game artificial intelligence has been repeatedly described in the scientific literature [4, 5, 25]. In the book "Artificial Intelligence and Games" it is proposed to use planners as the basis for such a system[25].

The problem of using automated planners to simulate meaningful activity in games has been around . The most striking example of the use of symbolic planning in games is the F.E.A.R. first-person horror shooter 2005 year. STRIPS planning (Stanford Institute Research Problem Solver) was used in this game [18, 25]. The game gained fame thanks to its advanced system of game AI, which simulated the behavior of a detachment of opponents through a complex consisting of goals and actions needed to achieve said goals [17].

The faster and more efficient planning method is HTN planning [5]. The main idea of this method is that the target task is decomposed into a network of subtasks, which in turn can be divided into other networks of subtasks, and so on, until a network of tasks consists of elementary actions is obtained [6]. This approach allows to interpret and construct tasks according to the "top-down" principle, which describes the domain knowledge using the cognitive approach to the problem of categorizing tasks included in it [11].

In an academic environment, HTN planning has been used and studied for a very long time. One of the first HTN planning systems, Nonlin, was developed and introduced in 1976 [22]. In Nonlin, the task network was represented as a node graph. There was no obvious distinction between composite and primitive tasks. Nonlin supported both ordered plans and partially ordered plans.

Since then, many HTN planning systems have been developed, including SHOP2 (Simple Hierarchial Ordered Planner 2)[14], which has been successfully used in crisis management and logistics [6], production planning [6], projects [13, 6] and videogames [8]. For example, in the game "The Elder Scrolls IV: Oblivion" a version of the SHOP2 system developed using Java programming language.

Also, HTN planning was successfully applied in games such as Transformers: Fall of Cybertron [5] and Horizon Zero Dawn [23], for which their own planning systems were developed. In academic environment, research on HTN planning in games is still relevant. For example, in [16], a high-performance gaming artificial intelligence system based on HTN planning was proposed. The researchers were faced with the task of making a search for a suitable plan as quick as possible. To do this, the size of the space in which the plan is searched was minimally reduced.

In the paper [21] an automatic planning system for a real-time strategy was proposed. Since real-time strategies have a very large space for a plan search, the researchers were faced the task of creating an effective algorithm for searching in it.

Thus, it can be concluded that HTN planning systems have been successfully used to develop games of various genres. Nevertheless, in the context of serious games about social engineering attacks, the decision is being applied for the first time.

That is partly due to the fact that there are no serious video games dedicated to teaching its users about social engineering attacks. The main pupose of the serious game presented by K. Beckers and S. Pape is created for elicitation of the company's vulnerabilities to the social engineering attacks [3]. And while it can be used to teach players, it is still a board game with no digital content whatsoever. Another game, "Playing safe", that was developed by M. Newbould and S. Furnell, is actually a video game and aims to raise awareness [15]. But it was designed to not have any NPCs, so naturally, there was no need in game AI.

3. Defining HTN planning

Hierarchical task network planning is based on the idea that any task in a set of tasks can be performed using a primitive action, or it can be decomposed into a sequence of tasks on which some constraints are imposed.

The HTN planning language is used to describe hierarchical task network planning prob-

lems.

3.1. Planning language

A *planning language* is a tuple of mutually disjoint sets of characters: a set of predicate characters P and a set of terms T. A set of terms consists of a finite set of constant characters and an infinite set of variable characters. Let Q be a set of predicates. A predicate is closed if its terms do not contain variable symbols. A state $s \in 2^Q$ is a set of closed predicates for which the *closed-world assumption* is adopted, i.e. any predicate that is not included in the set is considered false [6].

3.2. Tasks

There are two types of tasks in HTN planning: primitive and compound.

Primitive tasks are denoted by $t_p(\tau)$, where τ is a term. Primitive task set is denoted by T_p. Each primitive task is represented by an operator.

An *operator* o is a tuple $(t_p(\tau), f_{pre}(o), f_{eff}(o))$, where $f_{pre}(o), f_{eff}(o) \in 2^Q$ are the *preconditions* and *effects* of the operator, respectively. Preconditions can be both positive, then they are denoted by $f_{pre}(o)^+$, and negative — $f_{pre}(o)^-$. An operator is called *applicable to a state* s if $(f_{pre}(o)^+ \subseteq s) \wedge (f_{pre}(o)^- \cap s) = \emptyset$. The result of this application will be a new state $s *= (s \setminus f_{eff}(o)^-) \cup f_{eff}(o)^+$, where $f_{eff}(o)$ are the negative effects of o, and $f_{eff}(o)^+$ are positive.

A *composite problem* is denoted by $t_c(m) \in T_c$, where m is a decomposition method[6].

3.3. Task networks and decomposition

A *decomposition method* m is a tuple $(t_c(m), f_{pre}(o), tn(m))$, where $tn(m)$ denotes a task network, and $f_{pre}(o)$ is preconditions for the decomposition method.

A *task network* is a pair (T, Ψ), where T is a finite set of tasks, and Ψ is a set of constraints imposed on this set.

In the developed automatic planning system, the set of tasks of any task network is strictly partially ordered, which is the only restriction that is imposed on a task set by default.

A *decomposition method* with preconditions $f_{pre}(o)$ is applicable to state s in the same way as an operator with the same preconditions is applied to this state. The result of applying the decomposition method is a task network.

Let's define a decomposition. Let the state s be given and $tn_c = (T_c, \prec_c)$ be a task network ($\prec_c$ is a relation of strict partial order). The method m, applicable to state s, decomposes tn_c into a network of tasks tn_n, by replacing task t, if and only if the following conditions are satisfied:

- $t = t_c(m)$;
- $tn_n := ((T_c \setminus t) \cup T_m, \prec_c \cup \prec_m \cup \prec_D)$, where $\prec_D := \{(t_1, t_2) \in T_c \times T_m | (t_1, t) \in \prec_c\} \cup \{(t_1, t_2) \in T_c \times T_m | (t, t_2) \in \prec_c\}$[6].

3.4. Planning problem and solution

A planning problem is a tuple (Q, O, M, tn_0, s_0), where:

- Q — a set of predicates;
- O — a set of operators;
- M — a set of methods;
- tn_0 — the initial network of tasks;
- s_0 — the initial state.

A planning problem solution is a sequence of operators $o_1, o_2, \dots, o_n$, that is executable in s_0. That is, there exists a trajectory of states $s_1, s_2, \dots, s_n$, such that o_i is applicable to the state s_{i-1}, and for any $i \in 1, \dots, n$. The state s_i is the result of applying o_i to s_{i-1}. The solution is found by decomposing tn_0[6].

4. Used technologies

To create a serious game about social engineering attacks the Unity game engine was used. The choice of the game engine for this project was motivated by several considerations. First and foremost, to make the game more immersive and interesting to play and study, it was deemed necessary to create an imitation of a company or an organisation in a three dimensional environment. One of the benefits of this approach is that it allows the visual representation of the social engineering attacks. The Unity game engine supports three dimensional games which can be created using Unity game editor. The versatility and usability of the editor helped to concentrate on the development of game logic and social engineering attack imitation systems.

However, to implement completely a system for social engineering attacks imitation, it was important to create a system that controls the behavior of non-player characters that would imitate their meaningful activities. Since the imitation system was developed using Unity game engine, the automatic planning system should have been integrated into this environment, taking into account the features of the platform. In particular, taking into account that all function calls in the Unity game engine are tied to certain events, like game scene initialization or beginning of a frame rendering.

5. Implementation

A typical system implementation consists of 3 key components: agent, planner and task network. All of these depend on the implementation of the game environment, however, the logic behind task transition and decomposition can be made universal. Our GitHub project[1] implements such logic.

[1]Hierarchical Task Network planner for Unity. Simple scripts that help you to build AI systems based on HTN planning. GitHub project page URL: https://github.com/KrylovBoris/Hierarchical-Task-Network-planner-for-Unity

The UML class diagram depicted in figure 1 showcases the classes of the developed system. Class Agent acts as the domain knowledge holder for the planning problem. The predicates that describe preconditions of tasks and states of the environment are Boolean methods, which serve as sensors of the agent. Using sensors, agents and their associated task networks perceive the current state of the virtual environment. Sensors and agent actions are also provided by the Agent class. As the implementation of these components may vary from one game environment to another, it impossible to create a universal agent.

The HTN planner class is a scheduler in sense that it starts the task execution and handles the situations when it is interrupted. For the different planning problems, the scheduler may handle the interruptions differently, however out GitHub project page has an HTN planner class example that can be used in other projects with minor alterations.

Classes Task, SimpleTask, ComplexTask and Plan are the system's core components that are handling the planning process and various condition checks.

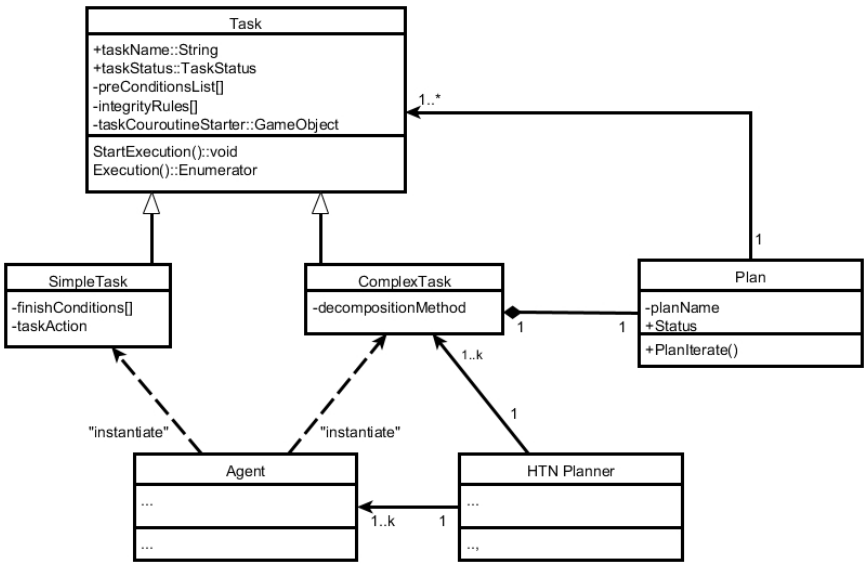

Figure 1: The structure of classes included in the automatic planning system.

For the guidelines on how to implement Agent and HTN planner class, please, refer to the project's GitHub page.

5.1. Tasks

As can be seen from figure 1, composite and primitive tasks are the heirs of the Task class, that implements the logic of checking preconditions, task execution, and also integrity rules for the task.

Integrity rules are an extension of the classsic HTN planning method within our implementation, they are the special conditions that should not be violated throughout the entire time the

task is being executed. These are helpful when we want the continuous task to be interruptible.

While the task is executed, it take different states, such as:

- **"Planned"** is an initial state which indicates that the task is in the task network, but is not being executed.
- **"InProgress"** is a state that shows that the task is being executed.
- The task is **"Complete"** after it is successfully executed and removed from the planning queue.
- **"Failed"** state is assigned to the task when its preconditions are not met or some of the integrity rules are violated.

5.2. Primitive tasks

The SimpleTask class implements logic related to operators and primitive tasks. Each instance of the class calls an agent method, which changes the state of the environment. If the effect of performing this action appears only after some time, we can postpone the transition to the next task using completion conditions. The transition to a new task will not be carried out if these conditions are not met.

To provide verification of completion conditions and task integrity rules, a Unity coroutine is instantiated for each task in the system. Coroutine in Unity game engine is a function which execution can be extended over several frames [24].

The effects in this model are not explicitly encoded, since a feature of the implemented system is that the task network is decomposed until the first primitive task is received, which is immediately executed, changing the state of the environment perceived by the agent's sensors. Such planning is called a plan search in the state space [18]. Thus, there is no need to encode explicitly the effects of operators.

5.3. Compound tasks and task networks

Compound tasks are implemented in the ComplexTask class. Each instance of this class is associated with an instance of the Plan class, which monitors the state of the task network and controls the transition from one subtask in the network to another. Each plan contains a task queue, which is formed on the basis of the task decomposition method.

Under the notion "task decomposition method" in this implementation is meant a function that takes an input state of the environment and returns an array of tasks.

Each plan can take one of this three states:

- "InProgress";
- "Complete";
- "Failed".

These states partially correspond with the task state. That is because of the fact that each ComplexTask instance state is tied to its dedicated Plan instance. While the plan is being executed it is "InProgress". If the planning queue is successfully cleared, then the plan and the task associated with that plan is "Complete". However, whilst executing the obtained plan, a situation may arise when one task in the plan became "Failed". In this case, the entire structure is assigned this status. Such cases when the execution of the plan is interrupted for some reason are handled separately by the planner in accordance with the specifics of the implementation of the environment. For example, the scheduler may restart the planning process after an error. Also, this component of the system is responsible for the initial launch of the planning process and the further behavior of the system after the successful completion of the plan.

More detailed description of the implementation can be found on the project's GitHub page. There you can also find documentation and simple tutorial on the integration of developed system into your Unity project. It is should be pointed out that the project is still in development and the project page will be updated in the future.

6. Graphical Notation

One of the most important tasks in the description of the planning problem is the description of a domain knowledge in accordance with the given semantics. For the planning system described in this paper, a task can be described using code, however, this leads to a large amount of boilerplate code. In this regard, a graphical notation was developed using the Microsoft Modeling SDK to make it possible to describe quickly planning tasks.

Figure 2 describes a simple planning problem using this notation. According to the compiled description, C# language code is generated using the T4 text template.

Using this graphic language, one can describe sensor methods and agent action methods (column "Agent Methods"). The specific implementation of these methods requires further determination using C# programming language and can vary greatly depending on the environment implemented on the Unity game engine. Primarily this graphic notation is designed to be used for the description of primitive and composite tasks. In this notation, it is possible to specify the preconditions, integrity rules and conditions defined in the first column. It can also assign action methods to primitive tasks, and decomposition methods can be defined for composite tasks.

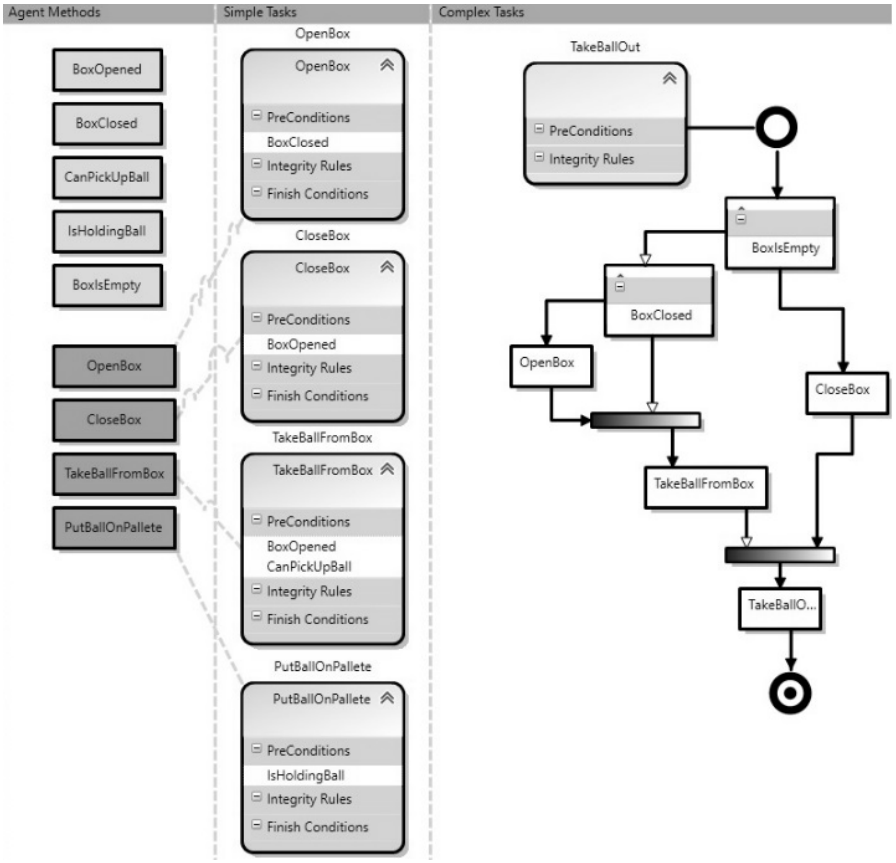

Figure 2: Graphic notation used to describe the task of extracting the ball from a closed box.

As mentioned above, task decomposition methods are the functions that determine which tasks a task should be decomposed into in a given state of the environment. To describe this function, several blocks of various types are used: blocks of a beginning and an end of an algorithm, task blocks (for determining an order of tasks in a resulting array) and branching blocks (for changing a composition of a resulting array under certain conditions). It is also possible to create recursive task networks by placing the composite task in the task decomposition method associated with it. With their help, we can express cycles within task networks. Thus, using the developed notation, all basic algorithmic structures can be expressed. One method of decomposing a problem can encode several tuples $(t_c(m), f_{pre}(o), tn(m))$. It is important to note that not all preconditions of the decomposition method are placed in the body of the function. Preconditions common to all tuples are placed in the preconditions of the composite problem.

The usage of presented notation can save time spent on coding. It omits some implementation features, allowing to concentrate solely on designing the planning problem. The visual nature of the notation and aforementioned omission allows users that has no experience in programming to encode task networks. It can be extremely helpful for the diverse teams of game developers. Game designers can design NPC behaviour using blocks of sensors and actions

that were prepared by the programmer.

But despite being a valuable asset during the development the serious game about social engineering attacks, this extension requires some polishing before it can be properly shipped.

7. Conclusion

This article describes the development of the game artificial intelligence system based on hierarchical task network planning, which is proposed to be used in a game dedicated to training users to counteract social engineering attacks. A graphical notation for describing a planning problem is presented. It is planned to finalize the graphical notation and ship a fleshed out extension for the Visual Studio IDE in the future.

It also seems promising to create a graphical planning problem editor for the Unity game editor, so that developers have the ability to edit agent behavior directly from Unity Editor. This will allow the developers to efficiently encode their planning problems without tying themselves to Visual Studio IDE.

Of the qualitative improvements of the system that are planned to be made in the future, the support of the unordered sets of tasks can be noted. This change will increase the number of different planning networks the developers can encode, while also reducing the amount of code necessary for certain planning problems.

And finally, the possibility of transferring the implementation of the planning system to other platforms is considered. Even though the current implementation relies on Unity Coroutines, other ways to check conditions can be implemented, making the system platform independent.

Acknowledgments

This work was carried out as part of the project on state assignment SPIIRAS No. 0073-2019-0003 and with financial support from the Russian Federal Property Fund (grants No. 18-01-00626, No. 20-07-00839).

References

[1] Abramov M., Tulupyeva T., Tulupyev A. *Social engineering attacks: social media and users security estimates.* SUAI, St.Petersburg, 2018 (in Russian).

[2] Azarov, A.A., Tulupyeva, T.V., Suvorova, A.V., Tulupyev, A.L., Abramov, M.V., Usupov R.M. *Social engineering attacks: the problem of analysis.* Nauka Publ., St Petersburg, 2016 (in Russian).

[3] Beckers, K., Pape S. *A Serious Game for Eliciting Social Engineering Security Requirements* 2016 IEEE 24th International Requirements Engineering Conference (RE), Beijing: 16–25, 2016.

[4] Champandard A. *AI Game Development: Synthetic Creatures with Learning and Reactive Behaviors* New Riders, 2003.

[5] *Game AI Pro: Collected Wisdom of Game AI Professionals* Ed. Rabin S. A. K. Peters, Ltd. Natick. MA. USA, 2013.

[6] Georgievski I., Aiello M. *HTN planning: Overview, comparison, and beyond.* Artificial Intelligence, 222: 124–156, 2016.

[7] Kaledina A. *The Central Bank is concerned about the growth of cybercrimes based on methods of social engineering* Izvestia, July 2019, URL: https://iz.ru/897320/anna-kaledina/nu-i-gadzhety-rossiiane-stali-samoi-legkoi-dobychei-dlia-kibermoshennikov (last available: 10.05.2020)(in Russian).

[8] Kelly J., Botea A., Koenig S. *Offline Planning with Hierarchical Task Networks in Video Games.* Proceedings of the 4th Artificial Intelligence and Interactive Digital Entertainment Conference, AIIDE 2008: 60–65, 2008.

[9] Khlobystova A., Abramov M., Tulupyev A. An approach to estimating of criticality of social engineering attacks traces //International Conference on Information Technologies. – Springer, Cham: 446–456, 2019.

[10] Korepanova A., Oliseenko V., Abramov M., Tulupyev A. *Application of Machine Learning Methods in the Task of Identifying User Accounts in Two Social Networks* Computer tools in education, 3: 29–43, 2019.

[11] Kuznetsov O. *Cognitive semantics and artificial intelligence.* Scientific and Technical Information Processing, 40: 269–276, 2013.

[12] Millington I., Funge J. *Artificial intelligence for games* 2nd edition, Morgan Kaufmann Publishers, 2009.

[13] Nau D., Au T., Ilghami O., Kuter U., Wu D., Yaman F., Munoz-Avila H., Murdock J. W. *Applications of shop and shop2* Intelligent Systems, IEEE. 20: 34–41, 2005.

[14] Nau D. S., Au T. C., Ilghami O., Kuter U., Murdock J. W., Wu D., Yaman F. *SHOP2: An HTN Planning System* Journal Of Artificial Intelligence Research, 20: 379–404, 2003.

[15] Newbould, M., Furnell, S. *Playing Safe: A prototype game for raising awareness of social engineering.* In Australian Information Security Management Conference: 24–30, 2009.

[16] Neufeld X., Mostaghim S., Perez-Liebana D. *HTN fighter: Planning in a highly-dynamic game* 2017 9th Computer Science and Electronic Engineering (CEEC). Colchester: 189–194, 2017.

[17] Orkin J. *Three States and a Plan: The A.I. of F.E.A.R.* Game Developers Conference, 2006, URL: https://alumni.media.mit.edu/~jorkin/gdc2006_orkin_jeff_fear.pdf (last available 25.06.2020)

[18] Osipov G. *Artificial Intelligence methods* PHISMATLIT, 2011.

[19] Peltier T. *Social Engineering: Concepts and Solutions.* Information Systems Security, 15: 13–21, 2006.

[20] *Forecasts for the near future.* Positive Research 2019, URL: https://www.ptsecurity.com/upload/corporate/ru-ru/analytics/Positive-Research-2019-rus.pdf (last available: 25.06.2020)(in Russian)

[21] Sun L., Jiao P., Xu K., Yin Q., Zha, Y. *Modified Adversarial Hierarchical Task Network Planning in Real-Time Strategy Games* Applied Sciences, 7(9): 872, 2017.

[22] Tate, A. *Project Planning Using a Hierarchic Non-linear Planner* D.A.I. Research Report

No. 25, Department of Artificial Intelligence, University of Edinburgh, August 1976.
[23] Thompson T. *The AI Of Horizon Zero Dawn (Part 1)* Gamasutra: The Art & Business of Making Games, January 2019, URL: https://www.gamasutra.com/blogs/TommyThompson/20190130/335238/Behind_The_AI_of_Horizon_Zero_Dawn_Part_1.php (last available: 09.05.2020)
[24] *Coroutines* Unity User Manual, 2015, URL: https://docs.unity3d.com/Manual/(last available: 10.05.2020).
[25] Yannakakis G., Togelius J. *Artificial Intelligence and Games* Springer, 2018.