Charles University in Prague
Faculty of Mathematics and Physics

DOCTORAL THESIS

Tomáš PLCH

Believable Decision Making in Large Scale Open World Games for Ambient Characters

Department of Software and Computer Science Education

Supervisor of the doctoral thesis: Mgr. Cyril Brom, Ph. D.

Study program: Computer Science
Specialization: Theoretical Computer Science

Prague 2017

I want to thank everybody who supported me on this quest and I apologize to everybody whom I forgot. First of all, I want to thank my parents for their never-ending support and inspiration. I want to thank my supervisor Cyril for his oversight and guidance for god-knows-how-long. I also want to thank all my friends (Kryštof, Dana, and many more) for cheering me on. All my friends at the faculty with whom I shared the same path to make some small scientific relevance in the world (Michal, Jakub, Martin). I want to thank my colleagues at work whom helped me make this work a reality (Matej, Mikee, Martin, Prokop, Pigi, Viktor, Baz and many more). I want to thank Warhorse Studios (Martin, Viktor, Dan and Petr) for providing me with the opportunity to have my ideas put to work. Also, my thanks goes to scripters and designers (Petr, Michal, Zigi, Dan…) who helped shape my ideas to make them work for actual production.

But my most sincere thanks go to Martina, my soul-mate, for being primary source of strength and raw power to push through this. She cared for me with all her love. Thank you for being at my side.

Thank you

I declare that I carried out this doctoral thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In…....... date............ signature

Název práce: Uvěřitelné rozhodování virtuálních postav ve hrách s velkým otevřeným světem

Autor: Tomáš Plch

Katedra / Ústav: Katedra softwaru a výuky informatiky

Vedoucí doktorské práce: Mgr. Cyril Brom, Ph. D., Katedra softwaru a výuky informatiky

Abstrakt: Hry s velkým otevřeným světem jsou obydleny populacemi s velkým počtem virtuálních bytostí, které se účastní na herním příběhu a obohacují virtuální svět svojí přítomností. Realizmus tohoto prostředí a uvěřitelnost chování jeho obyvatelstva je klíčová pro hráčův imerzivní zážitek. V první části naší práce prezentujeme vylepšení jazyka pro specifikaci chováni virtuálních bytostí tak, aby bylo možné specifikovat komplexní uvěřitelné chování jednoduchou a udržitelnou formou. Specifika jazyka jsme založili na kombinaci principů objektového programování a paradigmatu behaviorálních stromů. V druhé části práce popisujeme koncept Inteligentního prostředí schopného adaptivně reagovat na chování virtuálních bytostí i na hráčovy akce tak, abychom hráči prezentovali uvěřitelný svět. V třetí části specifikujeme koncept Smart konstruktů, které poskytují virtuálním bytostem kontextově správné chování, aby bylo ve specifických oblastech uvěřitelnější. Dále Smart konstrukty poskytují chování spjaté s používáním předmětů a účastí v příběhu hry tak, aby nedocházelo k narušení iluze uvěřitelného světa. V poslední části popisujeme funkci sémantické sítě informací, které umožňují poznávání virtuálního světa za pomocí jednoduchých dotazů. K ověření praktické aplikovatelnosti našich postupů jsme integrovali naši architekturu do velkorozpočtové hry Kingdom Come: Deliverance vyvíjené ve Warhorse Studios.

Klíčová slova: uveřitelné chování, virtuální bytost, otevřený svět, volba akcií, počítačová hra, behaviorální strom

Title: Believable Decision Making in Large Scale Open World Games for Ambient Characters

Author: Tomáš Plch

Department / Institute: Department of Software and Computer Science Education

Supervisor of the doctoral thesis: Mgr. Cyril Brom, Ph. D., Department of Software and Computer Science Education

Abstract: Large scale open worlds for computer games are inhabited by populations of Non-Player Characters (NPC). Believability of these NPCs is key in presenting immersive gameplay to the player. Managing complexity of NPC behaviors is a fundamental game development problem. This thesis is focused on increasing believability of NPCs' behaviors by providing an enhanced language for specifying action selection for these characters. The language is based on the Behavior Tree paradigm combined with object-oriented programming. We introduce our language's mechanisms that enable a developer to create complex, yet maintainable behaviors for individual NPCs. Second, we introduce our mechanism called Intelligent Environment aimed at maintaining a believable game environment able to adapt to player's actions and NPC's behaviors. Thirdly, we present our Smart construct concept which provides NPCs with context relevant behaviors from dedicated behavior containers to employ them when present at locations, using objects, or engaging in the game's quests. Fourthly, we present our semantic network to allow exploration of relations between objects, NPCs and in-game locations by means of predicate based queries. We integrated our architecture into the Kingdom Come: Deliverance computer developed by Warhorse Studios to evaluate the feasibility our approach in a real-life production of a big budget open world computer game.

Keywords: believable behavior, non-player character, open world, actions selection, computer game, behavior tree

Contents

1 Introduction ... 6

1.1 The Uncanny Valley of Believable Intellect ... 7

1.2 Virtual Worlds ... 8

1.3 Virtual Life ... 9

1.4 Thesis Goals ... 9

1.5 Thesis structure ... 12

2 Analysis ... 13

2.1 Computer Game Engine ... 13

2.2 Basic Concepts ... 14

2.2.1 Belief-Desire-Intention ... 15

2.2.2 Action Selection ... 16

2.2.3 Summary ... 20

2.3 Transferable Design Reasoning ... 20

2.3.1 Making a game ... 20

2.3.2 Engineered Virtual Worlds ... 21

2.3.3 Managing Complexity and Context ... 23

2.3.4 Summary ... 25

3 Scenarios ... 26

3.1 Life of Brian ... 26

3.2 Tavern ... 26

3.3 Death in the streets ... 27

3.4 Poisoned Jerry ... 28

3.4.1 Battle ... 28

3.5 Summary ... 29

4 Making Decisions ... 30

4.1 Behavior Tree ... 30

4.2 Stateful Behavior Tree ... 32

4.2.1 Tree Nodes ... 33

4.2.2 Execution and Evaluation ... 34

4.2.3 Budgeting ... 36

4.2.4 Parallelism ... 38

4.2.5 SBT Events ... 40

4.2.6 Data Model 43
4.2.7 Data Storage and Access 46
4.3 Messaging 50
4.4 The Message 51
4.4.1 States 51
4.4.2 Inboxes 52
4.4.3 Timeouts 54
4.4.4 Addressing 54
4.4.5 Processing Schemas 54
4.4.6 Summary 56
4.5 Synchronization 57
4.5.1 Locks 57
4.5.2 Semaphores 58
4.5.3 Barriers 58
4.6 SBT Actions 59
4.6.1 Run an Action 59
4.6.2 Asynchronous execution 61
4.6.3 Synchronized Actions 63
4.6.4 Action Chaining 65
4.6.5 Move and Act 67
4.6.6 Summary 68
4.7 SBT Summary 69
5 Decision Making Mechanism for NPCs 70
5.1 Architecture 71
5.2 Brain 72
5.2.1 Day Plan 72
5.3 SubBrain 74
5.4 The Player 75
5.5 Summary 76
6 Smart World, Intelligent Environment 78
6.1 Motivation 78
6.2 Relevant work 79
6.2.1 Smart Objects 79
6.2.2 Smart Environments 80
6.3 Analysis 80

6.4 Behavior Injection 82
6.5 Intelligent Environment 84
6.5.1 IE Object 84
6.5.2 IE Area 86
6.5.3 IE Mechanism 87
6.5.4 IE Virtual Observers 88
6.6 Smart Constructs 88
6.6.1 Smart Objects 91
6.6.2 Smart Quests 92
6.6.3 Smart Navigation Objects 92
6.6.4 Smart Areas 92
6.6.5 Summary 96
7 Knowledge network 98
7.1 Relation Knowledge Network 98
7.2 Static, Dynamic and Virtual Links 99
7.3 RKN Query 100
7.4 Query Predicate 102
7.4.1 Filter 102
7.4.2 Logical Operator 102
7.4.3 Analyzers 102
7.4.4 Sub-queries 103
7.5 Adding Dimension 103
7.6 Search Mechanism 104
7.7 Example 105
7.8 Scenario 1 – Life of Brian 105
7.8.1 Going to work 106
7.8.2 Having some fun 106
7.8.3 Player steals from the neighbor 106
7.9 Summary 107
8 Evaluation 109
8.1 SBT Use Evaluation 109
8.1.1 Qualitative Evaluation of MBTs 109
8.1.2 Quantitative Evaluation of MBTs 110
8.2 Comparing SA and SBT concepts 111
8.2.1 Results 112

8.3 Qualitative Evaluation ... 113
8.3.1 KCD Integration and Deployment ... 114
8.3.2 Personal Feedback via Interview ... 115
8.4 Industrial Deployment Evaluation ... 118
8.4.1 Frame Time ... 119
8.4.2 Overall numbers ... 122
8.4.3 Messaging ... 123
8.4.4 Intelligent Environment and NPCs ... 125
8.4.5 Updating SBTs ... 128
8.4.6 RKN queries ... 130
9 Summary ... 132
9.1 Stateful Behavior Tree Language ... 133
9.2 Three-Tier Deliberation Architecture ... 134
9.3 Intelligent Environment ... 134
9.4 Smart Constructs ... 135
9.5 Semantic Network ... 135
9.6 Evaluation and Integration ... 135
10 Future work ... 137
11 Conclusion ... 138
12 Bibliography ... 139
List of Figures ... 148
List of Tables ... 153
List of Abbreviations ... 154
List of Author’s Publications ... 155
_Publications relevant to this thesis ... 155
_Other publications ... 155
Appendix A – Attached Digital Content ... 158
Appendix B – Video commentary ... 159

1 Introduction

The evolution of computer games went hand in hand with the evolution of computer hardware. At first, only a handful of selected individuals could create and play games on expensive equipment mainly intended for scientific or military applications. As time progressed, Moore's Law (Moore, 1965) effect provided the common public with better and cheaper hardware and computer games took deep roots in our society. Known and loved classics like the game of Pong (1973) (Sellers, 2001) and Pac Man (1980) (Kent, 2001) are part our cultural heritage of the late 20th century.

With the progression of the Internet and more powerful hardware, evolution of games took a more global turn. The culture moved from single and split-screen played games to online gaming. At the dawn of the 21th century, games like World of Warcraft (2004) (Entertainment, Blizzard, 2004) started a social gaming revolution. Players could access gigantic virtual worlds, socialize with other players, play together against each other or kill creatures populating the environment's dungeons. Players enjoyed a complex in-game life of their virtual characters and do identify with them on some level. The role-playing aspect was ever more intrusive into every game genre, from Role Playing Games (RPG) to First Person Shooters (FPS) and Real Time Strategy (RTS) games.

Virtual worlds inhabited by both players and *Non-Player Characters* (NPC) have gotten bigger and more complicated with every new game. However, due to the ever slowing progression of visual fidelity, user focus has shifted to a more delicate topic of virtual world's believability (Umarov & Mozgovoy, 2012). Believable worlds with believably behaving entities tend to mimic the real world as much as possible to provide the player with the highest degree of immersion. It is an intricate illusion of intelligence which tricks the player into asking whether NPCs are exhibiting cognitive properties associated with humans – thinking for themselves to achieve their goals and desires in a deliberate way.

1.1 The Uncanny Valley of Believable Intellect

The Uncanny Valley Theory (Mori, MacDorman, & Kageki, 2012) postulates that when human-like artificial life is presented to a human observer the affection for it is increasing with the replica's ability to imitate human traits and properties. However, there is a major dip the observer's affinity called *Uncanny Valley* (Figure 1). This effect is further increased by reducing the consistency of the presented imitation (MacDorman & Chattopadhyay, 2016). There are different theories which have proposed explanations of this phenomenon like mate selection (Green, MacDorman, Chin-Chang, & Vasudevan, 2008) or pathogen avoidance (Craig, 2012). The common theme is the fact that humans are very sensitive about imperfections and subtle differences which are perceived at a subconscious level.

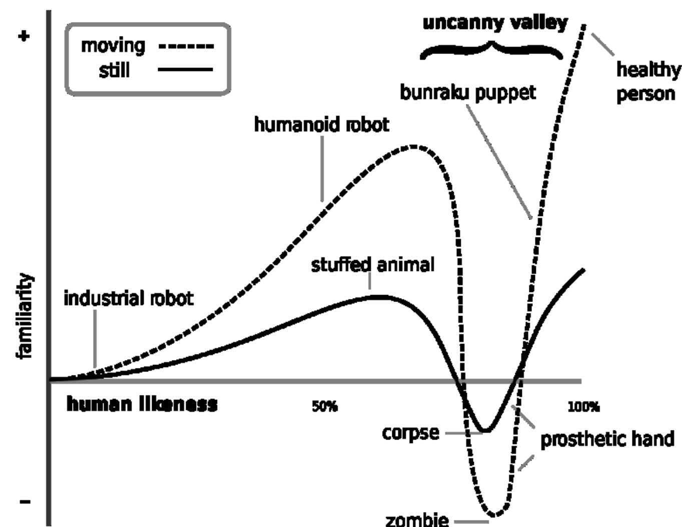

Figure 1: The Uncanny Valley diagram shows how human resentment is more intense with increase in human-like appearance (© MacDorman, 2005)

The theory is often applied to androids and humanized robots, but it also applies to animated virtual characters in movies and computer games (Tinwell, 2014). Since computer games are not limited to the physical capabilities of artificial skin and mechanical actuators (e.g. small servos to move an android's eyes), they can provide a more complex virtual representation that is more ahead into the Uncanny Valley.

In respect to computer games, a lot has changed over recent years. Polygon count of both environment and characters has increased dramatically – from few hundred to several thousand polygons per single character (Figure 2). Animation have advanced from simple sprite sequences to skeleton based animations with character skinning and muscle simulation (Geijtenbeek, van de Panne, & van der Stappen, 2013). Nowadays, characters utilize complex facial expressions to convey emotions and internal state of mind. Both in real-time (e.g. computer games) and offline rendered (e.g. movies) applications, the depth of the visual fidelity has increased almost exponentially.

Figure 2: Soldier NPC from the Half Life game series (Valve & Gearbox, 1998-2007). On the left a low polygon model with few animations (e.g. walking, kneeling, jumping). On the right a high polygon character with a large animation repository (e.g. jumping over obstacles, leaning etc.) (© Valve Software)

However, visual fidelity is only one part of the problem (Tinwella, Grimshawa, Nabib, & Williamsa, 2011). If characters are behaving in an unnatural or unrealistic way, it breaks the illusion and provides additional inconsistencies which throw the observer deeper into the Uncanny Valley. Therefore, the concept of believable behavior (Loyall, 1997) is key in getting games with amazing visuals more accepted by the players.

1.2 Virtual Worlds

Imagine virtual worlds – visually appealing, interactive, full of wonders, but lifeless. Nobody to talk to, nothing to see moving except leafs on trees and grass on meadows. Wandering such worlds would be boring after a while. Games are as much about interaction as they are about looks and mechanics. What would be a story without actors, a quest without anybody to give it, a dungeon without nothing to guard its treasures.

Imagine these worlds being populated by virtual life, virtual humans and animals, able to provide sense of meaning. Imagine actors for the stories and ambient life for the world to watch. However, if these virtual inhabitants would be hollow non-interactive husks, they would not be more than just better skinned set of moving statues.

Imagine believable acting, where reason, hidden or evident, might be suspected. Imagine a world where the player is stalked by the question if other characters are played by humans or not. A world where virtual humans go about their jobs, have hobbies, feuds with their neighbors or cheat on each other. Such an intriguing world might be fun to explore, to watch and live its stories.

Therefore, a virtual world is more than just a sum of all the graphical assets and animations. The true illusion of a virtual environment is the illusion of artificial life with its own principles, its own mechanisms and its own reasons. A successful

virtual world is the one that can capture the imagination of the player on what can be explored and what can happen and keep the consistency of what happens in the world, to avoid confusion.

1.3 Virtual Life

Large scale virtual worlds are commonly populated by both human players and intelligent NPCs. Where players follow their own game related agenda (i.e. quests in a storyline), NPCs perceive the fulfillment of their own goals (Russell & Norvig, 2009) and tasks. The agenda may be either a) *story bound*, or b) *ambient.* Story bound agenda is often related to activities in respect to the player's game agenda, where an NPC helps or opposes the player. There is little to no room to deviate from such predesigned paths, since game design tends to avoid situations which could break the game. On the contrary, an NPC's ambient agenda denotes the autonomous function of the NPC within the virtual world – *Virtual Life* (Figure 3).

Figure 3: Virtual life of a simple farmer in Kingdom Come: Deliverance (© Warhorse Studios 2017)

Virtual Life can be summarized as a set of available actions (e.g. go to sleep, go to work etc.) and given goals (e.g. have something to eat, chat with friends from time to time) to achieve on a day-by-day basis. A common peasant NPC at home in a village is the most generic example available. The NPC is not particularly interested in the player's quest and goes to work day-by-day and tends to have fun at the village's tavern from time to time. The believability of such ambient NPCs adds to the overall believability of the virtual world. If the NPC would be a simple automaton walking around the house all day, without any context specific reactions (e.g. it rains so avoid work outside), it would break the overall immersion.

1.4 Thesis Goals

The gaming industry mantra »visuals sell« implies that very limited time and effort is invested into creating complex virtual worlds inhabited with NPCs capable of believable human-like reasoning in everyday life aspects. It is much more desired to

utilize computing power and developers to facilitate a good-looking game, instead of complex and rich environments with emergent gameplay. Due to pressing technological limitations of today's hardware, visuals tend to advance at a slower pace, leaving room for more focus on believability of virtual worlds and their virtual life. In other words, it is ever harder to make the game look better, but they can be made more immersive and fun to play.

Our thesis focuses on RPGs which manifest within large scale virtual worlds facilitating a complex real time simulation of Virtual Life. We aim at providing a framework for creating and maintaining a persistent and believable NPC population which imitates a real-life environment with complex relations between participants and their believable reactions to emerging situations. The notion of behavior believability is imperative to our work.

Primarily we focus on six areas:

Goal 1) *Believable Ambient Environment* denotes the essence of Virtual Life within the large scale world – day-by-day life of NPCs which exist on their own within the world, living their virtual lives. We aim at producing a rich and diverse environment where NPCs have actual deliberation and purpose about their actions. Our focus is to provide an NPC architecture and mechanisms which allow for constructing day-by-day routines in a diverse and effortless way.

Goal 2) *Emergent Situations* may arise when something unexpected happens and NPCs need to deal with it in a reasonable and context specific manner. Fixing a broken cart wheel or investigating a murder should not be a big issue for both NPCs and the designers as well. Our goal is to provide mechanisms to create such situations easily in a controlled manner to maintain a proper situation context to participating NPCs.

Goal 3) *Story Driven Environment* is present in most RPGs, since having only an open world tends to get boring rather quickly. We aim at making the ambient environment able to adapt to a player centric narrative in a seamless way without disturbing their day-by-day lives

Goal 4) *Behavior Depth and Complexity* is necessary to facilitate believable human-like behaviors. We focus on providing mechanisms which can be utilized to create NPC relations with diverse interactions and communication. We want to provide NPCs with a more diverse rooster of actions without compromising the simplicity of the NPCs design.

Goal 5) *Large Scale Deployment* of NPCs is necessary to maintain an illusion of a vibrant and life-filled open world. Our architecture's goal is to be deployable in a large scale setting with hundreds of NPCs spread over a large landscape.

In summary, we aim at providing a complete solution to a large scal*e* open world simulation for hundreds of NPCs within a story driven game. Our main goal is to provide a framework capable of satisfying all the above described areas of interest in production grade quality.

To facilitate the above specified goals, we propose a set of mechanisms which we will investigate in our thesis in detail. These mechanisms will be implemented as an Artificial Intelligence Framework for the Kingdom Come: Deliverance (KCD) (Warhorse Studios, 2017) computer game, developed at Warhorse Studios. We have identified the following mechanisms:

1) *Stateful Behavior Tree Language* – we will discuss details of our extended behavior description language called Stateful Behavior Tree (SBT) language, which is aimed at providing a more complex deliberation mechanism than the commonly used scripting or Behavior Tree languages. We focus on behavior decomposition into blocks similar to function calls present in a programming language. This mechanism is primarily aimed at being the foundation to all our goals, since it is our way to express complex behavior patterns for a believable virtual world. Our primary focus is on Goal 1 and 4, where the expressiveness and complexity of the language are key to deliver believable behaving NPCs.

2) *Three Tier Deliberation Architecture* – we will discuss details of our deliberation architecture where high-level decisions are passed down to lower level decision making and action execution. We implemented it to be able to manage issues at Goal 2, 3, and 4 where adaptation to the situation's context is key and behavior decomposition is necessary to provide a maintainable overall solution.

3) *Intelligent Environment* – we propose to imbue objects, areas and other entities present in the virtual world with intelligence to provide more complex and context sensitive environment's behavior. It allows us to provide a context relevant reaction to emerging situations addressed by Goal 2. In respect to Goal 3, the Intelligent Environment enables us to provide custom tailored behaviors to player actions in respect to quests and the storyline. The required scale of the solution addressed in Goal 4 implies the necessity to decompose behaviors into maintainable partitions, which we intend to spread into the environment to avoid monolithic constructions which are hard to maintain.

4) *Smart constructs* – we propose to allow objects, areas and other entities to be a source of behaviors for any intelligent entity within the virtual world, displacing context and relevant (e.g. to an object, area or situation) behaviors for intelligent objects to make use of. This mechanic focuses on Goal 2, 4, and 5 where providing context relevant behaviors is key. This allows us to maintain simple and generic NPC behavior and spreading context relevant behaviors into the environment for use by NPCs. At Goal 2, emerging situations are a prime candidate to be the source of context relevant behaviors which provide more depth and complexity to NPCs required at Goal 4. This concept also provides a generic solution to the need to introduce new content, objects and mechanisms to a large-scale game (Goal 5). Overall, having context relevant behaving NPCs is also relevant to Goal 1.

5) *Semantic network* – we propose a graph based knowledge representation within the virtual world to allow for more natural acquisition of information about the environment and its capabilities. This is aimed at providing a supportive mechanism for Smart constructs (e.g. what can be done with an object) and the SBT language, to allow to query and modify semantics and relations within the virtual world (e.g. where do I like to go to have fun). It also provides a mechanism for the high-level deliberation (Goal 3) of an NPC to reason about the virtual world's dependencies (e.g. what is in my house is mine). Since a semantic network is fairly easy to

visualize, thus it is necessary for a large-scale deployment (Goal 5) of our architecture in a complex virtual world. The semantics introduced into the network (e.g. somebody who is not my friend present at my house at night is a threat) allows for a much more believable ambient behavior of NPCs aimed at by Goal 1.

In summary, all the above presented mechanisms are key to facilitate our goals as a whole, since they either support or complement each other (Figure 4).

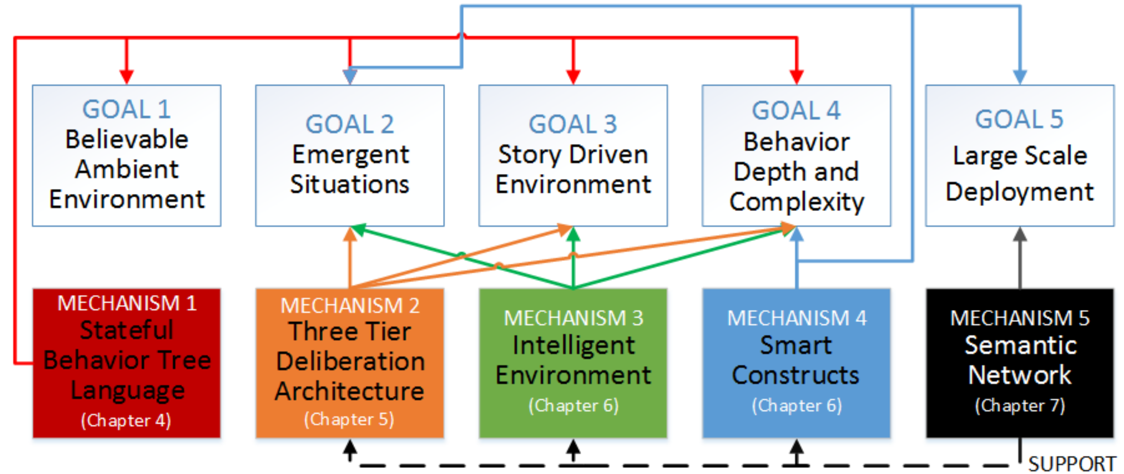

Figure 4: Dependency between Goals and Mechanisms

1.5 Thesis structure

We structure our thesis into chapters addressing the outlined issues in sequence. In Chapter 2, we focus on our analysis of the target environment (i.e. computer game), we present relevant work and setup fundaments for our thesis. In Chapter 3, we present a set of scenarios we used to develop the presented approaches[1]. In Chapter 4, we present our Stateful Behavior Tree language for describing behaviors. In Chapter 5, we present out Three Tier Deliberation Architecture. In Chapter 6 we present our approach on decomposing behaviors into our Intelligent Environment and Smart constructs. In Chapter 7 we present our Semantic Network used to access information in the world to convey behavior decomposition and logical inferences about the world. In Chapter 8, we present our evaluation of the presented mechanisms. Chapter 9 we summarize our thesis in respect to our goals. In Chapter 10, we discuss future work and Chapter 11 concludes our thesis.

[1] The presented scenarios are part of the KCD game proving the feasibility of our industrial integration.

2 Analysis

Our thesis focuses on enhancing Virtual Life within a large scale open world to make it more believable and thus more enjoyable for the player. RPGs are about player's character progression and often have a story to follow. We consider the ambient world and its population to be the key component in building such environment for the player to explore and enjoy.

In this chapter, we establish the basic concepts we build our architecture upon. We also address the relevant work influencing our approach in respect to action selection and deliberation.

2.1 Computer Game Engine

Computer Game Engine (CGE) represents the environment which hosts all the subsystems that make out a game (e.g. rendering, Artificial Intelligence, sound system etc.). It encompasses core mechanisms which may be required by other subsystems. Simply put, the CGE runs the game providing the overall architecture other subsystems respect.

There are many ways on how to design a CGE (Gregory, 2014), but one key conceptual trait has been present in most designs so far – *the endless game loop* (also called *frame loop*). The virtual world advances at discrete steps (i.e. frames), where inputs are collected, the world is transformed based on accumulated time difference from last frame (e.g. NPCs move in their current direction at their present speed), and all is rendered as soon as possible.

Since the game loop can be understood as the perpetual presentation of the virtual environment, it is frequent practice to present a framerate above the human perception threshold (i.e. 24 Frames Per Second (FPS) in movies) to keep the illusion of a continuous non-discrete runtime. Rates of 30 or 60 FPS are common target values for present game engines like Unreal Engine (Epic Games, 1998) and CryEngine (Crytek, 2002).

There are two key issues game developers tend to avoid – *frame rate drops* and *varying delta differences.* Rendering less frames per second than desired is called a frame rate drop. It may occur if some time-consuming action takes place within the CGE and as result, the virtual world advances at perceivably more discrete manner, thus losing the real-time feeling. It is similar to pausing and skipping over parts of a movie. Obvious it is not desired and has to be avoided at all cost by all subsystems. Varying delta difference issue is manifested if the difference between frames varies significantly and the overall feeling from moving objects is that they travel at varying speeds and the overall feeling of dynamic scenes is not good.

Therefore, one key feature of any component responsible for facilitating the Virtual Life's mechanisms must be designed so that it only consumes as much computational resources so that the frame loop is not hindered beyond the designated frame rate. Also, it is necessary that utilization of resources will be spread even to avoid frame rate fluctuations. It would be desirable to run all computations in parallel (i.e. on another thread) to other subsystems, so most of the frame time can be utilized. It is noteworthy that if target frame rate is 30 FPS, the overall time for one frame is not more than 30ms to advance the simulation to the next frame.

2.2 Basic Concepts

Open World Games (OWGs) can be abstracted as virtual environments that provide the player with the maximum available freedom in respect to both movement and actions. Simply put, the player should be able to go anywhere and do anything game mechanics and level design allows. A large scale OWG is presented to the player as a continuous world where everything happens simultaneously. Commonly such games cover a substantial virtual landscape, ranging up to hundreds of square kilometers.

Therefore we can abstract the large scale OWG as a system defined at any time by a *configuration* (CFG), which represents a set of *facts* valid within the world $\{F_1, ...F_n\}$. A fact can be seen as a logical predicate which holds true – e.g. »Frank is at Work«, »Hammer belongs to John«. When an *action* (a) is applied to the world, it changes the configuration of the world over time from one configuration to another $a(OWG,t): C_1 \rightarrow C_2$. Simply put, actions transform the world. We assume that when actions are applied, their effects can be serialized.

Every NPC has a given set of *available actions* (aa) $\{A_1, ...A_n\}$ which can be invoked. The contents of the set depend on the present OWG's CFG. Therefore, available actions are context specific (e.g. a sitting NPC has a separate set of actions than a standing NPC) in respect to the NPC and the environment. The player is similarly equipped with a set of available actions. Practically, actions may range from simple animations (e.g. picking up an objects) to complex constructs like »Find the Player«.

Further, every NPC has a set of desires and goals. These can be abstracted as a set of desirable configurations $\{CFG_1, ...CFG_n\}$ or as an evaluation function $fH:CFG \rightarrow Happiness$. Simply put, the NPC tends to execute actions which lead to accomplishing goals (de Silva, Sardina, & Padgham, 2009) (e.g. going to work, going to have fun). The overall goal of an NPC is to maximize the fH over time.

To achieve this, the NPC facilitates *Action Selection* (AS), which chooses the next action to execute based on the current configuration and value of fH. Therefore, the action selection can be abstracted as follows $AS: \max\{fH(a(CFG)); a \in aa\}$.

The action selection runs within a *Decision-Making Mechanism* (DMM), which is responsible for collecting CFGs, running AS and committing to the chosen actions. The DMM can provide additional interpretation of CFGs into an intermediate representation for a simpler AS processing. A DMM can also be responsible for information aggregation over time, to translate sets of CFGs into perception (e.g. slow moving NPCs are perceived as potential targets for the AS to choose to shoot them).

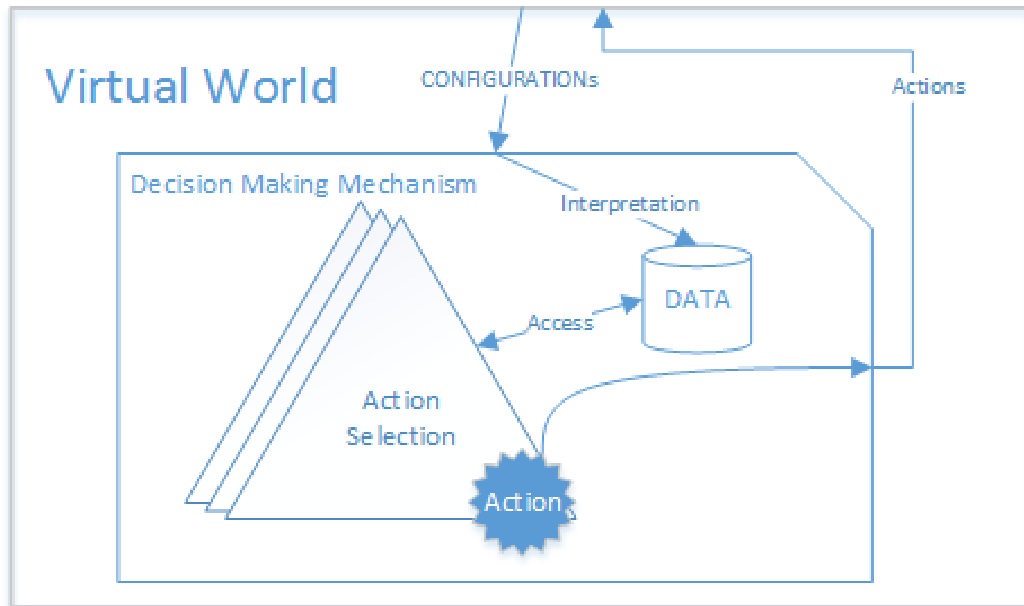

Figure 5: The DMM contains the Action Selection process which is queried about the next suitable action in respect to the constructed context of the environment's configuration. The produced action is committed to the environment afterwards.

2.2.1 Belief-Desire-Intention

The Belief-Desire-Intention (BDI) (Bratman, 1999) model is used to develop intelligent agents by separating the representation of the environment as an agent's informational state (*belief*), the motivational state to achieve objectives or situations (*desire*) and deliberative state (*intention*). In our case, the agent coincides with the NPC.

The *belief base* represents the NPC's perception of the virtual world's configuration. It is a collection of beliefs and inference rules which can be used to utilize forward chaining to create new beliefs. The notion of belief represents the NPC's interpretation of the observed world's configuration thus, it may not be accurate.

The desire conceptually represents the motivation of the NPC to achieve a set beliefs – e.g. »have shiny armor« or »find the player«. In respect to our prior formalism, the desire is the set of perceived configurations or the maximization of the `fH` function. If the NPC commits to a subset of desires, it adopts a *goal* to pursue.

The intention represents the agent's commitment to a goal, thus to commit resources to achieve it, i.e. to utilize the underlying action selection to reach the chosen intent. The action selection produces actions the NPC commits to – a *plan*. The NPC's plan represents a sequence of actions which when executed achieves one or more intentions. Plans can be nested, where one plan includes other plans.

The BDI also introduces *events* which are either utilized to trigger a reactive response from the NPC or to inform the belief system about changes in the environment.

As can be seen, the BDI model is a high-level abstraction of intellectual decomposition within a virtual agent's mind. It is often used in conjunction with action selection mechanisms, both reactive (Kaminka & Frenkel, 2005) and deliberative (Sardina, de Silva, & Lin, 2006), to produce agents for various purposes,

not limited to entertainment industry (Kaminka, Yakir, Erusalimchik, & Cohen-Nov, 2007).

2.2.1.1 Summary

We utilized the notion of BDI to design our three-tier architecture of NPC's DMM (Mechanism 2 – Three tier deliberation architecture). In principle, the notion of belief manifests as DMM's internal data structures which are modified based on the NPC's perception layer as well as other processing mechanisms (e.g. RPG updates of the NPC's stats like health, stamina etc.). These data structures are accessible to the underlying AS mechanisms which reside within the DMM. The notion of a desire is addressed as a high-level plan provided by the highest tier of our architecture. Further, the commitment to a goal provided by high-level planning is addressed at lower levels by selecting a dedicated AS which is responsible for choosing proper actions based on emerging contexts (e.g. the NPC wants to do some work (desire), however it rains outside (belief and perception), thus it chooses an indoor activity and respective actions (intention) like cleaning up, working on repairing clothes etc.).

2.2.2 Action Selection

As stated earlier, an OWG environment can be abstracted as a CFG of facts valid at a given point in time. An action transforms the CFG from a present state to a new set of valid facts. Therefore, the problem of action selection can be abstracted as a search within the search space (Russell & Norvig, 2009) of all available configurations. The goal of the search is either to reach a specific configuration or at least maximize the `fH` function. Conceptually, searching for the best next action can be visualized as finding a route over a changing landscape of CFGs. We distinguish two basic approaches to AS:

a) *Reactive action selection* is primarily focused on providing a decision about the next action in a timely fashion, mostly due to either lack of computational resources or the overall complexity of the reasoning in question (i.e. search space of possible choices may be too large). This approach utilizes none or a short lookahead in respect to further choices and anticipated changes in the CFG landscape. This means the AS only maximizes the `fH` function in a local fashion. For example, the NPC can choose attack or defend actions based on current NPC's and enemy's health ratio.

b) *Deliberative action selection* is focused on providing an action in respect to a lookahead considering what may happen or which actions may be chosen next (i.e. planning ahead). The deliberative approach is often more computational expensive, since it explores the search space of possible actions. For example, if the NPC is planning how to acquire food, it may consider various options like »stealing«, »doing work for food« or »selling unwanted items to get money«.

2.2.2.1 Reactive Action Selection

Reactive Action Selection covers a wide range of possible mechanisms on how to approach the deliberation about the next action. The following are to our knowledge commonly employed for NPCs within computer games:

1) *Finite State Machines;*

2) *Behavior Trees;*
3) *Scripts in an interpreted procedural language.*

Finite State Machines (FSM) (Fu & Houlette, 2004) are a simple and easy to use concept for AS for sole purpose NPCs (Isla, 2005) like soldiers, guards etc. It is easy to implement and is often paired with other AS or organized in a hierarchical manner (Girault, Lee, & Lee, 1999). In principle, a simple FSM AS can be defined as a tuple $\{S, s_0, T, ASF\}$, where S denotes the set $\{S_1, ...S_n\}$ of available states for the FSM to enter. The s_0 denotes the initial state the FSM start within and T is a set $\{T_1, ...T_n\}$ of available transitions between states. A transition is triggered by DMM relevant conditions (e.g. being under fire). The ASF represents a set of action selection functions, which map the FSM's current state and DMM context to an action from the NPC's action set (e.g. IDLE state means »sit down« in the forest, however it means »lie down« at home). The actual ASF is made in respect to the current state the FSM is at (Figure 6).

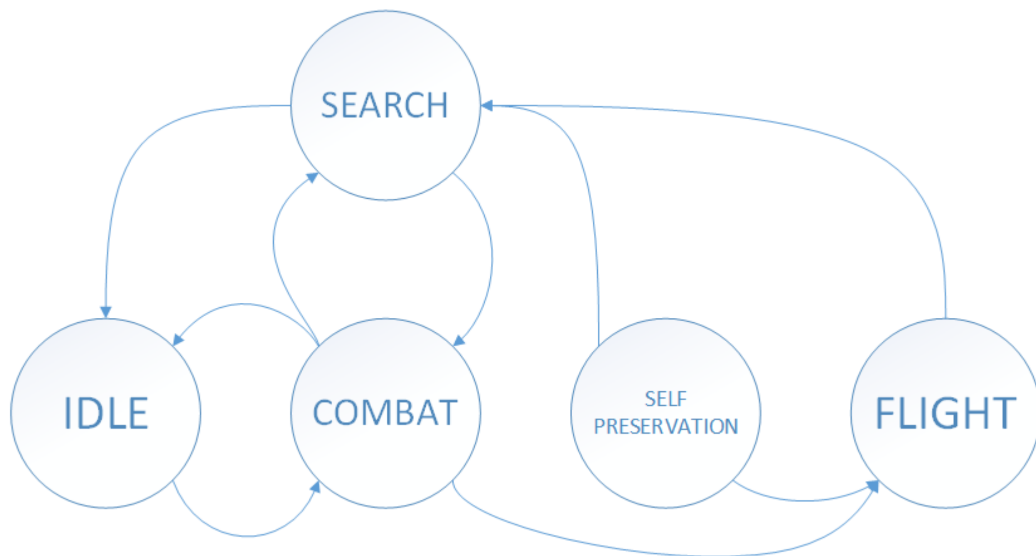

Figure 6: Simple FSM for NPC with a limited set of states and transitions focusing on four principal areas of function – Idling, Combat, Search, Self-Preservation and Flight. NPC has been used primarily as combat opponents

The set of ASF functions can be represented by other AS mechanisms, which will choose appropriate actions in respect to a hierarchical decomposition. For example, when being idle at full health has a different outcome than idling at low health. To our knowledge, FSM are employed in a variety of games (e.g. ARMA (Bohemia Interactive, 2006) game series uses a hierarchical FSM variant).

However, utilizing FSM as the main AS can be difficult over time, since a new state is required for almost any new game mechanic introduced into the NPC's repertoire (e.g. introducing flight capability to an NPC). Therefore, FSM tend to grow and are hard to maintain (Champandard, 2007.2).

Behavior Trees (BT) (Champandard, 2007) are a well-established reactive AS mechanism, which is built around the notion of representing the AS process in a tree-like structure (Figure 7). In principle, the tree is traversed during an DMM update to determine the next action. Non-leaf nodes are steering the selection process until a leaf node is reached. Commonly, leaf nodes represent actions or trigger another nested action selection mechanism (e.g. a scripting language call) which in return

chooses an action. Behavior Trees can also be viewed as an extension of Selection Trees (Quinlan, 1987).

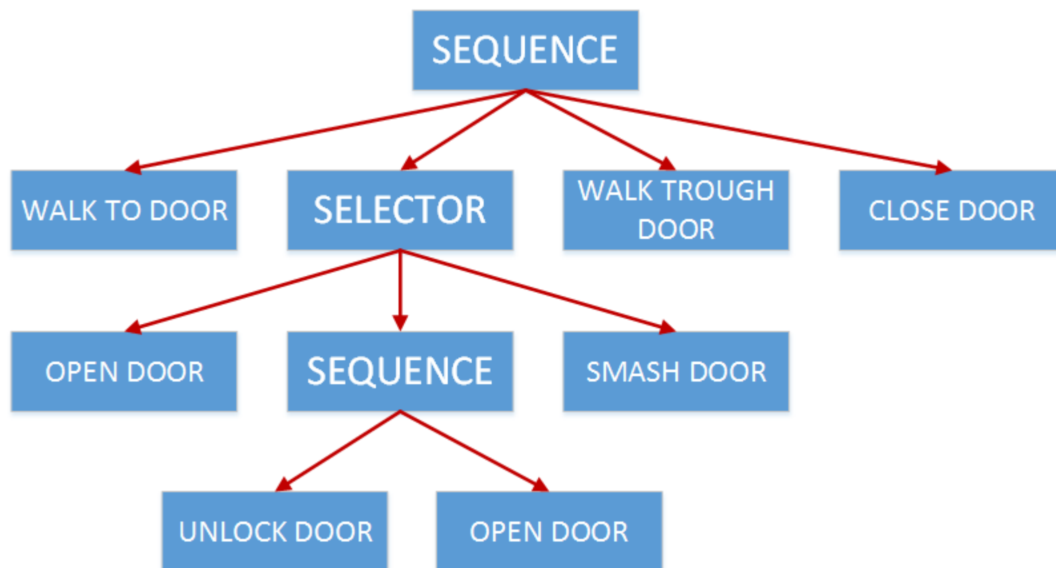

Figure 7: A simple Behavior Tree concerned with choosing the proper action to go through a door which may be closed

Another common paradigm is the *Behavior Oriented Design* (BOD) (Bryson, 2001) which represents a complete methodology on how to create agent's behavior based on behavior decomposition. BOD *Reactive Plans* perform action selection trough behavior arbitration – choosing the most suitable to satisfy a goal. The Reactive Plan decomposes into *action patterns*, *competences* and *drive collections*. Action patterns are simple sequences of low level actions (e.g. get → hold → throw). Competences satisfy goals and are priority ordered pairs of preconditions and either action patterns, competences or marking of the given goal being satisfied. Higher priority preconditions override lower priority preconditions, thus if a higher priority element fires (i.e. the precondition is activated), it receives control. Further, drive collections are perpetually checked collections of preconditions and goal triggers. These pairs are also priority ordered, where higher priority active goal receives control from a lower priority one. The BOD methodology allows to decompose agent's action selection in a hierarchical and structured manner, which is relatively easy to maintain and use (Brom, Gemrot, Bída, Burkert, S., & Bryson, 2006). We also did address issues with the BOD in our previous work (Plch, 2009).

Implementation details at the lowest level vary considerably. Interpreted procedural languages like LUA (Schuytema & Manyen, 2005) are often employed at the lowest level of action selection. Most interpreted procedural languages are like their compiled counterparts C++ or Java. However, their use heavily depends on the integrated solution and may vary from instance to instance. However, they are easy to comprehend but tend to lack expressiveness or efficiency.

2.2.2.2 Deliberative Action Selection

The deliberative approach is commonly referred to as *planning ahead*. Action selection is aimed at providing the next best action based on lookahead steering action selection towards achieving a goal. Where reactive approach is more concerned with local based maximization of the `fH` function, a deliberative approach

is aimed at maximizing the fH function over time, avoiding bad decisions. There are two major approaches a deliberative AS can produce either a*) single actions*, or b) *action plans*.

Single action deliberation choose a single action and the choice of the action is evaluated in respect to maximizing the fH over time and in respect to possible follow-up actions. However, the deliberation is always set to start anew if an action is required. This approach is often utilized in complex but dynamic domains, like combat, where the situation may change rapidly and short sighted reactive approaches may not provide satisfactory results. Often utilized techniques is Alfa-Beta pruning (Knuth & Moore, 1975). However, since even small domains tend to have an exponentially growing search space, the search is coupled with Monte Carlo (Ward & Cowling, 2009) techniques to create mechanisms that tend to evaluate the situation in both short term and long term perspective (Cerny, 2016).

Creating action plans revolves around the idea that if a plan is good enough, it can lead you step-by-step to your goal. Thus such approach focuses on creating a set of actions $\{A_1, ...A_n\}$ called a *plan* (p) which when applied to the environment in a consecutive fashion, will either maximize fH or reach the goal configuration: $p(CFG): A_n(...(A_1(CFG)))$. The produced plan is executed at the DMM step by step. The production of a plan is commonly referred to as *planning* (Ghallab, Nau, & Traverso, 2004) where various methodologies have been developed to search for either optimal or suboptimal plans within both small (Bartak, Brom, Cerny, & Gemrot, 2013) or large (Kolombo & Barták, 2014) domains.

There are many ways on how to search for a proper plan. Most commonly referred mechanisms are *forward* and *backward chaining* (Feigenbaum, McCorduck, & Nii, 1988) (Kapoor & Bahl, 2016). In principle, chaining refers to putting actions into a plan in a fashion that their results can enable the next action to be used properly. Since actions transform the world from one valid CFG to another, putting an action into the plan either disables or enables other actions to be put after it. Forward chaining aims at choosing the next action in sequence of the created plan, thus searching the domain space in a front-to-goal fashion. This approach is suitable for environments with dynamic situations (Hayes-Roth, Waterman, & Lenat, 1983). Backward chaining works in the opposite direction, when the target configuration is the starting point and actions are put into the plan in reverse order.

2.2.2.3 Goal Oriented Action Planning

The Goal Oriented Action Planning (GOAP) (Orkin, 2006) system integrated a modified classical planning paradigm STRIPS (Fikes & Nilsson, 1971) into a real-time environment (Orkin, 2005) of the F.E.A.R computer game (Monolith Productions, 2005) to avoid the complexity of a FSM for NPC action selection. The principle of GOAP is to create sequences of actions to satisfy a given goal. Actions are defined as a tuple (*cost, preconditions, effects*). The actions cost is an arbitrary cost of the action, representing a metric of either how time consuming or complex the action is. Preconditions specify the necessary states to be valid within the world (e.g. `atHome && haveMoney`). This can be either specified as enumeration of states or as logical predicates. When an action is executed, *effects* represent changes in the world's state, both positive and negative ones. This representation is common in the classical planning literature (Ghallab, Nau, & Traverso, 2004).

The actual planning for GOAP utilizes a modified STRIPS approach with cost of actions in mind, minimizing the aggregated value. Further the GOAP planner utilizes

an optimized representation of world state modifications by the investigated actions. One key difference is the introduction of *procedural preconditions* and *procedural effects* which represent more complex reasoning about the world's state (e.g. does a path from A to B exist) and the complex effects (e.g. run to a specific location).

Overall the GOAP system has proven that a planning can be used for action selection within a real-time environment (i.e. computer game). Other practical experiments in this area followed suit (Bartak, Brom, Cerny, & Gemrot, 2013) as well as tools to be utilized (Plch, Chomut, Brom, & Bartak, 2012).

2.2.3 Summary

The BOD methodology (Bryson, 2001) and our previous work (Plch, 2009) represent the foundation of our architecture. We utilize the notion of behavior decomposition into smaller more maintainable sub-behaviors. However, due to the fact that the BOD methodology less favored than common BT or imperative languages (C# or LUA), we combine the BT notation with the conceptual structure of a common imperative language (Mechanism 1 – SBT language) to provide a deployable solution. We utilize the techniques of reactive action selection to provide actions in a timely fashion in respect to a low-level context specific action selection (e.g. AS for day-by-day work differs from combat AS). We also build our NPC architecture to accommodate high level deliberative mechanism to create plans for the NPC to manage goals to manifest a believable virtual life (Mechanism 2 – Three tier deliberation architecture).

2.3 Transferable Design Reasoning

The *Transferable Design Reasoning* (TRD) is the notion of structuring the virtual world and its mechanisms so integrating new concepts and mechanisms does not require changing already present code or script. Design of a computer game's virtual environment can be seen as an engineering problem where new requirements are being introduced during development, thus requiring the underlying architecture to be adaptive and generic as much as possible.

TDR reflects the necessity to transfer designer's initial reasoning about features into the virtual world without triggering a major issue in the virtual world's conceptual architecture or practical implementation.

2.3.1 Making a game

Creating a computer game requires various professions – programmers, designers, screen writers, scripters, animators and many more. These professions can be split into several groups, based on how they contribute to the production's progress. At Warhorse Studios, we identified 1) *designers*, 2) *developers*, and 3) *content creators* as key participating groups.

Designers are the source of ideas about what should be in the game, producing proposals and requirements about what content should be present and how should it work. For example, a designer proposes a dice gambling mini-game between the player and NPCs which is introduced as a part the main story line. Beyond the mini-game's mechanics, there are constrains on which NPCs can participate and how to solve certain use-cases (e.g. cheating, lack of money etc.).

The design requirements are processed by developers (e.g. programmers, scripters) and integrated into the CGE. The actual integration can go beyond the initial design to accommodate the necessary mechanisms for the mini-game (e.g.

NPCs may require a subsystem tracking their gains and losses). Scripters develop subsystems for the NPC's DMM to utilize the new available features. The script subsystems may vary, ranging from decision about actual mini-game strategies to high level decisions about who to play with (e.g. if the NPC is poor, it would be unlikely to play at all).

The final integration into the game is done by content creators – animators, level designers, level scripters etc. From an NPC's perspective, the integration comprises of utilizing mechanisms already present for NPCs (e.g. everybody at the local tavern plays the dice mini-game). This may trigger further development of features for proper ambient integration (e.g. NPCs cheer when a game nearby has high stakes involved).

The overall process of developing a game can be summarized in iterations where designers describe how a feature should work, developers integrate the necessary functionality into the game's systems and provide tools for content creators to manifest the feature. The process than returns to the designer for review and is reiterated.

Based on our experience, introducing new features into the virtual world may have cumulative effects in respect to bad prior design decisions. Therefore, it is necessary to approach the overall architecture's design as generic as possible.

2.3.2 Engineered Virtual Worlds

Creating a virtual world within an OWG with changing requirements is not an easy task from an engineering perspective. It is much easier if necessary features are known beforehand any design of a framework development starts. However, game development is notorious for changing its core mechanisms as the development progresses and the game comes together as a whole.

Since our focus is to provide the virtual world with believably deliberating NPCs, we have to look upon the framework from an engineering perspective (Champandard, 2004). In principle, any AS can be viewed as an algorithm evaluating inputs and producing outputs. Therefore, we can abstract every NPC (or any other deliberating entity) as a collection of code which evaluates the environment and manifests by choosing actions within the virtual world.

2.3.2.1 World design principles

The actual notation (i.e. language or description) of an NPC's AS mechanism is negligible from a framework's high level design perspective. Therefore, we can abstract the NPCs and their AS from a software engineering perspective as components within a large software system. Software engineering as a research field has created many paradigms and approaches on how to handle *software design* (Gamma, Helm, Johnson, & Vlissides, 1995), *development management* (Sutherland & Sutherland, 2014) and *maintenance* (Swanson, 1976).

Our primary concern while designing the virtual world and our encompassing framework should be the software design paradigm we choose. There are several valid models for structuring our framework. The most notoriously known paradigm is the *Object-Oriented Design* (OOD) (Gamma, Helm, Johnson, & Vlissides, 1995) which capitalizes on discriminating everything as objects and interactions between objects. It is by far the most popular model to represent ideas and constructs within software design. We follow the OOD as our principle design paradigm for our architecture.

Software developers often employ the concept of *decoupling components* (Beck, 2011) from each other to reach a degree of independence, thus being resilient to changes within. Further, decoupled systems tend to be more adaptive when new mechanisms are introduced. Our framework is built on top of the philosophy of decoupling and decomposition into independent components.

2.3.2.2 Language Design principles

The internal workings of the AS may be irrelevant from a high level perspective, but are important to the overall virtual world's design. In principle language design impacts the way virtual world is perceived by developers, thus influencing design choices in return. The language design also influences patterns which address common design pitfalls. In software development, there are many relatable *patterns* (Larman, 2005) which have been explored in respect to good practices how to handle them. For decision making and action selection, known patterns have been proposed in robotics (Graves & Czarneck, 2000), autonomous agents (Kendall, Krishna, Pathak, & Suresh, 1998), and reactive reasoning (Weber, Mawhorter, Mateas, & Jhala, 2010).

The *Object-Oriented Programming* (OOP) (Lewis & Loftus, 2008) paradigm is commonly associated with the OOD paradigm. In principle, everything is an object with associated procedures and data in respect to *encapsulation* (Scott, 2006) which facilitates the information *hiding principle* (Mitchell, 1990). Objects are strictly isolated from each other and only their respective routines do modify object's data. To provide means for access, an object can provide a *public interface* which provides services to manipulate encapsulated data. This allows to employ *separation of concerns* (Laplante, 2007), where objects are responsible for certain dispositions of overall functionality. Having the software solution separated in this manner allows for a much more modular approach when designing features.

It is common that OOP is coupled with *generic programming* (Musser & Stepanov, 1989) paradigm, where code only denotes algorithms and is independent of data types. This reduces code duplication and formalizes certain algorithms (e.g. sorting of items).

Metaprogramming (Joshi, 2014) and *reflection* (Krauss, 2014) paradigms are concerned with adaptation of code. In principle metaprogramming falls under the generic programming paradigm and is in principal concerned with executing expressions that contain snippets of code. Such expressions can be modified during runtime allowing to change code as required. Reflection paradigm is similar to metaprogramming, but is aimed at host code inspecting and modifying itself during runtime. For example, choosing the called function on an object based on daytime (e.g. `CallDay` and `CallNight`).

It is noteworthy that there are other programming paradigms like *aspect oriented programming* (Kiczales, Hilsdale, Hugunin, Kersten, Palm, & Griswold, 2001) which focuses on extending functionality by adding code associated with specific aspects of existing code. We perceive it as adding another dimension to the already present code.

2.3.2.3 Conceptual perspective

Artificial intelligence within OWG tackles similar conceptual problems with deliberation as robotics (Bryson, 2001) and agent systems (Schurr, Marecki, Tambe, Scerri, Kasinadhuni, & Lewis, 2005) do. These environments can be classified as dynamic, adversarial, partially observable and populated by multiple entities. The

deliberating entities have only limited resources at disposal with the requirement to behave believably in a timely fashion.

Within robotics, the *subsumption architecture* (Brooks, 1986) provides the concept of behavior decomposition into sub-behaviors organized in a layered structure. High level behaviors utilize competences present at lower levels to manifest action selection based on received sensory input. The layers were populated by FSMs which were modified to hold data structures and have the capability to inhibit or suppress other FSMs by sending signals. This design influenced the development of artificial intelligence for both robotics and games alike.

2.3.2.4 Summary

One of our goal is to deploy our architecture in a real-life large scale production of a computer game. Therefore, we take design pipelines and production interactions into account. One of our key concerns is to make our architecture understandable on a principal level, thus abstractions must be relatable to common non-programmer's thinking (Mechanism 2 – Three tier deliberation architecture, Mechanism 5 – Semantic network). However, on a lower level, we adhere to principles used in software development using OOP (Mechanism 1 – SBT language), since they are commonly adopted as the most natural way to express human concepts in a computer's environment.

We adopt the OOP paradigm in our architecture's design, strictly separating entities from each other, providing low level mechanisms for communication. Our architecture is aimed at motivating developers to separate concerns into dedicated systems on both engine and script side (Mechanism 3 – Intelligent Environment, Mechanism 4 – Smart constructs). Further, we found the need to integrate reflections as a key component within our architecture, where objects can communicate their interfaces and capabilities to others at run-time (Mechanism 5 – Semantic Network), being able to adapt based on the current context (e.g. when it starts to rain, an NPC cannot go to work on a muddy field).

2.3.3 Managing Complexity and Context

Managing complexity is one of the key issues of developing any software. The field of software engineering (Pressman & Maxim, 2014) has identified and investigated many pitfalls and dead ends in respect to software design (McConnell, 2004) (Hunt & Thomas, 1999), code management (Fowler, Beck, Brant, Opdyke, & Roberts, 1999), maintenance (Feathers, 2004), and agile development (Humble & Farley, 2011).

Managing complexity of virtual worlds does not differ from managing software complexity. Most of AS mechanisms can be seen as part of a codebase which has to be designed, integrated and put together so the result is a working product and not »spaghetti code«. Interactions and competences of virtual world's subsystems have to be designed so they can be easily understood. Many projects which have built complex environments have stumbled upon these issues as well.

Within the computer game industry, several projects stand out in respect to managing complex virtual worlds in a believable fashion. Bethesda (Softworks Bethesda, 1985) has developed *Radiant AI* (Bertz, 2011) which decompose behaviors into isolated actions which are combined to satisfy goals. Decomposition helped to contain behavior complexity. The Sims game series (Ingebretson & Rebuschatis, 2014) utilized a system build on *affordances* (Gibson, 1977) where needs of NPCs (e.g. sleep, hunger, boredom) were satisfied by objects

or other NPC actions within the world. NPC actively searched or passively accepted presented influences by objects. The interaction was driven by engaged object (i.e. the object takes control of the NPC) thus allowing to introduce novel stuff without changing the underlying architecture.

The game of F.E.A.R (Monolith Productions, 2005) utilized the GOAP system, thus decomposing the world into actions and goals, there the NPC's deliberation comprised of a modified STRIPS planner (Fikes & Nilsson, 1971). This allowed for an adaptive introduction of new interactions, however was less manageable and developers did not have enough control over the NPC's AS outcome.

The Halo series of games (Halo Games, 2001-2017) combined FSMs and a variation of behavior trees (Isla, 2005) to handle the complexity of varying behaviors for different types of characters (e.g. soldier, enemy, civilian etc.). The overall complexity was handled by decomposing behaviors into sub-behaviors managing possible situations. However, scripts were chosen from a library in respect to impulses from the environment thus limiting any complex contextual reaction to the diversity of the impulses (e.g. player got into the car).

In Hitman: Absolution (IO Interactive, 2014) NPCs were coordinating with player *triggered events* to solve complex situations (e.g. searching for an intruder, engaging in combat etc.). The triggered event (e.g. shooting a gun in room) created a *situation* which coordinated a set of NPCs with assigned *roles* (Vehkala, 2012) (e.g. leader, point-man, panicking civilian etc.). The situation was responsible for communicating changes to participating NPCs to influence their AS. However, every NPC had to cover specifics of a situation it could be part of, thus creating a tightly coupled design which may prove fragile in respect to future changes.

Other projects also employ concepts of *opportunistic control* (Crytek, 2013). This concept allows for objects and places to take control of an NPC, which either enters an area (e.g. gate of a compound), uses an object or is in the proximity of an object (e.g. bench). The respective area or object takes full control over an NPC (e.g. full animation control) or instructs the NPC's AS to execute necessary actions (i.e. a bench tells the NPC to sit on it). Other games like FarCry 4 (Ubisoft Montreal, 2014), Castlevania: Lord of Shadows (Mercury Steam, 2010)(Parera, 2013), F.E.A.R. 2 (Monolith Productions, 2009), BioShock: Infinite (Irrational Games, 2013) used similar approaches.

In FarCry 4, not only objects could acquire control over NPCs, but also events triggered by the player. These events either used existing NPCs or spawned temporary ones. For example, the player could trigger a »tiger attack« which spawned a tiger and took control of the closest soldiers to use them as victims. In BioShock: Infinite, the player's companion Elizabeth engaged objects placed in the environment. For example, if the player was staying too long at an area, Elizabeth started to wander around investigating the environment and triggered opportunistic control from certain objects like levers etc.

In academia, behavior decomposition was introduced with the *subsumption architecture* (Brooks, 1986) which proposes the decomposition of behaviors into sub-behaviors. Sub-behaviors were utilized to handle specific functions (e.g. pickup an item) and can be further decomposed. Introducing a hierarchy and decomposition is similar to introducing function calls, object oriented design and differentiation of competences in software engineering.

Putting intelligence into the environment was also utilized in crowd simulations (Tecchia, Loscos, Conroy, & Chrysanthou, 2001), where the environment was overlaid with a grid of cells which instructed agents on their proper behavior in those

areas. The approach by (Sung, Gleicher, & Chenney, 2004) introduces *situation based behavior selection* (similar to Hitman: Absolution) where an area denotes a topological trigger, and a situation denotes a contextual trigger (e.g. NPC reaches 20% health in the village).

There is also the work on *smart materializations* (Brom, Lukavský, Šerý, Poch, & Šafrata, 2006) which is coupled with the BDI model decomposing intentions into hierarchical structure. Adopting an intention from the environment may trigger the NPC to adopt its sub-intentions. Intentions can be split until a low-level AS can provide necessary actions or action plans to satisfy the intention. This allows for a structured approach on designing the world for NPCs to manage their goals.

2.3.4 Summary

We adopt the notion of intelligent (Mechanism 3 – Intelligent Environment) and context aware objects (Mechanism 4 – Smart constructs) within the world. However, we avoid the notion of opportunistic control over NPCs, since we aim at maintaining full control over NPC's action selection within the confines of the NPC's DMM (i.e. encapsulate action selection as a conceptual part of the NPC). We build on top of the ideas of affordances, where we intend to maintain relatively simple and generic NPC action selection and be able to allow NCPs to extend their action selection based on what they encounter in their environment (Mechanism 5 – Semantic network). Decomposition of behaviors into the environment's components (i.e. objects, areas, etc.) aids our focus on reducing singular complexity (i.e. huge unwieldy action selection at an NPC) and on keeping a large code/script base maintainable. Further, decomposition of behaviors is also necessary to allow easy and straightforward integration of new features and behavior variations (e.g. sitting on a chair, sitting on a bed, sitting on a bench with several places).

3 Scenarios

We aim our architecture to be employed within a real-life production of computer games. Therefore, we focus the function and principles to go beyond a proof-of-concept. Based on our experience we established a set of common scenarios encountered in large scale open world role playing games. Our architecture is aimed at providing both conceptual and practical means to address the overall believability of the virtual world in such cases. These scenarios were coauthored by developers at Warhorse Studios[2] to utilize their know-how on how such scenarios would be engaged in the actual production. We explored situations which are present in the produced KCD game: 1) *Life of Brian*, 2) T*avern*, 3) *Death in the streets*, 4) *Poisoned Jerry*, and 5) *Battle*.

3.1 Life of Brian

Brian is a common peasant in the KCD game, who is not particularly interested in any questing or being hurt in any way. When something happens, he tends to run away as far as possible to avoid any harm. Brian is very responsible and goes to work every day, except for rainy days.

Brian, gets up in the morning, eats some breakfast, goes to work or does housework. After he is done, he likes to have some fun at one of the local taverns in his village. After he has had his fun, Brian returns home to get some sleep before the next day starts. When Brian gets too drunk, he wakes up his wife and they argue a little.

This scenario investigates the common ambient life we focus at Goal 1. Brian is a common representation of any day-by-day NPC which populate an OWG. Brian-like characters often avoid interacting with the player, since from a production perspective it is rather problematic to voice so many characters with different voices. Also, dialog topics would be rather limited, since game design tends to avoid repetitive dialogues. Therefore, it is common that the player triggers mostly reactive behaviors resulting in either generic responses (e.g. greetings, shouting etc.) or fight-flight scenarios (e.g. guards fight, civilians flee).

3.2 Tavern

The »Tavern« scenario revolves around NPCs seeking fun in the afternoon after a day full of demanding work on nearby fields. Some of them head straight into a tavern and some stop at home. Since the life of a single NPC is covered by the »Life of Brian« scenario, we will focus on the tavern's workings. Primarily the tavern is home and workplace for an innkeeper and his daughter who acts as the waitress. Their day-by-day work is to keep the tavern clean and when it opens, they are responsible for keeping the guests happy. However, since the tavern is their home, they tend to avoid any damages to it. Since most townsfolk are at work during the day, the daughter can clean up the place, and the innkeeper brings new beer barrels and repairs potential damages from last night. Before the tavern opens at 18:00, all potential guests are asked to come back later. As for the tavern itself, it contains some place to sit, drink, and play cards.

[2]Petr Ondráček, Martin Antoš, Michal Vrtílek, Petr Maláč and Viktor Bocan

There are three sub-scenarios that may happen:

1) *A guest enters* the tavern and wants to sit somewhere, it may happen that the tavern is full and so he goes to have fun at a different place. He may prefer to sit alone far from everybody or with his friends to socialize.
2) *A guest orders* some food and drinks. After he finishes his beer, he orders another one, until being drunk. Some guests may start to sing, others may get upset about losing a card or dice game and start a fight.
3) *Fighting guests* may attract a guard to sort things out. If the situation gets out of hand, the waitress will run for help. Other guests leave as soon as a bigger disturbance occurs, since they do not want to be part of it.

The tavern closes at midnight so everybody can go home. This scenario covers Goals 1 and 4. We aim at providing a believable ambient simulation of how a tavern is managed from both a processing point of view (i.e. managing drink orders etc.) as well as from an individual perspective (i.e. going for a drink). The complexity of the presented behavior is important for a believable feeling.

3.3 Death in the streets

This scenario revolves around a common issue within almost any RPG game – the player kills an NPC either by accident or on purpose. We focus the scenario on reactions of bystanders:

a) *Guards* – are responsible for maintaining law and order within densely inhabited areas. They are allowed to stop, interrogate and engage hostile targets. They even can throw a criminal into prison.
b) *Citizens* – inhabit nearby houses and tend to run away from any potential situation, since they want to avoid any bodily harm. They are also very investigative and like to watch.
c) *Soldiers* – are similar to guards, but tend to have much less understanding for criminal activities and tend to engage first, and not ask at all.

One important notion is how the scenario starts from a bystander's perspective:

1) *Murder is witnessed directly*, thus there is no question about what may have happened. Guards tends to arrest the player, and if resisting, engage him in combat. If the player flees the crime scene, guards will chase after him. Citizens run away, preferably seeking a guard to report the crime. If no guard is found, citizens tend to forget what they saw. Soldiers tend to engage the player directly, but will not chase him.
2) *Dead body is found,* thus a question arises who did it. If a criminal can be inferred (e.g. the player stands nearby with a drawn sword), the outcome is similar to the first case. Otherwise, guards try to investigate crime scene, ask around and eventually clean up the body. Citizens are investigative and tend to stand around and talk about what may have happened. After a while, they get bored and go after their business. Soldiers tend to be uninterested in a dead body, since they have seen their fair share already.

This scenario covers Goal 1, 2 and 4. The reactions of bystanders creates a believable emergent situation which may affect ambient life nearby (e.g. after a murder people avoid going to taverns for some time). Overall mood in the village

may change due to the fact that some murderer is running around. Also, NPCs can chat about what and where has happened and maybe be less friendly when talking to strangers, creating a more complex experience providing the illusion of consequences for actions.

3.4 Poisoned Jerry

The scenario of Poisoned Jerry reflects a complex quest within an RPG game. The storyline revolves around a villager called Jerry who sits in front of his house and is sick to his stomach. When the player comes along, Jerry asks for help. He wants to get rid of his stomach pain and find out what has happened to him and who is responsible. Jerry provides the player with a set of suspects – 1) his wife, 2) his sweetheart, 3) his neighbor, and 4) a herbwoman living in the woods. The player must collect evidence, talk to other NPCs and help Jerry to get healthy. There are several ways on how to make Jerry healthy: a) befriend the herbwoman and get an antidote, b) try to steal an antidote from the herbwoman or Jerry's enemy, or c) buy an expensive antidote in the nearby city. Discovering who poisoned Jerry requires the player to talk to most of the suspects and observe their behavior in respect to Jerry. The player may not get all the information and may guess whom to accuse. However, there can be dire consequences for wrong accusations.

After the quest starts, all the participants change their daily routines to be more in line with the quest. The herbwoman goes for occasional walks into the woods to let the player search for her. But after they meet she will stay more at home to avoid being robbed if the player bullies here instead of befriending her. Jerry's sweetheart will be working longer hours at the tavern where she cooked rotten meat (i.e. the actual culprit) by accident. The neighbor will get sick next day after eating at the tavern too, so the player gets a clue about what may have happened. Jerry's wife will be around not being friendly at all, having some antidote with her. After the player comes to a conclusion, he decides to accuse someone of mischief. If he accuses Jerry's wife, she stops talking to Jerry and Jerry starts to drink a lot more. If he accuses Jerry's sweetheart, they stop seeing each other and some future quests in respect to their relationship cannot be acquired anymore. If the herbwoman is accused, she will not come into the village anymore and there will be no more potions for sale. If the player takes too long to investigate, the neighbor will die eventually.

This scenario covers Goals 3 and 4, since the behaviors are rather complex, having story driven effects on the NPC behaviors. We aim at providing custom changes to NPC behavior based on the storyline. However, these custom behaviors are integrated into the day-by-day NPC behavior, to avoid behavior artifacts common in other games, where NPCs on a quest tend to wait for the next player's move no matter what happens (i.e. they stand in front of their home day and night not caring for anything else than the quest). Further, the quest's results and the player's approach have long term effects on both the world and the relationships between NPCs. This provides a more believable environment where the player acquires a feeling of actions having consequences. We want to avoid the common OWG pitfall where ending of a quest has no impact of what happens in the world.

3.4.1 Battle

The Battle is a simple scenario covering a large-scale employment of many NPCs at one location. All NPCs are put into formations and march towards the enemy and a

small skirmish is conducted. The evaluation of the battle is semi-random in nature (e.g. which flank collapses is based on actual combat between NPCs). The player is dispatched to help at problematic locations.

This scenario mostly covers Goal 5, since most of the fighting and low level decision making (e.g. where to go, how to attack, etc.) happens in code due to optimization. The large-scale nature of this scenario determines how well our architecture copes with large amounts of NPCs present at one location. It will also show how our architecture scales in respect to NPC counts.

3.5 Summary

The above presented scenarios are to our knowledge the most relevant to OWG. They were our primary test bed for both the implementation as well as the feasibility of the core mechanisms. Since they were developed for use in the KCD game, Warhorse Studios used them to evaluate and educate candidates for the scripting and level design departments. Last but not least, these scenarios helped us to outline the overall picture on how a believable virtual world should manifest and how designers and screenwriters can make use of our architecture. These scenarios also helped us to identify common subsystems at the script level, which had to work across scenarios to provide specific functionality (e.g. crime system works the same in Scenario 4 (Jerry), Scenario 3 (Murder) and Scenario 2 (Tavern), since there is the possibility of the player conducting a crime – murder, theft, false accusation).

In the KCD game, these scenarios are covered by a large collection of behaviors spread across the virtual environment. Decomposing and isolating specific functionality into small behaviors helped scripters to maintain a fairly large (above 3000 individual behavior trees) behavior base.

4 Making Decisions

Action Selection (AS) is the process of choosing the next proper action based on evaluating the *context* provided by NPC's DMM. DMM context specifies NPC's *interpretation* of the environment – it combines *sensory perception, internal state* and *goals*. Action selection is aimed at either reaching a desired configuration of the environment (e.g. to move from point A to point B) or maximize a utility function either momentarily (e.g. having enough health to avoid death) or over time (e.g. to experience some fun at least once a week).

Conceptually speaking, AS is the lowest level decision making mechanism within an NPC, except for automated subsystems like collision avoidance or path following. There are many different ways on how to execute AS – FSMs, BTs, procedural scripting languages, classical planning etc.

Based on our prior research (Plch, 2009), we chose to further develop our concept of Stateful Behavior Trees (SBT), which builds upon the well-established notion of Behavior Trees (Champandard, 2007) (BT).

This chapter discusses our low-level action selection language. We introduce the language's tree-like structure and explain its internal mechanisms. We provide insight into our budgeting system, to allow users to employ our language in a large-scale simulation where computational resources are limited. We also address parallelism issues to allow for concurrent AS code execution. We also discuss our data model, expression evaluation and data type handling. As part of this chapter, we introduce our communication mechanism utilizing a messaging system, where NPCs send complex data to each other. We discuss our synchronization mechanisms to create synchronization schemes between NPCs (e.g. locks and semaphores). Lastly, we discuss our low-level action system to provide more streamlined execution of actions.

Our main goal is to provide a generic and expressive language to manifest complex behaviors for both the environment (Goal 3 and Goal 4) and NPCs (Goal 1). We combine the approach on how to structure code in an imperative language (e.g. C++) with the structure of a BT. We build on top of our prior research (Plch, 2009) which tackles the issues with the BOD methodology (Brom C. , 2005). The use of BTs is aimed at non-programmers to avoid complex written languages and provide visual means to abstract behaviors.

4.1 Behavior Tree

Behavior Trees (BTs) (Simpson, 2014) represent a well-established approach describing action selection for NPCs. BTs decompose AS into a tree-like structure (Figure 8). The tree is evaluated from the *root* in a consistent (i.e. child nodes are always access in the same order) depth first approach (Cormen, Leiserson, Rivest, & Stein, 2001) until an action choice is reached. Every non-leaf node aggregates its result from evaluating its respective children. Every node can report three results 1) *processing*, 2) *success*, or 3) *failure*. The duration of the evaluation is not limited to being atomic, thus can take time to finish. If no new action is chosen, the NPC either enters an idle state or continues with the currently executing action (e.g. playing a looped animation).

The leaf nodes represent action choices and non-leaf nodes are used to create a path through the BT for a leaf node to be reached. The non-leaf nodes are split into four types:

1) *Sequence* node evaluates its children in order from first to last. If the child node returns Success, the evaluation continues to the next child. When a child Fails, or is Running, the sequence adopts this result as its own and reports it. Since the evaluation is perpetual, the sequence keeps track of which child is to be evaluated next. Further, if all children return Success, the sequence also returns Success.
2) *Selector* node evaluates in the same manner as the sequence does, with the difference when processing of a child node. If a child node returns Fail, the evaluation continues to consecutive child. In case of Success, evaluation stops similar to what happens at the sequence in case of Failure.
3) *Decorator* modifies the result of the underlying subtree which is rooted under its single child. Common uses are to invert evaluated values of a child node from Fail to Success and vice versa.
4) *Parallel* behaves similarly to either a sequence or selector (depends on the specified parameters) however it evaluates all children concurrently. When in sequence-like mode, the evaluation of its child nodes is stopped if at most one child returns Failure.

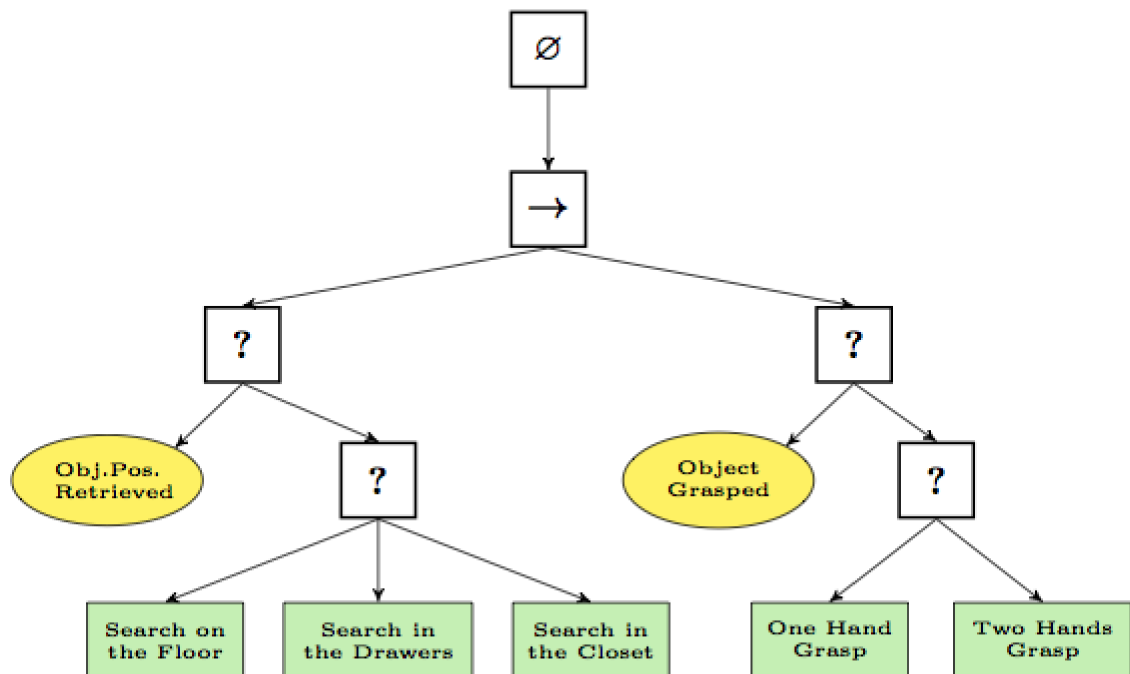

Figure 8: A simple Behavior Tree representing the action selection of searching an item and grabbing it with either one or both hands. If the item is not found, it is searched on the floor, in the drawers and in the closet. The block leaf nodes represent action choices, the rounded leaf nodes represent sense actions (determine if something is true or not), boxes with question marks represent selectors and arrows represent sequences. (©Wikipedia)

The Action Selection represented by Behavior Trees can be used to create rather complex constructions with little effort. Contrary to FSMs or other mechanisms, structuring of the thought process is very close to how designers think about humans solving action selection.

We have identified downsides to the basic idea and improved both the syntax and the semantics of the trees to better express complex behaviors. The key issues were:

a) *Node execution model* – nodes are only limited to reporting their current evaluation, thus being limited in representing their internal state. This limits nodes in respect to different states representing their inner mechanism to other nodes without reflecting the node's purpose. Further nodes cannot suspend and resume when either being passive (e.g. waiting for a pathfinding request to finish) or being suspended by parent nodes. Nodes also lack the capability to resume their suspended state. The issue of switching between behaviors was addressed in our previous work (Plch, 2009). A more complex execution model may allow for a more detailed control over a BT's evaluation.

b) *Limited variable support* – the traditional BT representation has only limited support for variables as part of the BT description. Often employing a blackboard data architecture (Corkill, 1991) to store relevant data. We aim at providing local data for the BT to be able to store partial evaluations, temporal and persistent data to be used when executing. This also follows the methodology of encapsulation and information hiding present in the OOP paradigm.

c) *Missing synchronization* – the BT scheme lacks any complex synchronization mechanism between ASs, either from NPC to NPC or within one NPC's AS. To provide complex behaviors, synchronization is key to executing coordinated actions (e.g. one NPC waits until another NPC finishes an evaluation).

d) *Missing communication mechanisms* – BTs also lack any complex communication mechanism beyond using a blackboard. This limits the capacity of NPCs to convey complex data structures to each other in organized fashion.

e) *No explicit execution awareness* – commonly BT are executed so all nodes can evaluate their result and report it to its parent node. We need BTs to be able to function in a manner when their evaluation and execution can be interrupted at any time and budgeted in a similar fashion as common CPU run code. This is a key mechanism due to the fact that running large numbers of responsive NPCs on limited hardware requires more intricate execution schemas.

The above presented issues are addressed in our Stateful Behavior Tree (SBT) model and its internal structures and mechanisms. In the further subchapters, we will present our architecture's low level language implemented as the primary AS for DMM for a large scale OWG. All the presented mechanisms have been implemented into the KCD game and utilized in the game's production.

4.2 Stateful Behavior Tree

The SBT model is an extension of our prior approach on extending the BT formalism (Plch, 2009). We took our primary inspiration from known procedural language structures like C++ or Java. In principle, we view the SBT as set of named language elements (i.e. nodes) organized into a tree-like hierarchy of nested blocks, similar to code written in C++. Our aim is to provide the language with more flexibility in both execution and evaluation. We also focus on being able to simply add new language elements (e.g. loops, switch statements, try-catch constructs etc.).

4.2.1 Tree Nodes

Every tree node within the SBT model represents an isolated language element. These can be simple elements like »if« or »for« statements, or more complex one »follow the player«. The notion of »language element« is synonymous with the notion of »tree node«. All language elements are organized into a tree-like structure similar to a code block organization in a procedural programming language. We have designed our nodes with two principles in mind:

1) *Abstract isolation* of nodes is focused on separating nodes from each other so they are not affected by what is happening within other nodes. Ideally, nodes are unaware of what other nodes do and are only concerned with their own agenda. This provides us with the flexibility of decoupled code.
2) *Abstract control* of nodes is focused on providing only abstract interfaces to nodes to be influenced by other nodes. The individual control allows for more detailed execution control and debugging capabilities (i.e. we can stop AS at any node at any time provide inspection tools to evaluate it).

Based on these requirements, we designed the workings of a node as small FSM (Figure 9).

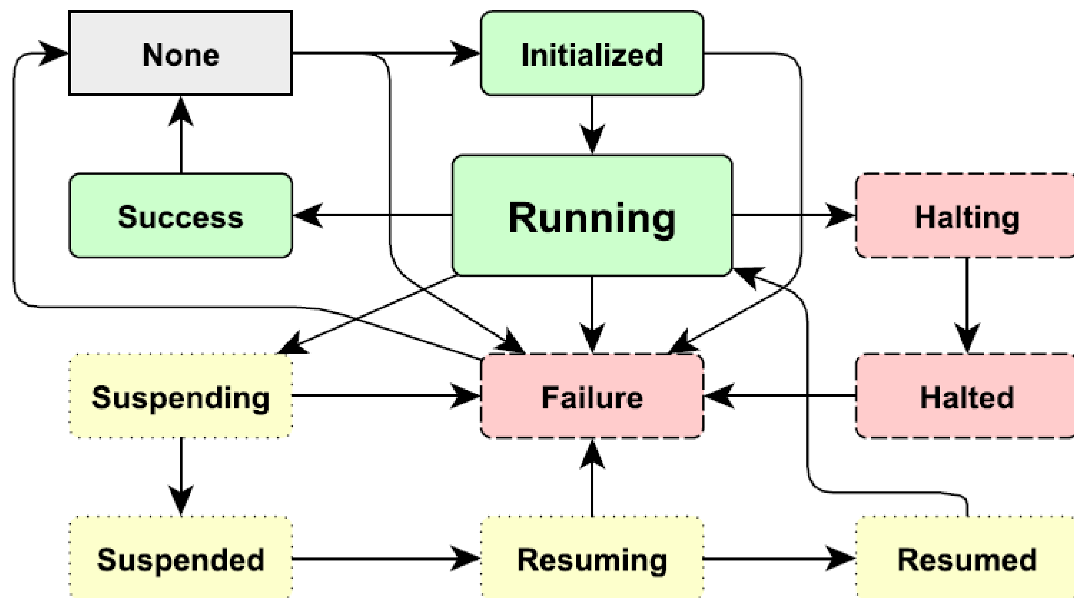

Figure 9: The FSM of a language element (node) within the SBT structure. The node starts at the None state and traverses over the available states until it reaches the Fail or Success state.

We designed our node's FSM to start at a state called *None* where the node is inactive. When transitioning trough states, every FSM finishes at either a *Success* or *Failure* state (i.e. terminal states). The internal reasoning about when to get to a termination state may differ from node to node. Generally speaking, a terminal state can be reached from almost any state of the node's execution. Every node can be *Reset* to get from a termination state to the None state.

To enter the *Running* state, the node has to be initialized first by its parent. In our case, the initialization is a synchronous process, but it could be reasonable to have initialization that takes more time to happen. After the initialization is finished, the

node enters the Running state, from which it may terminate on its own only by means of Success or Failure. To provide the parent with some governance over a node, the parent may enact *Halt*, *Suspend,* or *Abort* on the executing node. When halting, the node is asked by its parent to enter a Fail state at the earliest possible occasion. Halting can be viewed as an asynchronous request and Abort as a synchronous one, where the parent node needs the child node to end instantly. A third option is to Suspend an executing node, by asking it to enter a specific suspended state where the node stops receiving updates for its execution and waits for a *Resume* to happen. In our implementation, Resume is synchronous, however it may be reasonable to have it asynchronous. All asynchronous operations lead to necessarily dealing with conflicting situations between transitions (e.g. a node Suspends, the parent requests a Halt.).

4.2.1.1 Summary

We chose this structure of nodes to provide other nodes and subsystems with a more complete information about their internal state. Further, other nodes and subsystems do not need any knowledge of the node's internal mechanisms and can control it by a simple interface. Adding new states allows us to handle additional practical issues with proper initialization and termination of the node in respect to the AS and other NPC subsystems (e.g. the node can register to callbacks when initialized and when terminating, unregister in a paired fashion).

Another upside of such node structure is the easy implementation of new nodes by programmers, maintain existing ones over time and adapt to changing requirements by scripters and designers (e.g. the follow-the-player node has to have a chasing policy).

4.2.2 Execution and Evaluation

The language structure defines how the language is shaped, its interpretation shapes the evaluation's outcome and use. Popular decision trees have an execution model, where all predicates are evaluated at once and the action following the highest priority predicate is chosen to be executed. On the other hand, an imperative language like C++ has an execution model where branched instruction streams are packed into calls which may be executed within a stack in a very linear fashion. Other languages like Prolog (Clocksin & Mellish, 2003) are based on periodic evaluation of a predicate set to meet a query.

Evaluation represents the contract between language elements and the ordering of *data operations* which we trivialize as reads and writes to memory. In SBT reads can occur concurrently at any time, since they do not affect data. Write operation have to be ordered in respect to both other write operations and read operations (i.e. a write operation is executed after all the preceding reads and writes are finished).

Execution represents the contract between language elements and the interpreting system on how and when individual elements are executed and their effects are committed. In SBT we distinguish two effect types:

1) *External effects* are directed outwards in respect to the NPC. These affect the NPC's external properties, the environment or other NPCs. They are commonly executed at the end of a frame and taken into account at the next frame.

2) *Internal effects* can be either directed at modifying NPC's internal properties or can be read/write operations into variables or data containers contained within the NPC.

Since our language elements are small FSMs, we view the execution model of SBT as executing every *active node* (i.e. Running, Halting, Suspending or Resuming state) in a concurrent fashion. All effects of node decisions are applied as they come along. Practically, execution happens within a frame, based on a given time budget. Since every node is isolated, the communication up and down the language's tree structure is limited to *reporting of state changes* – either being a report from a child to a parent or being an operation forced from parent onto child (i.e. do halt, suspend, resume).

With respect to evaluation, we require SBTs to have partial execution ordering, since nodes share memory scopes which are also organized into a tree like hierarchy i.e. parent nodes provide scopes for child nodes. Branching statements (i.e. conditions, switch statements etc.) and their evaluation can lead to different results if at least some order of evaluation is not kept. To provide a manageable sense of evaluation, branching elements are guaranteed to evaluate in order of their appearance in execution – i.e. parents are evaluated before child nodes and child nodes are evaluated in an ordered fashion (i.e. sibling nodes are strictly ordered).

Primarily, we assume the SBT execution engine of every NPC receives a given time maximum it may use – a *time quantum*. If the execution exceeds the quantum's limit, the execution is required to cooperatively return control to the governing system (i.e. DMM mechanism of NPC's AS). This is also known as cooperative multitasking (Silberschatz, Gagne, & Galvin, 2008), where executed elements need to cooperate to a certain degree to avoid stalls and provide an illusion of parallelism. To further optimize overall performance, we allow nodes to be excluded from execution – *falls asleep*. This concept is also known as *passive waiting* (Tanenbaum, Modern Operating Systems, 2014). The sleeping node can be either forced out of sleep (e.g. by other nodes who want to halt it) or can wake up on its own (e.g. a timer is finished and the node wants to terminate). We have developed three ways on how SBT can approach the order of execution within the tree structure:

1) *Pass Through* is an execution ordering, where the tree is always traversed from the top to all its eligible leaf nodes. Every node at the traversed paths is given time to process. Every node executes within the execution of its parent, thus providing a sequential ordering of any effects. This approach has major drawbacks in respect to effective code execution and is heavily impacted by the depth of the executed tree. To avoid unnecessary execution, some nodes may enter a passive state until a governing system wakes them up (e.g. timers).

2) *Linearization* of node execution creates a queue of nodes. Nodes are pushed onto the queue *by request* (either by them or by their parent), or *automatically* after they wake up from passive waiting or their child node changes its state. The queue executes sequentially in a round robin fashion. All nodes are uniquely present in the queue. This approach allows for a more streamlined execution which can be interrupted at any time at the granularity of a single node's execution. However, there are complex corner cases based on implementation details – e.g. an already suspending child is halted by its parent etc.

3) *Contracted tree* represents the original tree in a fashion where sleeping or not-executing nodes get *contracted* – are temporarily removed from the hierarchy. All nodes within the contracted tree are executed in a breadth traverse pattern. If a node is required to reengage execution it is reintroduced into the contracted tree at its original position. Conditions for a node to reengage execution are similar to the Linearization execution principle.

At first, we implemented SBT execution as a Pass through approach, since it is easier to implement and maintain. However, due to optimization requirements, we implemented the Linearized approach. We prototyped the Contracted tree approach but the operations on top of the tree structure were too computationally costly.

4.2.3 Budgeting

Since processing power is a limited resource on any computer system, it is necessary to employ some sort of budgeting system for the execution of DMM to avoid stalls (i.e. FPS drops) or non-responsive reactions. Even on multi-core and threaded environments, the hardware limitations require to interleave multiple DMM executions to provide the illusion of parallelism.

The *Budgeting* concept is built upon a simple idea, that a governing system splits the overall budget (i.e. part of the frame limit) between all governed systems based on fair approach and priority. A budget can be set arbitrarily (e.g. 30ms) or estimated based on system load.

However, since our current system has no means to enforce preemptive multitasking (Bovet & Cesati, 2006), we have to rely on a cooperative multitasking approach (Bartel, 2011), where every component of the executed system is aware of the notion of being budgeted and cooperatively returns the control to the system as soon as possible. Otherwise, the system's update will be stuck within the non-cooperative subsystem. This was a frequent problem in MS-DOS (Paterson, 1983) operating system, before processors supported preemptive multitasking.

Since our framework is run on a separate thread, parallel to the rendering thread, it determines its budget adaptively, based on the assumption that the framework's thread and renderer thread should be aligned at the end of each frame to maximize resource utilization. Since renderer thread is beyond our control, the framework's budget is changed to align to it. However, there is also a hard limit to avoid stalls – a single frame should not go over 33ms to reach 30 FPS. Our multitasking is based on following concepts:

1) *Priority* is aimed at having various levels of importance in respect to update. Higher priority DMM receive more time to execute in respect to others. At the same level, execution takes shape of a *Round Robin* queue. However, to avoid starving, every DMM has to receive at least some budget over a given time frame. This can result in either a larger portion of a budget being given to higher priorities, or having an absolute addition to the budget (e.g. priority 0 DMM receive a +1ms boost). Priorities may change over time, based on various circumstances – e.g. player interactions should be more responsive than ambient presences. There is also the concept of priority boost, where a DMM receives a temporal priority gain if it was suspended and resumes execution. We utilized both approaches, when NPCs in closer proximity to the player are in a higher

priority queue and certain actions (e.g. message is delivered) trigger a temporal boost to the budget given to an NPC.

2) *Debt* denotes the overuse of an assigned budget. Since we employ a cooperative multitasking paradigm, it may happen that an DMM does not return control in time and thus generates debt. This is tolerated misuse of cooperative approach, however the debt has to be paid at the earliest possible moment. Therefore, if a DMM has debt, it is primarily consumed at the next DMM's update (e.g. the next frame) and the remainder is provided to the DMM as a budget. If a DMM constantly overuses its budget, it may receive a penalty either a negative priority boost or as a more severe budget cut. However, this may have a cumulative effect, since a DMM may require a budget for processing and stalling its processing creates even more demand to catch up. We only implemented a simple debt mechanic and avoided any penalty system, since we consider these hard to tune to behave predictable.

3) *Fairness* is a straight forward principle, where every DMM receives an even portion in respect to its debt and priority. This avoids differences between DMM executions.

4) *Interactivity* is aimed at providing the user with the most responsive experience, since humans are sensitive about response times. Therefore, it is crucial to take player centric interactions into account, either by boosting priority, budget quantum or overlook debt. We implemented a mechanism to boost priority if an interactive action is triggered.

5) *Proximity* is similar to interactivity, only difference being the topological nature in contrast to triggered action. The player is sensitive about their immediate environment, thus if something far away is amiss, it is less distracting than something wrong happening up close. Therefore, proximity to the player is crucial to provide adequate reaction times. We divide priority queues based on proximity, where closer NPCs are moved into queues with higher priority.

6) *Adaptability* is aimed at providing changes to the budget based on the current execution pattern of a DMM and a collection of DMMs. It is built around the idea that most DMM do some baseline decision making, occasional demanding computations and tend to sleep plenty (i.e. wait for input from external sources). The baseline decision making cannot be avoided; however, it should be marginal and stable in nature. Occasional spikes in demand for computational power should be met with boosting budget or priority to allow AS to get through the spike as quickly as possible to get back to its baseline decision making. The spike indicates a short-term necessity; therefore, it should be prioritized. Spikes can be dealt with temporal budget increases which are smoothed over time, to avoid overfeeding an AS with computational resources, triggering a cascade of spikes as effect. Eventually we did not address these issues since spikes in AS were not our primary optimization concern.

7) *Yielding* is a concept build upon the capability of the executing SBT node to forfeit its budget. It can be viewed as a self-imposed budget restriction. This can be either *intentional* or *proactive*. Intentional yielding is triggered at the node, based an in-node analysis (e.g. too many child nodes were

updated). Proactive yielding is triggered at the DMM level based on varying reasons, for example the tree is being too deep, or too many language elements have been executed. We implemented intentional yielding into specific nodes like Loop and Parallel.

4.2.3.1 Summary

Budgeting is key in respect to delivering responsive and thus believable behavior from NPCs. We implemented a multilayered round robin scheduling scheme taking, budget debt, proximity, interactivity and yielding into account. Without budgeting, Goal 1, 4 and 5 could not be accomplished in any meaningful manner, since we could not afford to manage large numbers of active NPCs with complex SBTs.

4.2.4 Parallelism

Executing tasks in parallel is often a desired feature for a system managing multiple concurrent constructs. Often employed in environments where encapsulated entities execute their code and effective utilization of hardware is key. Presently most modern hardware configurations contain multiple processing units – e.g. 8 core CPUs able to provide 16 virtual cores. Therefore, we aim at designing our architecture to support parallel execution and multicore systems. We have employed the paradigm of parallelism at different tiers of our design:

1) *External parallelism* is aimed at having our architecture's designed so it can be run independently from other engine subsystems in a parallel fashion.
2) *Internal parallelism* is aimed at handling concurrent execution at the DMM's level, where active subtrees can be executed concurrently.

4.2.4.1 External Parallelism

One of key mechanisms of our DMM's architecture is the capability to run in parallel with other engine subsystems to take as much advantage of hardware resources as possible. To run independently of the engine, we employed a *cache-commit* mechanism in respect to data access. Data from the engine is cached at the DMM before AS's update is executed. The AS is executed with cached data at its disposal, thus it is not required to synchronize the data with the engine while accessing it. The downside is the possibility of outdated data. After the AS is done, the changed data is committed back to the engine. We employ the caching phase at start of every frame and commit the aggregated changes at the frame's end.

4.2.4.2 Internal Parallelism

Internal parallelism denotes our aim at executing parts of the SBT in concurrent fashion. A simple example is how new inputs from the environment are processed while executing actions. In principle, the solution are divided into two basic approaches:

a) *Sequential* approach – sequences of checking for inputs and executing actions are executed in a loop. New inputs can be processed only after an action finished its execution. This may lead to untimely reactions to new inputs or reacting to outdated inputs. However, this approach is often favored due to its simplicity, since there is no need to synchronize access

to shared resources or synchronize execution of actions with emerging high priority inputs.

b) *Parallel* approach – has two branches of code, where one branch passively waits for an input and the other branch deliberates and executes chosen actions. The processing branch can filter out unwanted or outdated inputs and deliberate about interrupting the execution branch if a timely reaction is necessary. This approach allows for timely reactions and more complex processing of events and actions. However, it is less favored due to complexity and an inherit need to manage shared resources and communication between branches executed in parallel (e.g. the input processing branch requests actions to be executed at the execution branch).

In our case, we allow to use both approaches to solve problems, since in some cases sequential processing is satisfactory. Sequential processing is supported by use of sequences in the SBT language. To support parallel approaches, we provide two mechanisms at the SBT language and DMM level:

1) *Parallel branching* introduces a node into SBT language structure, which has at least one child node and all child nodes are executed in parallel. If a child node enters a passive state (i.e. waits for a system wakeup), its branch is not executed. However, if the node is signaled and awakens, its execution is resumed. The Parallel node waits for all its children to either succeed or fail. The setup of the parallel node may wary, terminating when at least one succeeds/fails or all succeed/fail. This approach is commonly known as *coroutines* (Conway, 1963).

2) *Asynchronous execution* represents the concept where a piece of SBT (i.e. subtree) is marked as asynchronous, which leads to pulling (i.e. copy) it out of the current tree and placing it into a separate AS running in parallel. It can be also described as spawning the sub-tree separately and executing it. However, there is the issue of utilizing data, since the origin SBT can stop to exist at any time in respect to the pulled SBT. Thus data (i.e. variables) is either copied *by reference*, *by value* or using a copy-on-write approach[3] (i.e. the data exists as a reference until it is modified at the origin). We utilize a copy by value approach, which is slower but easier to implement and maintain.

The parallel branching approach is relatively easy to implement by utilizing a specific node which handles scheduling of its child nodes. In contrast to a sequence which executes one child node after another, the parallel node simply executes all of its child nodes and periodically (i.e. when a child reports a change of state) evaluates their state. However, it represents a static way to describe parallel branching, thus limiting its expressiveness. Also, since the parallel node has to be within a SBT, the parallel executing branches are limited to the lifetime of the parallel node. Thus, it is not suited to handling tasks which require to complete the handler when started. Further, the parallel branches share the data[4] scope thus race conditions on data have to be taken into account. There is also necessity to synchronize the execution of the parallel branches if they depend on each other (e.g. one branch evaluates actions the

[3] Referencing data and asynchronous execution is explain in detail in 4.6.2

[4] Data scoping and variables are explained in 4.2.6

and other branch executes them). However, this approach has proven to be popular due to its simplicity and easy maintenance.

The asynchronous execution approach requires a node to denote a subtree to be executed asynchronously. This allows for to trigger execution of isolated asynchronous handlers to manage emerging situations (e.g. the NPC is hit by something and it has to be handled as soon as possible). Since the SBT is pulled from its origin and put into a separated AS within the NPC's DMM, the execution in individualized (e.g. it can be prioritized etc.). The separated execution also provides a separated memory envelope which limits the capacity of developers and scripters to introduce data related bugs (e.g. writing into a variable without synchronizing). However, this requires the data management at the SBT to handle moving referenced data (i.e. when accessing data in the origin tree), which can be quite inefficient. Further, since the pulled SBT is executed in a separated fashion, it can be executed multiple times with little binding to the origin (e.g. one asynchronous handler per every event). This allows for proper response times and well separated and decoupled SBT code. However, it is not simple to use the pulled tree to report back to the origin SBT, since they run independently (e.g. the origin tree runs an asynchronous handler for every enemy in sight and the handlers report their evaluation of the target's threat level to the origin tree to choose a target). This approach is popular for handling events and triggers which are independent of the core NPC's AS.

4.2.4.3 Summary

Our architecture provides both of these approaches to the user, since we need the NPC to deliberate in a timely fashion. The branching approach is aimed at providing more complex behaviors (Goal 4), where an NPC can evaluate and execute at the same time. The asynchronous approach is aimed at managing the variety of events which the NPC can encounter in complex environments with emerging situations (Goal 2). We also utilize the asynchronous approach to deliver complex behaviors (Goal 4) coupled with actions[5] where asynchronous SBTs manage an action's internal events. Both approaches allow us to provide the SBT language with more capability to handle design of complex believable behaviors (Goal 1), where NPCs act in a more complex fashion (i.e. NPCs can do more actions – go to work, greet the player and make remarks about his armor).

4.2.5 SBT Events

SBT node states are designed to inform about the particular high level state (e.g. suspended) of a node thus avoiding any detailed information about results or internal specifics. This works well in respect to node management, where no more detailed information is needed. However, there is understandable desire to communicate more complex information than simple running/success/fail of a node. Beyond the concept of returning a result, the node may need to inform the user or other DMM systems about specific situations (e.g. it encountered an exception while evaluating) or partial results.

One possibility for a node to communicate is over *shared data*, where the parent node provides the child node with variables to use either as return value or to report specifics during execution. However, this approach has limitations, due to the fact

[5] Actions are discussed in 4.6

that the parent node has to inform the child about which variables are designed for which information (e.g. `$pos` should be filled with the target position for movement). Further there is the necessity to synchronize changes to those variables between the child node and anyone interested in the results (i.e. some other node or subsystem may be interested in that information). One possible solution is to always provide a commonly named variable (e.g. named »returnValue«), but this creates unwanted memory consumption and requires providing a proper type for the variable. Since we do not want to introduce a *reflection contract* (i.e. being able to tell what is the node's or subtree's return value type), we avoid the principle of forwarding contracted return variables as much as possible. We do however forward contracted data into subtrees as parameters (i.e. a subtree of a particular node has variables the subtree's root fills with data).

To avoid the dependency between parent nodes and child nodes, we implement an *Event System* within the SBT. The Event System is easy to use and versatile up-the-tree (i.e. traverses the node from child to parent) communication mechanism. An *event* is a tuple (*name*, *type*, *custom data*) and is either provided by the node's code or by the SBT script (i.e. similar to a C++ *throw* statement). Events are not limited to returning values after node termination and may be spawned by any running node at any time.

Events are propagated from their origin (i.e. node within the SBT) to its parent. The parent either removes the event, or propagates it upward. If not stopped, the event propagates into the DMM (Figure 10). This system follows an approach known as exception unwinding (Cubbi, 2013).

To our knowledge, this mechanism is not utilized in any of the known AS mechanisms, ranging from BT to scripting languages like LUA. However, these are common in high level programming languages like C++, Java and C#. Therefore, we consider it a much-needed extension to our SBT architecture, to be able to provide more expressive capability to our nodes and the AS. This allows us to introduce mechanisms like a Try-Catch construction (Persson, 2012) in C++, where events can be coupled with asynchronous event handlers (Figure 10). For example, a movement node while following a path, engages in one of two situations 1) NPCs is stuck, or 2) player forces a dialog. The movement node throws an event for each such situation and the vents are handled within the parent structures of SBT the movement resides within.

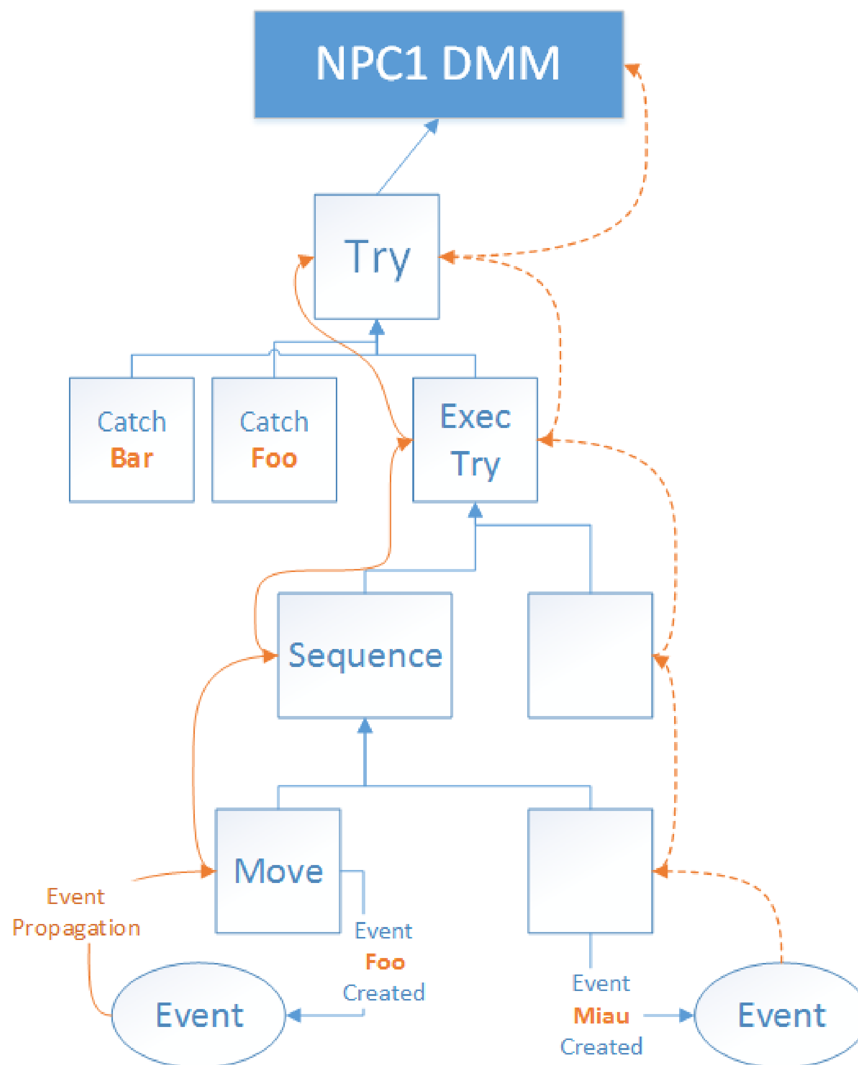

Figure 10: Event propagation for delivering the »Foo« event which originates at the Action node. The event is caught by the Try node and the respective Catch branch is activated. If the event would be something else than »Foo« or »Bar«, the propagation would continue until reaching the DMM.

4.2.5.1 Summary

SBT events is primarily aimed to provide more complex behaviors (Goal 4) for a believable environment (Goal 1). Being able to handle complex internal states, including various failures or execution specifics (e.g. Move node can report why the movement has failed – no path found, target unreachable, NPC stuck etc.). Furthermore, events allow us to produce information within the SBT to be handled by the scripter, which can be manipulated by the already present means (e.g. expression evaluation, data operations etc.) since the event internally contains custom data being SBT variables (e.g. the catch branch of a try-catch can send the event's data as a message[7]). Overall, the SBT events provide our architecture with capability to communicate more information to the scripter and designer about what is happening around the NPC and within the CGE. They have proven to be a

[7] Messages and the Messaging System are addressed in 4.3

valuable tool for debugging purposes, since debug events can carry complex information (e.g. which animation was not found) which can be processed and viewed in our tool-chain.

4.2.6 Data Model

Every language which evaluates expressions (e.g. `x > y + 3`) requires a *Data Model* to manage conceptual representation. Conceptual constructs are commonly represented by organized data aggregations. These aggregations are distinguished by being associated with named *types* (e.g. Human, Actor, Car etc.) (Parnas, Shore, & Weiss, 1976). *Variables* represent a dedicated portion of memory where such data is stored and the type describes the memory's use (i.e. where which type member is stored) (Jackson, 1977).

There are many ways on how to represent data using types, either in a *strong* or *weak* typed form (Aahz, 2003). The classification revolves around being able to identify data types more or less precisely and determine parameter requirements and possible conversions between types that have predictable results (Cardelli, 1991):

1) Strongly typing of data means that every partial data is represented by a fundamental type representation with a given set of available transformations – i.e. conversions. Fundamental types can range from integral types, trough floating point representations up to more complex constructs like 3D vectors and strings. Fundamental types can be aggregated into more complex types to provide a more coherent representation. Typing of parameters is important for proper expression evaluation. Further, a function call uses a definition to specify its parameter's and return value types to avoid invalid use. Type checking manages run-time or compilation checking of proper function use (Liskov & Zilles, 1974). A common example of a strongly typed language is C++.
2) Weakly typed languages are less concerned with data type specifications, where implicit conversions between data happens during runtime. Function calls and data operations may have varying results based on varied factors like ordering, used operations etc. (e.g. `a+b` used on strings has a different result than `a*b` – the addition produces a string conjunction `ab` and the multiplication a transformation to integers). In some cases, there is no concept of type present at all. Often there are some fundamental types, thus they are implicitly convertible. However, this can lead to issues, for example `1+1` may mean `2` or `11` depending on the interpretation and the current state of the given variables. An established representative of a weak typed language JavaScript (Crockford, 2008) or Perl (Sheppard, 2000).

Our data model is focused on a strongly typed data environment. We consider this approach more error prone, easier to debug and simpler to comprehend. From our perspective, having type knowledge provides necessary insight into the design decisions used (i.e. a 3D vector denotes position, and is not used as an arbitrary triplet of floats). To provide more expressive evaluation, we provide *explicit* and *implicit* conversions for data types. As for data type consistency, we chose to make use of a *dynamic* type checking scheme, since we also support introducing variables at runtime. To provide some static analysis, we allow scripters to utilize simple

forward declaration of variables (Wikipedia, 2017). Overall, our Data Model goes is heavily influenced by data models utilized by procedural languages like C++ and Java.

4.2.6.1 Primitive Data Types

Primitive Data Types (PDT) (Fog, 2010) represent the fundamental low level representation of data. We distinguish the following PDT types:

a) *integer* (32bit) and *long integer* (64bit) – represent a whole signed integral number of the given bit width;

b) *float* – represent floating point number with a 7-decimal digit precision, similar to a C language POD (Bazzy, 2012) type;

c) *boolean* – represent the logical true/false value;

d) *string* – represent a dynamic array of 8bit characters;

e) *polymorph* – represent all the above types as one type, able to change between representations, however we abandoned this type in later development to avoid user confusion.

We provide most common conversions between types. In the case an operation would result in data (e.g. float converted to integer) or precision loss (e.g. long integer converted to integer), we provide users with a run-time warning. We do not allow to convert any type to string type to avoid user confusion.

4.2.6.2 Structuring Types

Aggregating types into structures is a common feature of any more advanced language commonly known as Advanced Data Types (ADT) (Liskov & Zilles, 1974). To avoid a flood of PDT variables and user confusion, we utilized the mechanism of creating *structures* commonly known in the C language. However, we do not allow member methods to be declared.

Creating an aggregated type is based on one rule, where every aggregated type has to have at least one member having a PDT or already defined type (Figure 11). Further, every member may have a default initialization specified in its description. If an initialization is not specified, it will be composed of prior existing member initializations. Every PDT type always has an initialization specified in code (i.e. all zero initialization). Members within a structured type are ordered by their appearance in the type's definition.

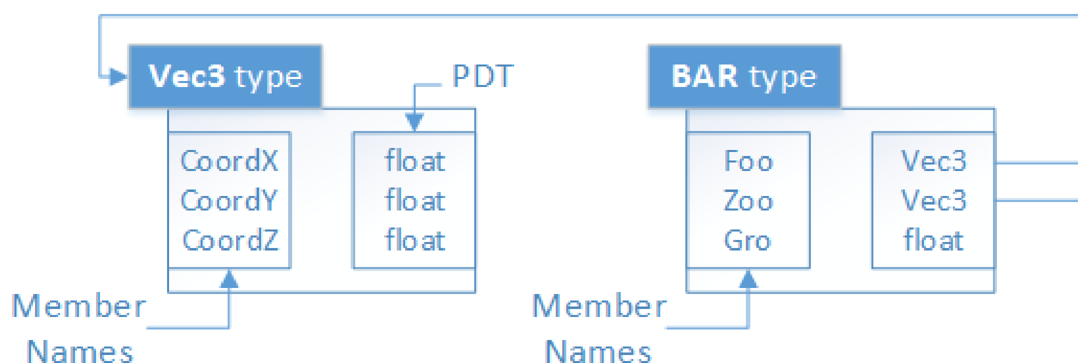

Figure 11: Schema of structuring types based on aggregation of either EDT or already defined types

This allows us to create a more complex representation of ideas, starting from 3D vectors and ending in complex sensory information. In respect to language evaluation, we define a *staged dot description*, where members at every level are separated by a ».« character. For example in (Figure 11) the defined type contains a member of another type, thus accessing the 3D vector is denoted by `$Variable.Foo.CoordX`, where `$Variable` represents the variable of the `Bar` type, and Foo represents the first member being a `Vec3` type. Furthermore, the `CoordX` represents the floating point X coordinate.

4.2.6.3 Inheritance

We also introduce the concept of *Type Inheritance* (Wegner & Cardelli, 1985) where one structure can be inheriting its members from another structure, thus extending it by providing additional members. Languages like Java and C++ build most of their internal concepts around inheritance. Furthermore, inheritance is the key concept for OOD and OOP.

In respect to our language, we allow for a simplistic inheritance model, where one structured type can extend an already existing structured type, by creating a new type, which at least contains already existing type's members. The new child type is not required to add new members. If no members are present, the extending type is a *type alias* of the parent type. We only allow single inheritance, where one type can only have one parent type.

We utilize a colon notation, where the child type has to be fully identified by its parent types. For example if a `Foo` type inherits from a `Bar` type, which inherits from a `Miau` type, the `Foo`'s complete type identification would be `Miau:Bar:Foo`.

With inheritance comes the necessity to provide conversions, both implicit and explicit. Imagine there are two variables – `A` of type `Miau:Bar` and `B` of type `Miau:Bar:Foo`. The `Miau:Bar` type is a different type than the `Miau:Bar:Foo` type. However, we could execute two basic operations onto A and B a) `A = B` and b) `B = A`. In the first case `A` being of a superclass type, we need to implicitly convert `B` to a `Miau:Bar` type, and *data slicing* will occur (part of the information stored in `B` will not be stored in `A`). We support the implicit conversion of types at such occasions.

In the second case, only a part of the `B` variable may be addressed by the `A` variable, due to if being its superclass. This requires a conversion, where the `A` has to be changed to a `Miau:Bar:Foo` type and the extended members will be initialized by default. This results in overwriting the members not present in the superclass of `B` by their default values – *whipping*. This may be considered undesirable and in some languages, requires specifying an explicit conversion. In our case, we avoid converting the `A` into its subclass type and recognizing the inheritance, assign only the members present in the superclass, leaving the subclass members intact.

4.2.6.4 Summary

Having a more complex data model than the common BT or LUA allows us to translate design ideas into a simpler and more maintainable data representation within the SBT language. This allows to more readable code base, where scripters can infer the intended use of a variable by its type (e.g. `Vec3` is commonly used for describing positions). This also allows us to provide users with run-time and static

type analysis in respect to node use (e.g. a node parameter is limited to input `Vec3` and `Foo` types). Overall, the strict typing of the languages data allows us to provide a more believable environment (Goal 1) since the SBT language can be used to express more complex design constructions in an easier, understandable and manageable fashion.

4.2.7 Data Storage and Access

Almost every programming language requires to store data in temporal or persistent form. However, FSM do not use variables at all and express everything in states. To our knowledge, this can lead to too many states if variety of the environment is too complex. Further BT can be without variables, however, similar to FSM they tend to grow into unmanageable proportions (Rasmussen, 2016).

Based on our experience we chose to include typed variables to represent constructs within the AS. This allows us to manage the decision-making process with more ease and utilize common programming paradigms. Almost all users of our architecture will have some experience with common programming languages like Java and C++ thus providing them with a known mechanism on how to express data is beneficial.

4.2.7.1 Variables

Variables can be described as parts of the AS where users store data for either immediate or later use. They can be either *temporary* or *persistent*, having a *volatile* or *constant* content. Temporary variables exist for the purpose of evaluation and are removed after a statement's end (e.g. statement `z>1+x` creates a temporary variable for `1+x` which stops to exist after the statement is evaluated). Persistent variable's existence is not influenced by statements (i.e. they exist within a SBT variable data scope[8]). Constant content of a variable prevents expressions to modify their content. All other variables are volatile in nature. In our case, variables represent instances of the structured types and may take the shape of four different forms:

a) *Single* variable is the simplest form, representing only one instance of the specified type.

b) *Indexed* variable represent an adaptively growing array of instances of a particular type. Indexing of such arrays starts at 0 and is continuous. If an instance is added at an index beyond the end of the array, the void between the latest item and the end is filled with instances.

c) *Associated* variable is similar to an indexed variable, with the difference that their instances are indexed by string names, not being in any way continuous. If an instance is added at an address, it does not create new instances to fill a void.

d) *Custom associated* variable works the same way as an associated variable, with the difference the key being another structured type.

To specify access to our variables, we utilize a simple mechanism of *Variable References*, for both constant and volatile ones. It is a simple adapter pattern

[8] Data scopes are addressed in 4.2.7.2

construction (Freeman, Sierra, & Bates, 2004) for accessing the data stored in variables. It consists of three parts (Figure 12):

1) *Variable specifier* can be simply understood as the variable's name. It also represents a key into an indexing structure holding the variables in the SBT.
2) *Selector* represents the selection of a specific variable instance in respect to the variable's form. It represents the key to search for the variable's content (e.g. in an associated variable it is a string to search by).
3) *Member specifier* is a collection of by-dot-separated member names which coincide with members within the variable type's definition. The collection represents a path within the tree structure of the type. The resulting member is not required to be a PDT (e.g. `$A.Foo.Pos = $B.Bar.Miau.Pos` is an assignment of two 3D vectors within the `A` and `B` variables).

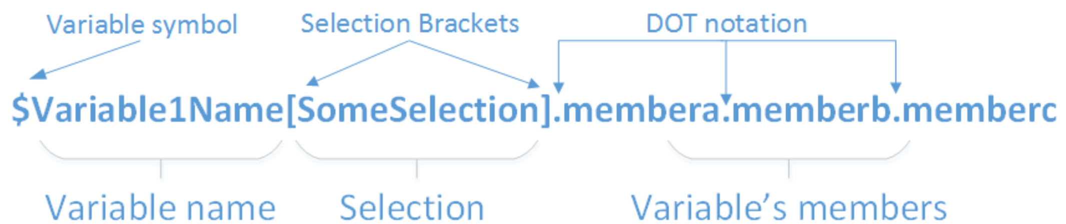

Figure 12: Variable Reference consists of a variable specifier (name), a selector to locate a specific instance within a complex variable form (e.g. array) and a member specifier, which denotes the exact data part of the accessed variable

4.2.7.2 Data Scopes

There are several ways on how to approach storing and access to variables. One common approach is to utilize a Blackboard architecture (Corkill, 1991), where all data, both constant and volatile, are kept accessible for everyone. Variables may be added and removed, requiring coordination either between developers or within code. There may be distinct levels of blackboards – e.g. world blackboard, entity blackboard etc., but the paradigm of everybody being able to modify data in a shared container remains.

We utilized a different approach, commonly used in modern programming languages – *Data Scopes* (ISO/IEC 9899:TC3, 2007). Since SBT language is structured as a tree, it was natural to structure data containers in the same manner – as a tree of data scopes (Figure 13).

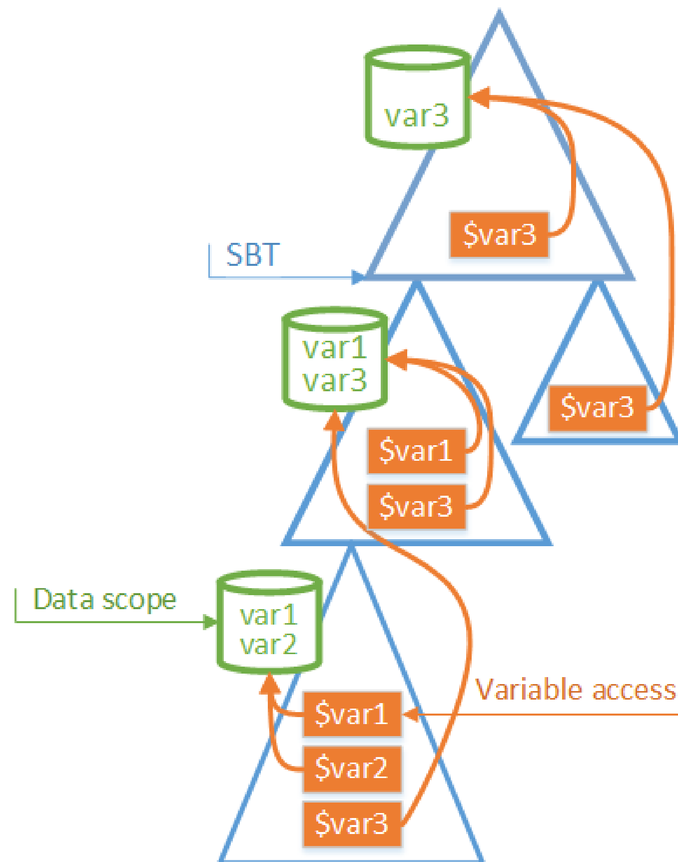

Figure 13: Nesting of data scopes for a SBT and access to that data via variable references. Variables are stored in Data Scopes. These can be at any level of the SBT hierarchy (i.e. at any node). The access searches the data scope hierarchy until a variable is found. If not, it results in a failure to locate the variable.

Data Scopes are constructed as simple data containers, where every data scope has one parent scope and may contain any number of named variables. There is *Global Scope* which has no parent scope. This scope is used to store constant or restricted (i.e. only specific subsystems can alter these) data, which should be accessible to everyone. Further down the hierarchy, nested scopes are present within the DMM and the underlying AS (i.e. SBT). At the SBT language level, scopes are present in every node, similar to data scopes within programming languages like C++. This allows for nodes to introduce data for their respective subtrees. This serves three purposes:

1) *Forwarding data* for later use by subtrees (e.g. a move element provides the target position as a named variable to be used within the subtree as parameters for looking and targeting).
2) *Return values* (e.g. a spawned subtree computes a targeting solution and writes the result into a provided variable).
3) *Temporary* and *Local variables* to avoid polluting parent scopes.

The *lookup* of variables is done in a bottom-up manner, first the local scope of the node is asked, and if not present, the search is moved upwards in the hierarchy. The first match is returned as a result of the search. It may occur that a found result is restricted (e.g. private within the scope), thus leading to an error in access. If any variable name is present in a parent and a child scope, and the child scope is

precedent in the search – the child variable *shadows* (Kildaegi, 2008) the parent variable. While shadowing, it is not required that parent and child variables have the same type, since shadowing is only concerned with variable's name.

One disadvantage of a hierarchical lookup approach is the necessity to search for variables over a possibly large set of containers. We introduce a *lookup cache* where a variable found above a scope's index can be stored within that index as a reference to avoid further lookup. Since it is unlikely that a variable in parent scopes is removed, a consolidation (e.g. clearing of caches) occurs less frequent. This leads to increased memory consumption but provides an efficiency boost. This is most prominent when global constants are accessed. To avoid unnecessary memory consumption, lookup caches may be introduced at key points (e.g. root nodes) within the SBT's hierarchy.

We also define *forward definitions,* where SBT subtrees can specify a variable required to be present in a parent's scope. All forwarded variable definitions are checked prior to using a tree as a subtree for a node. This allows for run-time and compile-time checking for correct evaluation of expressions and correct parameter use at nodes.

There is also the possibility to access variables by their fully qualified name (Weik, 2000) to circumvent shadowing – e.g. a variable named `Foo` is present in a subtree's scope and in the global scope as well. To access the global scope we need to specify a fully qualified name of the variable `global::Foo` to skip the search within the hierarchy of scopes. The naming of scopes is up to developers and scripters (e.g. global, NPC etc.).

4.2.7.3 References and Closures

Variables are named data containers which exists within a data scope which reside within the data scope hierarchy. A variable can exist in two forms in a data scope:

1) *By value* – variable and its internal data containers reside within the data scope. Therefore, any access to the variable from the underlying SBT execution is contained within the data scope.

2) *By reference* – variable's actual internals reside within a different data scope, possibly not even in the actual hierarchy path from the scope having the reference to the global scope.

Variable references may exist in different modes:

1) *Concurrently accessed* variable is unguarded except its constant-volatile specification (i.e. being modifiable). The reference is only a proxy to the referenced variable. However, the reference is not protected against variable destruction and invalidation.

2) *Shared access* behaves in the same manner as a concurrent reference, but in a case of invalidation or destruction of the referenced original, a new original is created instead of one of the references

3) *Copy-on-write* (COW) variables is instanced if the original is either modified, invalidated or destroyed.

Creating of references most commonly occurs while a *closure* (Landin, 1964) is created i.e. a part of the tree structure is copied or moved for asynchronous execution. In SBT an environment's closure is done either by *naming a capture list* (similar to a C++ lambda capture (Bazzy, 2012)) or by *examining* the SBT. Naming a

capture list is an enumeration of variable names which are to be captured from the source environment. Examining the SBT expressions requires an active cooperation of nodes, which have to be able to analyze their content and provide a listing of variables to reference.

Creating referenced variables can provide us with benefits in respect processing and memory optimization. However, it is more demanding on the developer to properly use this concept and avoid hard to maintainable code constructions.

The choice of variable representation within the captured SBT closure depends on either *explicit specification* by the user (per variable), or is based on the intended use of the SBT closure.

4.2.7.4 Summary

Variables as storage is a cornerstone of the SBT language (Mechanism 1). Storing persistent data is imperative to providing complex behaviors (Goal 4) and a believable environment (Goal 1). Without variables, it would be hard to express almost any design decision in an easy to understand and easy to maintain way.

4.3 Messaging

Mathematical models of message exchange (Hewitt, Bishop, & Steiger, 1973) have been proposed to model communication between actors (Agha, 1986) (i.e. universal artificial agents) in a concurrent environment. Communication (Ferguson & Terrion, 2014) has also been identified as one of primary human ways to frame ideas and concepts. From our perspective, designing believable behaviors require a messaging system to allow for relatable problem solutions, both between NPCs as well as within an NPC (i.e. between parallel branches of execution). We have designed a messaging mechanism on top of SBT and our data model, to allow scripters to transfer data (i.e. messages) using several methods (e.g. synchronous sending, mass sending, context specific delivery etc.). In this subchapter, we will discuss the basic principles integrated into the SBT. We also present our concept of *inboxes* and *delivery methodologies*. There are three components to our *Message Delivery System* (MDS) (Figure 14):

1) *The Message* is a data container holding a typed variable which can be used by the sender to provide data to the receiver.
2) *The Sender* provides the delivery system with a message, utilizing an addressing and processing scheme.
3) *The Recipient* is responsible for message pickup and processing. It is also possible to engage in an exchange process with the sender.

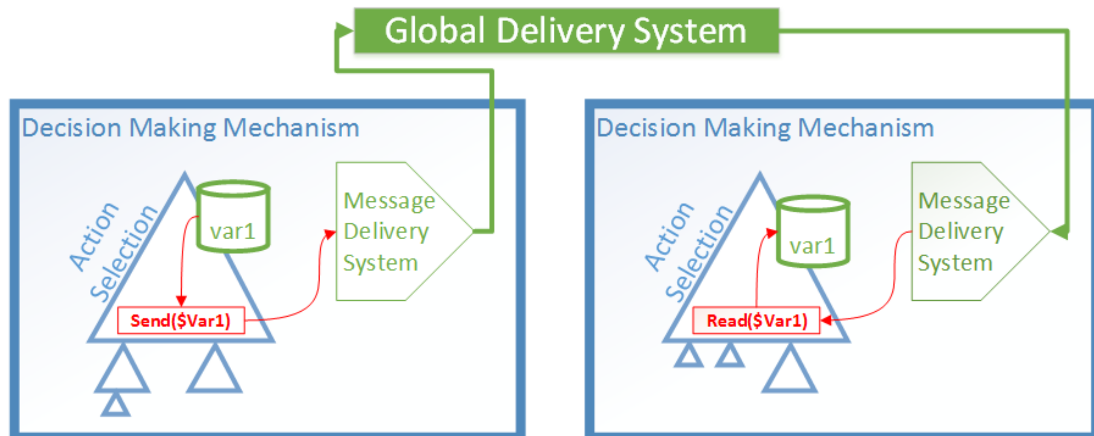

Figure 14: Message Delivery System provides a subsystem to be used by the SBT to deliver messages containing data to another NPC and its respective SBT

4.4 The Message

The message is primarily aimed at providing a vehicle for data delivery from the sender to the receiver/receivers – it holds a typed variable with message's data. If the message is destined for multiple receivers, every receiver is provided with their own copy of the message's data. The data can be stored in a shared reference to avoid any unnecessary copy of the actual source data.

In respect to our scenarios, we utilize messaging almost everywhere. In Scenario 1 (Brian), we use messages to communicate changes in the daily schedule of Brian. One SBT handles the schedule, watching the daytime changes (e.g. morning at 6:00 wakeup time, or 18:00 end of work) and when a different activity is to be scheduled, it sends a message to itself to process the change. Another SBT subsystem within Brian's AS processes these messages and changes Brian's behavior accordingly. Using messages leads to proper decomposition to pass information between parallel processes in a clean and well organized manner.

In our Scenario 2 (Tavern), NPCs who attend the tavern send messages to the Innkeeper to order beer and drinks. The Innkeeper collects and organizes these requests (e.g. by the table the NPCs are sitting at) and sends messages to the waitress about tables that need service. This also illustrates how messaging allows for a *front controller* pattern (Fowler, 2002) to be implemented via SBT using messages. The waitress may get bored over time by not getting any orders to handle, so she start to sweep the tavern's floor.

4.4.1 States

Beyond the data, the message has an internal state (Figure 15):

1) *Ready* – initial state, all data is instantiated and necessary checks are made e.g. receiver exists, data is valid etc. If something would be amiss, the message would switch to a *failed* state.
2) *Sent* – after being accepted by the MDS. It is removed from the sender's competence (i.e. object ownership).
3) *Delivered* – after being accepted by the receiver for processing.

4) *Picked up* – after being accepted by processing and moved into the SBT to be handled.

5) *Processed* – the SBT handler can explicitly state the message's evaluation has finished.

6) *Returned* – the SBT handler may provide a response message to be returned to the sender. The original message is paired with the response.

7) *Dropped* – the message is refused by the receiver at any stage after being delivered.

8) *Failed* – it could not be delivered or was explicitly rejected.

9) Finished – processing has finished. In the case of a return message, both messages enter this state synchronously.

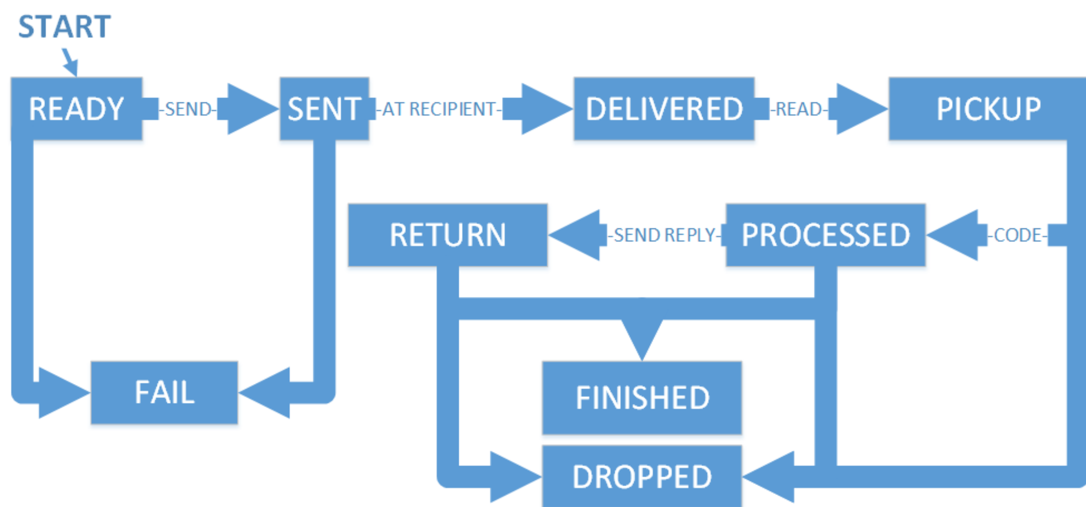

Figure 15: Message states and their respective transitions in the process of delivering the message from Sender to Receiver.

4.4.2 Inboxes

An NPC's internal MDS can handle incoming messages two ways:

1) *Delivery to a listener* – only accepts messages to which somebody is actively waiting to receive – i.e. a »read« node at the SBT. If such listener is not present, the message is rejected. This can lead to issues when a schema of »read and process« in a loop can miss out on important messages just because they have arrived at the wrong time. The upside of this method is the simplicity of use and integration at the SBT. The downside is the necessity to distinguish concurrent listeners at the DMM, and to whom to deliver the message.

2) *Delivery to an inbox* – provides the MDS with a set of user specified inboxes. The MDS provides the recipient's DMM with the message and the internal delivery subsystem runs it through a *set of rules* for accepting or dropping the message. If at least one inbox accepts the message, it is considered to be delivered. Every inbox may be either *limited* or *unlimited* in respect to number of stored messages. Messages reside within the inboxes until they are either cancelled by the sender or fetched by requests

from a SBT node. A »read« specifies the type of message data it wants to fetch from an inbox into the internal SBT memory scope. Therefore, all inboxes have types they may store. Since our types form an inheritance hierarchy, an inbox may contain types and their respective child types, due to looking at the inbox as a form of an array of typed variables.

Further we provide the notion of providing inboxes for subtrees. This provides us with inboxes that are introduced for the lifetime of a given subtree, thus providing the subtree's »read« requests with concrete inboxes to work with. This however leads us to the creation of an inbox hierarchy (Figure 16), where inboxes may be introduced similarly as data scopes. The organization of inboxes into a tree-like structure carries additional problems with which inboxes to prefer in respect to delivery and fetch requests, if multiple inboxes of a similar type are present. We order the new inboxes prior to the existing inboxes, since it is reasonable to expect that these inboxes will be used for processing messages for the particular subtree that introduced them – thus ordering of inboxes is based on their appearance in time, for those which come later have higher priority. Fetch requests inspect the higher priority inboxes first. It is noteworthy that requests are type specific and the internal MDS tries to match provided data as much as possible avoiding data slicing.

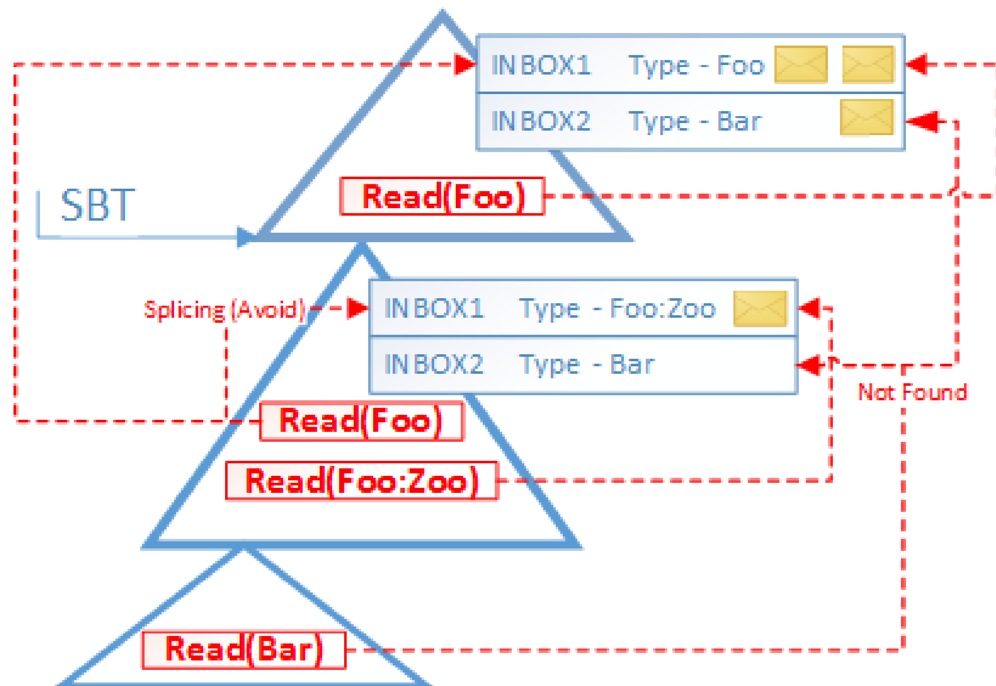

Figure 16: Inbox hierarchy within the DMM. Fetching of messages is based on the hierarchical ordering, where the closest inboxes are checked first

Delivery to inboxes is common in the Scenario 1 (Brian), where notifications about new activities are stored within an inbox and processed sequentially. It would be incorrect to use the delivery to a listener, since the SBT may not have a pending »read« request and the notification would be lost.

Scenario 5 (Battle) illustrates how delivery to a listener can be properly used to simulate believable behavior – commands from the Commander are only received if a trooper NPC has nothing else to do (i.e. he is not fighting other NPCs). Thus, a command can be lost in the heat of battle.

4.4.3 Timeouts

Further the message can have various timeouts:

1) *Delivery timeout* denotes how long a message can wait for delivery. This can be viewed as how long the message is available to be picked up by the receiver's MDS. For a message that is to be delivered based on a predicate (e.g. all NPC in an area), this denotes the time of its existence.
2) *Hang timeout* denotes how long the message is willing to stay delivered but not processed.
3) *The process timeout* denotes how long is the message available for a return message to occur. Timeout's expiration is handled by a failure of nodes involved in the message's processing.

For example, in Scenario 5 (Battle), whilst in battle, the Commander sends a message to some of his troops to engage on left flank. He annotates the message with a hang timeout and a process timeout. The hang timeout will tell him how many troopers are able to fight at the left flank, since the message was not processed, the NPC may be dead or too occupied by fighting. If the message gets processed but is not processed in due time, the order for that trooper will be dropped (i.e. message fails) and the Commander sends other troops onto the flank. It can be seen, that timeouts are a necessary mechanism for more complex behavior to manifest, like the presented *Command pattern* (Freeman, Sierra, & Bates, 2004).

4.4.4 Addressing

The address of the receiver is a vital component of any messaging system. We chose several schemas of how to approach the issue of addressing receivers:

1) *Direct addressing* is aimed at providing the MDS with an exact address to whom to deliver the message, most commonly being a unique identifier. It is not limited to one recipient.
2) *Predicate addressing* is aimed at providing the MDS with a predicate, which when satisfied, will trigger an attempt to deliver the message. Such predicates may address area presence, health of an NPC, performed actions, observed actions, time spent in an area etc.

In our Scenario 4 (Jerry), we use direct addressing to send notifications to NPCs about what the player told Jerry during the quest. Jerry forwards these information to his wife, so she knows how to change her daily schedule to accommodate to the quest. In Scenario 3 (Murder), we use predicate addressing to tell everybody within the radius of a crime that they should come and look. Guards investigating the crime often tell (via message) every citizen within the area of the crime to go about their business.

4.4.5 Processing Schemas

A processing schema is concerned with how messages are sent, processed and received:

1) *Instant processing schema* from a sender's perspective requests the delivery from the MDS, only checking for being valid (e.g. recipient exists, predicate can be validated etc.). The sender's request may be

processed later, but the request is instant in respect to SBT's execution and sender does not care about the message. From a processing view, the message's status is not reported to the sender and effort is made to deliver it. From a recipient perspective, instant fetching of messages does only check and possibly pick up a message from an inbox. If the message is not present, the request fails. Both sender and receiver instant schemas can be summarized as being non-blocking in respect to SBT execution.

2) *Temporal processing schemas* are bound to timeouts in respect to requested operations. From a sender's perspective, a given timeout is provided for either delivery or pickup of a message by the recipient. If this timeout runs out, the message is dropped by the sender. Therefore, the sender must be informed about changes in the message's state, thus the request is blocking in respect to the SBT execution. From a processing perspective, the message changes states as it transits from the sender to the MDS and to the recipient's inbox and is processed within the recipient's MDS and SBT respectively. These changes must be adequately reported. The MDS should be able to recall the message from the recipient, if the sender runs out of time. The timeout is also part of the message to avoid concurrency issues, where the sender wants to cancel a message being processed. From a recipient's perspective, temporal processing is done by a blocking node at the SBT waiting for a suitable message.

3) *Coordinated processing* establishes a contract between sender and recipient about a message being sent, received and processed in a manner where both parties are synchronized. Simply put, the sender's perspective is that the message is being sent in a temporal fashion, where the release of the blocking node happens after the recipient reports the message has finished processing. On the other side, the recipient must process the message as part of it being fetched from an inbox (i.e. the »read« request is a root of a process subtree). The message can be cancelled by the sender until it reaches recipient's processing stage. Further, the recipient can provide an answer to be delivered to the sender if the sender allows this to happen. From a processing perspective, both parties are connected via the established connection of sending and answering. This allows for a cleaner solution to a send-acknowledge schema of communication, where only temporal or instant schemas would not perform well, since the both sender and recipient have no idea about the current counterparty's state. Binding their execution together provides a clear mechanism on what is happening on both ends (e.g. one side's execution is halted due to a hit reaction, the other is notified and both fail).

Instant processing can be utilized for example in Scenario 5 (Battle), for commands by the Commander, where he does not care if a specific command to retreat was followed or not, since the Commander is fleeing himself. In the Scenario 2 (Tavern), temporal processing is used to model discomfort of guests with untimely fashion of handling their orders. If the Innkeeper has too much work to do (i.e. too many orders), he will miss out on some order's timeouts, thus signaling to guests that they should get angry. In Scenario 5 (Battle), the Commander may want specific commands to be acknowledged by his troops, thus he will not issue further orders until they are done processing already issued orders. Coordinated processing

makes it easier to implement a Command pattern with acknowledgments, where the Commander will wait before the soldier has finished processing on his end.

4.4.6 Summary

Messaging represents one of the key concepts for communication between NPCs and between parallel executing SBT within an NPC. Without it, some of the more complex behavior patterns presented above could not be easily implemented or not possible at all. Implementing synchronized execution of two or more BT or FSMs would be rather complicated and would require special modifications to those paradigms. In our case, implementing a synchronized execution takes only one inbox and use of read and send nodes. Messaging is fundamental in respect to our Goals 1, 2, 3, and 4.

Goal 1 manifests in our Scenario 1 (Brian) where the messaging is employed as an internal scripted mechanism to be able to provide adaptable behavior for Brian in respect to his day-by-day activities. Brian's AS responsible for selecting the present activity sends messages to other AS parts to execute it. In Scenario 2 (Tavern), the messaging is employed to facilitate tavern management. Sending messages between customers and tavern's manager provides a believable system where player's interactions (e.g. he distracts the waitress with a dialogue) may trigger believable reactions from other guests (i.e. the waitress is talking thus does not process messages from her manager). Scenario 3 (Murder) employs messaging to convey believable behaviors of civilians who are informed by messages about something happening nearby. Scenario 4 (Jerry) utilizes messages to provide communication between NPCs about how to change daily activities to suit needs of the quest. NPCs are informed via messages what to do next in their day-by-day life to be able to participate in the quest. Scenario 5 (Battle) uses messages to deliver commands to troops in a believable fashion. Commands to troops are delivered via messages and troops occupied with combat ignore them. Killing the Commander puts the troops into disarray, since no one is issuing commands anymore.

Goal 2 is mainly manifested by Scenario 2 and 3. Messaging allows Brian in Scenario 2 (Brian) to change his behavior based on how the environment changes. For example, if the player steals all food from a nearby tavern, Brian will receive a message about the tavern being closed and removes it from his list of favorites. Also, when Brian receives a message from the environment that it started to rain, he will change his daily schedule to avoid working outside. In Scenario 3 (Murder) the messaging system is used to convey information about the emerging situation to the nearby NPCs. If guards identify someone as the murderer, they will send out messages to everybody nearby to chase him. NPCs can spread those messages further to other nearby NPCs until everybody knows that the player killed somebody. However, the information can change slightly so after few broadcasts, recipients only know that the murderer had a helmet and a sword. Thus, messaging provides a natural way how to spread information within the virtual world and how to create new emergent situations based on these information. Guards could also use messaging to query nearby NPCs about seeing the murder and use this information to reason about the suspect. As can be seen, messaging is used to create an emergent world in natural and understandable fashion.

Goal 3 is mainly manifested in Scenario 4 (Jerry). Messaging is used to facilitate communication between NPCs about what the player does and to adapt the quest participant's behavior. The questing manager can use messaging to send commands to NPCs so they can enrich their day-by-day behavior in ways to suit the quest.

Messages can be used to report results of dialogues and even local NPC decisions to the quest's manager to change the quest's structure (e.g. the player is spotted when stealing the healing potion from the herbwoman, closing off some ways to finish the quest).

In principle, Goal 4 is reached by employing the messaging paradigm. Scripters and designers can provide a more complex set of behaviors where NPCs communicate easily with each other. It allows for deeper interactions with the player, since NPCs can coordinate their actions in an intricate fashion. The downside is implicit SBT complexity which is harder to maintain over time. Also, debugging tools within our tool chain have to be able to track communication between NPCs to discover issues and mishaps.

4.5 Synchronization

One key communication mechanism in any concurrently executing system are synchronization primitives (Tanenbaum & Woodhull, 2006) – commonly known as mutexes, semaphores and read/write locks. Since our architecture provides both internal and external concurrency, where internal concurrency is concerned with parallel executing SBT branches and external concurrency is aimed at parallel executing NPC's DMMs, we introduce *locks* and *semaphores* to our architecture. Both locks and semaphores are designed to stall execution until the invariant they guard is reached.

4.5.1 Locks

A lock can be abstracted as a simple device which unlocks after a given set of key turns is reached. In respect to SBT a lock is represented by a SBT decorator node where the node's subtree represents the guarded execution. Our Locks have 5 basic characteristics:

1) *Identification* is a shared information amongst participants, which identifies the lock uniquely.
2) *Participant limit* gives the count of necessary lock participants for the lock to open.
3) *The meeting timeout* denotes how long a participant is willing to wait for the lock to open. If the lock opens, all participants are signaled and continue with their respective SBT execution into the lock node's subtree. Everybody exits in a synchronized manner (e.g. the lock is closed and reactivated).
4) *Reopen timeout* marks how long a participant is willing to wait for the lock to reactivate.
5) *Break behavior* specifies how the lock should behave if somebody leaves it before others finish execution of their respective subtrees associated with the lock.

This mechanism allows for simple constructions where several NPCs synchronize on executing their respective subtrees. This also allows for guarding resources for mutual exclusion, by providing the lock with a participant limit on one NPC.

A simple example for using locks is when two NPCs want to coordinate on an activity at a specific location. For example, 2 soldiers in Scenario 5 (Battle) prime

the trebuchet to fire at the enemy castle walls. Both execute an animation which when started at various times will look bad – they will not synchronize at turning the trebuchet's mechanism. Since they start executing the animation independently, we need to put a lock on top of the subtree where they execute the node which runs the animation. The lock allows for further execution only after both soldiers have reached it in due time. To our knowledge, such synchronization is not possible in any other scripting language for computer games.

4.5.2 Semaphores

Semaphores can be abstracted as gateways with limited tokens at their disposal to be consumed by execution entering the sequence guarded by the semaphore. In respect to SBTs the semaphore is represented by a decorator node with a single subtree representing the guarded sequence. When existing the sequence, the consumed token is returned to the semaphore. If there are no more tokens, the semaphore node stalls the SBT execution until a token is available. Since the semaphore is concurrently accessed, we guarantee that all token acquisitions are atomic. A Semaphore consists of:

1) *Identification* is a unique identification shared amongst participants.
2) *Token limit* indicates how many semaphore subtrees can be executed in parallel.
3) *Wait timeout* indicates how long are participant willing to wait until failing the request to enter the semaphore.

For example, if the scripter wants to limit the number of NPCs present in a room, he will provide them with a semaphore which represents how many NPCs can enter the room. If the semaphore is depleted (i.e. all tokens are consumed), no other NPC enters and they wait for someone leaves. To our knowledge, implementing such behavior in other scripting languages is not possible by easy means or depends on exploiting implementation details of the language in question.

4.5.3 Barriers

A barrier represents a decorator node which waits until a predicate is true, to allow its subtree to execute. If the predicate cases to hold, based on the barrier's setup, it may either halt the subtree's execution or keep it running. Some barriers may wait for certain events to occur either within the NPC, or from other entities (e.g. a navigation spot is occupied). Barriers can be used for simple synchronization between NPCs and the environment.

For example, a barrier may check the predicate if a NPC's health drops below 20% in combat. The barrier's subtree sends a message to the high level decision making of the NPC's DMM. This may trigger getting out of combat and running away. However, the fleeing may not commence until the NPC's barrier for »having an available flee route« is not valid. Thus, the barrier will allow the NPC to stop fighting and run away only after a search for a fleeing route has finished.

4.5.3.1 Summary

Synchronization primitives are a key mechanism to allow development of complex behaviors required at Goal 4. It would be much harder to implement utilization of shared environmental resources without using semaphores. Further, reliably scripting complex behaviors incorporating multiple coordinated NPCs would be much more

demanding without using locks. Where locks and semaphores are more focused on communication between NPCs, barriers are key in respect to effectively communicating specific changes in the environment to a reactive AS. Barriers also proved to be valuable in respect to adapting to internal changes of NPC's stats, like health, stamina etc.

4.6 SBT Actions

The common denominator of all *SBT Actions* is that they have a set of *requirements* for their successful execution and when executed, have a set of *effects* which change the configuration of a virtual world. They either can be a result of AS or be invoked by the environment (e.g. when an NPC is hit by a rock, it causes a hit reaction, which puts the NPC into a ragdoll state).

We can split the actions into two basic categories in respect to their manifestation a) *atomic*, or b) *temporal*. *Atomic* actions are executed synchronously with the DMM of the host NPC. *Temporal* actions take effect over time, either in a *synchronous* or *asynchronous* manner. When an action is executed synchronously, the DMM waits until the action is finished. The asynchronous approach lets the action play out on its own, so the SBT can continue execution. Effects of an action can be either *atomic* or *temporal*. Atomic effects happen instantly. Temporal effects may take time to manifest e.g. moving an NPC.

Atomic actions with atomic effects are the simplest to utilize since they happen instantly and can be evaluated instantly. Temporal synchronous actions with atomic effects are also simple to utilize, since the SBT has to wait for the action to finish until it can continue execution. Asynchronous actions are the most complex in respect to executing them. In our case, we execute the action within the NPC's DMM with the capacity to have its own SBT to handle specific states[9] (e.g. the action wants to attach an object to the NPC).

4.6.1 Run an Action

When an action is executed within an NPC's DMM, it is handled by an Action Manager (AM). All actions coexist within the AM and compete for control over the NPC's facilities (e.g. movement, manipulators, etc.) The AM owns actions and is responsible for managing their lifetime (e.g. construction, initialization, destruction) and facilitates event handlers (e.g. action attaches a tool to the NPC's manipulator).

Actions may either originate within SBT nodes or are forced on the NPC by the environment (e.g. hit reactions). When executed, the action is queued at the Action Manager. Within the AM are *Facility Scopes* (FS) an action can occupy (Figure 17). FS cover NPC's facilities an action can take control over – e.g. movement, full body control, left or right hand, vision, etc. An action cannot be executed if at least one of required FS cannot be satisfied by the AM (i.e. it is occupied by another action). Every FS can actively be occupied by at most one action at any given time. We denote two or more actions competing for an FS to be »in conflict«. The AM's primary purpose is to resolve conflicts (e.g. by terminating a running action or rejecting a requested action) so all the running actions are non-conflicting.

[9] We addressed the asynchronous execution in 4.2.4

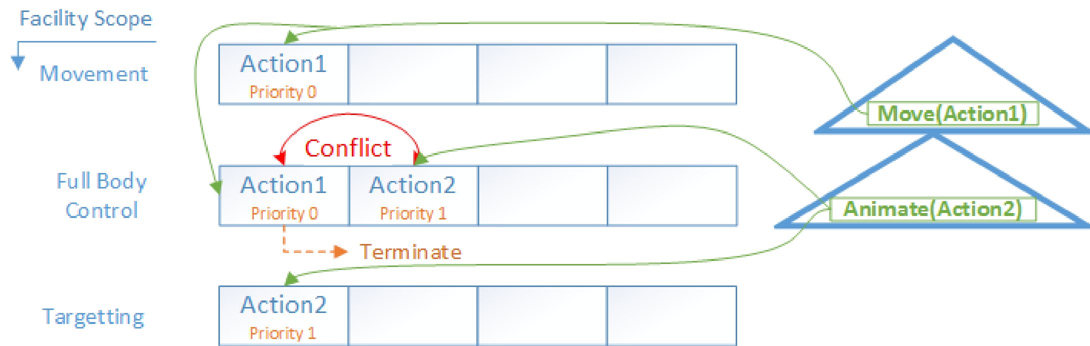

Figure 17: Action Scopes within the Action Manager denote various facilities the Action can occupy – e.g. movement, vision, hands, full body etc. Actions which want to take control over a FS are in conflict which may be solved by terminating the already present action. All FSs for an action have to be satisfied to run the action properly

Actions may be further enhanced by setting a *priority*, where a lower priority action cannot terminate a higher priority action, an equal priority action can terminate an already executed action at the earliest possible point, and higher priority actions can terminate lower priority actions immediately.

If an action is validated for execution and all the conflicts have been resolved in favor of the new action, the action is submitted to the Action Manager and enters its lifecycle (Figure 18).

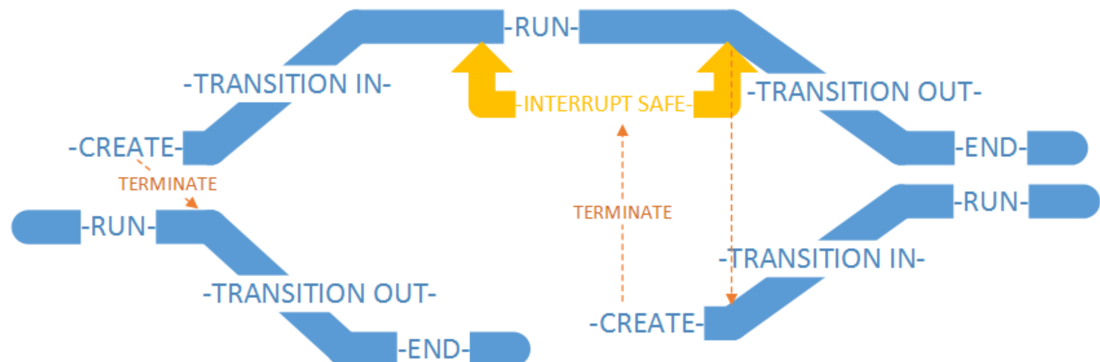

Figure 18: Action Lifecycle starting at the Initialization and ending at being Interrupted or Finish on it own. An action in conflict triggers a termination of a previous action, however it may be postponed due to the previous action being in an Interrupt Safe state.

First every action starts at the *initialization* phase, where the action's internals are set up for valid execution (e.g. the database of action preprocessing is queried to get more information about the action effects etc.). After being initialized, the action is *queued* into the proper FS. If there are any conflicting actions, the AM triggers their termination. Both the old and new action may enter a *transition* phase which denotes a controlled way to go from one action to another[10] (e.g. holding a rock may transition into holding a sword for combat via a »throw rock at enemy« transition). However, transitions over multiple FS are tricky, since the action can only continue to *enter* execution phase if all its transitions have finished. When executing, the

[10] We cover the topic of transitions in respect to BT in our previous work (Plch, 2009) where we discuss the downside of the BOD and BT approaches when transitioning between behaviors.

action may abandon FSs, but it has to retain control over at least one. Control over a FS cannot be reacquired during action execution. The action may finish its execution in two ways, either being *interrupted* or *finishing* on its own. Interrupting happens when the action's conflict with another consequent action has been resolved in termination. Finishing an action means that all scopes are released and the action enters the *end* phase. While ending, the action either executes a transition to a following action or puts the AM's FS to an *idle* state (i.e. no action follows). An NPC can enter a specific *idling* state if all AM FS are idling (e.g. play an idle animations). In the case of equally prioritized actions, we provide designers and scripters with the ability to enter an *Interrupt Safe (IS)* state on demand during an action execution. This state protects an action from being terminated, however postponing the termination to the closest possible occasion (i.e. when IS ends). The IS state is ignored when a higher priority action wants to terminate a lower priority action.

During execution, an action can also trigger an event which is distributed to the AM for processing. These are processed within the AM utilizing either an automated (e.g. play a sound) or designer defined SBT handler. These handlers are running within the NPC's DMM in an atomic fashion, to guarantee minimum delay (e.g. when attaching an object has to be guaranteed with a few frames tolerance to avoid awkward visuals).

Most common example of an action is an *animated action* – which triggered by the *PlayAnimation* node within the SBT. Animated actions commonly occupy the full body and movement facility scope. One of the most common animations are executed for picking up items. The animation triggers an »attach event« which is processed by linking the picked-up object's pivot and the hand bone together. The animation is marked as IS until this attach event occurs, to avoid interrupting the action before the item is in hand. If a follow-up animation would utilize the object (e.g. to strike at somebody in combat), it would transition into the striking animations seamlessly, creating a much more fluid feeling of the two actions combined.

4.6.2 Asynchronous execution

Synchronous actions have a downside in respect to their executing SBT – execution cannot continue to the next node, until the action is finished (Figure 19). So, in our previous example with pickup animations, two actions following each other would lead to the first finishing and going the idle state and the second starting from the idle state. It would be much more believable if the actions would transition from one to another (Figure 20).

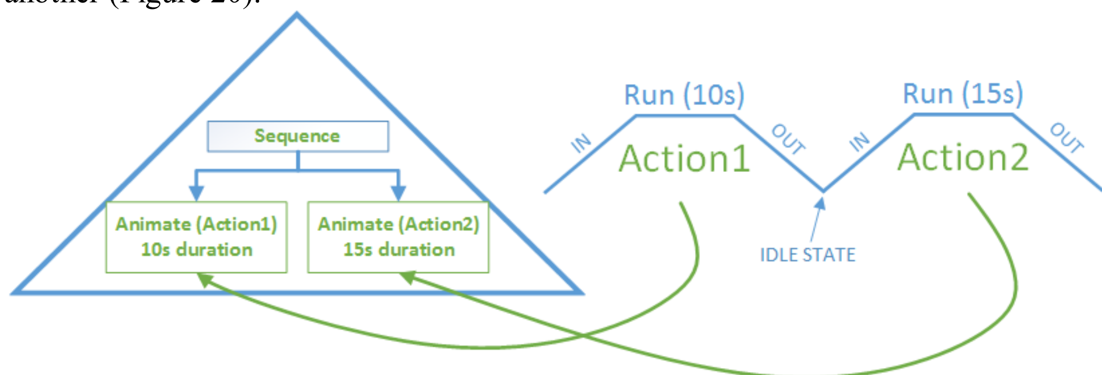

Figure 19: Synchronous Execution of Actions which ends in an Idle state since every action has to end for the SBT to run a node triggering the next action.

To compensate the issue of synchronized actions ending in the idle state, we introduce *asynchronous actions.* Asynchronous actions are executed decoupled from their origin SBT's execution, thus allowing to proceed with evaluation of the SBT tree and possibly triggering another follow-up action.

We also identified the need for asynchronous actions due to the fact that some actions are executed as loops (e.g. looped animations). Since looped actions would run indefinitely, they cannot be executed as synchronous actions and have to be interrupted by other actions.

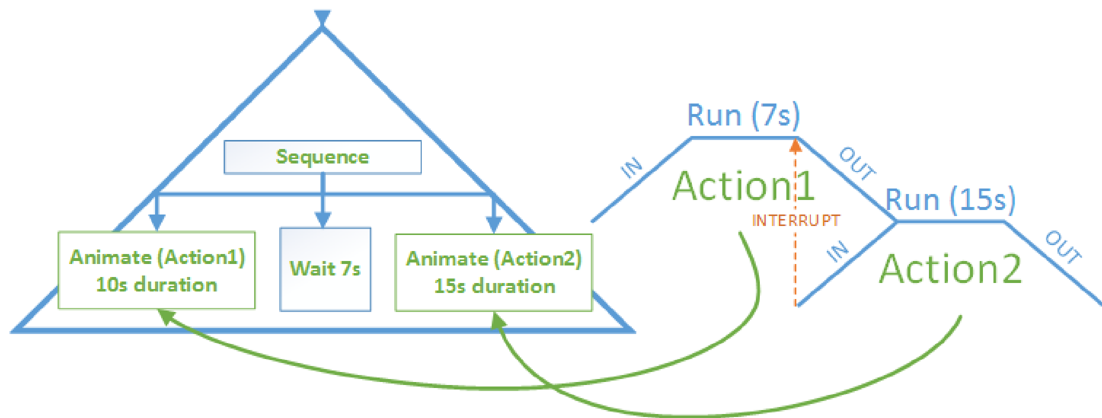

Figure 20:Asynchronous Execution where two actions interleave. The Action1 is started, the AS waits for 7 seconds and Action2 is started, which interrupts Action1. Both actions would run for 10 seconds when not interrupted.

We also allow asynchronous actions to define SBT handlers which are handled as *function closures* (Turner D. A., 2012) with all the referenced variables copied by value into the SBT handler. These handlers are executed (Figure 21) in an atomic fashion within the DMM, however in parallel with all other AS mechanisms.

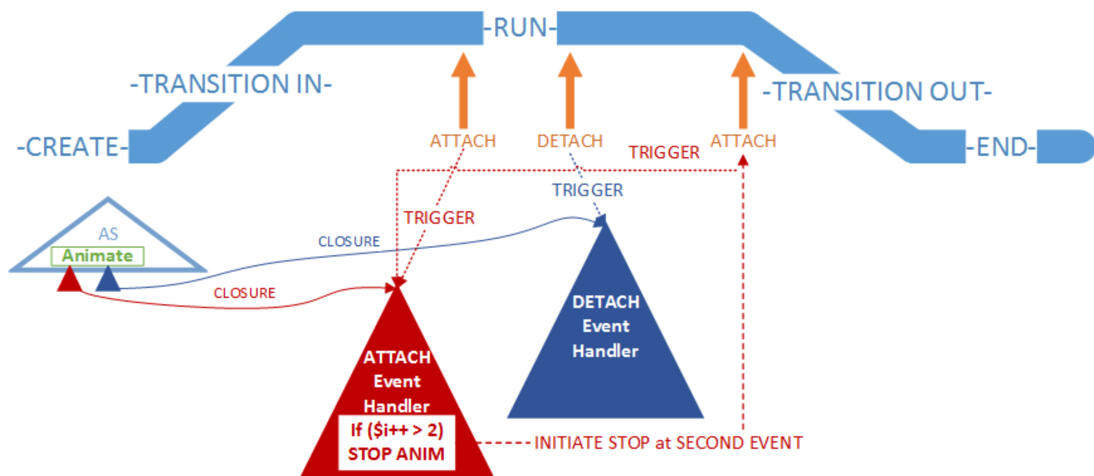

Figure 21: Executing Asynchronous Actions with its event handler closures in a parallel manner within the NPC's DMM. The Action1 has two handlers Attach and Detach, where the Attach handler when executed more than once, terminates the animation. The NPC's AS runs in parallel with the handler code.

4.6.2.1 Summary

Asynchronous actions are a key mechanism in providing a seamless execution of actions, since returning to an idle state may produce undesirable visual or practical results. Transitions between actions are for example key in combat where an action of using an item may transition into combat by throwing the held item at the assailant. Without asynchronous action, the NPC would first put the item into its inventory and then start fighting or running away from the assailant. Executing asynchronous handling of action events also decouples the action from the origin SBT logic, thus promoting a more separated way of designing an NPC's behavior.

4.6.3 Synchronized Actions

In some cases, there is the need to execute actions on different entities (e.g. 2 different NPCs) in a synchronized manner i.e. the action's start is aligned. One example is an animated action, where two NPCs pickup up a wooden log and try to break down a gate.
There are numerous ways how manage it via already presented mechanisms – e.g. create a lock, wait for everybody to enter and the first action following the lock is to be the animated action (Figure 22). However, this works only if everybody gets to execute their animated action at the same time and few frame difference can produce a lot of discomfort in the final visualization.

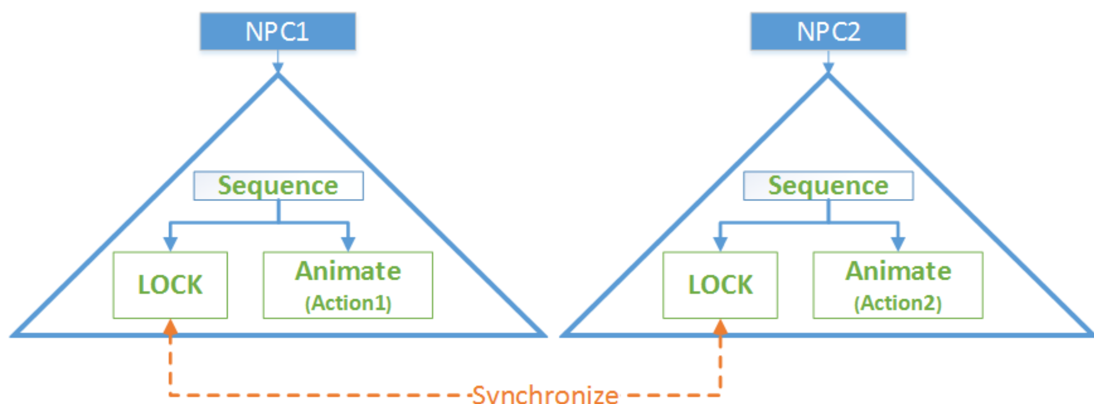

Figure 22: Simple approached for executing animations in a synchronized manner by entering a synchronization lock

Therefore, we require these actions to manifest in a more synchronized manner. Also, we address the issue if a participant (e.g. one of the two NPCs breaking down the gate) terminates his action (e.g. he is hit by an arrow). It may be required that all other participants terminate as well – it would look silly if two men carried the wooden log and one would get killed and it looked like both are still holding the log's ends and running towards the gate.

However, we want to avoid introducing new SBT mechanism, so have we focused on enhancing already present SBT principles. Commonly, actions are created and scheduled at the AM by their respective origin nodes (e.g. PlayAnimation) where the node directly contacts the AM and provides the action.

To allow for synchronized actions, we first introduce event bases startup, which utilized the already presented Event System. When a node wants to trigger an action (e.g. »Pickup« node wants an NPC to pick up a tool with an animation), it creates a »start action« event which it throws into its respective origin tree. This event travels

over the SBT hierarchy until it reaches the top most root node and is forwarded to the DMM, where the AM catches the event and schedules the specified action.

However, if a node along the path from the origin node (i.e. origin of the startup event) decides to modify or consume the startup event, it may do so in respect to being synchronized with other nodes at another NPC's SBT. Thus, a synchronization mechanism already present in the SBT (e.g. locks) is used to delay the startup event until all other participants enter the lock. After everybody has entered their locks, all locks re-throw their respective startup events thus triggering the startup of the action at the same time (Figure 23).

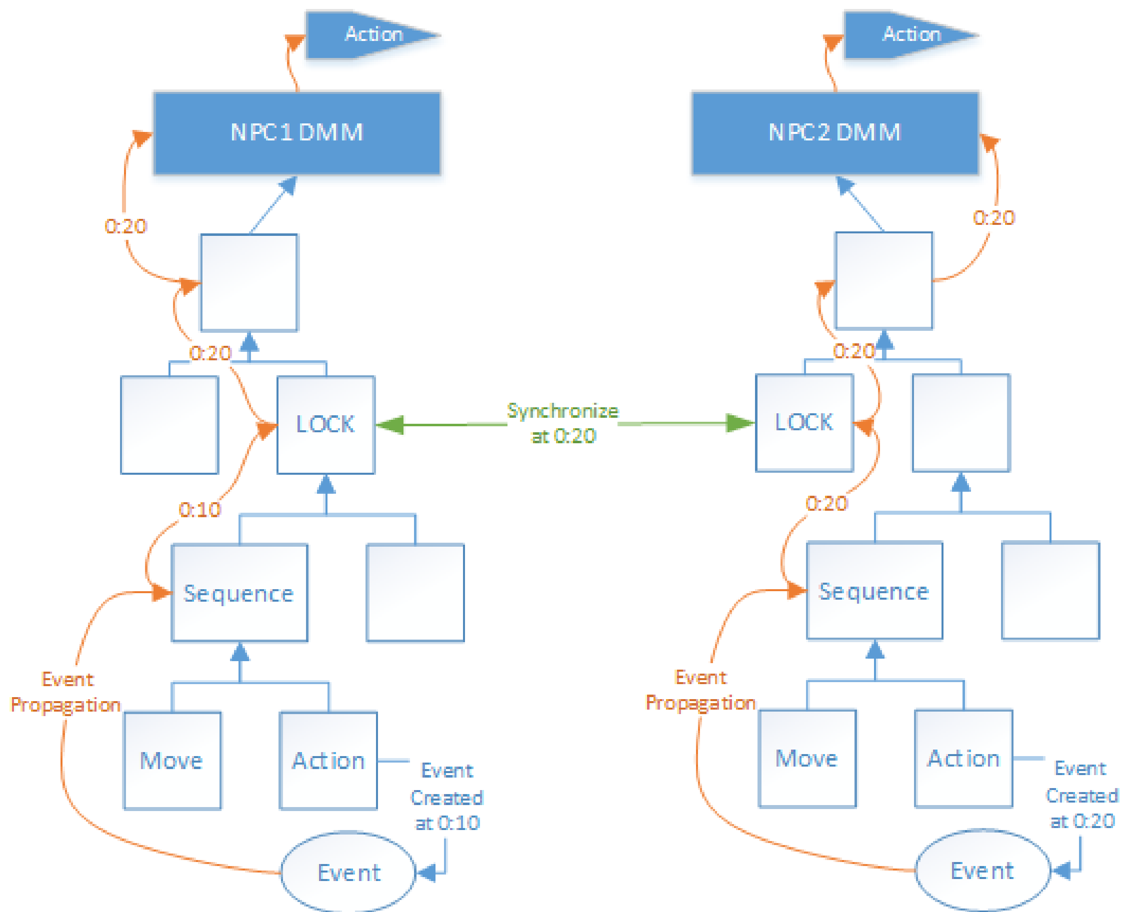

Figure 23: Synchronizing Actions between 2 NPCs which use the synchronization lock as a mechanism to postpone the delivery of the startup event for an animation

One common example is when multiple NPCs have an animation for picking up large items. Since it is unlikely that both NPCs get to their respective spots at the same time, they synchronize at the pickup action. The movement and pickup action are both within the subtree residing under a lock. So, after they arrive, the faster one waits for the slower one to start its action. After both action startup events arrive at the respective locks, the locks trigger both actions at the same time.

4.6.3.1 Summary

Synchronizing actions is a key mechanic in respect to starting a coordinated set of actions amongst several NPCs in a controlled and repeatable manner that is not dependent on the current computational resources (i.e. lower FPS means less NPCs are scheduled to run in a single frame). Further, the mechanism provides an internal

termination semantic for group execution of actions (e.g. if one of the participant's action is terminated, all actions get terminated). This allows for simpler SBT code and a more complex overall behavior (e.g. if one NPC is shot and lets the carried log fall, all other NPCs let go of it too).

4.6.4 Action Chaining

Our action system handles scope conflicts by either terminating the prior executed action or refusing to run a low priority one. However, in some cases it may be appropriate execute actions after each other without terminating the presently executed one – *action chaining*. We implement this concept within the AM as well as a part of the action's inner mechanisms.

In principle, the newly committed action decides on how to resolve the scope occupancy conflict with the prior action – either by requesting it to terminate or wait for it to finish. If no resolve mechanism is specified, the AM reverts to terminating actions by default. In case the new action decides to wait for the current action to finish it enters a *chain* (Figure 24). For example, a movement action can wait for an animated action, if the animation moves the NPC around. However, if the animation does not move the NPC around, movement terminates the action as soon as possible.

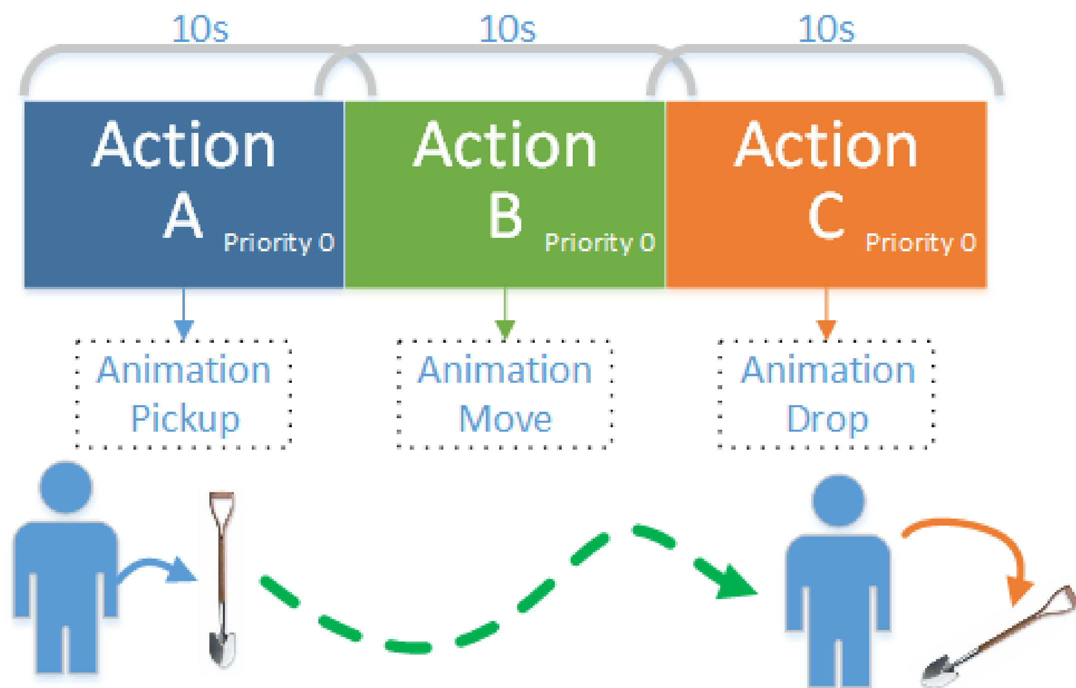

Figure 24: Actions Chains are created by Actions which do not require termination from their predecessors but can work with the results of those actions or even influence them

Within a chain there are three major concepts involved (Figure 25):

1) *Dependency* within a chain means that actions execute with the expectation of the prior action's successful finish. Therefore, actions in a chain can be abstracted as one merged action consisting of sub-actions. However, a conflict between a chain (i.e. at least one action in the chain) and a new action would lead to termination of all actions within the chain.

2) *Forecasting* of action results is applied on actions in a chain, where a previous action provides the next action with effects (e.g. expected end position). Consider the following example of a chain of 3 actions *Animation*, *Movement*. Animation may change the position of the entity, its expected effect will be forecasted to the movement action which can plan the path beforehand based on the provided end location. This pathfinding query can be executed during the Animation's execution, thus spreading the computational consumption over the action's execution. The following transition from Animation into Movement can result in a seamless animation transition (Figure 24).

3) *Inverse influence* represents the capacity to influence an already executing action (e.g. Movement) by a newly scheduled action (e.g. Animation). The effect of the Animation has a set of limitations on where the end point for the movement has to be to properly execute the actual animation (e.g. limiting the approach points only from front and back to be able to align the animation properly). The Animation sends the necessary changes to the Movement action to try to influence the action. However, this may fail (e.g. the movement is not willing to change its end destination) and the influencing action has to either terminate or compensate.

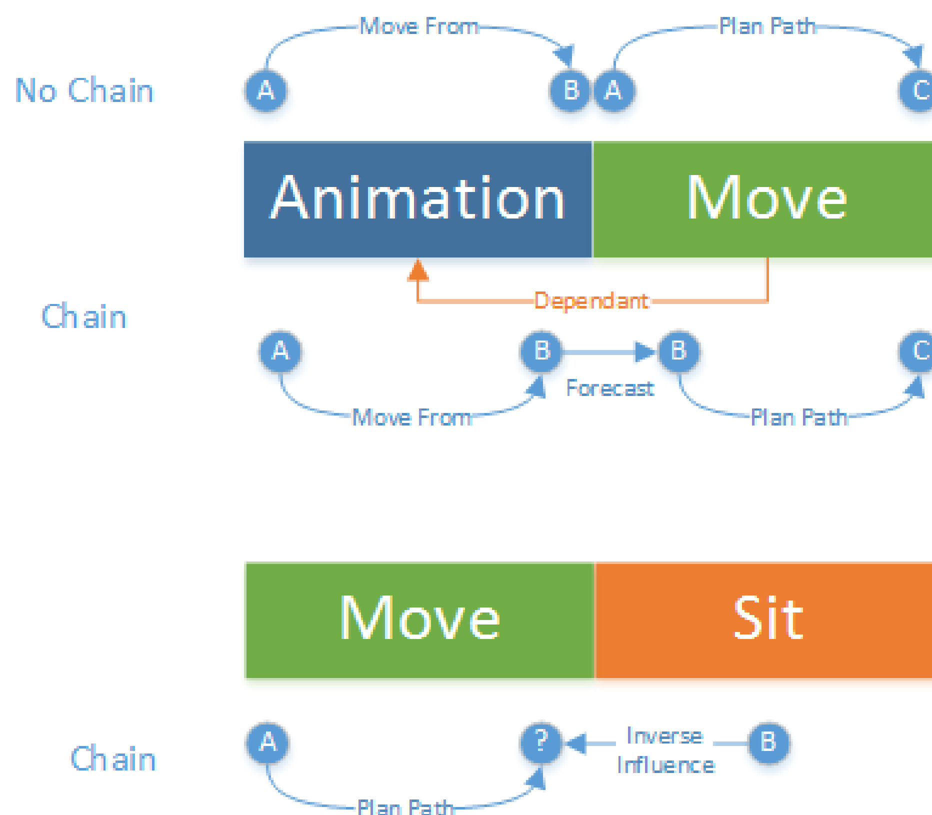

Figure 25: Actions Animation and Movement when not in a chain lead to the animation moving the NPC from A to B, where the move when executed before the animation ends, searches the path from A to C. Forecasting provides the move with the end location of the animation. When a Movement is followed by a Sit, the sitting animation provides the necessary target position for the move to plan to via inverse influence.

4.6.5 Move and Act

Moving and *animating* is one of the key examples of action utilization and chaining. The movement of an NPC requires having at least a valid start and end position to plan the path in-between. However, the end position may change over time, thus triggering a creation of a new path to follow. Since movement is an action within the AM system, it has its preconditions (e.g. valid positions, existing path etc.) and effects (change of position over time, final position of the entity within a tolerance etc.). Playing animations is utilized by designers to choose a specific animation for the NPC to play (e.g. sit down on a bench, jump over an obstacle), which has preconditions (e.g. existing animation, enough space to execute it without collisions etc.) and effects (e.g. NPC ends up sitting on a bench).

Combinations of both actions is a common pattern in the NPC's execution where all the above principles manifest. We will inspect the following combinations of *Moving Action* (MA) and *Playing Animation Action* (PAA): a) {PAA, MA}, b) {MA, PAA}, c) {PAA, MA, PAA}.

When a PAA precedes a MA, the MA may choose to enter a chain with the PAA if a set of conditions is met: a) the PAA is not a looped action, b) the PAA moves the entity around. If the PAA is a looped action the MA may choose to wait for the loop to finish its current iteration and chain with the action at this point requesting a synchronized termination. If the PAA moves the entity around, it would be complicated to chain with the PAA and determine the exact position or state the action would be in. If this is possible the MA can request a termination at the given point being reached. After the PAA finished, the MA can chain its execution to the position being reached and continue with its execution. The MA, when chained, utilizes the forecasting mechanism to determine where the PAA will end to plan its path accordingly from the reached point (Figure 26).

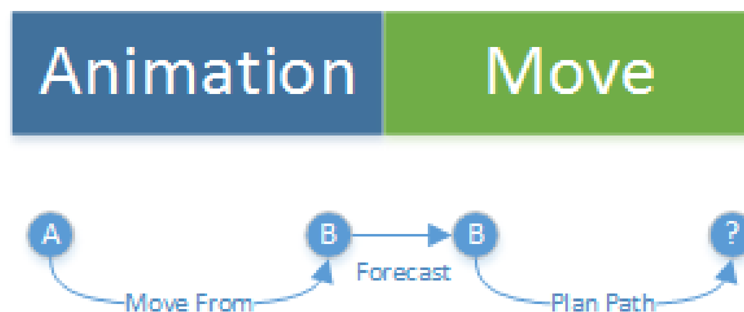

Figure 26: The Animated Action provides the Movement action with a forecast on the end of the animation, allowing for movement to pathfind during the execution of the action taking the end location as the start position for the path. If these actions would not be chained, the movement would be planned from the current location of the NPC when the movement action was invoked.

When the PAA precedes the MA, it will chain with it by default. However, depending on the situation, the PAA may have a limited spectrum of animations to play at the end location of the MA (e.g. there are only 2 available aligned animations to sit down on a bench), thus the PAA will inverse influence the MA to change its target to reach the proper destination to execute the PAA (Figure 27). The inverse influence may repeatedly happen, since the conditions the MA tries to reach the end location may change (e.g. the NPC gets injured and the available animations change with their respective positions to align from). If the PAA and MA actions are created

and chained at the same time, the PAA can influence the initial setup of the MA based on its internal demands on the end location, avoiding an on the fly re-plan of the path.

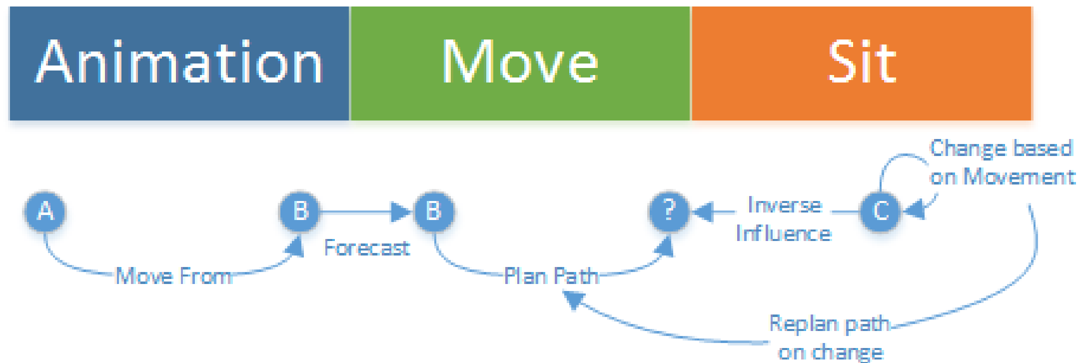

Figure 27: Adaptive inverse influence by the animation to change the destination of the movement action to suit the necessary alignment requirements. The movement goes in the general direction of the Bench, which has two points from which the Sit animation can be executed (Front and Back). Since path planning may get the NPC from either direction, the Animation Action (Sit) monitors the progress and chooses the end destination (i.e. either front or back) when movement gets close enough. After the move reaches the end location provided by the animation, the animation already knows which animation should be executed.

The last case of an MA being in the middle of two PAAs leads to the combination of both approaches, where the MA is both influenced at the start and end location by the chained PAAs. The MA can optimize its start point based on the first PAA – for example if first PAA would jump over a fence, it may shorten the actual MA by a substantial distance (no need to walk around the fence). Thus, the MA can wait for the PAA to finish jumping to execute a shorter path to its target. It may also happen the other way around, where the MA may terminate the PAA prior to finishing the jump, to avoid going back around the fence.

If any of these cases would be executed in a non-chained approach, the end result would look much worse. For example, when reaching a bench to sit down, the NPC would first come close to the bench, then get back to the required alignment location and then execute the animation. The back and forth communication between actions s necessary to provide a well-orchestrated and seamless execution.

4.6.6 Summary

Proper action manifestation is as much important to deliver believable behavior as proper action selection is. Therefore, having a seamless, well executed set of actions is key to achieving our Goal 1 and 4. The necessity of chaining actions is key to delivering a believable visual performance of NPCs to avoid the feeling of robotic execution. The transitions between actions with inverse influence provide the illusion of actions depending on each other, thus the deliberation process manifests in a more complex fashion. Actions allow us to further decompose behavior into isolated components (i.e. actions) on both the code as well as the script side.

4.7 SBT Summary

We presented our SBT architecture aimed at providing low level decision making for NPCs within a large scale OWG. The SBT architecture with the supportive mechanisms (i.e. messaging, synchronization primitives, actions) focuses on our Goals 1, 2, 3 and 4. We focus on enhancing the believability of the presented ambient virtual life in an emergent and story driven environment. Our focus was to maintain and promote decomposition and encapsulation on both the architecture's as well as level design side. The presented SBT architecture allows to introduce more complex behaviors utilizing complex and more generic SBT constructs (e.g. messaging, actions etc.) allowing for a more fluent transition from design to in-game script.

In our implementation for the KCD game the SBT language's resemblance to a OOP language has proven as a benefit, since most developers and content creators are familiar with the OOD and OOP principles[11].

[11] Chapter 8 focuses on evaluating developer feedback.

5 Decision Making Mechanism for NPCs

Conceptually we view the Action Selection (AS) being a low-level component which transforms inputs (i.e. configurations of the world) into outputs (i.e. actions) in a streamline, program like fashion. Having a believable AS at the low level is key in making the proper believable choices in respect to short term decisions, reactions and executing predesigned sequences of actions to achieve a task the NPC has committed to (e.g. go to work). From a design perspective, the low-level AS is only responsible for behavior specifics, e.g. how to acquire tools for work, how to get home etc. The downside of such approach is often a rigid structure which results in repetitiveness and lacks long term structure or continuality. Therefore, to achieve more believable and long term focused behavior, we introduce additional higher level mechanisms into the Decision-Making Mechanism (DMM), to provide a feeling of a deliberating and thinking NPC.

The DMM represents the enclosing mechanism for Action Selection (AS) to reside within. Thus, it is only natural to decompose the mechanism into a tiered architecture. In principle, we aim at following simple decomposition principals, similar to what we learned from Hierarchical Task Networks (HTN) (Erol, Hendler, & Nau, 1996). In principal, HTN utilizes the notion of designed problem decomposition into subtasks which can be further decomposed, until atomic operations are reached. The key concept is that decomposition of tasks is done by a designer with domain knowledge who provides the DMM with a structure a human may come up with when solving a problem. Solving high level goals by arranging tasks into a plan is done by a planning mechanism. Tasks are principally structured similar to STRIPS (Fikes & Nilsson, 1971) actions, have preconditions and effects, thus can be arrangement by various different methods, ranging from classical planning (Ghallab, Nau, & Traverso, 2004) to hybrid approaches (Kambhampati, Mali, & Srivastava, 1998). The key difference to classical approaches is the designer's domain based knowledge which act as a form of principal heuristic, which allows the planning mechanisms to approach problem solving in a much more structured way.

Our principal inspiration is the decomposition which goes from *goals*, to *task*, to *actions*. A goal can be viewed as a long term (e.g. life-long) or short term (e.g. for one day) commitment to satisfying a given set of conditions e.g. »be at work«, »do not feel hunger«, »feel safe«. Since our NPCs live their life on a day-by-day basis, we avoid overreaching their ambitions to longer than one day purpose. We abstract goals as parts of a *Day Plan* every NPC has for every day. The NPC tends to follow the plan, utilizing its environment to satisfy goals in a decomposed manner, one task at a time. The tasks are executed by the AS at the lower levels of the DMM.

We further decompose the functioning mechanisms of the DMM into dedicated components for different areas of expertise e.g. combat, day-by-day life, player interaction etc. Our practical reasoning behind adding a layer of functional decomposition is that distinct types of behaviors have different requirements, as well as various subsystems they employ. For example, when in combat, the NPC may employ anticipation, tracking and targeting subsystems, which are of no use in a day-by-day life. This decomposition of functionality is motivated by the notion of having a specific mindset for different situations.

In this chapter, we will present our NPC's DMM architecture with our high-level plans and their executions. Our primary focus is on providing solutions to our Goal 1 and Goal 4, where we focus on providing more believable and complex behaviors by

adding the illusion of NPC purpose and long term reasoning. Introducing Day Plans also suits our Goal 3, since long term goals and tasks can be utilized by a questing mechanism to modify NPC behavior to suit a quest's needs.

5.1 Architecture

To provide a more believable behaving NPC, we decided to structure the DMM into a three-tier architecture, where levels range from abstract to specific. We are also concerned with the modularity of our approach, where components within the architecture can be added and removed on-the-fly during runtime. One of our motivations was to structure the architecture in a comprehensive and easy to understand way, to be able to employ it in an industrial application of the KCD game's ambient environment. The architecture is organized in a tree like fashion (Figure 28):

1) *Brain* represents the high-level manager, containing shared subsystems, planning mechanisms and a set of by-priority-ordered SubBrains.
2) *SubBrain* (SubB) is a mid-tier DMM component which focuses on managing a specific area of decision making, ranging from day-by-day life to combat. The SubB contains a set of Action Selection mechanism
3) *Action Selection* represents the low-level decision making manifested by a SBT or any other similar technology.

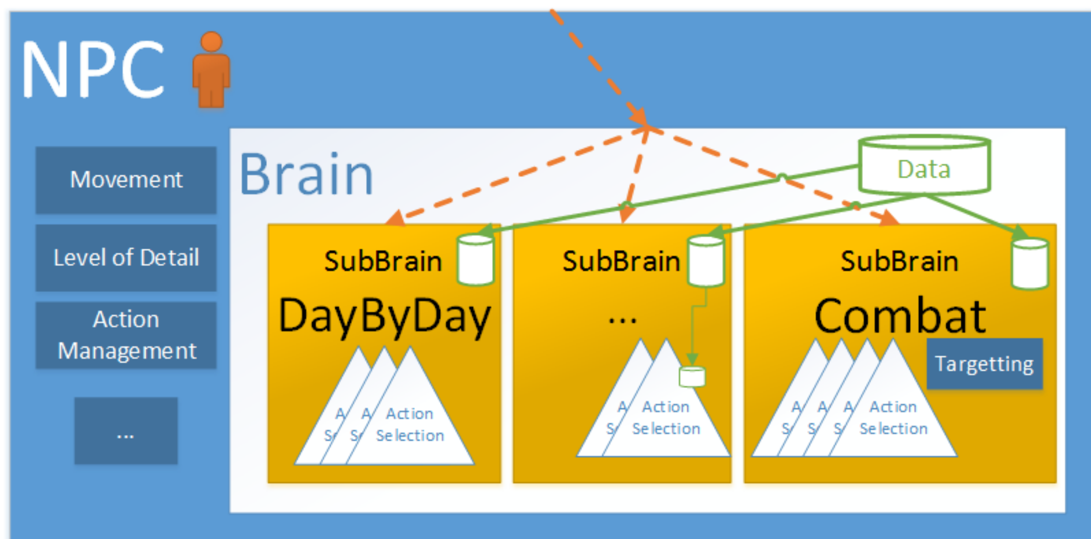

Figure 28: NPC Architecture divided into the Brain, SubBrains and AS mechanisms

In principal, we employ *graceful governance* at our architecture – higher level tiers within our DMM (including the SBT's nodes) are responsible for execution of lower level tiers. Lower level tiers are utilized to accomplish the high-level goals managed by their respective tiers. A higher-level component can terminate a lower level component in two ways – 1) *graceful,* where the lower level tier has time to ends its execution in a meaningful but timely fashion, and 2) *abrupt*, where the higher tier terminates the lower tier's execution without any time to bring local

affairs into order. Thus, the lower level tiers have to be able to terminate instantly without compromising the NPC (e.g. putting it into an undefined state).

5.2 Brain

The *Brain* represents the top-level tier responsible for managing NPC's DMM subsystems and the SubBrain layer. There is a multitude of support subsystems present in the Brain, mainly pathfinding, combat movement, level of detail management, and many others. The Brain also holds the top-most DMM memory scope and the messaging subsystem. It also contains a *high-level planning* mechanism which creates the day-by-day plans for the lower tiers to handle.

The Brain's core functionality is to provide SubB management. There is a set of by-priority-ordered SubBs, which may run either *exclusively* or *concurrently* to each other. The Brain is responsible for maintaining the invariant that no two conflicting SubBs run at the same time. Thus, when two SubBs are exclusive to each other, the higher priority one prevails. A SubB can be activated either by a different SubB or by an automated trigger.

There is one SubB called the *Switching SubBrain* (SSubB) which can run concurrently with every other SubB. The SSubB evaluates external stimuli, percepts and internal state of the NPC and decides which other SubBs to activate. For example, if the NPC perceives an enemy NPC, it triggers the activation of a Combat SubB (CSubB). However, if the NPC is hit by something (e.g. an arrow), an automatic trigger activates the CsubB to handle the threat. But most of the time, the NPC lives its life having the Day-By-Day SubB activate, which handles the necessities of the Day Plan produced by the Brain's planner.

5.2.1 Day Plan

One of our key motivations for making NPCs more believable is to create the illusion of long term deliberation which can be explored or investigated by the player. From an OWG perspective, NPCs require having at least some sort of mid-term planning perspective to avoid being only reactive to what the player does. In our architecture, we utilize the Brain as a high-level decision making platform to provide goals for lower tiers of the architecture.

Since our NPC are day-by-day oriented, we focus on providing them with a mid-term plan specifically tailored for their role within the world (e.g. a baker differs from a soldier). These plans consist of *Activities* (Figure 29) the NPC has to follow in respect to the capabilities of its local environment.

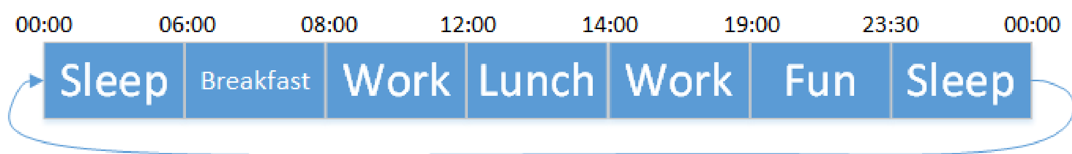

Figure 29: Day Plan of a common peasant NPC. The Goals are annotated by simple strings which are evaluated within the Day-By-Day SubBrain

The Activities, being a basic building blocks of a *Day Plan* consist of: a) *start time,* b) *end time*, c) *start variation*, d) *end variation*, e) *start tolerance, f) end tolerance*, and f) *priority*. Start and End time specify when the activity should be scheduled. Variations are used to add a sense of randomness to the world. Activities are stacked into priority lanes (Figure 30). If an activity of a higher priority is to be

activated, the NPC is notified and switches to executing it[12]. The start tolerance is used to avoid starting an activity when a known higher level activity will happen soon enough. The end tolerance is utilized when resuming activities which end time is close, to avoid resuming activities when there is no time to finish them.

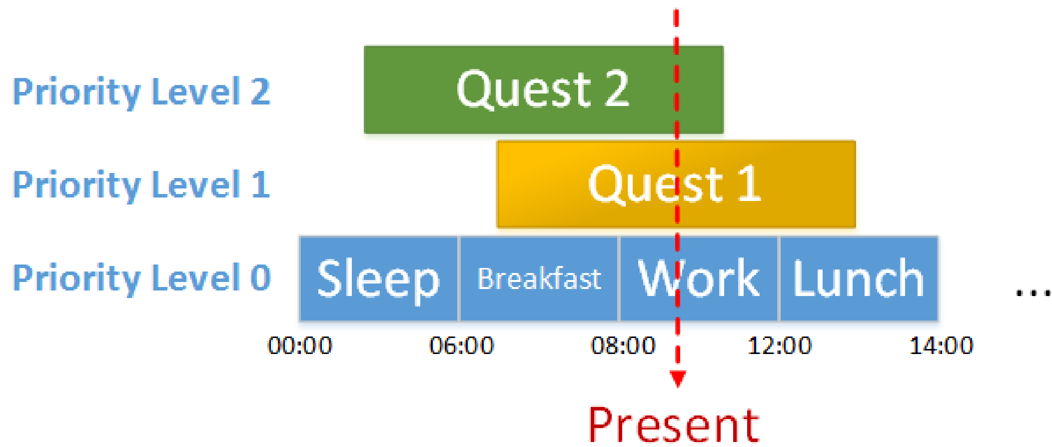

Figure 30: Activities are organized into priority lanes where the activity with the highest priority at the current time is to be executed.

When the NPC's internal clock reaches midnight, a Default Plan is created for the next day. The planner takes the NPC specific set of Activities from a designer filled database and puts them in a given order based on their start/end time applying start/end variations to those values. The Activities are scheduled back-to-back and if a time gap is created, both ends of an activity are extended to meet in the middle of the time gap. This ensures that the entire day's plan is covered and no idling should occur. The initial activities have the lowest level of priority. The Default Plan ensures the NPC has no downtime and it looks like it is always doing something meaningful.

When an Activity is activated (i.e. it should run), the Brain sends a message over the MDS to the NPC itself. There is an in-build message inbox within the Brain's messaging context, which stores these messages. If a running AS picks up the message, it can handle the situation accordingly. In most cases, this is the responsibility of the Day-By-Day SubB. Using the messaging approach allows us to handle Activities after the NPC has been doing something with a higher priority (e.g. combat).

Changes to the plan may occur from various sources, either from within the NPC, when a SubB decides that there is something more important to do (e.g. recover wounds after combat) than following the Default Plan, or from an external source (e.g. quest system needs the NPC to go somewhere). These alterations are called *patches*. The invariant for adding patches to a running plan is that at every priority lane no Activities can overlap (i.e. a conflict). When a patch is committed to the Day Plan, it turns into an Activity.

If the higher-level priority activity is not active anymore, the system reverts back to a lower priority activity. However, it would be unreasonable to return to a lower level activity if there is very little time to finish it (Figure 31) – e.g. the default

[12] We use messaging to deliver the notification to the Switching SubB to handle the change

activity was working on the field, a quest provided a patch to go visit a lover and it ended 5 minutes prior to end of the work activity.

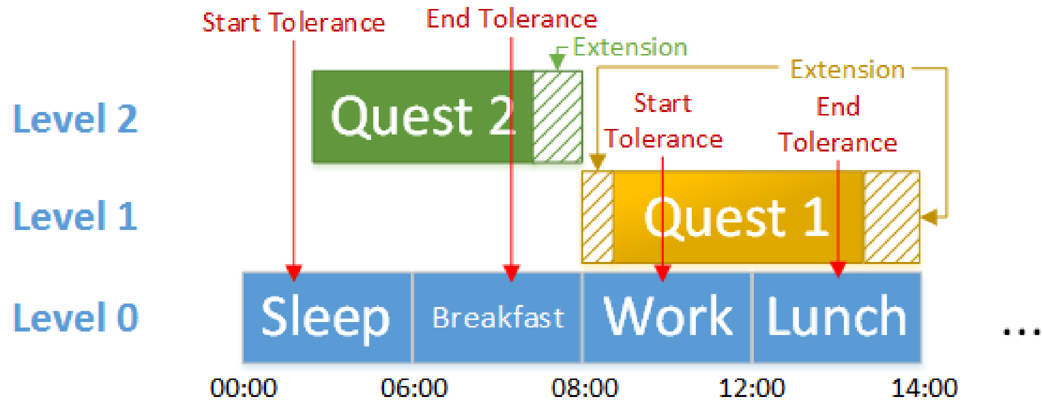

Figure 31: Start and end tolerances change the chosen activity based on how close a higher-level activity is to either the start or an end of another activity that would be executed prior or after it.

We solve this issue by specifying the end tolerance where the planning mechanism looks at the remaining time and if it is lower than the tolerance, it continue with the next activity on the priority lane. This mimics the human approach of not doing work when there is not enough time to finish it. We observed a similar approach in ENTs (Brom, Lukavsky, Sery, Poch, & P.Safrata, 2006).

5.3 SubBrain

A SubBrain (SubB) denotes a dedicated mid-tier component within the DMM hierarchy, which is responsible for managing a certain type of activity – e.g. combat, day-by-day life, social interactions with other NPCs, dialogues, quest specifics, action handlers etc. Every SubB provides an environment for the underlying AS mechanism (i.e. SBTs). For example, the Combat SubB (CSubB) provides the SBT with a dedicated targeting and target tracking subsystem with the capacity to make automated reasoning about the target's proximity and estimate movement.

SubBs within a Brain is specified as a tuple `(prio,CON,EXCL)`, where `prio` specifies the priority within the Brain's set of SubBs. The `CON` enumerates SubBs which can be run concurrently (Lamport, 1978), and `ECXL` enumerates exclusive SubBs which have a conflicting execution. Concurrent SubBs are considered those who do not require the same facilities at the Brain level, for example the Day-By-Day SubB can run concurrently with an Animation Event handling SubB. Exclusive SubBs cannot be executed at the same time within a Brain, since they will clash on use of some exclusive resources, for example movement of the NPC is used by both the Day-By-Day SubB and the Combat SubB. Therefore, the Brain manages the execution of SubBs in respect to favoring the higher priority ones which are non-conflicting.

There are three groups of states a SubB may take:

a) *Inactive* – the SubB does not compete for NPC control, it only checks for its activation predicate to be valid

b) *Active* – signals the Brain the SubB wants to enter a running state and acquire control over NPC facilities. Active states cover the transition from

being Inactive to Running. The SubB has to be Queued, the conflicts resolved when the SubB is about to run. When transitioning back to being Inactive, the SubB may go through a Suspended state, when a same priority SubB is in conflict.

c) *Running* – the SubB takes control of the NPC's facilities while its internal AS is being executed. SubBs in this state receive a fair share of the update time for their internal execution. A SubBs execution is guarded by a Switch In and Switch Out state, where the SubB handles initializations, transitions and cleanup.

Transitions between inactive and active states are in full control of the SubB while the Brain controls the transitions to running states (Figure 32). This system is inspired by previous academic research (Plch, 2009).

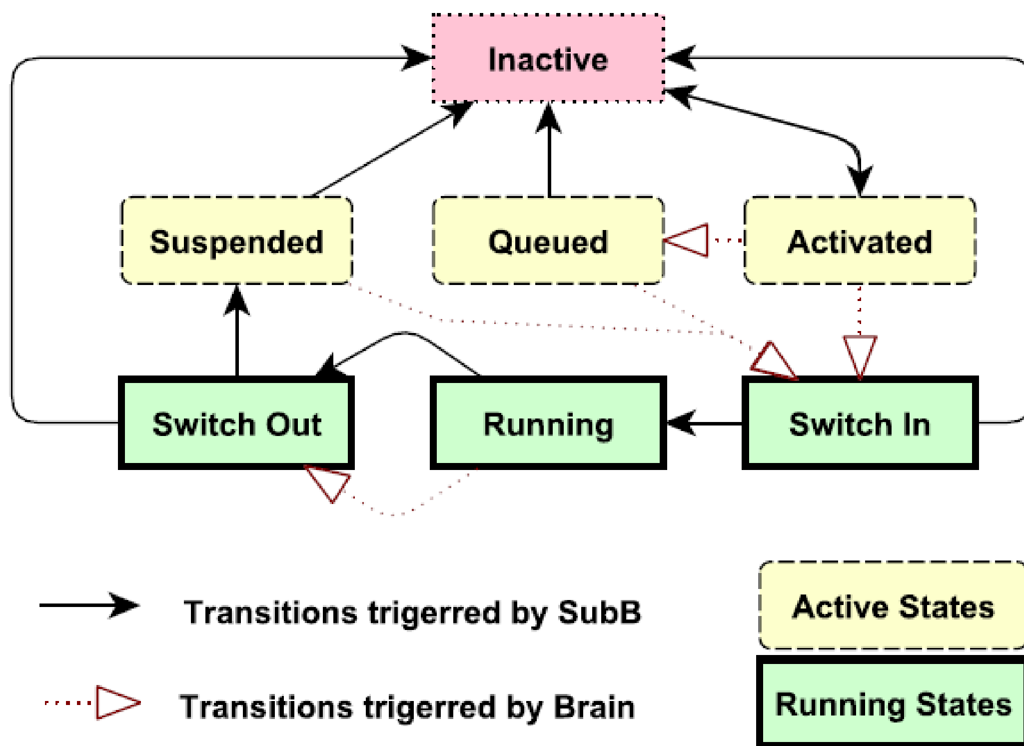

Figure 32: Every SubBs default state is the Inactive state, into which it may enter when an activation by the SubB or external source (e.g. other SubB, environment etc.) is requested. The SubB is Queued to run. While waiting in the Brain's queue, all conflicts are resolved. Before entering and after leaving the Running state, the SubB transitions over the Switch In and Switch Out respectively. A running SubB may enter a suspended state if the conflicting SubB is at the same priority level.

5.4 The Player

One key concept we utilize in our ambient architecture is the notion of the Player being part of the world. In most games, the Player represents a singular special entity which is apart from all other NPCs. In our case, we take a different approach and view the Player as another NPC having a human decision making as well as a virtual subconscious. In principle, the Player exists as another NPC within the world,

sharing the same mechanisms and capabilities as any other NPC. The Player's avatar actually has its own Brain-SubBrain-Action Selection (BSA), which can take control over the Player's NPC if necessary or required (e.g. Player triggers a context action like opening a door or shooting a trebuchet). It also provides us the benefit for NPCs to view and communicate with the Player in a more systemic fashion e.g. send messages to his BSA to trigger responses, which can be evaluated by his AS.

An example is the Player going to sleep in someone's house. When the NPC that wakes everybody up wanders the house, it wakes up both the Player and other NPCs in the same fashion. However, the Player's AS can decide to ignore the wakeup due to being too tired and in need to sleep-off injuries.

5.5 Summary

We utilized this architecture in all our presented use-cases for the KCD game. Every NPC within the game is build using the decomposition into the BSA architecture. Presently there are more than 600 fully active NPCs within one instance of the game, with additional NPCs being streamed in during runtime as ambient additions. Almost every NPC has a BSA complement of at least 6 distinct SubBs with specific AS:

a) *Switching* SubB is responsible for managing other SubB activation, in respect to perception and internal requests from other SubBs.

b) *Combat* SubB is responsible for fighting, manages weapon use, target tracking and everything necessary to get out of a deadly situation.

c) *Day-By-Day* SubB responsible for maintaining daily routines according to the Day Plan.

d) *Planning* SubB is responsible for evaluating the Day Plan and tasking the Day-By-Day SubB. It also manages further decomposition of Day Plan's Activities if necessary.

e) *Situation* SubB is responsible for managing social interaction. These may arise when for example NPCs meet and greet in the morning.

f) *Event Handler* SubB is responsible for running emerging events (e.g. from Animated Actions) by means of a SBT

However, the content of the BSA architecture can be specifically tailored to a given NPC. For example, soldiers in our Scenario 5 (Battle0 have no need for any other SubB than the Combat SubB, since they will die anyway. However, in Scenario 4 (Jerry), Jerry and all the other NPCs involved in the quest can receive a specific quest handling SubB, which replaces their Day-By-Day SubB for the duration of the quest. This allows for a much more tailored development of the behaviors necessary for the quest and provides means to avoid polluting the generic Day-By-Day SubB.

In respect to goals, our three-tier architecture focuses on Goals 2, 3 and 4. Emerging situations in Goal 2 are commonly handled by mid-tier SubB which handle different emerging situations in either a generic (e.g. CSubB) or custom manner. Our capability to introduce a SubB on the fly to the Brain at runtime allows us to adaptively enhance NPC capabilities to handle new situations in the environment. The existence of a Day Plan and a planning mechanism on top of a Switching SubB allow us to adaptively react on changes introduced by quests and the questing system in a story driven game. The Day Plan also focuses on providing mechanisms to provide a deeper and more complex behavior providing the illusion of a long and

mid-term goals. Since a daily routine is a common human trait, it allows for a more believable ambient behavior (Goal 1). The NPC's daily routine can also be exploited by the player for questing. Further, various SubBs utilized for solving various situations within the virtual world can provide designers with the capacity to encode more complex, custom behaviors to enrich the NPC's capabilities.

6 Smart World, Intelligent Environment

Creating huge monolithic code base has been proven to be inefficient, hard do maintain and inherently prone to bugs. Programming languages and environments have evolved concepts like encapsulation (Scott, 2006), object oriented design (Gamma, Helm, Johnson, & Vlissides, 1995), function calls (Svenk, 2003), dynamic linkable code (Hart J. , 2005) and many others to avoid those issues. The overall theme is to split code into understandable subparts which can be coupled with other code to provide functionality, storage and other services. It is the rule of thumb, that a loosely coupled code (Beck, 2011) is more resistant to failure and simpler to maintain. Loosely coupled code denotes the notion of having independent code constructions that can be put together so changes in either of the parties does not affect the other.

In this chapter, we focus on addressing the issue of behavior decomposition in respect to simplifying action selection. We present our method of *Behavior Injection* (BI) into the SBT structure. We couple the SE concept with our *Intelligent Environment* (IE) approach on how to enrich the ambient environment with more context aware entities beyond the notion of NPCs. Further we address our concept of *Smart Entities* (SE) which represent context aware providers of SBT to be injected into a running SBT AS.

6.1 Motivation

One of the key issues with believable large scale OWG is the principal broad spectrum of situations an NPC can encounter. The player experiences the world in a unhindered fashion, thus the necessity for proper reactions to complex emerging situations is key in respect to overall believability of NPCs' behaviors. However, the sheer combination complexity of such environments puts immense requirements on the amount of code and script that should be working together flawlessly.

Our primary motivation is to provide mechanisms and constructs for scripters and developers to avoid giant monolithic code and script bases custom tailored for every NPC and situation combination. We focus on decomposing behaviors so they can be reused and utilized in a much more fluent and natural way. We focus on providing mechanisms that developers can relate to on how humans think about human thinking. From our experience, it is much easier to produce believable behaviors if a designer can relate to the technology solution utilized to deploy it.

From a designer's standpoint, our goal is to provide believable behaviors so no NPC just wanders around randomly without any purpose, hidden or obvious. To our knowledge, almost any large scale OWG has regressed some of their characters to just wander around and play random animations, simply due to the complexity of creating custom tailored behaviors for each of those NPCs. We want the player to explore NPC behavior patterns to use or exploit them (e.g. when the NPC goes to buy groceries, the player can sneak into its house and steel a quest item he would need to buy otherwise).

Our overall goal is to create a virtual world full of NPCs with actual agenda and traceable reasoning about goals and purpose. We focus on making such world producible and maintainable in a real life setting of the KCD computer game.

6.2 Relevant work

Computer games industry and academia have worked on tackling behavior decomposition and complexity management in numerous ways, commonly inspired by programming languages. We present our principal inspiration found in *Smart Objects* (SO) and *Smart Environment* (SEnv). We already discussed some details in Chapter 2. We build our work on joined research collaboration in this topic (Cerny, 2016) and our previous work (Plch, 2009). We deployed these solutions at Warhorse Studios for the KCD game.

6.2.1 Smart Objects

Conceptually, Smart Objects (SO) (Kallman & Thalmann, 1999) represent actual objects in an environment (e.g. computer game, internet, physical world etc.) with enhanced interactions with other objects and humans. The concept of »smart« refers to the capacity of the object to describe its interaction capabilities. From a practical standpoint, SO refer to physical objects connected to the internet (Kortuem, Kawsar, Fitton, & Sundramoor, 2010) enhancing human interaction by understanding the host environment.

The SO concept caught easy traction in the computer game industry, since it allowed to annotate the world in a more convenient and controllable way. Within a computer game, a SO can be abstracted as anything non-NPC with the capacity to provide extended interaction for possible users (e.g. NPCs, player, other SOs). This extended interaction may range from providing information about animations necessary to engage the object, to taking complete control over the NPC. These interactions can be exploited within various other mechanisms, beyond AS, for example while moving the NPC or engaging in idle behaviors (e.g. a companion NPC stands there without any commands from the player).

One example of utilizing a SO is during executing path planning and path following for NPC movement (Reed & Geisled, 2004). The SO (e.g. door, fence, trench) provides contextual information based on the NPC type on how to traverse over it (e.g. jump, crawl, open). The traversal method can be considered while planning and executing the path.

Another commonly utilized SO principle is the already discussed opportunistic control[13], where the object takes control of the NPC if the NPC satisfies a set of predicates (e.g. is close enough). These are being utilized as means of enriching the behavior, for example in CryEngine (Crytek, CryEngine, 2002) games like FarCry (Ubisoft Montreal, 2014).

Smart Objects may also have a form of *triggered control*, when they only take governance over an NPC if the NPC enters a specific state. For example, if the NPC is in a dungeon as a player companion and the player stops for some reason, the NPC gets bored. After a time, the NPC starts to explore the immediate environment and triggers various SO which provide animations (e.g. pulling on levers) or provide audio tracks for NPC commentary (e.g. »there is an interesting door«).

Within the computer game Sims (Ingebretson & Rebuschatis, 2014), the concept of SO is built on top of *affordances* (Gibson, 1977), where objects are viewed as providers for *needs* (e.g. ease hunger) or provider of *opportunities* (e.g. open box). In the Sims game, NPCs traverse the world with their needs (e.g. toilet, hunger, joy) to

[13] Discussed in 2.3.3

be satisfied. The player fills the environment with objects to satisfy those needs. The more NPC's needs were satisfied, the happier it is. Behaviors are decomposed into *interactions* which were associated with objects. Every interaction utilizes a decomposition into *blocks* which are associated with NPC animations and change of NPC's state. The NPC can engage in multiple interactions where the particular blocks may interleave.

In principle, the SO concept is key to employ when decomposing behaviors, since it provides behaviors relevant to the use or engagement with a particular SO. However, to our knowledge, these interactions are often very limited in nature and SO in games are often limited to the notion of sole purpose behavior providers.

6.2.2 Smart Environments

Smart Environments (SEnv) are a variation on the SO principle, where the decomposition is aimed at topological (e.g. a house) or contextual areas (e.g. a designated area to shoot arrows in a battle). The principle is the same as with SO, SEnv provides information about possible interactions within the confines of its region (e.g. an NPC should move differently when an area is on fire).

Within the game S.T.A.L.K.E.R. (GSC Game World, 2007) the virtual world provided long-term goals for NPCs residing within specific areas annotated by *smart terrain*. In further development, *smart zones* (De Sevin, Chopinaud, & Mars, 2015) were proposed for the game's environment. However, these approaches use the opportunistic control over NPCs which entered them, thus bypassing their high-level reasoning and on-going tasks.

Another approach was presented in the Hitman: Absolution game (Vehkala, 2012), where the NPCs utilized coordination objects called *situation* which instructed them on their role for a particular event (e.g. attack on a guard). These instructions are evaluated within the NPC's DMM where all possible instructions were reflected, rendering this method prone to bloating the host NPC code. Since different NPCs can vary in their DMM (e.g. guards, civilians, police officers), new situations should be introduced into all NPCs types, making it a tedious process.

6.3 Analysis

Large Scale OWGs are complex and unpredictable environments inhabited by numerous varying NPC. The environment is filled with interactive objects and situations NPCs either use or walk into eventually. The overall complexity of the possible interactions can be summarized as `|N|*|OBJ|*|ITR|`, where `N` represents the set of all distinct NPC types (e.g. guard, farmer, soldier), `OBJ` represents the set of all object types (e.g. hammer, spade, fork, sword) and `ITR` represent the set of high level interactions (e.g. use, throw etc.). It is obvious that the overall complexity increases dramatically with either of these sets increasing in magnitude.

We aspire to provide a believable environment where NPCs behave in a believable manner, thus it is necessary to be able to cover most of the interactions that could emerge. However, we need to maintain a manageable architecture, so adding a new tool or a new situation should not trigger the necessity to change every NPC type in the game.

Since believable behaviors are the key issue, we focus them as the principal manifestation of our system. We need to address the issue of *Behavior*

Decomposition applying the proven concepts of OOD for having a more manageable and easier to understand set of behaviors and entities within the virtual world.

Primarily the necessity to *decouple* and *encapsulate* is the key concept. Decoupling behaviors is critical since we need to deploy isolated solutions to specific situations accessible for NPCs within the environment. We want to avoid coupling behaviors as much as possible to avoid situation where introducing a new mechanism or situation causes rework of other behaviors or ones already in-use. Decoupling is also important in respect to *proper contextual execution*, where NPC behavior depends on combining context specifics and NPC's capabilities to provide a proper solution to an emerging situation (e.g. having fun at a wedding is completely different from having fun at a tavern).

Encapsulation is also key in respect to isolate NPCs and other objects in an OOD manner, hiding internal mechanisms and focusing on providing functionality via well-established interfaces. Therefore, if a behavior related functionality changes, the NPC who uses it does not need to change its internal AS if the interface and contract (Meyer, 1991) is kept unchanged. We address decomposition and encapsulation by providing several concepts:

1) *Behavior Injection* (BI) allows to introduce a SBT subtrees at run-time into an already running SBT. The selection of the injected SBT may be based either on static or run-time data. The injected subtree can be abstracted to a late binding (Booch, 1994) function call.

2) *Intelligent Environment* (IE) represents a collection of non-NPC constructs (e.g. areas, objects, quests etc.) within the OWG, having their own perception and AS.

3) *Smart constructs* are various entities (e.g. areas, objects, quests etc.) within the environment which can provide BI for other SBT running entities. Smart constructs do not employ a control scheme over BI users, they only represent context specific containers for BIs.

Our Scenario 1 (Brian) shows how decomposition of behaviors is key to maintaining believable behaviors. Brian, having a simple AS mechanism wakes up in the morning based on his Default Plan. He wanders around his house asking the house how to satisfy the need for a breakfast. Since the house is part of the IE, it can tell Brian to go to the kitchen to get something to eat. Brain's wife made a meal a day before, thus the house knows about it. However, there may be no more food left, since Brian's wife already woke up and ate it, Brain receives a hint to go and grab something from the pantry. Let's assume, Brian's wife is still asleep so Brain goes into the kitchen and finds food in a kettle. Since Brain does not like cold food, he wants to heat it up. He asks the kettle how to make the food inside hot. The kettle, being a Smart kettle, knows that Brain has to heat it up at the stove. Brain than asks the Smart stove to give him hints on how to heat up the kettle by getting wood and start a fire. When Brian finally puts all these together, he can eat and go to work. This example shows, how even a simple ambient behavior can be complex in its design if approached in a believable fashion.

However, proper decomposition comes into play, when either Brian's house changes, or the kettle and stove are put into Jerry's house by a level designer (i.e. a different instance of those objects). For example, when either level designers or Brian's wife introduce a new object into the house (e.g. a indoor fireplace), Brian can

use it employing the same mechanism as with the stove and kettle. And Brian has no need to enhance his core AS to accommodate novelties, he explores them on-the-fly.

Since we follow the encapsulation principle, both Jerry and Brian have no idea how the kettle and stove work on their inside, and only utilize a reflected interface (e.g. make hot, eat, use etc.) which satisfies their needs. So, if the same kettle is used at Brian's and at Jerry's house, they work the same.

6.4 Behavior Injection

Behavior Injection (BI) is a simple mechanism on how to extend the SBT language to introduce run-time changes to the SBT's structure. A BI can be compared to the concept of calling a late bound function within a programming language. In principle, the request to inject a SBT subtree originates within an existing and executing SBT (Figure 33). The injected SBT is connected to all the host's SBT contexts (i.e. memory and messaging context) so all data access and operations happen as they would within the host SBT.

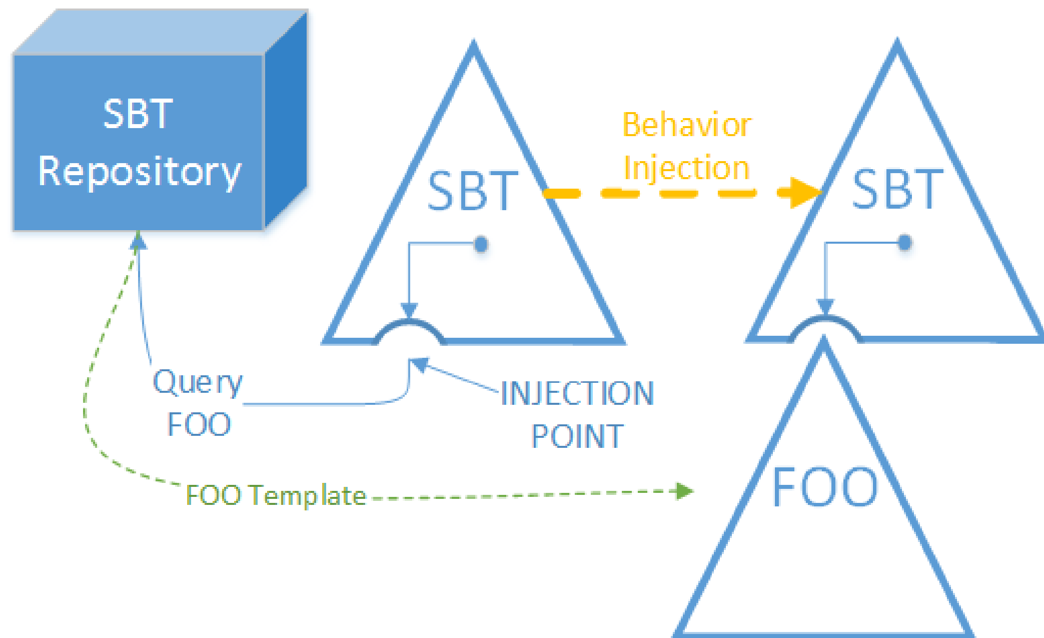

Figure 33: Injecting Behavior SBT into a running SBT is done at the Injection Point (IP) node. The BI manifests after the IP is executed. The IP continues to run until the injected SBT runs. After the injected SBT fails or succeeds, it is discarded.

The actual structure of the injected behavior is not known prior to injection. The Injection Point (IP) may request a BI based on various parameters, ranging from a simple name (*Behavior Tag*) in a behavior library to a set of parameters which to use to generate the behavior.

Since the injected behavior is connected to the host behavior, all its internal memory accesses are reevaluated to be correct in respect to the host tree. This mechanism is similar when shared code libraries (e.g. dll files) are injected into host code. However, the BI SBT can specify its own local data environment (i.e. local variables) as well as communication context (i.e. inboxes). Further, to follow the encapsulation principle, the BI SBT can provide a list of expected variables and their types within the host SBT, as well as necessary inboxes. We have not considered any other constrains, but the principle stays the same – when the injection preconditions

are not met, the injected code is *refused*. The injection can be nested, where the injected SBT contains another injection. After the SBT has finished, the IP returns the behavior to its source – we call it a *behavior drop*.

This simple mechanism allows us to decompose behaviors and inject them at any required spot within the host SBT. Further, validation/rejection mechanism allows us to create SBT code independently but in a contained manner, so errors can be easily identified. However, the run-time nature of the mechanism limits the possibility to identify possible issues during SBT design.

One key feature is the capacity to provide SBT *generators*, which can take parameters for the BI and either select an appropriate tree to inject or can construct one on the fly. We utilize this mechanism in our Smart constructs described later.

Another feature is *revoking* the injected behavior based on either an external input (e.g. the repository retracts the injections) or when the IP's predicate does not hold anymore (e.g. the NPC's opponent is dead, the combat relevant BI has no need to run anymore). This allows to introduce some designer control over how BI are used and can identify and handle badly behaving host SBTs at runtime (e.g. a NPC should send a message to another NPC when using a BI SBT but fails to do so in due time and the BI is revoked).

Further, the IP (i.e. a node within the SBT) can provide a place to manifest opportunistic control over an aspect of the NPC. In principle, the NPC's IP publishes a capacity to receive BI. Thus, an external source may trigger a forced injection at the published point (Figure 34).

Figure 34: The NPC wanders around with a published IP called »Fear« which can be used by the Hunted Church to force a behavior to act afraid in the area

Overall, utilizing the BI provides us with a strong tool on how to effectively decompose behaviors so we can create more complex behaviors in comparison to current methods of static AS's behavior declaration.

6.5 Intelligent Environment

The *Intelligent Environment* (IE) represents a collection of non-NPC entities, which perceive the environment and exhibit deliberation to satisfy their individual agenda (i.e. short and long term goals). We took our inspiration from the SO concept (Kallman & Thalmann, 1999), providing NPC deliberation mechanics to objects, areas and game mechanics (i.e. quests). Our main motivation is to populate the world with thinking entities which can aid NPCs and enhance believability of the ambient world. In principle, we introduce a SBT as a component of an non NPC entity. We split the IE into four categories:

a) *Object* represent a physical manifestation of game objects within the world e.g. hammers, forks, swords, doors.

b) *Area* represent a region within the virtual world having a dedicated meaning e.g. city, home, tavern, work place.

c) *Mechanism* represent game mechanisms which require an intelligent proxy to manage their influence in the virtual world (e.g. quests, dialog system etc.).

d) *Virtual observer* is a run-time entity who observes the virtual world and makes deliberations and communicates with other AS capable entities (e.g. crime observers watching player's actions and informing nearby guards, battle observers monitoring how many NPCs are alive).

All of the above presented categories receive a custom DMM and are managed in a similar fashion like NPCs. However, run-time execution of NPCs and IE is separated to avoid running the IE on behalf of NPC execution budget. NPCs also differ from IE components in respect to capabilities (e.g. they cannot move around, play animations etc.) and forms of perception. Some IE components may require custom code to provide their percepts (e.g. aggregated health of a group of soldiers).

6.5.1 IE Object

The *IE Object* represent the most common manifestation of an IE component within a virtual world. These are simply done by adding an AS mechanism to an existing object (e.g. a hammer). Further, a IE Object is enhanced by a perception mechanism. There is either the possibility to specify the mechanism by design (e.g. a mug receives percept information in form of messages if somebody picked it up) or are used as in-parallel executed barrier nodes (e.g. a node waiting for somebody picking up the mug) within the AS's SBT (Figure 35).

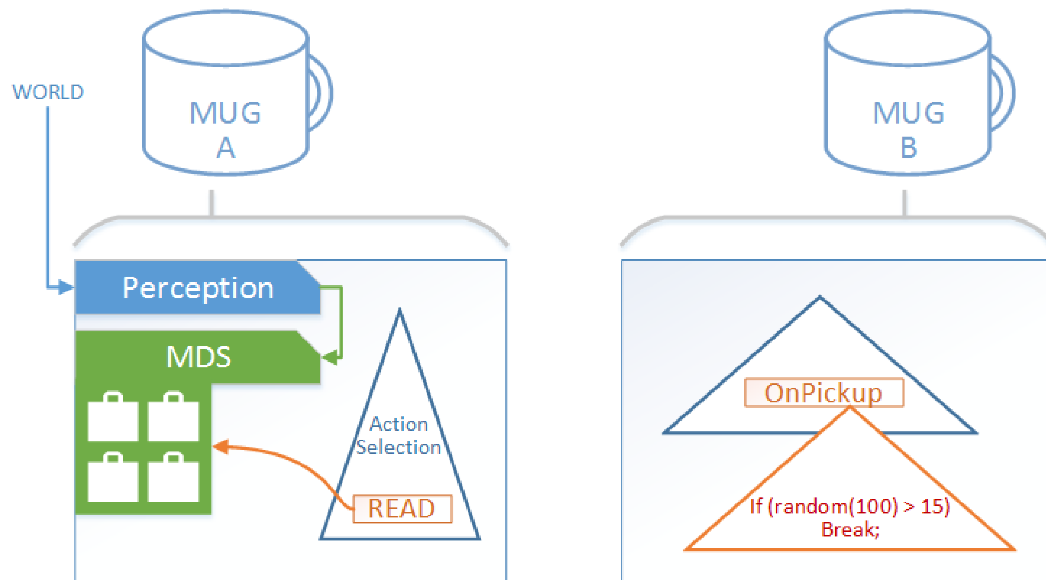

Figure 35: How to specify an IE Object a) the Mug object's perception informs the AS via messages, b) the Mug object waits on pickup events to happen to break at random

An example of the use is within Scenario 2 (Tavern), where beer mugs are a common item. Our intent here is to provide the Tavern with realistic mugs whose contents depend on how many times have they been used so they can be realistically refilled when empty. Further the mug keeps track who is its current owner, so it can distinguish who is using it legitimately and who is not. Further the mug keeps track of its content (e.g. beer, water, poison). Thus, it can forward this information (e.g. being drunk, poisoned etc.) to the present user. In our case, the effects of using the mug manifest in putting temporal effects (Buffs, 2017) on the user.

The use of an IE Object provides us with all the above mechanisms without the necessity to make any changes to the user's (i.e. NPC) AS. First, the mug has a form of perception which is informed when either a 1) new customer takes control of the mug, 2) the mug is used for drinking, and 3) the mug is filled. These percepts can be manifested by simple messages to the mug's MDS so the AS can take care of them. Internally the mug stores persistent information (i.e. variables) about a) the owner, b) the level of its content, and c) effects of its content. The actual AS of the mug is relatively simple, maintaining a reactive approach on handling percepts based on mug's present state. When someone uses the mug, they are checked against the current owner. If they are not the same person, the owner is informed via a message to handle the situation (i.e. complain about someone using his mug). Further, use of the mug triggers creation of effects on the user (e.g. poisoning), making it possible for the player to poison someone's mug, or even other NPCs to poison the player's mug. When the mug gets empty, it reports this to its owner, so a refill can be requested.

It can be seen how IE Objects provide a simple and convenient way to enhance the believability (Goal 1) and complexity (Goal 4) of the virtual environment by decomposing the behavior into several components at various entities. In a common approach, all the mug's logic had to be present in the NPC's AS or at some managing system which takes care of mugs and drinking. The strength of the system is the overall separation of competences, e.g. when the mug detects drinking, it triggers an effect on the user. This mechanism also benefits the overall production, since

introducing new liquids and innovative ways of using the mug is independent of the user (i.e. NPC or player). Thus, having IE Objects adds a dimension of modularity to the virtual world. This provides the capacity to create new emergent situations (Goal 2) based on object use, without the necessity to heavily modify NPC behavior.

6.5.2 IE Area

The *IE Area* is a similar concept to IE Object. An IE Area represent a 3D region within the virtual world[14]. IE Areas are responsible for managing topological based deliberation. The IE Area is equipped with 3 SBTs to manage its own deliberation and region relevant events a) *entering* and b) *leaving* of entities.

When a NPC enters the area, it triggers an inbuilt *Enter* event which is associated with an SBT. The local SBT's data scope is filled with the necessary information about the entering entity (e.g. its name, identification, health etc.). The *Leave* event is handled in the same manner, when an NPC leaves the area. The event handling SBT executes in atomic fashion, to provide a strict time ordering of all events (Figure 36).

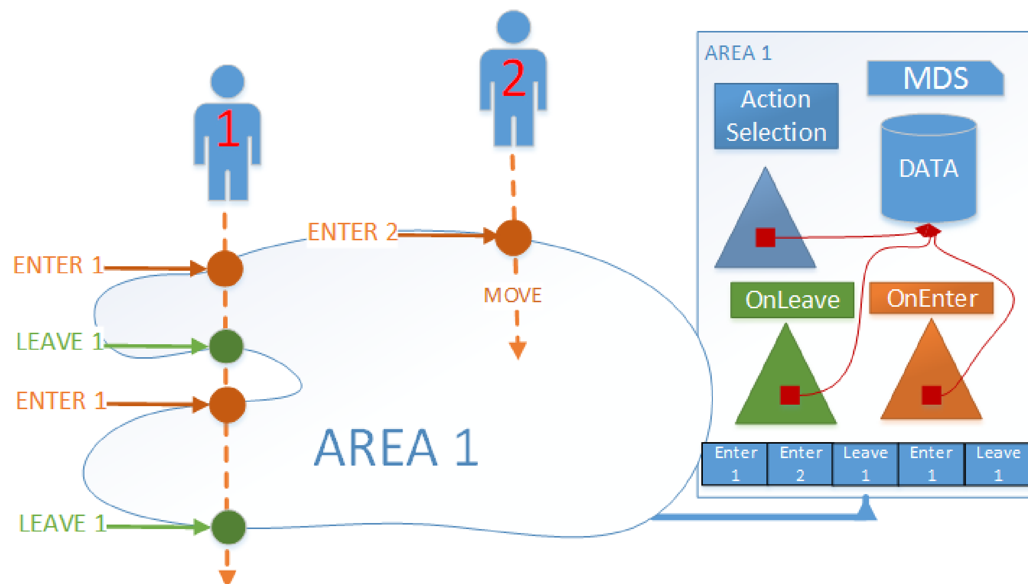

Figure 36: NPCs enter an IE Area triggering Enter events which are handled by the area's AS. After leaving the area, they trigger the Leave event. The events are strictly ordered. The IE area has three AS mechanisms (Enter, Leave, Action Selection) where the Action Selection runs the internal AS of the area, Enter and Leave are triggered by NPCs. All the AS share the areas data and communication context to store variables and send messages.

In Scenario 2 (Tavern) the IE Area is the Tavern itself. It tracks who enters and who leaves the room, thus creating a list of clients the Tavern's employees have to serve acting as a manager. When someone orders a beer or food, the Tavern is responsible for handling the message by its AS and deliberate about proper actions (e.g. when some NPC is too drunk it will not be served). When an NPC leaves the Tavern, and does not pay his bills, the IE Area can trigger an alarm and designate that NPC as being a thief.

As can be seen we can easily decompose behaviors in an OOD manner and simply implement a »command« and »front controller« design pattern (Freeman, Sierra, &

[14] In KCD, the region is specified by a 2D polygon and its height

Bates, 2004). It also can be seen that the decomposition of the scenario is natural in respect to how humans perceive competences and management.

6.5.3 IE Mechanism

IE Mechanism is similar to IE Object with the distinction that they are associated with systems within the game's engine which are not materialized in the virtual world. For example, quests are a common RPG mechanism which is ideal to manifest as part of the IE being a IE Mechanism. In principle, IE Mechanisms are proxies for relevant engine objects providing additional functionality in respect to the virtual world.

For example, a quest being an IE Mechanism can receive updates from the relevant quest (e.g. bring Jerry his medicine) and influence the quest mechanic based on what is happening in the virtual world (Figure 37).

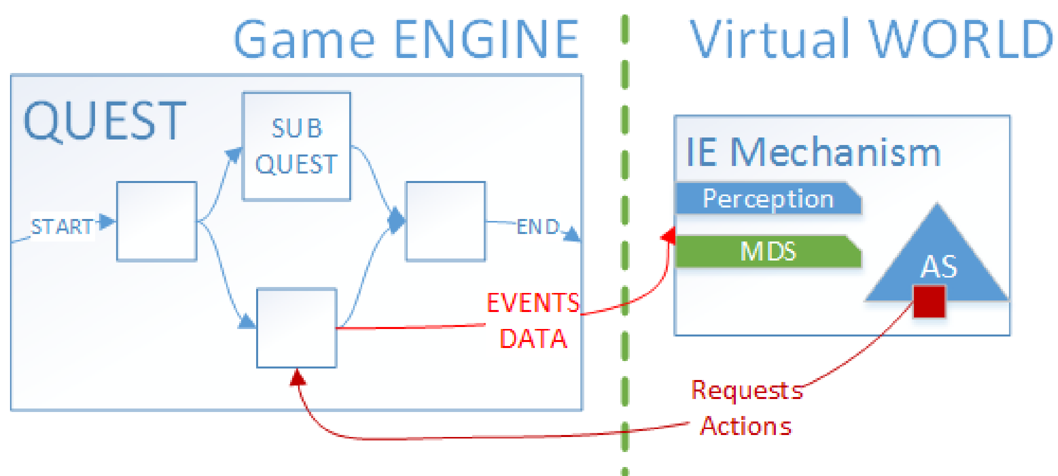

Figure 37: The IE Mechanism providing a proxy for a Quest, which is a part of a quest chain. The AS within the IE Mechanism can »cancel« the quest if for example someone kills Jerry (see Scenario 4). The quest also informs the IE Mechanism about specifics (e.g. the event's timeout expires) so the IE Mechanism can react accordingly (e.g. tell all NPCs to get back to work).

In our Scenario 4 (Jerry), the IE Mechanism is used to represent specific parts in the quest. When Jerry sends the player on a mission to get him some medicine, the player is tracked by the custom designed IE Mechanism for this quest. On the quest's part, there are timeouts (i.e. Jerry cannot wait all day) and conditions (e.g. the correct antidote in the medicine) which have to be communicated to the virtual world. For example, if the timeout is running out, Jerry may send someone looking for the player and medicine. On the other hand, if something happens in the virtual world the IE Mechanism perceives (e.g. the player steals the medicine from the herbwoman) it may inform the world about it to act accordingly (e.g. the herbwoman is not friendly to the player anymore).

It can be seen, providing IE Mechanisms is vital to integrating other mechanisms into the virtual environment to provide a two-way proxy. Having IE Mechanisms mainly focuses on quests at Goal 3. Binding perception and action selection to sub-quests within a quest allows us to manage more complex quest development as well as deeper and more managed involvement of NPCs (Goal 4 and Goal 1). At Goal 1, NPCs change their day-by-day routines based on requests and day plan patches thus

providing a more context specific set of reactions (e.g. when the player investigates a murder, NPCs tend to stay at home because they are afraid).

6.5.4 IE Virtual Observers

All the above concepts of IE are primarily bound to some object, area or an engine mechanism. However, in some cases, the situation, script or environment may require an abstract entity which observers and deliberates about the world around it. The *IE Virtual Observer* (IEVO) is a virtual entity with no physical or observable (i.e. from the virtual world's perspective) manifestation within the virtual world. In principle, the IEVO acts the same way a IE Object does, having the same capabilities and internal mechanisms, but represents only the essence of deliberation about the world.

As an example, in Scenario 5 (Battle) both fighting sides are observed by IEVO. Each of those manages the necessary target pairing between NPCs, choosing the appropriate targets for every combatant. The IEVOs are also responsible for balancing the battle out so it does not end too soon or differs from the by design intended story line (i.e. the bad guys win). Thus, both observers act as battle managers. The player is also watched by an individually attached IEVO, which provides him with opponents in respect to design requirements (e.g. on easy difficulty only 1 at a time, on hard difficulty 5 at a time).

The IEVOs are necessary for our Goals 2, 3 and 4. At Goal 2, they provide the roaming player with emergent situations. There is always an IEVO watching and evaluating if a nearby emergent situation is feasible to be activated. Thus, when the player roams the woods, he comes upon bandit camps, small side quests and interesting situations. At Goal 3, the IEVOs are responsible for managing small side quests which happen in specific places or at specific circumstances (e.g. the player meets a NPC at night). The IEVOs are spawned by other quests and may be removed after some time (e.g. a main story line quest is completed in a specific way creates IEVOs with small side quests around the town to provide some story background for what happened after the main story quest). At Goal 4, the IEVOs can provide more complex context specific situations for the player to engage.

6.6 Smart Constructs

One of the key issues with believable behaviors is the capability to handle situations in a contextually sensitive way. NPCs acting in odd or unnatural ways may break believability in respect to the situation at hand. Another key issue with believable behaviors is the overall design complexity, when encoding all environment specifics into one NPC's AS, thus rendering the individual AS unmaintainable over time. Introducing new mechanisms, items or situations to the environment would facilitate a rework of the present NPC AS. *Smart constructs* is our concept to tackle issues of:

a) *Contextually relevant* behaviors, which can be viewed as the notion of behaving in different situations differently, following the human reasoning on how a reaction should look. Consider two situations where NPCs observe the player walking around in full plate armor with all weapons on display. In a peaceful village, almost everybody would run away and hide and soldiers would be questioning the player's intent. In an army camp just before battle, soldiers would cheer and comment on the armor's quality.

b) *Behavior decomposition* in respect to using objects and handling situations is necessary to avoid aggregated solutions. In principle, encoding every possibility into a AS of an NPC may bloat the description immensely over time (e.g. covering all combinations of items and jobs) providing a unmaintainable mess.

We take our primary inspiration in OOD, where every concept is wrapped in an object encapsulating data and functionality. Communication between objects is done by providing interfaces which can be used to change objects in a controlled manner.

In our perspective, we turn this concept around – if a Smart construct (e.g. object, area, situation, quest etc.) provides functionality, it is focused at influencing the requesting entity. This influence is aimed at providing contextually proper approach to handling the interaction between the requesting entity and the Smart construct.

Simply put, the Smart construct is a behavior container, where every behavior has a name and number of instances for each behavior. The requesting entity (i.e. a NPC) asks for a behavior by name, and injects it into its own AS utilizing the BI principle. The injected code is part of the host AS until the IP or the host AS are willing keep it there (Figure 38).

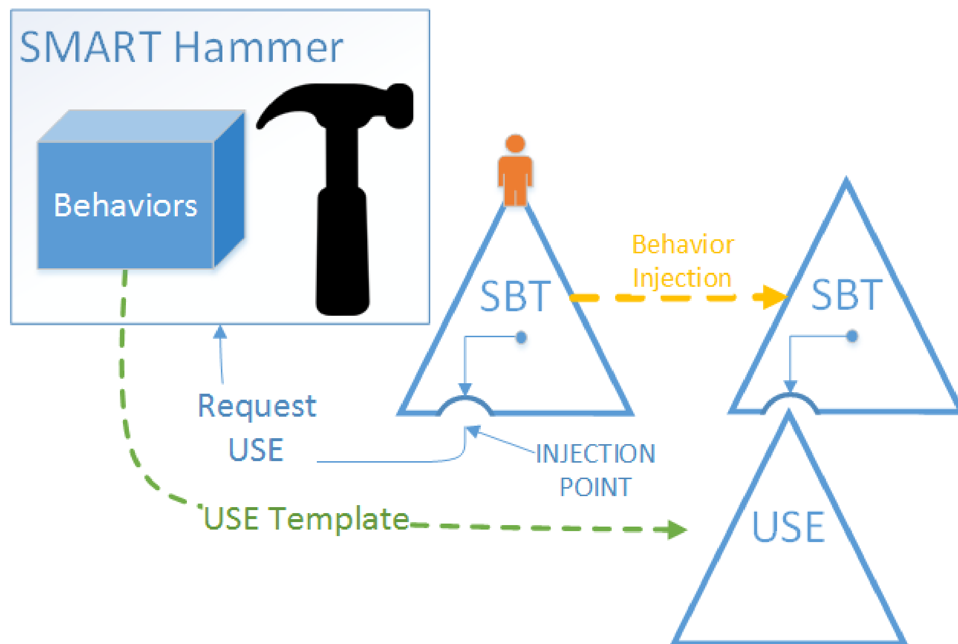

Figure 38: Smart construct contains behaviors which can be provided for BI at host AS. The behavior is identified by a name tag and there may be a limited amount available to be injected at the same time.

Another key mechanism at the Smart construct is that it can deliberate about the request for BI by the to-be-host AS. For this purpose, the Smart construct is enhanced with a simple DMM which handles two events 1) *requested*, and 2) *returned*. The requested event is invoked when someone requested a BI of a behavior. The behavior request is automatically rejected if there are no more instances of the behavior, or there is no such behavior available. When the host SBT drops the behavior (i.e. discards it after it finishes its execution), the behavior is returned to the Smart construct and the returned event is invoked. Smart constructs can associate both events with a SBT (or any other AS mechanism) (Figure 39). The event handler also has access to the data and messaging context associated with the

Smart construct (i.e. modify persistent variables and send/read messages). For example, a Smart can count how many times somebody requested a particular behavior. The data context is also used to forward information about the host and the requested behavior into the event handler (i.e. SBT).

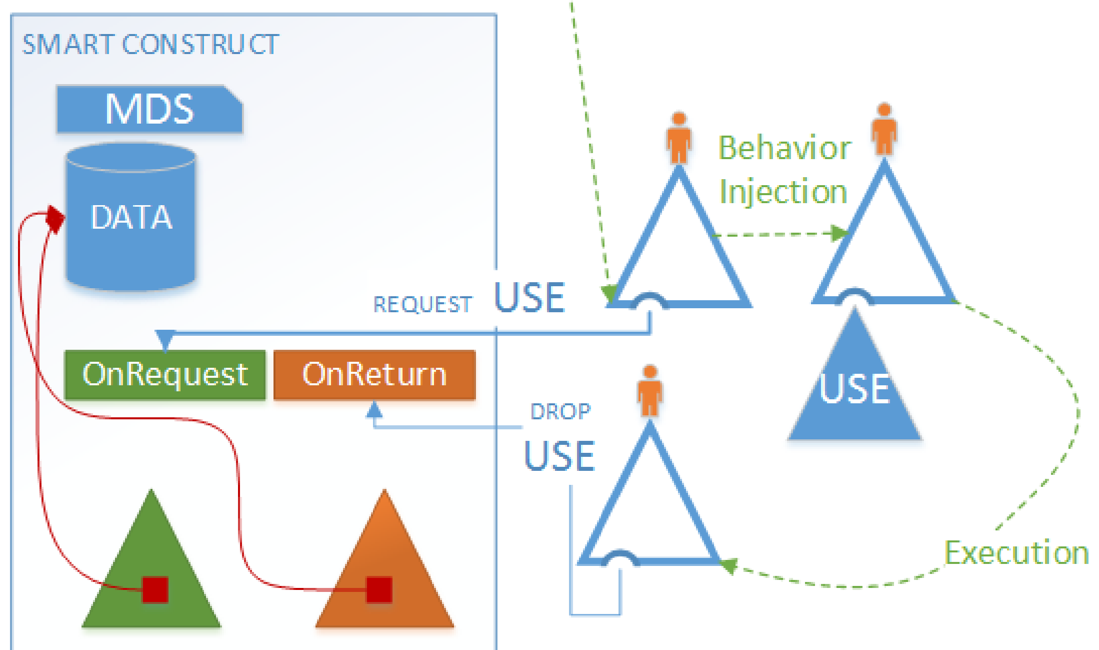

Figure 39: Requesting a BI from a Smart triggers the Requested event which is handled within the Smart construct. When dropped, the injected behavior returns to the Smart and the Returned event is invoked and handled

Our Scenario 2 (Tavern) can be used as a simple example of utilizing Smart constructs (Figure 40). This scenario we will take 3 entities into account – a NPC, a beer mug, and the tavern. First, we have to establish that the beer mug is registered at the tavern as being an item within it and it also has knowledge to which tavern it belongs to. We will omit all other NPCs and entities taking part in this scenario. Both the tavern and the beer mug are Smart constructs.

At first, the NPC enters the tavern and asks for a context specific behavior »guest« to behave in a way the tavern requires. Different taverns may have different guest behaviors at disposal. A fancy tavern may even reject a low life NPC by denying it the behavior. If there are enough instances to BI into the NPC, the tavern satisfies the request. Within the injected behavior, the tavern provides the NPC with the means and logic to use the beer mug it assigns to him. We can simplify this to a sequence of »Take, Drink, Ask for Refill If Empty«. Both »Take« and »Drink« are behaviors requested from the beer mug, since it knows best how to use it. However, »Ask for Refill If Empty« represents a behavior that is requested at the tavern. Within it, the tavern checks the status of the beer mug and if empty, tells the tavern via message to get a refill. Note that all this is happening within the NPC's AS, thus it has to communicate with the tavern for the refill. However, the code above the Refill behavior provides the data on how full the beer mug is via local variables.

It can be seen that rather complex believable behaviors can be decomposed into components which can be put together in a very loose fashion. The whole concept would work if the beer mug and the NPC would meet in a restaurant (i.e. being a

fancier tavern). It also shows how the beer mug can be replaced with a glass of wine without the NPC changing any part of the composition.

It also shows how stacking behaviors can lead to complex behaviors, where injected behaviors can inject behaviors based on evaluating the current context (e.g. if all beer is gone, lets order wine).

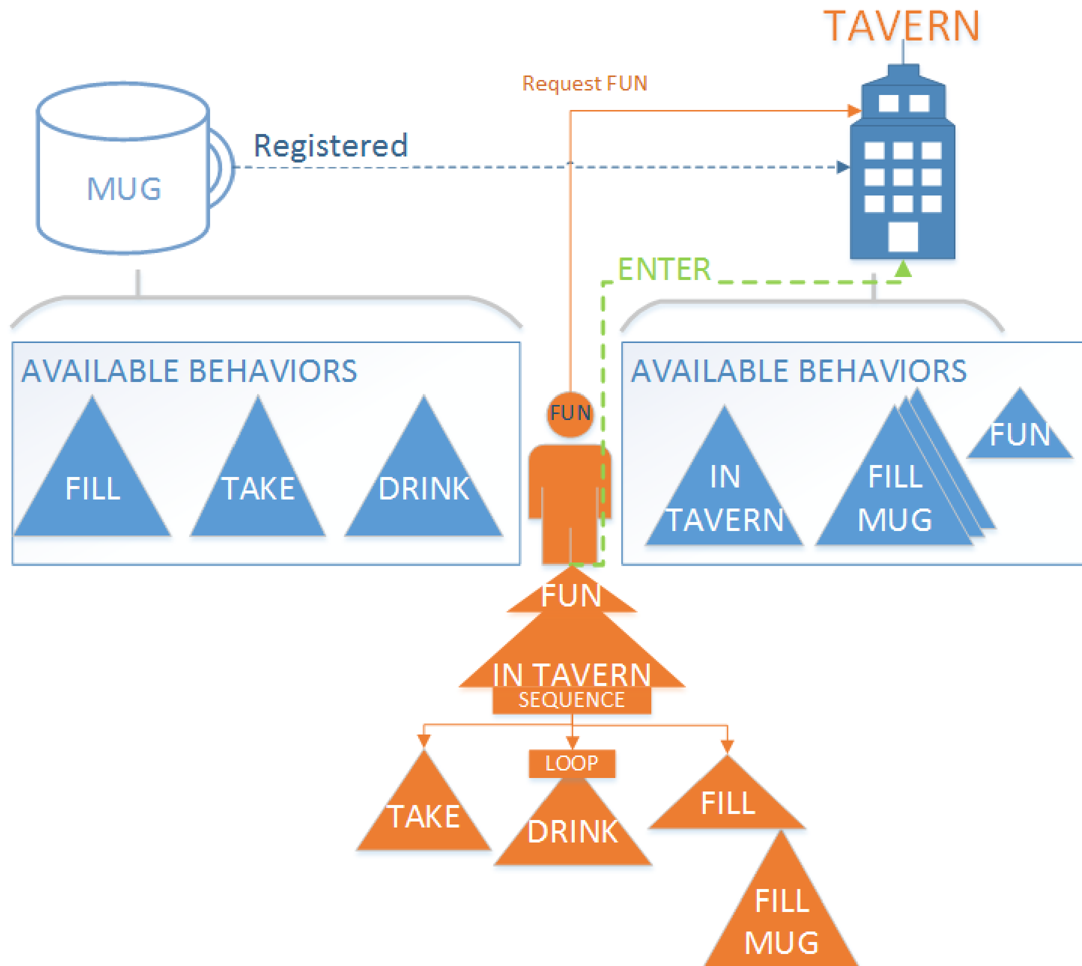

Figure 40: When the NPC enters the tavern it only requests the »Fun« behavior, which further makes use of requesting behaviors from the mug and tavern. The NPC does not know of any other behaviors except for Fun.

We recognize four main representatives of Smart constructs: 1) *Smart Objects*, 2) *Smart Quests*, 3) *Smart Navigation Objects,* and 4) *Smart Areas.*

6.6.1 Smart Objects

Smart Objects (SO) are associated with in-game materialized objects, like hammers, beer mugs etc. These objects are utilized to provide behavior decomposition in respect to their intended use e.g. »use« behavior, »drink« behavior etc. These objects are also coupled with other constructs (e.g. Smart Areas) to provide proper context of their use. If used alone, these are often responsible for providing proper context for activities in their immediate surroundings (e.g. cross near the road makes people bless themselves).

6.6.2 Smart Quests

Smart Quests (SQ) are associated with game mechanisms, like questing (i.e. a SQ represents a particular quest) or emergent situations (e.g. murder in the streets). The SQ acts as a source for BI for *participants*. When some NPC (or any other AS capable, e.g. an IE Object) wants to take part in a quest or is pulled into a situation, SQs are responsible for providing context relevant behaviors. For example, when Jerry gets sick in Scenario 4 (Jerry), his behavior is provided by an associated SQ. Emergent situations provide specific behaviors based on the situation type (e.g. surprise attack on the player provides attackers with behaviors how to engage) and the ongoing circumstances (e.g. if the player's health drops below 50% no more participants are allowed in arresting him to give him a chance).

6.6.3 Smart Navigation Objects

Smart Navigation Objects (SNO) are a specific purpose SO associated with navigation in both path finding and path following. The SNO represent a complex behavior necessary to plan or traverse over a certain navigation element (e.g. door, trench, fence). The SNO is also responsible for providing context information about the possibility to traverse over them (e.g. when path planning wants to go over an SNO but the host NPC would be too heavy, the host SNO denies the passage while path finding) to avoid valid paths being planned over them but being rejected while following the planned path. During execution, the SNO provides the movement mechanism with behavior to be used as a part of the movement process – the BI happens as part of the »move« node (Figure 41). The used behavior is implicit. However, the SNO can have different variant of the implicit »Traverse« behavior for distinct types of NPCs or different path planning or path execution requirements (e.g. a fully armored knight does not care for a locked door, he just walks through).

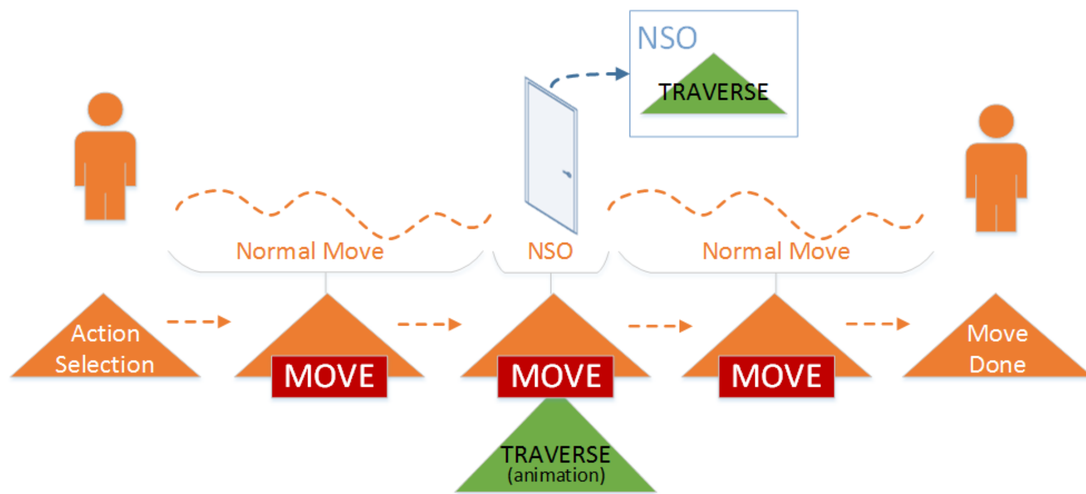

Figure 41: The NPC walks along the planned path with a SNO (Door) being part of it. The SNO provides the »Traverse« behavior to get through the door using specific animations and handling issues like locked doors.

6.6.4 Smart Areas

Smart Areas (SA) represent a region, i.e. topological information in respect to the context they represent and the region coverage they are responsible for. In principle SA behave similar to SO, with the difference of having a volumetric coverage of the

virtual world, thus NPCs and other constructs can enter, leave and reside within them.

Therefore, we provide the NPCs with an enhanced way to request behaviors in respect to a reference point – *topological request* (TR). The principle of a TR is the same as a direct request from any other Smart construct (including a direct request from a SA), it provides a name of the behavior to inject at the IP. However, the reference point is used to determine which area is to be contacted to provide the behavior. Often the reference point is the same location the NPC is at.

In principle, the SA represents a contextual annotation of the virtual space and provides specific behaviors on how to behave in that context. The spatial decomposition helps to associate certain behaviors with certain areas (e.g. having »fun« in a city is different from having »fun« at home). However, a key issue with the TR is the possibility of SA to overlap (Figure 42).

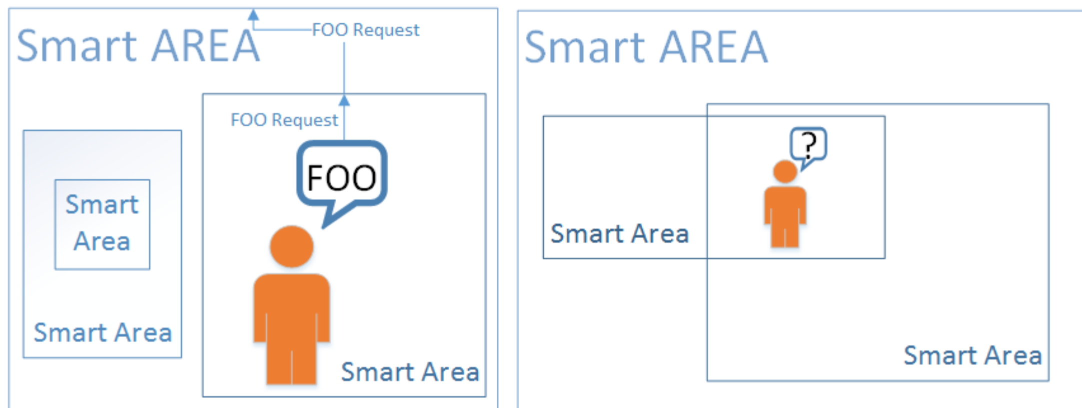

Figure 42: Overlapping SA are ambiguous in respect to a TR for BI. Non-overlapping areas can be used in a hierarchy, where »FOO« when not found at an area, can be searched at its enclosing (i.e. parent) area.

When overlap happens, the possible areas to contact are ambiguous to the provided reference point. Therefore, we require the SA to be ordered in a *strictly exclusive hierarchy*, where every area has to be either without overlap with other SAs, or completely contained within another SA. As a result, we can specify a hierarchy for the TR to follow while contacting SA to satisfy the BI request. Since the SAs are exclusively contained within each other, the reference point has a given set of SA it can contact, from the current area it is within to the topmost containing SA (Figure 43). We avoid having a SA forest by introducing a *World SA* (WSA) which represents the whole virtual world, thus every SA without a parent SA resides within the WSA.

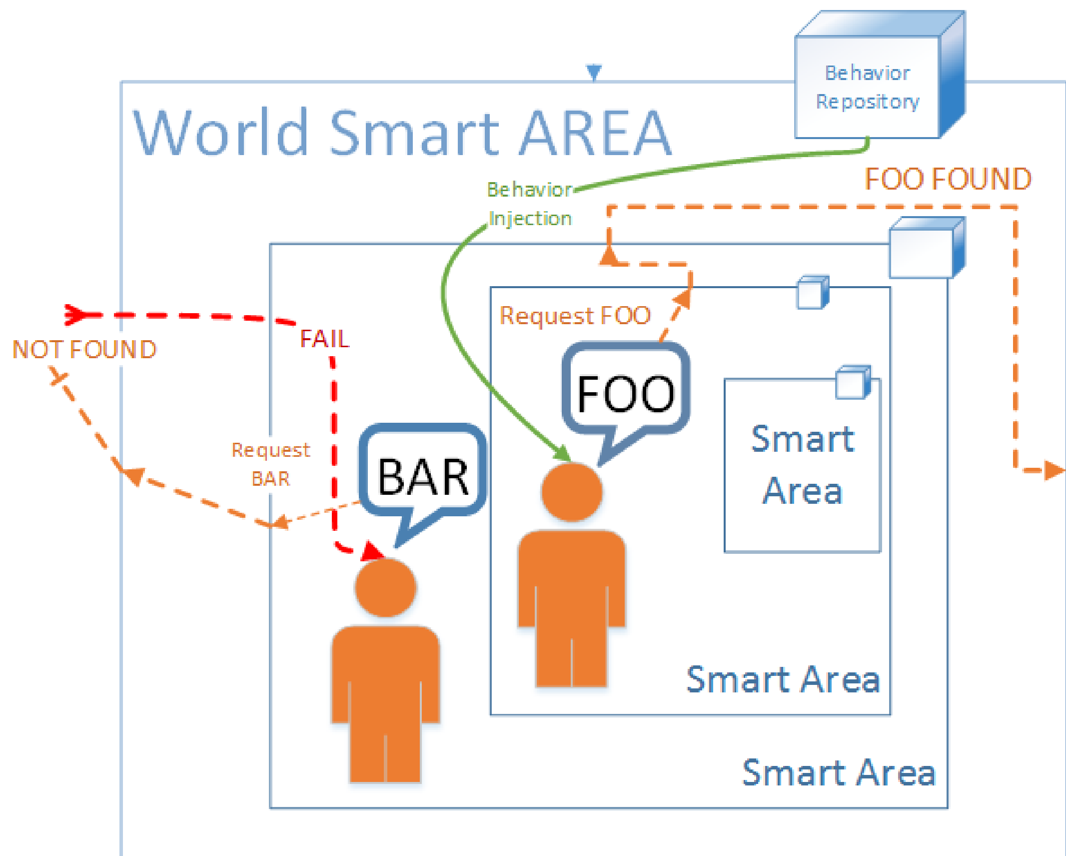

Figure 43: Exclusive strict ordering of SA provides the TR for a BI with a set of areas to ask in a given order from the most bottom to the topmost SA

Due to the fact that a specified behavior may not be provided by a SA, we allow two variants of TR – 1) *explicit* and 2) *transitive*. The explicit TR only searches the bottom most SA (i.e. the smallest area it is within). This allows to explicitly ask for a behavior from a SA. The transitive TR is aimed at providing a behavior from the SA that is within the set from the current (i.e. most bottom area) to the WSA. The SA closest to the current SA is the preferred choice for providing the BI for the TR.

To illustrate the use of SA we use the Scenario 2 (Tavern). In principle, we can topologically divide the world (Figure 44) and search that world based on the NPCs current location.

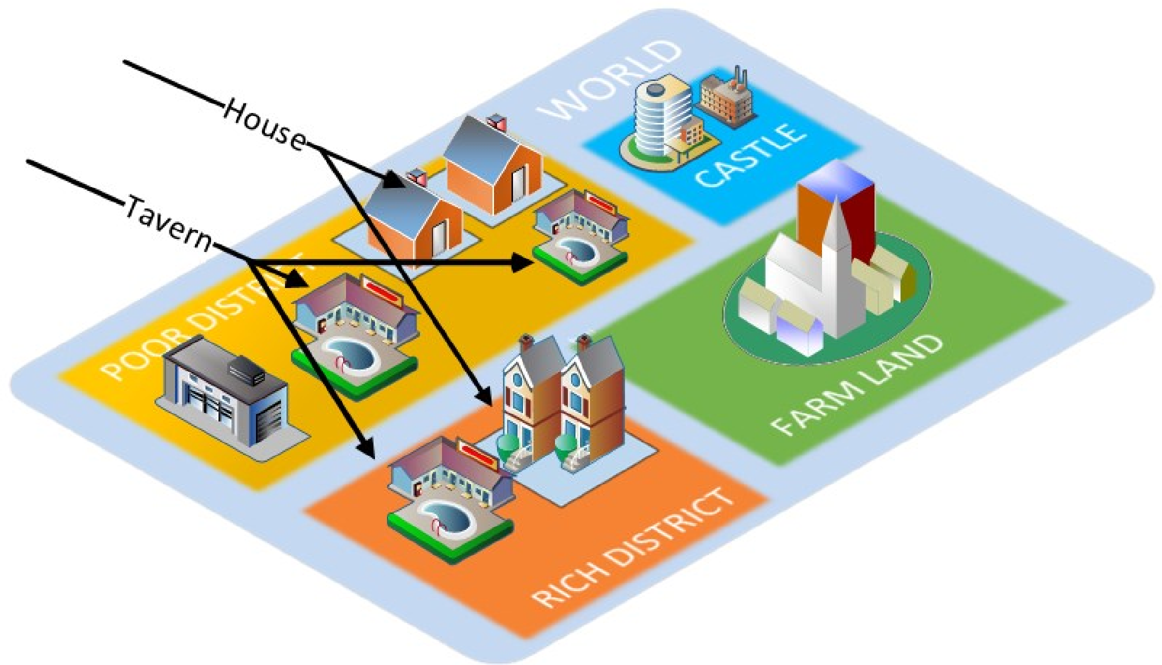

Figure 44: The topological division of a part of the virtual world into a Land, City, Districts within the City, and Taverns and Homes within each District. There is also o Castle where only soldiers are.

For example, if the NPC would be at outside of the »City« looking for »Fun« it would get no behavior, since there is no fun outside the city. However, the WSA may respond with a generic behavior for »Fun« which seeks for the nearest city to go into. After the NPC arrives at the nearest city, it would again search for »Fun« (i.e. as a TR within the WSA's generic fun search). When being within the »Castle«, the provided »Fun« behavior would again end up in a search like behavior directing the NPC to go into a city district based on the NPC's wealth. When the NPC arrives at the wrong district, the pattern repeats, sending the NPC to the proper district to have fun. When arriving at the Poor district, the NPC receives a behavior »Fun« to choose a tavern based on the proximity and some randomness. The NPC goes to the nearest tavern and requests »Fun« finally getting what it asked for. The resulting structure of the SBT within the NPC's AS may look something like (Figure 45).

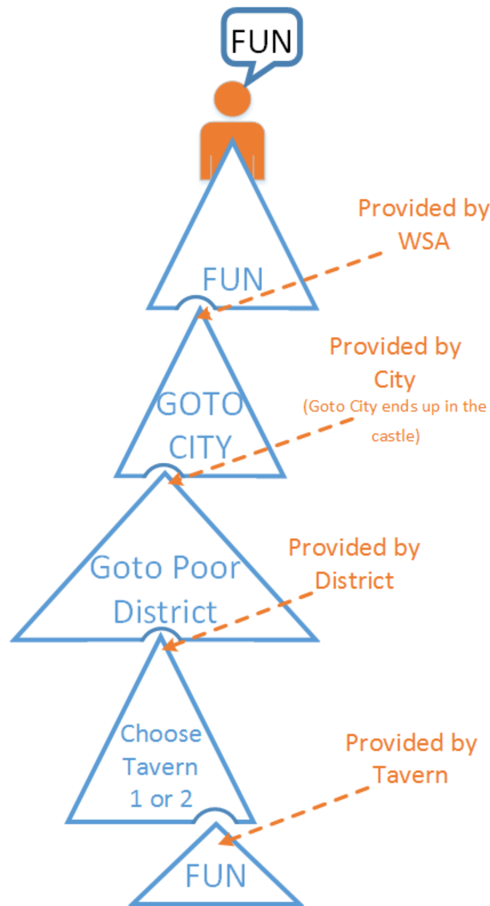

Figure 45: The SBT of the NPC seeking Fun in the City

Utilizing the SA allows us to manage the behaviors in a more in the sense of topological decomposition in respect to the context specifics of certain locations.

6.6.4.1 Trigger Area

In some cases, the restriction to have SAs exclusively contained within parent areas may be limiting, either in respect to spawning specific areas during runtime, or as a design limitation. Therefore, we utilize a parallel system of *Trigger Areas* which are SAs but are taken out of the SA hierarchy. However, all trigger areas are exclusive to each other. If a request for behavior is executed, first the Trigger Areas are queried and if not being able to satisfy the request, the SA Hierarchy is queried to satisfy the request.

6.6.5 Summary

In this sub-chapter, we presented our Smart construct approach on decomposing and distributing context relevant behaviors into the virtual environment. This approach is primarily aimed at helping to mitigate the overall complexity of believable behavior in respect to context relevant behaviors in specific regions, utilizing objects, participating in quests and traversing the environment. Employing the decomposition of behaviors in conjunction with behavior injection provides an adaptive tool how to tackle context relevant believability, where adding new behaviors, objects and quests does not influence how the already present behaviors interact with the NPC, thus keeping the NPCs basal SBT as generic as possible.

Smart constructs are primarily focused to achieve our Goals 2, 4 and 5. In respect to Goal 2, Smart construct provide specific behaviors for emerging situations, either in a static manner, when some situation the area or object is designed to handle occurs and NPCs can utilize the Smart construct to behave in a contextually aware fashion. For example, when the Scenario 3 (Murder) happens in a tavern or in the streets, surrounding NPCs will be inquisitive in a different fashion than when the murder happens in a church or at a cloister. The hierarchical structuring of Smart Areas allows us to manage emerging situations at a more generic level with every parent layer (e.g. a city handles murder in a more generic fashion than a church).

The complexity and depth of behaviors aimed at in Goal 4, is enhanced by utilizing both Smart Objects and Smart Areas. SOs can provide complex handling mechanisms which can make complex use of objects within the game's world. Using a simple hammer can lead to searching for nails to hammer some wood together. Smart Areas can provide context on how to handle complex and structured behaviors, like cooking at home, where the NPC (e.g. Brian) has to find food, make use a of a stove and find a place to eat.

Goal 5 aims at supporting development and easy maintenance of a large scale open world. Decomposing behaviors into SO and SAs allows for more structured and reusable behavior base (e.g. SBTs for chairs and benches can be reused at almost every place within the virtual world where benches and chairs are). However, it requires a more generic approach when designing behaviors (e.g. for sitting on chairs) but it pays off in a long run where one script (i.e. SBT provided by a chair) can be reused and maintained only at one place (i.e. chair).

The Smart construct concept are also focuses on Goals 1 and 3. Since Goal 1 is aimed at providing a believable ambient environment, the SO and SA help to maintain the proper mechanisms on how to employ objects and behave in regions. SOs provide proper behaviors for their correct use, so NPCs do not need to know in detail how to employ objects for their activities (e.g. using a shovel to dig a hole). SA provide guidelines (i.e. behaviors) on how to discover places for activities (e.g. a city helps to find a tavern) or provide context and proper behaviors on how to behave at those places (e.g. tavern provides a behavior on how to be a good guest). NSO are also employed to help the NPC traverse in a believable way through objects (e.g. jumping over fences, opening locked doors etc.).

Goal 3 is mainly managed by our Smart Quests, which provide specific behaviors for NPCs to believably engage in the respective quest or it's part. Injecting these custom behaviors helps designers to manage NPCs in a controlled and desirable fashion to provide more believable integration of quests into the ambient virtual life.

7 Knowledge network

There are many ways how to convey information about the world to a DMM. Without accurate information about the environment, the NPC would be only a simple automaton with limited capacity to adapt to the changing situation:

1) *Perception* provides knowledge as percepts which are interpreted within the internal DMM's mechanism. This limits the entity in respect to only perceivable information from immediate surroundings. This approach is feasible for reaction based DMM.
2) *Static Information* is introduced to the DMM during the design phase by a designer and has limited use, since it is not updated during run-time. For example. the NPC's fraction is considered a static information in KCD.
3) *Databases* may provide the designer with the capacity to query more complex relational information and store additional information. However, the downside is the necessity to have database schemas prior to game deployment and in some cases a database query is not simple to process and may take plenty of time and memory to answer.
4) *Graph* based information representing entities and relations (e.g. ownership, family relations etc.) between them provides a different approach on how to conceptualize the information within the OWG[15].

In our case, we make use of all the above specified information sources on different system levels. Every DMM capable entity which has a perception subsystem provides the DMM with messages about the perceived situation within the world. Since visual perception is commonly considered computationally heavy, we apply it only to entities in close proximity to the player. Static information is encoded by designers into the SBT as constant literals – e.g. player being the primary enemy. We also utilize a *database* exported alongside level data e.g. armor sets for NPCs, animation preprocessing, etc.

In respect to a graph based information database, we introduce a run-time adaptive (i.e. links can be added and removed) semantic network between entities. We use the network to annotate relations and capabilities (e.g. how much does John like Jane, which behaviors are available at a SA and how preferred are they (e.g. drink, sing and fight at a Tavern)). We integrate the network with SBTs to provide AS with the capacity to explore the world on a quantitative, semantic and functional level.

7.1 Relation Knowledge Network

The *Relation Knowledge Network* (RKN) represents the semantic network annotating the virtual world. In principle, the RKN is a directed graph between engine entities (Figure 46). World entities (e.g. spots, roads, NPCs, quests) are edge endpoints. Every edge may be annotated with *Tags* (an empty tag is allowed), thus creating a *directed multi-graph*. Every edge's Tag can be associated with data which is stored on the edge (i.e. a variable). These edge data are volatile in nature, therefore can be modified from any AS.

[15] To our knowledge, no OWG or RPG uses a graph based network of knowledge

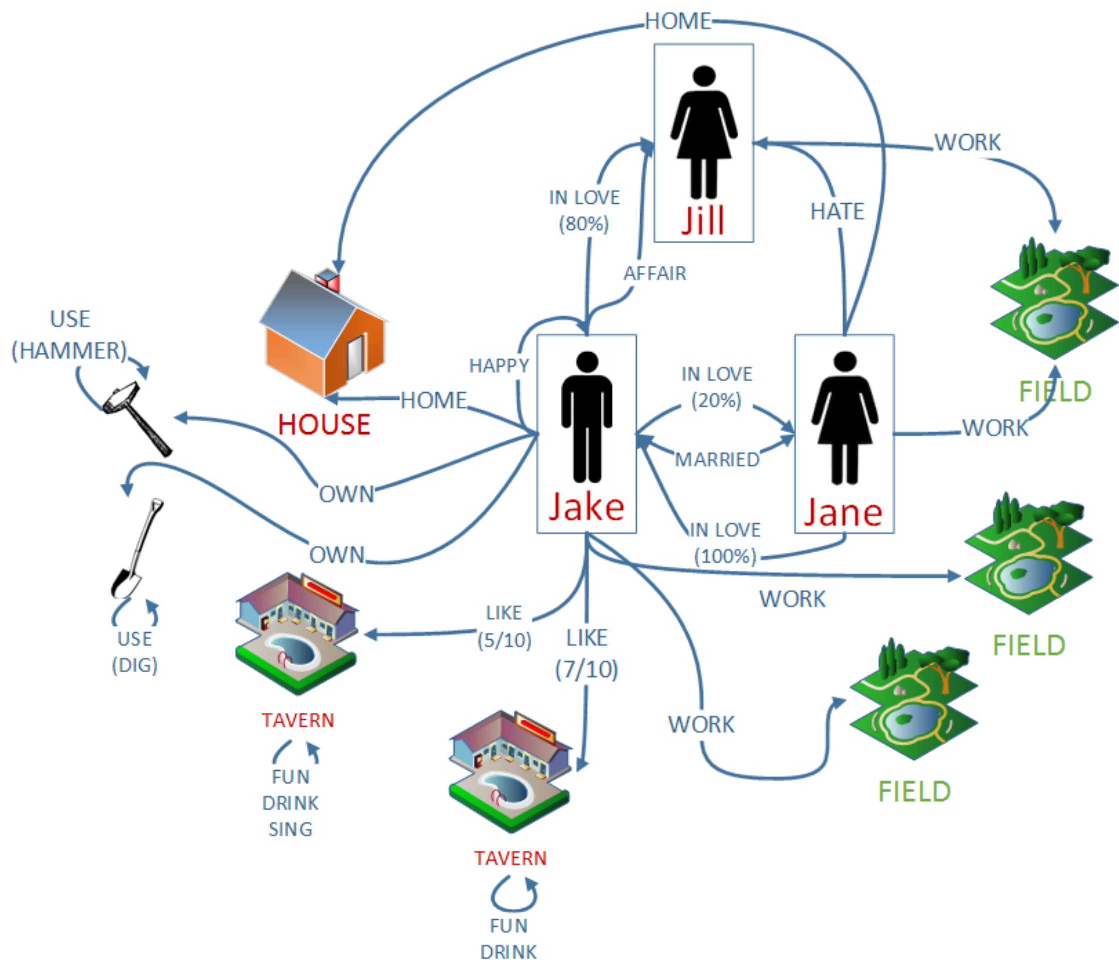

Figure 46: The Relation Knowledge Network annotates the semantic bindings between entities. John has a binding (i.e. edge) to Jane and Jill. John is also married to Jane but he likes Jill more. John also has two jobs and one house. John owns some tools which are SO who annotate themselves with their available behaviors. John also has a binding to himself, on how happy he is.

The use of the RKN is on one side to store information about the world to be queried and explored. For example, if John would like to go to a place where he can have »Fun« he simply explores his bindings to the world. However, someone else can also inspect the world from John's point of view, to see where John goes to have fun and who is his wife.

Furthermore, the idea behind RKN is the capability to adapt based on modifications. For example, if the player would tell John about a new tavern in town, John could add a binding to it and from that point onward, have the possibility to choose from 3 instead of 2 taverns.

7.2 Static, Dynamic and Virtual Links

There are three types of *links* (i.e. bindings) within the RKN:

1) *Static* links are present in the exported level data and cannot be removed or modified. These links represent the initial setup of the world to work with. These links may only change due to changes in the level's makeup – e.g. when loading or unloading level layers.

2) *Dynamic* links are links which are introduced to the RKN during runtime. These links freely accessible and modifiable from any AS either directly or by a query. These links represent the current emerging bindings between entities within the virtual world.

3) *Virtual links* exist to avoid too many links in the environment. These are created while executing queries into the RKN. For example, the links between NPC's inventory and its content is not necessary to maintain, mostly due to the fact it tends to change a lot and it is only interesting when it is queried. The virtual links are removed after a query has been resolved.

7.3 RKN Query

The *Query* represent a mechanism which explores the RKN and provides results to the hosting AS. Conceptually a query is the »cognition« about the environment and its semantic properties. In practice, any result produced by the query has to satisfy a provided *set of logic predicates*. The query searches the RKN from an origin using a pattern (e.g. depth first search, breadth first search) and checks edges (i.e. bindings) in respect to the provided predicates. The query continues to expand the searched space until it either cannot expand or is terminated. For example, a query can be verbalized as »Find the closest spot to shoot a bow not further than 50m« (Figure 47). Since commonly such searches are uninformed, they may be computationally heavy if badly shaped.

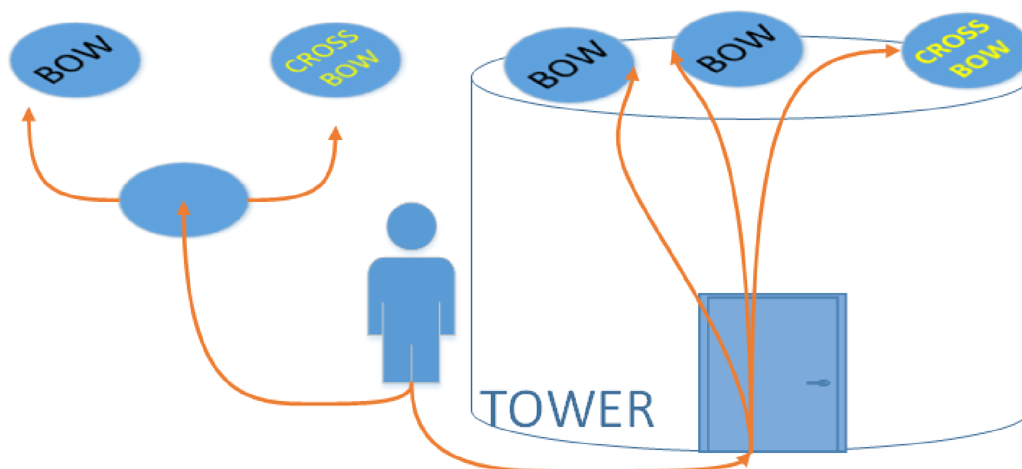

Figure 47: The RKN annotating the NPC's knowledge about the available shooting spots for both Bow and Crossbow. Some of them are at a Tower, some of them are in the open.

Every query has several key parameters to start with – 1) *origin to start the search from*, 2) *maximum depth of the search*, 3) *search pattern*, 4) *tag subselection*, and 5) *logical predicate*. The origin of the search is obviously necessary, since the search mechanism traverses the RKN to satisfy the given predicate and it has to start somewhere. A properly identified origin of the search may produce less computationally heavy pass-through. In some cases, the designer can anticipate the maximum depth the search needs to travel to either be successful or failure, mainly due to the designer's knowledge on how the RKN is structured in respect to that particular search. The search pattern provides the search engine with a mechanism on how to approach the order of expanding edges for the search engine to validate. Most

commonly, RKN queries utilize *Breadth First Search* and *Depth First Search* patterns (Figure 48), which however are uninformed search patterns (Shimon, 2011) thus may lead to unnecessary expanding the RKN's edges. The user may also specify a heuristic based search pattern which would lead the search expansions based on the provided heuristic – e.g. an A-Star search pattern (Hart, Nilsson, & Raphael, 1968). Additionally, a search pattern's choices for expansion may be either static or random. The static selection always follows the same order of expansion on every node, whereas the random selection chooses the next edge to investigate by random.

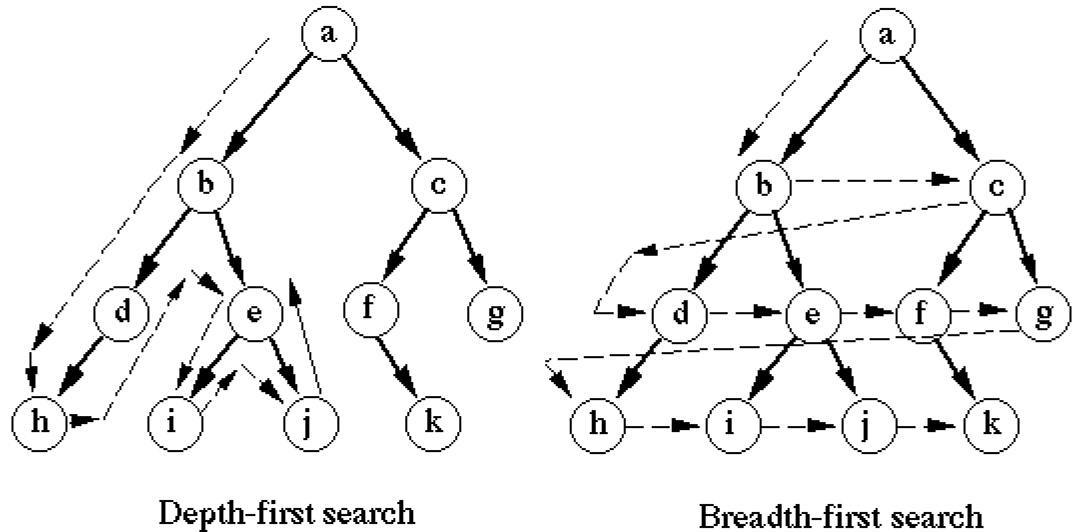

Figure 48: Depth First Search and Breadth First Search

Since the RKN is a multigraph, the designer may want to execute the search query over a subselection of the available edges by filtering out unwanted parts of the graph. The subgraph is specified by the tag subselection which filters out the unwanted edges (Figure 49). Tags are link annotations describing the semantic relation between entities.

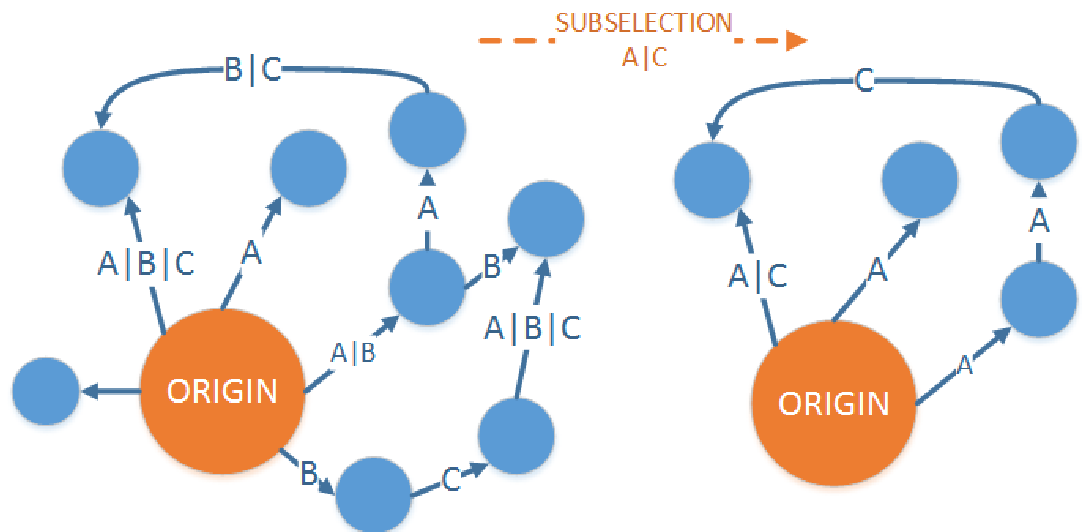

Figure 49: The Subgraph from the RKN based on the Tag SubSelection of (A) and (C) excludes all other edges (B) from the search.

The logical predicate provides the mechanism for the search algorithm to validate edges to be included in the query's result. It can be abstracted as the question the cognition has to answer by exploring the RKN's semantic bindings.

7.4 Query Predicate

Every logical predicate within a RKN search query consists of a non-empty set of Predicate Components (PC). These components are a) *filters*, b) *logical operators*, c) *analyzers*, and d) *sub-queries*. In principle, the evaluation engine asks every PC to answer its particular cognition about an inspected edge in an isolated manner (i.e. PCs have no idea how other PCs have answered). A PC can opt out from being questioned, if it cannot be answered (e.g. a question »Has health above 50%« cannot be answered neither true or false on an entity that has no health).

Beyond asking about the PC's cognition about an edge, the PC answer two more questions about *edge pruning* and *ending* the search. If a PC knows that it is not feasible to continue the search on a particular path within the RKN, it can request to prune the search to avoid further traverse. If a PC thinks there is no need to continue the search, it can request for the search to be terminated.

These PC cognitions about the search are aggregated and their results drives the search algorithm. They represent a triplet `(found,prune,end)` of answers every PC has to provide to the query mechanism. We utilize three valued logic (Bergmann, 2008) to represent the values of all answers (e.g. `prune` → `yes/no/unknown`)

7.4.1 Filter

Filter is simple mechanism that evaluates a provided edge. They range from filters evaluating edge endpoint properties (e.g. »is alive«) to evaluation of edge associated data. Commonly they answer a simple yes/no question on top of their predicate and when positive, they require the search to terminate. Otherwise, filters want to continue with a search. Pruning is specified as a parameter of a particular filter.

7.4.2 Logical Operator

Logical operators are simple PC aggregators either being a) *and*, b) *or*, c) *not* clause. Logical operators simply aggregate the evaluation of their components and present the results as their own.

7.4.3 Analyzers

Analyzers are special PC which do not take part in the actual evaluation of an edge. They are mostly concerned with the aggregated result. Their primary function is to process or output (i.e. write to SBT variables) the validated edges. Secondly, they are responsible for organizing the outputs based on an internal predicate – e.g. sorting the edges based on the proximity of one of the endpoints. Thirdly, analyzers are responsible for managing the amount of positive validated edges since they control the output of the search. They either can provide a quantifier constrain a) *for all*, or b) *exists*, or c) *absolute constrain* (i.e. not more than 5 results) which mainly influence the question about ending the search. Commonly analyzers do not have answer the question about pruning the search.

7.4.4 Sub-queries

Sub-queries are a PC which triggers a separated query within the currently executed search. In principle, the origin of the sub-query can be either the currently validated edge, or a designer specified origin. For example, a query may be verbalized like this »Find a house which has at least one chest which contains a sword« where the search for the sword is to be considered a sub-query conducted from an edge which has »house« as its endpoint. Another query could be verbalized as »Find a house which is targeted by our trebuchet«, where the sub-query runs from the Trebuchet after the search has found a house at an edge's endpoint.

7.5 Adding Dimension

When looking upon the RKN visually, it can be perceived as a graph being on a 2D plane. Therefore, we introduced a concept of adding another dimension to it, by connecting other RKNs via a given set of *transition* edges. These edges have to be explicitly mentioned for the search query to be able to traverse over, since they are filtered by default. Further every transition edge has to be a singular *bridge* into the RKNs component which it connects to the main body of the RKN. Transition edges cannot be combined with other edges (Figure 50).

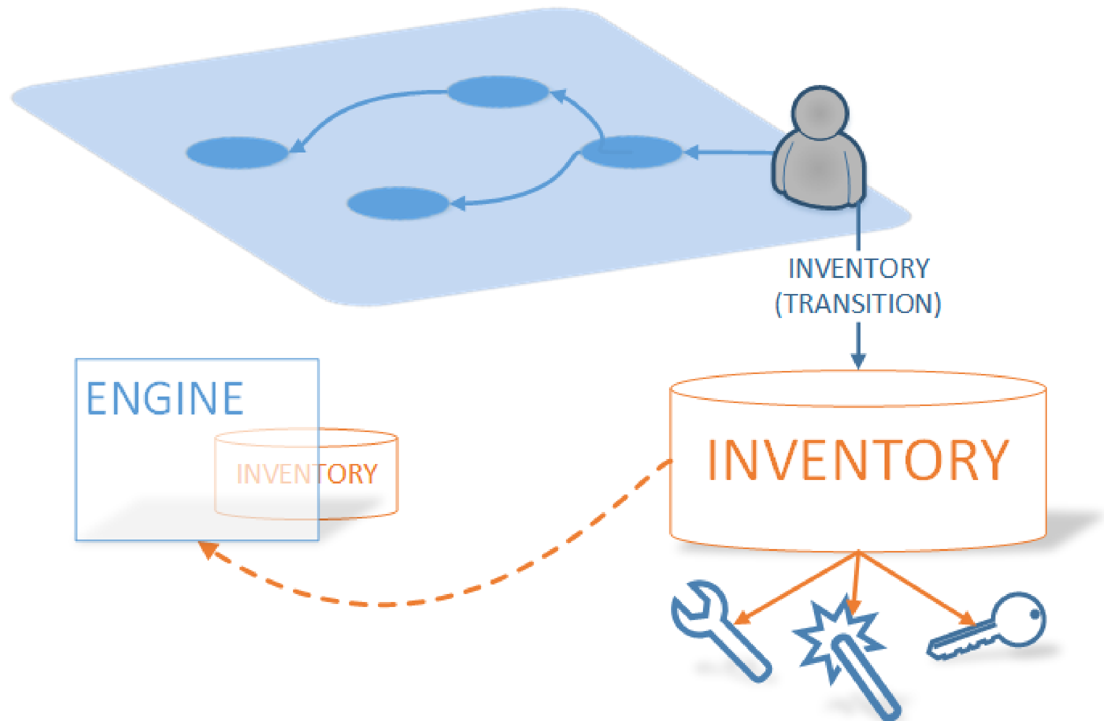

Figure 50: Adding bridge separated components to the RKN graph provides the search with an additional dimension to work with

Further, the components isolated by the transition edges are called *Isolated Graph Components* (IGC). These may be either designed or generated by code. In specific cases, we can make assumptions about the IGC's internal structure, thus optimizing the search, or even caching the previous results of searches made upon them. By engine generated IGCs can provide a unified interface between the SBT query and the engine. An example of a generated IGC is the representation of an NPC's

inventory in KCD. The actual IGC is generated for every search and it is comprised completely out of virtual links. We know that the inventory's IGC is a tree structure, thus we can avoid checking the processed edges in a closed list (to avoid loops). This provides us with an optimization benefit, since we can avoid making costly lookups into a closed list, where already visited edges are stored (to avoid endless loops).

7.6 Search Mechanism

The search mechanism is in principle a continuous expansion of the searched space, where new edges to be evaluated are taken from the search horizon (Figure 51), which represents a set of edges adjacent to already evaluated edges that have not been evaluated. The outline of the algorithm is as follows:

```
1. -> RKN query (origin point, expansion policy...)
2. Prepare Search
3. Initialize Predicate Components
4. Loop until Edges are available
   4.1.   Take open Edge from the Horizon
   4.2.   Pre-Search Step On All Predicate Components
   4.3.   For All Predicate Components
      4.3.1.   Pre-Evaluation
      4.3.2.   Evaluate Edge
      4.3.3.   Post-Evaluation
   4.4.   Post-Search Step On All Predicate Components
   4.5.   Aggregate Results
   4.6.   If Evaluate Search Finished
      4.6.1.   (True) End Search
      4.6.2.   (False) Continue Search
   4.7.   If the Search Be Expanded Beyond the Edge
      4.7.1.   (True) Expand the Horizon with adjacent
          edges. Exclude edges already evaluated
      4.7.2.   (False) Throw edge away
5. Finalize Predicate Components
   5.1.   Produce outputs into local SBT variables
6. Cleanup Search
```

Algorithm 2: RKN Search algorithm

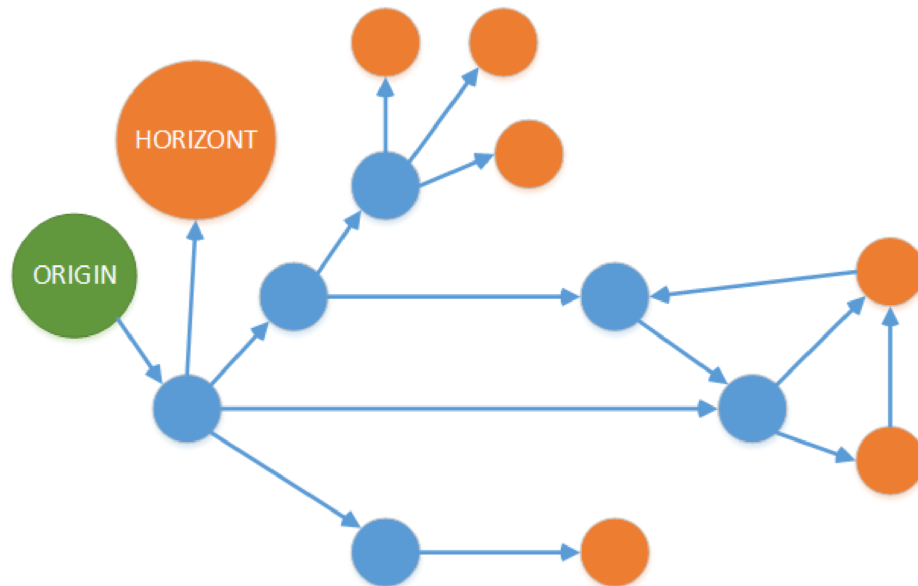

Figure 51: Horizon of expanded edges which have not been evaluated by the Search Mechanism's predicate

As we can see, the search mechanism is fairly simple in respect to traversing the searched space and providing the PC with edges to reason about the search results. There are varying PC with different internal mechanism.

7.7 Example

To illustrate the use of our RKN, we provide a simple example in respect to our Scenario 1 (Brian). We will outline how the RKN can be employed to represent and acquire information about the world. We will also present how the RKN can be used to provides adaptive solutions without the need to change AS descriptions at the NPC level.

7.8 Scenario 1 – Life of Brian

In Scenario 1 (Brian), we focus on Brian, who lives in a small village. The village consists of few houses, a smith's shop, a baker, several field nearby and two taverns. It is populated by at most 50 people, who live their lives in a similar fashion as Brian does. Brian is a simple farmer, who occasionally helps out at the bakery, has few friends and has a feud with his neighbor. He likes to hang out with at the tavern and play cards[16].

As can be seen, Brian's life is quite common. We focus on three basic use-cases Brian's AS can come across 1) going to work, 2) having some fun, and 3) witness the player steal something from the neighbor's house. Brian has the following bindings (i.e. edges in the RKN) to places with the data representing specifics about the binding (Table 1). The presented data is only schematic.

[16] This description reflect the initial stages of design for NPCs within the KCD's world.

Table 1: Brian's bindings (edges) to others. The Target represent the entity the edge points to. Annotation represents the tag associated with that edge, Data represents the associated variables

Target	Annotation	Data
Building	Home	(no data)
Building	Tavern, Fun	Last Visit; Popularity
Building	Tavern, Fun	Last Visit; Popularity
Person	Friend	Popularity, Last Seen
Person	Friend	Popularity, Last Seen
Person	Neighbor, Feud	Popularity
Place	Farm, Work	Progress
Place	Farm, Work	Progress
Place	Bakery, Work	Progress
Place	Smith, Supplies	Inventory

7.8.1 Going to work

When Brian wants to go to work, he simply queries for at least one destination (i.e. Target) that has the annotation »Work« which has the least »Progress«. If Brian is too tired, he order's these places based on distance and chooses the closest one, however dismissing places with »Progress > 80%«. Adding a new work place requires linking Brian to it. Note that the Places are represented by SA and provide a uniform behavior structure, so Brian only calls for BI »work« and the SA takes care of the rest. However, sometimes when Brian needs to repair something, he asks for the closest »Smith's shop« where the necessary supplies are in its »inventory« and the NPC linked to the Smithery is at home (i.e. it is present in the area of the Smithery). It is noteworthy, that the BI provided by the SA can be inferred from a link from the SA on itself, denoting what behaviors it has at disposal (i.e. the reflection principle).

7.8.2 Having some fun

If Brian wants to go to have fun, he chooses the Tavern where most of his »Friends« are. He avoids the Taverns having his less likable friends as customers (i.e. Brian likes to avoid conflicts). When Brian gets to the Tavern, he talks to his friends in the order of him seeing them lately, to catch up with his friends. Again, Brian can be introduced to new friends very easily, just by adding a new link. Brian can also loose friends by removing links. It can be seen the network provides the means to make the virtual world a socially dynamic environment. Brian can use his bindings to other people to determine to whom will he talk and who will he avoid. Thus, the player might need to first gain Brian's trust to be able to talk to him about Brian's other friends, since Brian does not like to talk to strangers.

7.8.3 Player steals from the neighbor

When Brian stays home, he may witness the player trespassing on his neighbor's property. If Brian does not know the player as a friend, he will sound an alarm. However, when the player befriends Brian he can benefit from Brian being angry at his neighbor. Brian does not care for anything happening at anybody's house when the mischief is done by his friends.

7.9 Summary

The RKN represents the necessary means to annotate the virtual world so NPCs can deliberate about the properties and opportunities present in the environment. Since visual perception is computationally heavy, the RKN provides the means to encode easy to access information about nearby entities without the need to employ virtual senses. Queries into the RKN are also useful to discover and infer relations between entities. Our RKN query language allows scripters and designers to encode complex logical information into the environment's annotation. For example, a guard can query the contents of the player's inventory to stolen items that originate from his village, omitting items stolen at location he does not know.

In principle, the RKN is mainly aimed at providing the information about the large-scale world addressed in Goal 5. Without the RKN, we would require encoding information either statically into the NPC's SBTs or into some form of relational database (i.e. SQL). Since SBT execution depends on the results from the RKN queries it is easy to deploy generic NPCs into the world, just by connecting them to their respective environment and let them query for what they need. For example, an NPC has a generic home related behavior, thus the only requirement is to have its home be a SA and link it to the NPC. Everything else will be arranged by the script invoking BI from the linked SA. Another example is to link owned items (SOs) to the NPC. The responsible SBT script at the NPCs AS will discover the attached objects and choose one best suited for the work they intend to do to satisfy a Day Plan's goal. The actual search for an item can be present in the BI provided by the destination area where the work is conducted. The area's BI SBT can query for any item owned by the host NPC limiting it to specific provided behaviors (e.g. a field limits the SO to providing »farming« behaviors).

Further, the RKN supports all the other Goals providing the means to explore the world and its properties. For Goal 1, the ambient environment can be more believable if the semantic relations represent the actual relations between NPCs (e.g. love, hate, parent, child etc.), or represent the affinity to places (e.g. like the tavern, dislike the prison) or activity relevance (e.g. workplace, home, fun and friends). The designer can add new connections between entities to further enrich the environment by new and more believable constructions. The presence of data on the links between entities allows to encode data in respect to the specific relation (e.g. how much do I like a tavern). These can be used to further refine NPCs' and IEs' behaviors .

The capacity to add and remove links provides a mechanism to utilize support for Goal 2, where new emerging situations can be introduced into the virtual world by simply adding or removing links or changing link data. Within our deployment for the KCD game, SBTs have barrier nodes, which watch over the changes in the RKN. This can lead to NPCs reacting on the changing setup within the world's RKN.

RKN is an important support mechanism for Goal 3, where the story can utilize the current state of the RKN network, to choose optional participants (e.g. Scenario 4 (Jerry) choosing people who will get food poisoning at random from the village occupants who go to the same tavern as Jerry). Quests can also modify internal parameters based on specific configurations within the RKN, for example adding guards for quests if the player is too strong at that point in the game (i.e. to provide a more challenging sub-quest).

The overall depth and complexity addressed in Goal 4 is covered by the capacity of the RKN to describe complex relations between entities within the world. For example, the NPC makes choices based on the setup of the nearby RKN (i.e. part of

the RKN close to the NPC in respect to how deep the NPC searches get). The environment and player can change the setup and data of the RKN to influence the NPC (e.g. it starts to rain and the NPC avoids workplaces too far from its home). Furthermore, chaining SA and SO can be inferred in various ways (this depends on the designer's intent and setup of the RKN). For example, tools in a RKN chain with a tag »next tool« can be processed as a sequence of tools to be used to manufacture something. When the player steals a tool, it will disrupt the capacity of an NPC to manufacture the respective product. These can be put into combinations and are easily extensible. Simply chaining the SOs in a SA can lead to rather complex behaviors. All in all, the RKN has proven to be a key mechanism in our KCD integration to facilitate convenient world discovery and environment annotation.

8 Evaluation

We deployed our architecture at Warhorse Studios to be used during KCD's production for the large open world environment. All the above presented mechanisms have been deployed as a standalone artificial intelligence module within the KCD's game framework. In this chapter, we present our architecture's evaluation, to provide insight into its properties and practical use. We split our evaluation into three major topics:

1. SBT Use evaluation;
2. Comparative evaluation of use of SBTs and SAs;
3. Qualitative evaluation of SA and SO use and deployment in a long term large scale production of an OWG;
4. Deployment and integration evaluation.

8.1 SBT Use Evaluation

We have performed two types of evaluation with SBTs – qualitative evaluation where scripters tried to implement the same scenario with different tools and quantitative performance evaluation. In this chapter, we present the summary of our evaluation covered in (Plch, Marko, Ondracek, Cerny, Gemrot, & Brom, 2014).

8.1.1 Qualitative Evaluation of MBTs

We utilize two of our scenarios for evaluating Scenario 1 (Brian) and Scenario 2 (Tavern) since we consider them adequate for testing common use of SBTs in a real-life application. Two of Warhorse Studios scripters implemented both scenarios in SBTs to be used in full production. They also implemented scenarios using other available technologies provided with CryEngine. The first one was plain behavior trees with conditions evaluating only boolean variables and allowing for FSM action selection as tree leafs (BT1). The second comparative implementation was behavior trees with boolean based conditions (i.e. no relational operations) retaining node states (BT2). Both technologies had support for communicating with LUA scripts. Both BT1 and BT2 were hard to develop and eventually regressed to use of lengthy and hard to maintain LUA code.

Both BT1 and BT2 solutions suffered from disadvantages in respect to language expressiveness, where certain concepts inbuilt into SBT had to be supplanted by LUA code. Beyond that, BT1 and BT2 suffered from being too specifically tailored to explicit problem instances, having significant issues in adapting to instance variants (e.g. no two taverns have the same furniture dispositions) or changing requirements (e.g. Brian's life is enriched by one additional activity in the morning – eating breakfast).

Our architecture managed well, since both scenarios were solved without using any custom LUA code as a crutch. The code for Brian had managed his simple life (e.g. »sleep, go to work, have fun, go to sleep«) as a set of reactions to messages from Brian to Brian. Part of his DMM followed the activity schedule and dispatched notifications to other DMM parts to handle them. Adding new activity required only to implement a handler for the activity and it was plugged into the present code without issues. This also showed how our architecture coped with decomposition of behaviors in a manageable way.

Our architecture was able to solve the issues of instance variations robustly, since adding a table in the tavern or changing required only change in data but not of the actual behavior.

Moreover, the scripters considered SBTs relatively easy to learn and did not have trouble understanding its semantics. They were also very fond of the debugging features which were superior to other systems as well as to the LUA implementation we provided. In the end, we started instructing our scripters to use LUA as little as possible for two reasons: 1) invoking LUA environment was computationally expensive, and 2) LUA code was less readable to the scripters than the visual structure of SBTs. On the other hand, nodes invoking LUA proved very useful as a tool to prototype new functionality that will later be added as a specialized SBT node.

8.1.2 Quantitative Evaluation of MBTs

For our quantitative evaluation, we utilized two scenarios 1) simple behaviors, and 2) Scenario 1 (Brian) variant. In first scenarios (simple) we used many NPCs with a simple tree (10 nodes, depth 4). The NPCs moved to random positions at various speeds, while the tree was enlarged by spurious decorators and composites. The Scenario 1 (Brian) was a production variant where several NPCs carried out daily routines – hoeing fields, visiting pub and eating (SBT were populated by with 60+ nodes and maximal depth > 10). Aside from the NPCs, the environment in the day cycles scenarios contained 142 non-NPC entities which also may have some SBT logic to coordinate with NPCs.

Both scenario sets were tested with different numbers of NPCs (Table 2**Chyba! Nenalezen zdroj odkazů.**). All NPCs were updated every frame in full detail. To keep the results meaningful, no CPU budgeting restrictions were enforced — the trees always ran until an action was executed. Data was gathered running the game for 3 minutes, resulting in 3000–6650 captured frames. To reduce noise caused by interrupts from the operating system or other processes, up to 10 outlying frames were removed from each measured category. We imposed a budget of 5ms per frame (on a single core) for the whole AI system's update, including other subsystems as well (e.g. pathfinding etc.).

Even with plenty NPCs on the scene, the average and the .99 quantile performance is far below the limit, although the peak performance is not satisfactory. But as there are high peaks in SBT evaluation, enforcing CPU budget restrictions should effectively cut the peaks, postponing some of the workload to next frame. Since at maximum 1 in 100 frames is over the limit, this will not have any detrimental effect on the resulting behavior.

Table 2:The results of the quantitative evaluation. The table displays mean (left), and .99 quantile and maximum times (right)

	NPC	SBT		DMM		Perception		Other		Total	
Simple	100	0.33	0.6 0.7	0.09	0.2 0.5	0.02	0.2 0.4	0.34	0.6 1.7	0.79	1.3 3.9
	200	0.53	0.9 2.5	0.20	0.3 0.5	0.03	0.3 0.7	0.42	0.8 1.8	1.19	2.0 4.3
	300	0.75	1.1 4.0	0.30	0.5 3.3	0.05	0.4 1.1	0.59	1.0 4.0	1.71	2.6 6.8
Brian	10	0.39	0.7 1.4	0.12	0.2 0.6	0.02	0.1 0.5	0.22	0.5 1.0	0.76	1.1 2.0
	20	0.46	0.7 1.6	0.12	0.2 0.4	0.02	0.6 2.8	0.22	0.6 2.8	0.84	1.4 3.4
	30	0.56	0.9 1.8	0.14	0.2 0.5	0.02	0.1 0.3	0.23	0.7 2.7	0.96	1.6 3.9

8.2 Comparing SA and SBT concepts

Our comparative evaluation is aimed at comparing the capacities of both the SA and SBT approaches in respect to producing believable behaviors in respect to our Scenario 1 (Brian) and Scenario 2 (Tavern). We aim at manifesting behaviors where individual behaviors and their decomposition is connected to the problems within everyday life. In this chapter, we present the summary of our published work (Černý, Plch, Marko, Ondracek, & Brom, 2014). We are evaluating four hypotheses based on the typical use cases in industry development:

1) Learning to create day cycle like behavior is harder using SA than SBTs;
2) New behaviors from scratch are developed faster using SBTs;
3) Existing behaviors are modified more convenient and faster using SAs;
4) SA decomposed behaviors are easier to read.

We conducted the evaluation by using small scale group of 6 males and 2 females. We used the methodology developed by our colleagues (Gemrot, Cerny, & Brom, 2014) for comparing design tools and mechanisms. We also used the evaluation to target usability issues with the production tool chain. The studied group was rather small; however, usability research has shown that even smaller scale groups can provide significant results (Turner, Lewis, & Nielsen, 2006). We also had limited human resources with access to the proprietary technology used for KCD development at Warhorse Studios.

In our experiment, we have used a within-subject due to the small number of participants. We proposed a set of simple tasks for the participants to develop to emulate standard development in a time constrained production environment:

1) Create a simple NPC having a daily routine of 4 distinct behaviors with a fixed order and manifested predefined places.
2) Create two additional NPCs with different daily routines, one having 4 and the other 6 behaviors. One new behavior type was added to the set used in the first task.

3) Make modifications to the behaviors that the daily routine is composed from.
4) Add new places for behaviors to manifest. Let the NPC choose by chance, which place to use.

Our focus is not on the behavior design, since we provided the test subjects with the possibility to use already prepared low level SBT designs. We composed our test group out of two Warhorse employees with experience working with the tool chain and having prior experience designing NPC behaviors. One of the participants from Warhorse may be considered having expert level knowledge about SA use and NPC behavior design. Six of our subjects were university students, only two of those having limited experience with the used tool chain and having no NPC design experience. The students had no prior experience with the SA concept. Two of our subjects had extensive programming experience (more than 90 man-months) and two had limited programming experience (less than 5 man-months). We focused on measuring how long did it take for our subjects to complete the given tasks, with an approximate limit of 90-180 minutes for all tasks. We also evaluated their qualitative feedback via a questionnaire. We analyzed the Task 4 solutions in respect to quality (i.e. by expert inspection) and quantity of used SBT nodes.

Our experiment was divided into two parts where subjects solved the tasks using 1) plain SBTs, and 2) SAs. To test SBT readability, we provided the subjects in Part 2 with the solution to for Task 1 while they worked only on Tasks 2–4. For Part 1, we used only half of our subjects (i.e. the less experienced), where the other half used SA. We also provided a template for the structuring of the SA in part 2, similar to our example in (Section 6.6.4), where the »city« SA was responsible for sending the NPC to the respective SA to perform their daily routine. We provided the subjects with a prepared level, with all NPCs and SA present in the tool chain.

8.2.1 Results

We summarize our quantitative results in (Table 3). On average, subjects took much more time to finalize Task 1 using SAs than SBTs. This can be explained by the lack of experience with the SA concept, thus the subjects had to figure out the mechanisms. We also consider the possibility that utilizing SA requires creating a much more structured content, where SBTs are much less structured in respect to behavior interactions. The data seems to support Hypothesis 1 »that SA are harder to learn« and Hypothesis 2 »that new behaviors are faster to develop using SBTs«. Our subjects using plain SBTs were capable to solve the Task 1 very quickly.

Further, the time necessary to learn to use a novel technology was reasonable (less than 90 minutes for all subjects). The learning time was further reduced if subjects were presented with a working example. All users took less than 30 minutes to replicate or modify the SA example shown to them at Task 2.

The modification tasks (2–4) were faster solved by using SBTs than SA, except for Task 3. Thus, it does not support our Hypothesis 3 »that modifying SA is faster«, although the difference in Task 3 is statistically significant ($p = 0.01$, Wilcoxon paired test), while other differences are not. However, Tasks 2 and 4 when solved using SA incorporate the necessity to learn the new technology. The total time spent on Tasks 2-4 slightly favors SAs over SBTs, however the significance is on the verge of statistical significance ($p = 0{:}05$, Wilcoxon paired test).

Table 3: Summarization of our experiments comparing the Behavior Trees and Smart Areas

	Behavior Trees		Smart Areas	
	Duration in minutes	Standard deviation	Duration in minutes	Standard deviation
Task 1	10.00	3.5	47.25	20.3
Task 2	12.00	1.5	13.50	6.0
Task 3	12.88	5.5	4.88	1.5
Task 4	9.38	3.7	10.63	6.8
Sum 2-4	34.25	9.4	29.00	12.5
Node count	85	14	56	4

In respect to subjective evaluation (Table 4), subjects have identified the task of modifying SBTs more tiresome than modifying SA. Our qualitative data supports the view that modifications to SBTs are prone to manifest mistakes due to the repetitiveness of the task. Thus, the ease of use is compensated by more time-consuming testing and debugging. In general, our data promotes that SAs are better in real production environment, where repetitive modifications and better code structure is required. Our data also showed that utilizing the SA produced almost identical code in all subjects. The SBTs allowed for more freedom to solve the issues at hand. From a production and development perspective, the SA are to be favored since they lead to common design patterns. We also observed the increase in decomposition for the SA approach, where the nodes were distributed between 14 simple trees, in comparison to 3 large trees for the SBT approach.

Table 4: Subjective qualification of the tasks difficulty when creating new SBTs/SA or modifying existing ones. Scale is from 0 – 3, (easy – hard)

	Assignment difficulty	Recreating	Modifying
SBTs	2.7	1.3	1.1
Smart Areas	2.6	0.1	0.3

In conclusion, both our Scenarios (1 and 2) were evaluated in a realistic setting and support our approach as being feasible in large scale development. The learning curve is acceptable for untrained users with or without programming knowledge. Further, both technologies (SBTs and SA) were adopted easily and did not hinder design decisions. The Behavior Displacement was also very easily understood and used.

8.3 Qualitative Evaluation

We present our qualitative evaluation in respect to our collaboration with Warhorse Studios on Kingdom Come: Deliverance during the last 5 years. In this chapter, we present the summary of our published work (Černý, Plch, Marko, Ondracek, & Brom, 2014). We split it into two parts:

a) We present the practical achievements within the KCD game made by utilizing the concepts and mechanisms presented in this thesis.

b) We present a set of interviews with some members of the script department who daily engage with our technology.

8.3.1 KCD Integration and Deployment

Presently our technology and concepts are integrated[17] as a key system running the entire virtual world, both ambient and individual NPCs. In principle, the AI Engine which is mainly build on top of ideas presented in this thesis, represents one of the game's fundaments. Presently all game mechanics in respect to the virtual world are implemented using our approaches and methods, on how to specify and decompose behaviors. More than 80 quests are implemented via the use of SA and SO, where SBTs are the primary language for scripters to express ideas and manage the world. The 15-people strong script department uses our technology to perpetuate the world to be as believable as possible. Other technologies, like a script based crime system were built on top of SBTs combined with SAs and SOs. This proves the quality of the overall approach, since such long deployment presents a unique capability test.

From out observations, the concepts of SA and SO are a natural way to express design decision. They allow for a natural decomposition of behaviors in respect to both places and objects in an encapsulated way. We have learned that the management design pattern is a strong and useful mechanic to employ for coupling SA with another SAs or SOs. In our common case, the SA represents a management entity which governs the SA and SO contained within. The SO are responsible for managing low level interactions, and instruct NPCs on how to directly interact with them. Interactions between SOs is commonly carried out under SA supervision, which orchestrates the proper use and cleanup of the utilized SOs. This provides strong control over what is happening in the world and who is responsible. The BI provided by the SA includes the further BI by a governed SO, thus creating deeper SBT structure and not a broad one.

We also did observe that scripters make use of the Intelligent Environment concepts, often combining both Smart constructs and IE principles. It is common to populate the world with objects that provide BI to their users and within those behaviors, they introduce a communication mechanism to their IE part. A »chicken leg« eaten by both the player and NPCs provides the functionality to »eat« where animations are specified. However, the injected behavior also sends messages back to the chicken leg to tell it how many times an animation was played, so the IE Object may change its visual appearance (e.g. the meat actually is vanishing). All the respective behaviors are encapsulated (both NPC and IE Object) and only loosely coupled. The NPC has no knowledge about the communication mechanism involved in its action. In some cases, we introduced such complex interaction at a later stage of development, and there was no need to alter the NPC's AS, only the SO's and IE Object's internals[18].

We employ the SO concept in every interactive entity within the world, and since the player is an NPC from the SO's perspective, he utilizes the same mechanisms as NPCs do. Further, every door, chair, table and bed (and many more) are SO, being more than 15000 within the game. These provide various mechanisms (e.g. people are coordinated when somebody wants to sit on a bench and they need room to scoot by) and contextual behaviors (e.g. doors provide complex context sensitive behaviors

---

[17] The AI integration team headed by Tomáš Plch consist Matej Marko, Martin Černý, Petr Smrček and Martin Štýs, with support by the scripters department's Petr Ondráček, Michal Vrtílek, Martin Antoš, Petr Maláč and many others within Warhorse Studios

[18] The chicken leg is both an SO and IE Object at the same time

on how to handle their locked state, they also can recognize who is walking through and who waits on either end).

The entire KCD quest system using SO and IE Virtual Observers to manage complex quests with multiple endings. Quests reside within the world as hidden objects and are both integrated into the RKN and influence what is happening in the world. The Quest SO communicate with the questing backend over messages. The Quest SO represent an entire world on its own, commonly residing deep under the landscape.

We also used SBTs to prototype the low-level AI for combat behaviors. Due to optimization necessity, we had to rewrite it into C++ code, to allow large scale battles. But the targeting system for combatants still runs as a SBT implementation in the Combat SubBrain of every combat capable NPC.

One of our most interesting achievements is the integration of a believable, research based (Pitnerová, 2008), sheep AI implementation fully written using SBTs. We focused on realistic sheep behavior, in respect to their daily routines and their heard behavior. This realistic behavior allows us to actually include the natural behaving sheep into our quests concerned with helping sheppards and finding lost sheep.

Overall, from a usability standpoint, our architecture was proven repeatedly to provide well manageable results in large scale production of an OWG game.

8.3.2 Personal Feedback via Interview

In this sub-chapter, we present the summary of our published research (Cerny, Plch, Marko, Gemrot, Ondracek, & Brom, 2016). We conducted two rounds of interviews with 6 Warhorse scripters who were engaged with our technology on a daily basis for more than 4 years. Our selection in subjects was limited due to the fact that there are no more scripters with such long experience at disposal and invited subjects would be of no practical use. We chose these scripters due to their experience and lasting exposure to our technology. However, their individual responsibilities in respect to utilizing our technology differ.

We build the structure of our interview based on five basic usability measures (Shneiderman & Plaisant, 2005): time to learn, speed of performance, rate of errors by users, retention over time and subjective satisfaction.

Our *first set of questions* (Table 5) is focused on typical use of SBTs and Smart constructs in the KCD's AI system. Each interview took from 30 to 60 minutes. We try to frame the questions in the manner to avoid bias in respect to the used technologies (i.e. Smarts, SBTs) and investigate the possibility of alternative solutions. Our intention is to evaluate the system as a whole.

Table 5: First set of questions. Every question is stated in the upper part, where the reasoning behind it is in the lower part

Question 1	What were the tasks you worked on recently?
	Frame the interview and provide source for specific examples for the rest of the interview
Question 2	What activity consumes the most of your development time?
	Discover the main bottlenecks for production.
Question 3	Give an example of a code segment/snippet that is often repeated across behaviors and has to be copied each time and a segment that is well reused across behaviors.
	Discover a situation where Smart constructs are not applicable in practice, although they should be in theory. Understand the potential for AI code reuse.
Question 4	Describe the process of implementing a behavior from a design request to the final code.
	Discover how Smart constructs fit (or do not fit) in the overall production pipeline.
Question 5	How would your behavior code change if you could only use plain tree injection (without Smart constructs).
	Understand what features of Smart constructs are considered important.
Question 6	What was the most complex/difficult task you have worked on in this company?
	The most challenging tasks are likely to demonstrate the full power (or lack thereof) of a system.
Question 7	Describe the process of resolving an issue reported by the Quality Assurance department.
	Discover whether Smart constructs help/hinder debugging.
Question 8	What do you dislike about the scripting tools?
	Gather all the problems scripters face when writing code.
Question 9	Describe your ideal scripting tool.
	Gather constructive suggestions and let the scripters compare Smart constructs to hypothetical alternatives.

Answers to questions 4, 6 and 7 did no yield any useful insight into how Smart constructs were used. Answers to questions 3, 8 and 9 provided us with feedback on the overall usability of the system, mostly focused on the tool-chain (i.e. editor and debugging tools). Problems with debugging larger SBTs were mentioned by 4 scripters overall (e.g. »large trees do not fit on a single screen«). This is observed mainly due to the fact that the SO/SA decomposition with the management pattern employed (often used technique at KCD) leads to deep trees. A single scripter complained about the need to connect the SOs representing quests into the RKN even when there is no natural connection[19]. Overall, Smart constructs were not presented as the source of scripter's frustration.

Answers to question 2 have pointed out that debugging and refactoring of SBTs due to code changes were the most frustrating and time-consuming for scripters. Both issues were mentioned by 4 scripters in total. One scripter complained about the complexity of creating of synchronized behaviors. Another scripter reported that figuring out how to translate a broad design specification into a SBT implementation is his most time-consuming task. Answers to question 3 provided no valuable data since the examples for encapsulation were too specific to our system, to be able to

[19] This is a implementation specific necessity in the KCD game

compare them with other similar systems. Our subjects provided us with 7 examples of repetitive small scale SBT code (around 8 nodes) which was not suitable to be implemented as BI via SO/SA use. This indicates that the SO/SA are more suitable to decompose problems into larger structures, but a different approach has to be provided to create small reusable SBT snippets.

Best input in respect to using Smart constructs was provided in answers to question 5, where our subjects reported that if not for Smarts, they would reimplement the a) capacity to connect behaviors and data (4 mentions), b) messaging between entities (manager pattern) where handling of context specifics are handled (3 mentions), c) central NPC logic (i.e. brain) for managing a set of behaviors (2 mentions) and d) a container aggregating related behaviors (1 mention). One of our subject stated that he would implement a similar architecture. The principles of decomposition and encapsulation, as well as a hierarchy of control and coordination were recognized as important capacities of our system.

Our *second set of questions* (Table 6) was aimed providing information about the utilization of our system to achieve design goals. This part of the interview took from 10 to 30 minutes.

Table 6: Second set of questions aimed at feedback in respect to design problems when creating the believable ambient environment

Question 10	When writing code, do you take into account the possibility of interruption by quest/combat? How?
Question 11	Is there a difference in using BTs and BOs in quest logic and in ambient AI?
Question 12	What are the necessary steps to place a new instance of an SA/SO in the game world?
Question 13	Have you implemented any behavior where NPC attributes would change the way the NPC behaves in a given context?
Question 14	What was the most difficult synchronization/coordination task you implemented? Why?

In respect to question 10, only one scripter reported the need to implement a complex enough SBT logic to accommodate for being interrupted by the Dialog SubBrain by suspending the Day-by-Day SubBrain. The primary issue was the necessity to resume the entity into the interrupted task so it would seem uninterrupted. Since this issue was mainly concerned with animation continuity, we solved it by using a twin character for the dialogues and pausing and hiding the talked-to NPC. Four of our subjects have implemented simpler variants of a interruptible behavior by halting the subtree within the Day-By-Day SubB not needing to resume. Two scripters stated halting the SubBrain was easy to integrate. This shows that the NPC architecture can handle interrupts well.

Our question 11 was aimed at inspecting the coupling between the ambient AI and the questing system. No issues were reported. However, one of the mentioned issues was the complexity of the adjacent systems like Quest Management and Dialogue System. Scripters are required to interact with several different systems to integrate a quest and the logic can be distributed amongst these system in various measures, thus creating different approaches for handling quest design. Quest SO's SBTs were considered simpler to implement than SBTs for NPCs. In principle, scripters noted that the Quest SO acts as a central entity that coordinates quest participants as well as the quest's progression. This indicates that the design of the system is followed as intended.

All scripters have reported that creating a new instance of a SO only requires linking the SO to the appropriate environment within the game. Two scripters explicitly stated the process was quick, however two other scripters reported issues with the tool chain responsible for managing the RKN. This indicates that the system allows for logic decomposition to separate data and code, where reusing an SO is as simple as putting it into its proper context.

Only one scripter implemented a system investigated by question 13, where the NPCs behaved differently based on their attributes. The use-case was a military camp where different work was assigned based on the NPC's rank. The implementation was straight forward without any issues,

Question 14 was focused on synchronization and coordination between NPCs. All scripters encountered the issue of explicit synchronization. Two scripters encountered this issue in a simple context. One scripter needed to address this issue outside of a Smart construct's scope and described it as being difficult to handle. One scripter mentioned Smart constructs being helpful with maintaining coordination. Three scripters considered the synchronization to be non-problematic. However, two scripters complained about debugging issues. One scripter reported that he considers parallel behavior's a challenging principle. Two scripters reported that writing message oriented SBTs as tedious.

In respect to quests, two scripters have addressed the issue of reducing available NPC states to be able to coordinate in respect to a quest. NPCs participating in a quest are instructed to stay at specific places and maintain only simple and limited activities, so simple assumptions about the overall setup can be made. One scripter utilized the messaging system for synchronizing use of a door (i.e. Navigational Smart Object) when traversing it.

Overall, scripters are not complaining about using SBT and Smart constructs and have naturally absorbed the system's principles in respect to behavior decomposition and encapsulation.

8.4 Industrial Deployment Evaluation

Since our system was deployed as being part an industry project Kingdom Come: Deliverance, we also provide an evaluation of overall performance and utilization of our architecture. We have conducted measurements within the running Beta development stage (Chandler, 2009) of the game.

The game consists of one large scale level being 16km^2 of surface area. The level is loaded as a whole, but parts (e.g. animations, object meshes, textures) are streamed into memory and Level of Detail (LOD) mechanisms (Brom, Poch, & Serý, AI Level of Detail for Really Large Worlds, 2010) are used for all entities (i.e. NPCs, SA, SO ...). The world's population is split into persistent NPCs (around 600) and semi-persistent NPCs. Persistent NPCs inhabit the world, where semi-persistent are spawned to provide the feeling of the world being filled with life. However, semi-persistent NPCs also have a Day-by-Day life but with more limited goals and are more generic in nature. Persistent NPCs have persistent recollection of player's activities and maintain a relationship to him and an attitude towards his actions. We also have non-persistent NPCs, which exist only at specific locations and most often represent wildlife within the game's woods.

We measured the game's performance at three different location in respect to level and game progression:

a) *Start of the Game at Skalitz village* – the game starts in an isolated environment where the player is tutored about to the game's mechanics. The number of active NPCs is limited to the village's population, where other NPCs within the world are disabled (i.e. the open world is not accessible at this point).

b) *Big city of Rataj* – the open world is fully accessible and all persistent NPCs are active (but may be in LOD behavior). The city of Rataj is full of people who engage in their daily life.

c) *Battle* – two groups of soldiers engage in a battle. The focus of the game is on the battle and all other parts of the open world are not accessible and tend to be LODed out or disabled.

Our setup PC is a Core i7 4790K 3.6GHz, 32GB of memory, running a release version of the game (i.e. the game's source code is optimized but all debug and error messages are logged). We used the GeForce GTX760 as the system's graphics card. The game was stored on a SSD drive for faster access to streamed resources.

8.4.1 Frame Time

We measured the frame time in 5 measurements in 1) Skalitz standing around, 2) Skalitz roaming around, 3) Rataj city standing around, 4) Battle and 5) Long term gameplay.

During a common day in Skalitz, when looking around (Figure 52), frame time is acceptable around 25-30 milliseconds (i.e. around 30 FPS) except for spikes when more NPCs are present in the main village's square. We identified these spikes as waiting for rendering to finish. The frame time is fairly consistent thus avoiding frame rate variation and frame rate drops are only occasional.

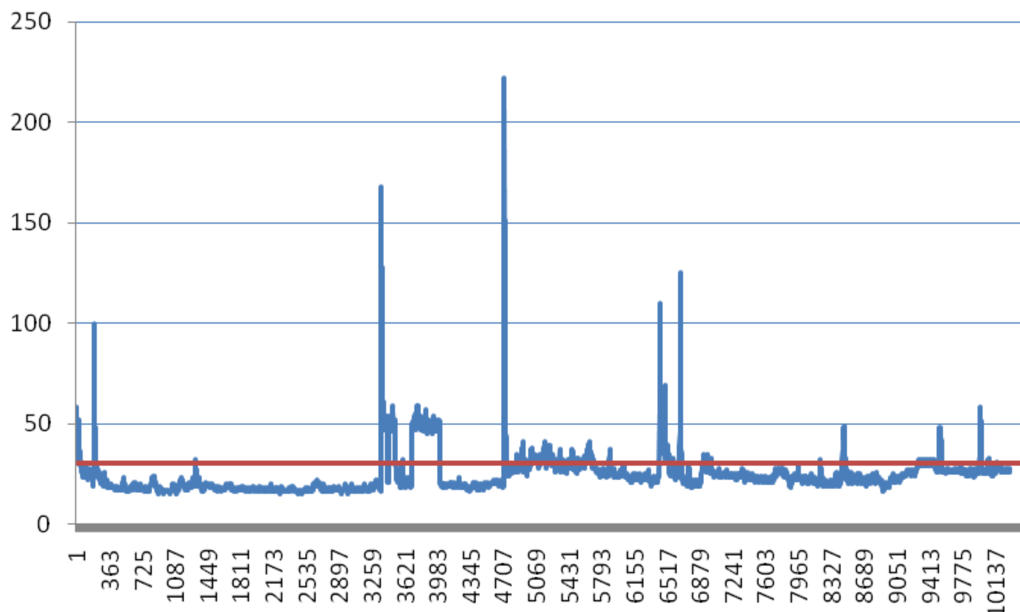

Figure 52: Skalitz, Common day, frame time in millisecond measured over 10000 frames, standing in one place and looking around. Red line denotes 30ms.

When roaming around (Figure 53), the frame rate is more stable because the player encounters less NPCs in one location, thus the render is less under load from the environment. The spikes at the beginning of the measurement can be attributed to the initial location at Skalitz being full of NPCs.

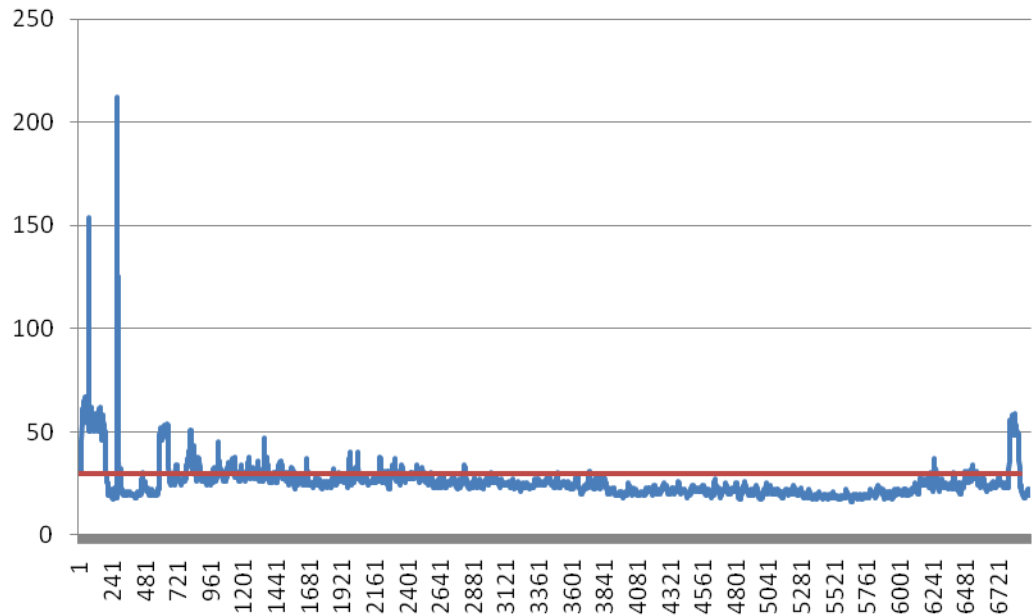

Figure 53: Skalitz, Common day, frame time measured over 6800 frames, roaming around. Red line denotes 30ms.

Since the City of Rataj (Figure 54) has more NPCs and is much more complex in respect to geometry and number of rendered entities, the frame times increase thus proving more frame drops and the FPS fluctuates more above the 30 FPS target border. The spikes can be linked to rendering, where multiple animated characters burden the rendering engine too much.

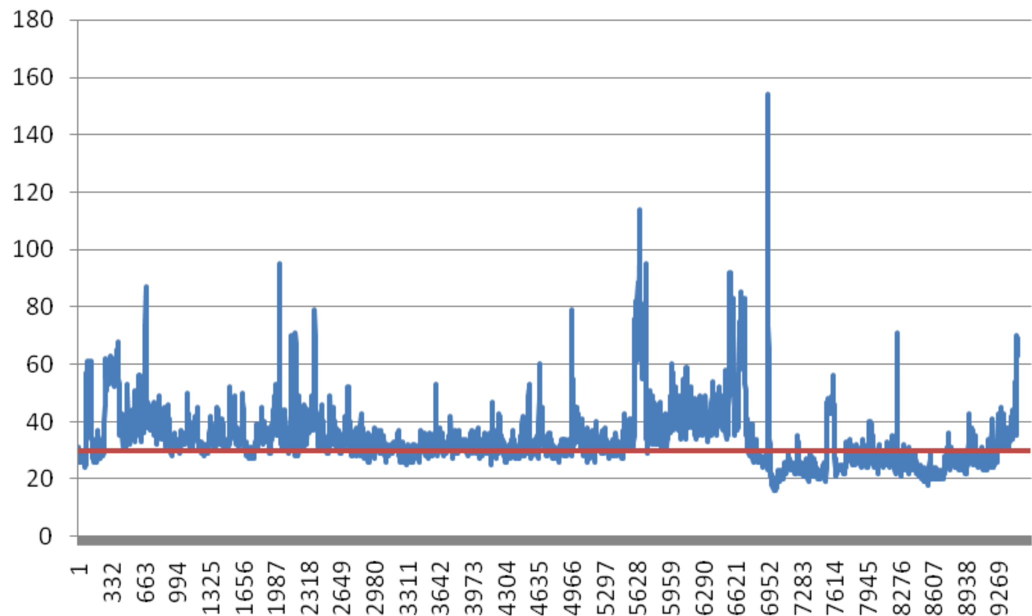

Figure 54: City of Rataj, Common day, Frame time measured over 10000 frames, standing and looking around. Red line denotes 30ms.

In a short and small battle (Figure 55) (around 30 soldiers in total) the frame time fluctuates steadily around 20 milliseconds with occasional spikes. Stable FPS is due to the fact that most of the fighting NPCs are handled by automated systems within their decision making and are the overall load is spread by a scheduling system.

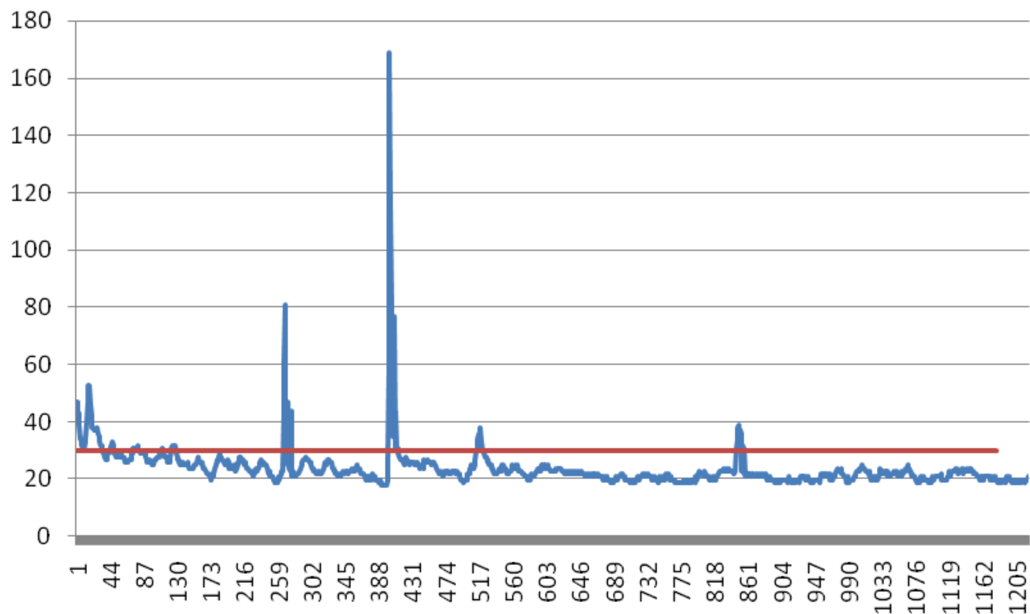

Figure 55: Battle of 30 soldiers, Frame time measured over 1200 frames, taking part in the battl. Red line denotes 30ms

During a long-term gaming session (Figure 56) we were running around in the country and visiting villages and talking to NPCs. Overall the frame times were well below 30ms threshold and the fluctuations were kept to a spread of 5ms around a mean value of 20ms per frame. This is due to the fact that the animations of characters take a heavy toll on the rendering process. Also, there are less NPCs outside of the bigger cities, thus our architecture can spread the resources in a more efficient manner.

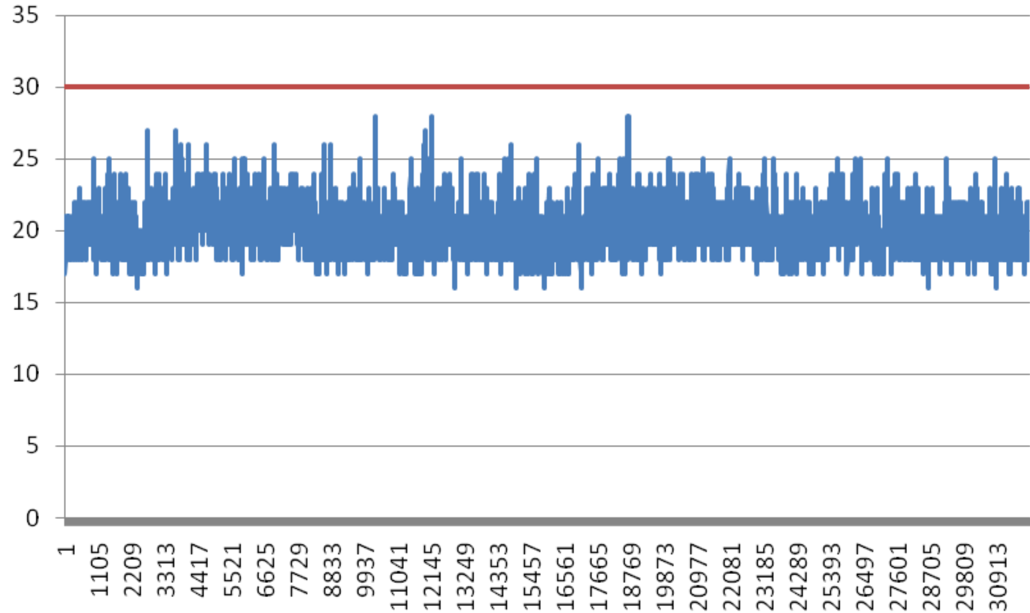

Figure 56: Long term game play running around in the country side, visiting villages and talking to NPCs, occasionally engaging in a fight. Red line denotes 30ms.

8.4.2 Overall numbers

In the KCD game we collected statistics about how many instances and templates of given classes are present. Active NPCs are all NPCs presently doing action selection within the virtual world. These NPCs' DMM can be suspended due to all their AS nodes being passive and waiting (e.g. message read is waiting for a message to arrive at an inbox). All AI Objects contain all the relevant objects within our framework, accounting for all objects which can be accessed over the RKN (e.g. tag points, paths, NPCs, Smart Objects, Player, NPC actions etc.). Smart Objects and Smart Areas have a varying number of instances within the environment, since they can be spawned on demand or present within level layers loaded during runtime. The minimum amount of SAs and SOs represents how many are present in the virtual world from start (i.e. at the load of the game). All SAs and SOs have their respective template from which they are instanced into the world. Further, every SBT within the game has a specific template from a repository of templates. These are loaded or constructed on demand, thus their range can vary. Variables include all typed data containers, including variables in messages and on links. The amount of links in the RKN fluctuates due to the fact that we account for virtual links. However, adding and removing links is an often-used practice to store run-time information about NPCs and their respective relations. The inbox count varies based upon which trees are injected. The node count also varies in the same manner, but the base of 90000 represents the minimum instanced nodes aggregated over all SBTs.

Table 7: Overall amount of entities and constructs within the KCD data

Active NPC Count	600 – 640
All AI Objects	72000 – 79000
Smart Object Instance Count	15180 – 15300
Smart Object Templates	380
Smart Area Instance Count	964-987
Smart Area Templates	256

Loaded Tree Templates	3919 – 4267
Variables	~ 320000
Links in the RKN	37830 – 39092
Inbox Count	44346 – 48887
Node instances	90000 – 100000

8.4.3 Messaging

The load of the Messaging system heavily depends on what is currently happening in the world. We have focused on 3 scenarios 1) Skalitz, 2) Rataj, and 3) Battle.

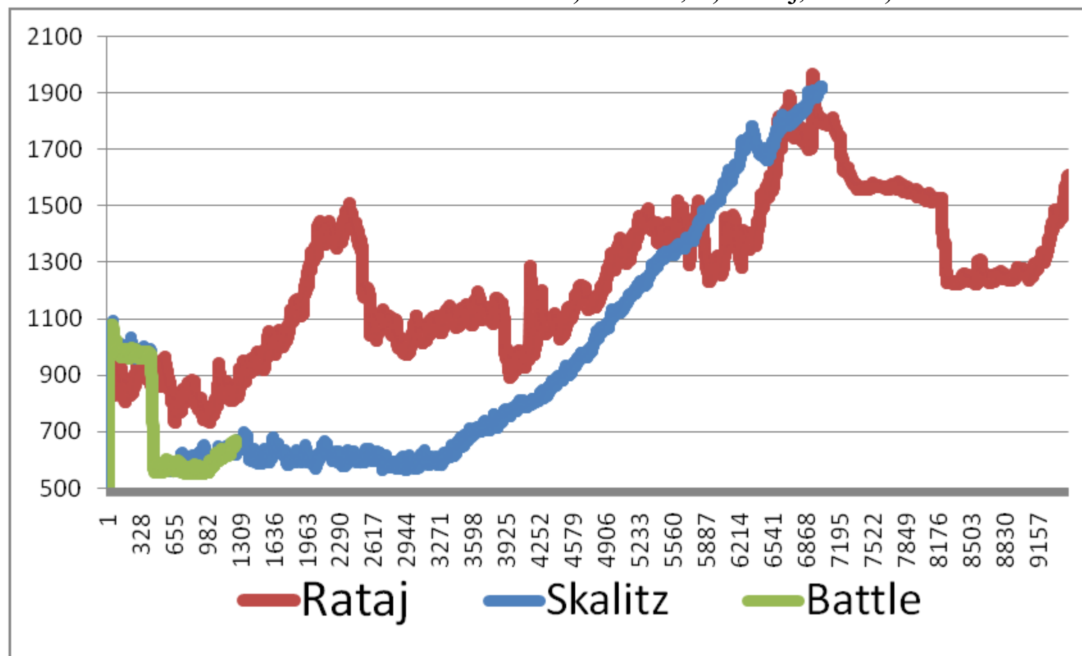

Figure 57: Inbox congestion for Skalitz, Rataj and Battle scenarios. The battle is short in comparison to other scenarios.

In (Figure 57) we show how the congestion of inboxes behaves over time of up to 9000 frames. In the Battle scenario, messaging is used to inform soldiers about what will happen in battle, thus the congestion of their inboxes is high at the beginning but when processed, the inboxes stay relatively empty. In the Skalitz scenario, the inboxes are saturated due to the fact that the NPCs figure out what to do in the world and communicate with other entities (e.g. IE areas and IE objects). At the Rataj scenario, the inbox congestion oscillates in respect to what time of day it is and what the player does in the city. It is important to note that one message can be present in more than one inbox, if multiple inboxes at one NPC accept it.

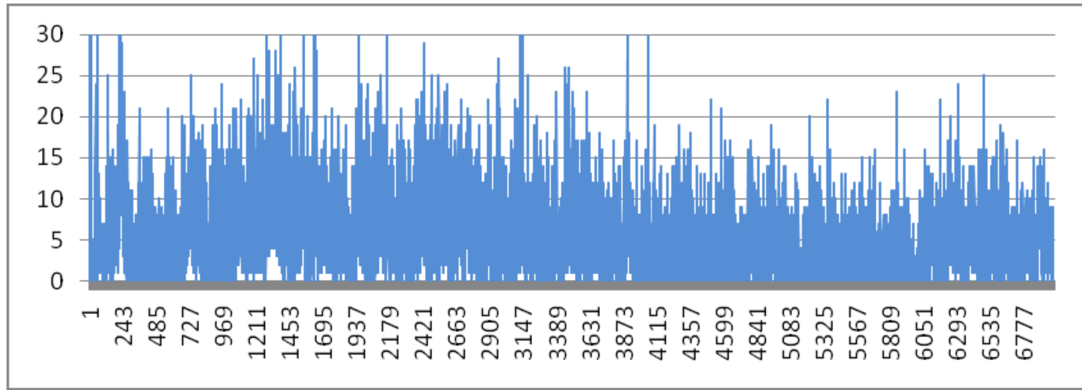

Figure 58: Message pickup and processing for 6800 frames at the Skalitz scenario

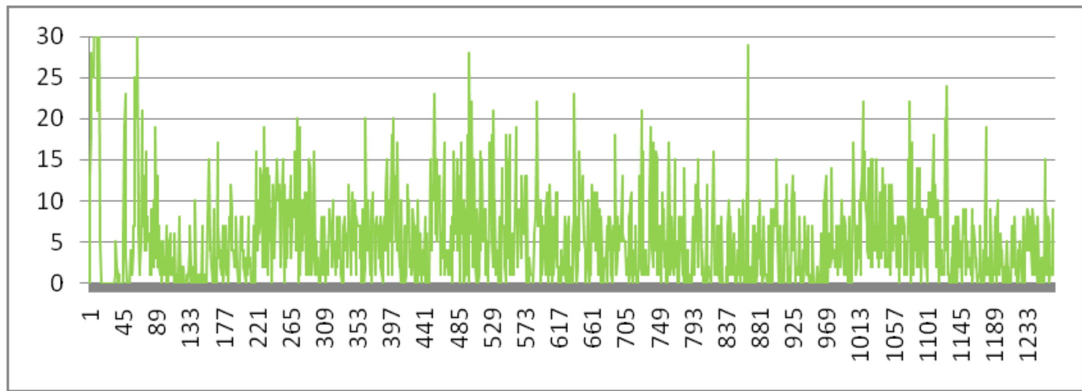

Figure 59: Message pickup and processing for 1200 frames at the Battle scenario for 1200 frames

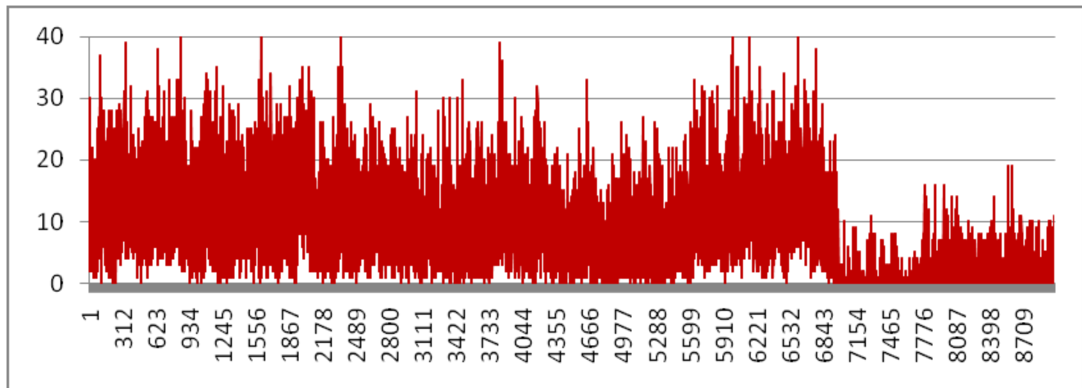

Figure 60: Message pickup and processing for 8700 frames at Rataj

In (**Chyba! Nenalezen zdroj odkazů.**)(Figure 59)(Figure 60) we can see the constant pickup and processing of messages in respect to (Figure 57). It can be seen that the processing is more intense if there are more messages present. This allows for a constant capacity to respond to environment's stimuli.

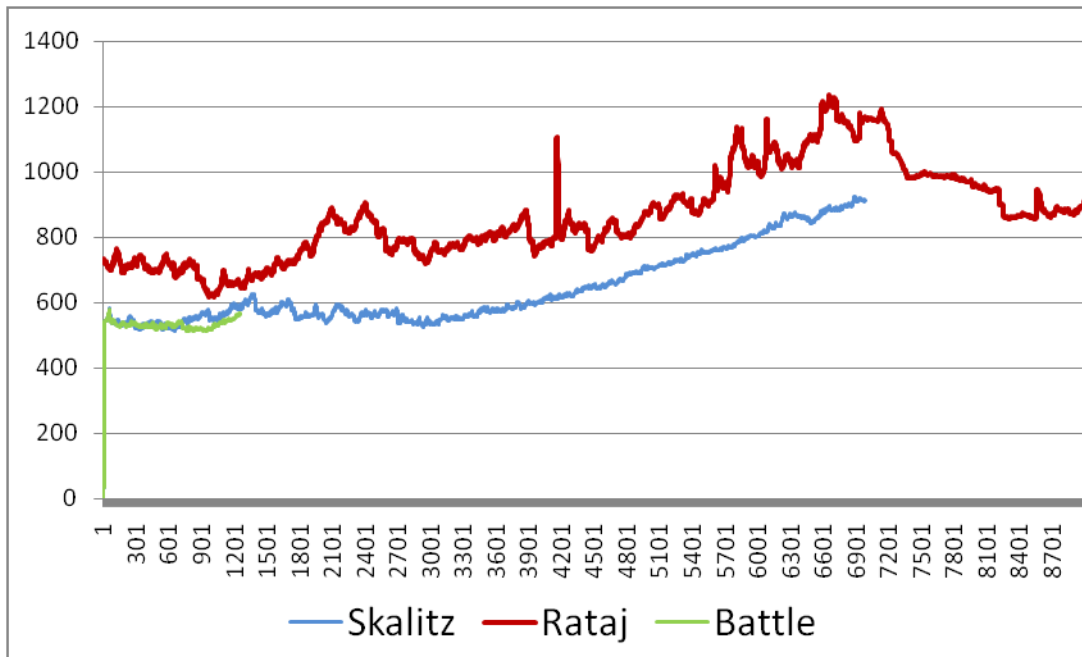

Figure 61:Message count exchanged over the messaging system

As expected the congestion of the messaging system (Figure 61) is structured in the same fashion as the inbox congestion presented in (Figure 57). The drop-in message processing in (Figure 60) mirrors the drop at the end of (Figure 61) since the messaging systems processing is in tandem with the messaging congestion and inbox fill.

8.4.4 Intelligent Environment and NPCs

As can be seen in (Table 7) there are plenty of NPCs, Smart Objects, Smart Areas and components of the Intelligent Environment. To be able to handle such large-scale loads of active entities, we have to manage computational resources in a very strict manner. We focus on updating only those entities which require computational time to advance their decision making, thus if an entity is only waiting for an event (e.g. a message arrives) we suspend its DMM to avoid CPU consumption.

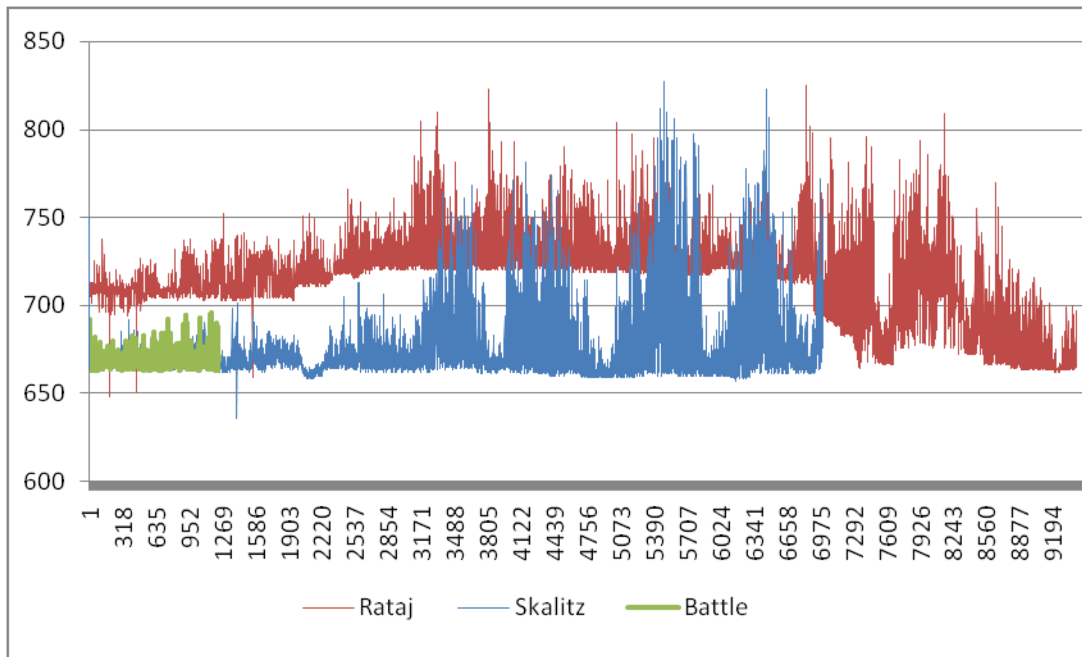

Figure 62: Active AI entities in different scenarios. All these entities receive an update from our framework

In (Figure 62), we present how many active AS capable objects can run within our architecture. These objects receive an update call and can utilize the given budget. In the Battle scenario, only a fairly limited amount of entities is updated, since only those participating in the Battle are of interest. In Rataj, we can see an overall spike in active objects, since the city is much more crowded. However, the overall spikes in Rataj are fairly contained to avoid overconsumption of computational resources. The scenario in Skalitz has multiple spikes due to the fact that this is the start of the game and many entities are using this time to discover the world and setup their respective role within it (e.g. the Tavern communicates with all the associated objects). In all cases, the running NPC count is stable (Figure 63).

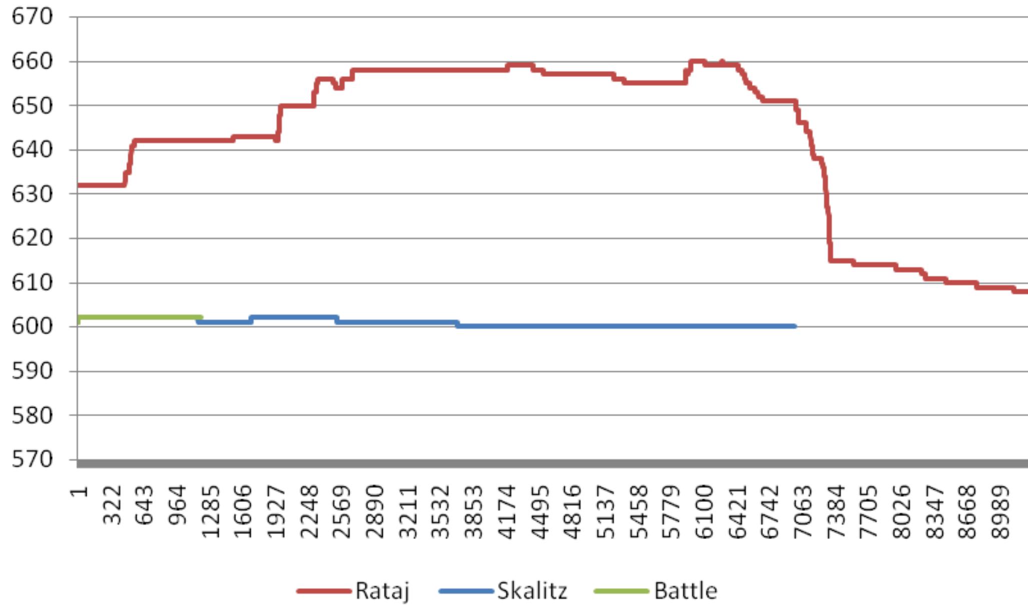

Figure 63: NPC counts in various scenarios tracked over 9000 frames

The updates of the IE Objects (Figure 64) and IE Areas (Figure 65) heavily depend on where the player or NPCs are at the moment. In principle, most of the IE Objects and IE Areas are passive most of the time, waiting for someone triggering an event or entering an area. However, IE Virtual Observers are often actively (e.g. every 5 seconds) scanning for some condition to be valid, thus triggering something as a follow-up. We can see that in all the presented scenarios, the overall update of SAs and SOs is identical and kept on a very strict budget, to avoid overconsumption of resources.

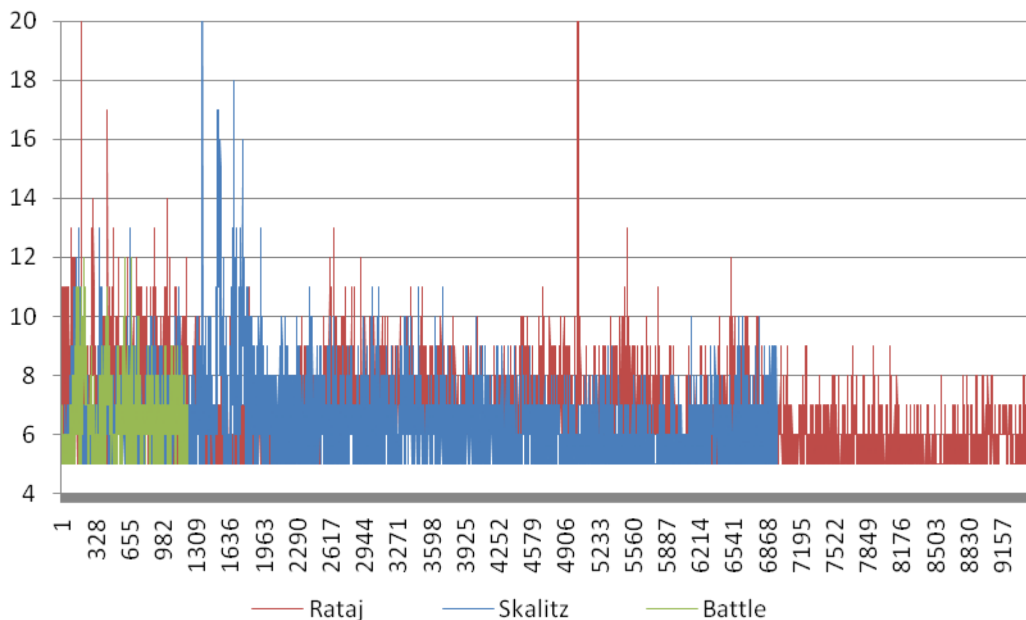

Figure 64: Intelligent Environment Objects updated per frame

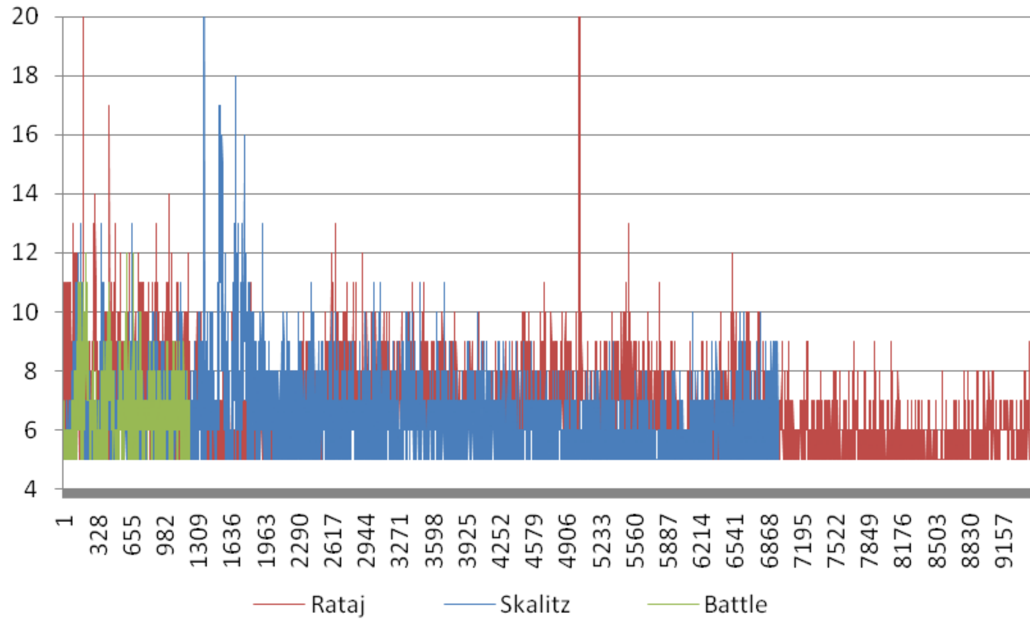

Figure 65: Intelligent Environment Areas updated per frame

8.4.5 Updating SBTs

The SBT is the primary action selection approach in our tool chain. Besides using the SBT we utilize custom build mechanisms for specific tasks which require complex analysis (e.g. combat movement and cooperation) or are computationally heavy. However, SBT nodes (Figure 66) are the primary mechanism to make action selection for NPCs and other AS capable entities.

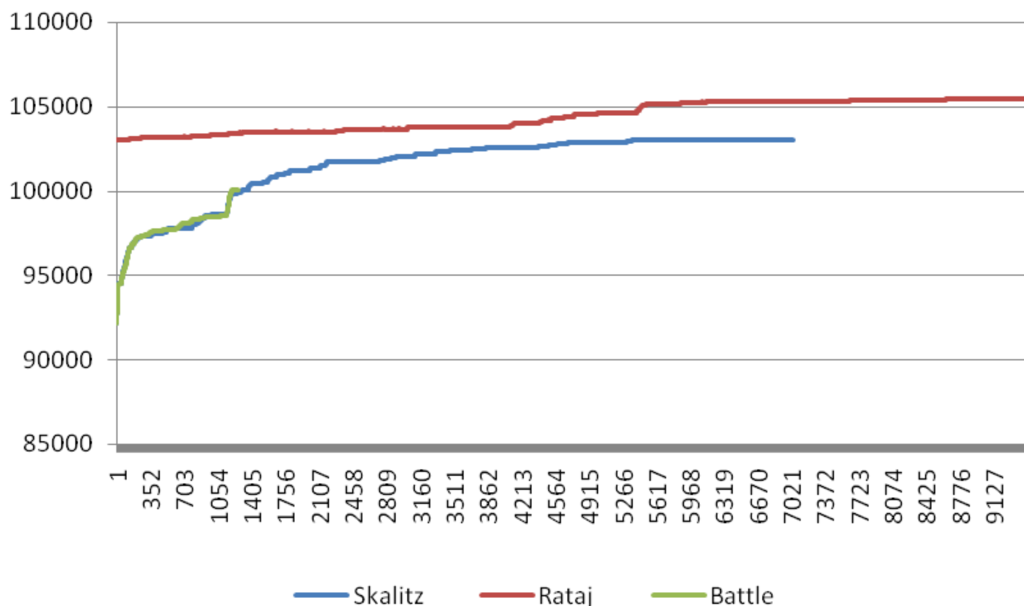

Figure 66:: Total number of node instances aggregated over all SBTs

In (Figure 67) we show how many node updates can be processed every frame. To maximize responsiveness, it is important to maximize the throughput of node

updates. We employ a scheduling mechanism with several layers of priorities and a budgeting system to allow for as hundreds of nodes to be updates per frame.

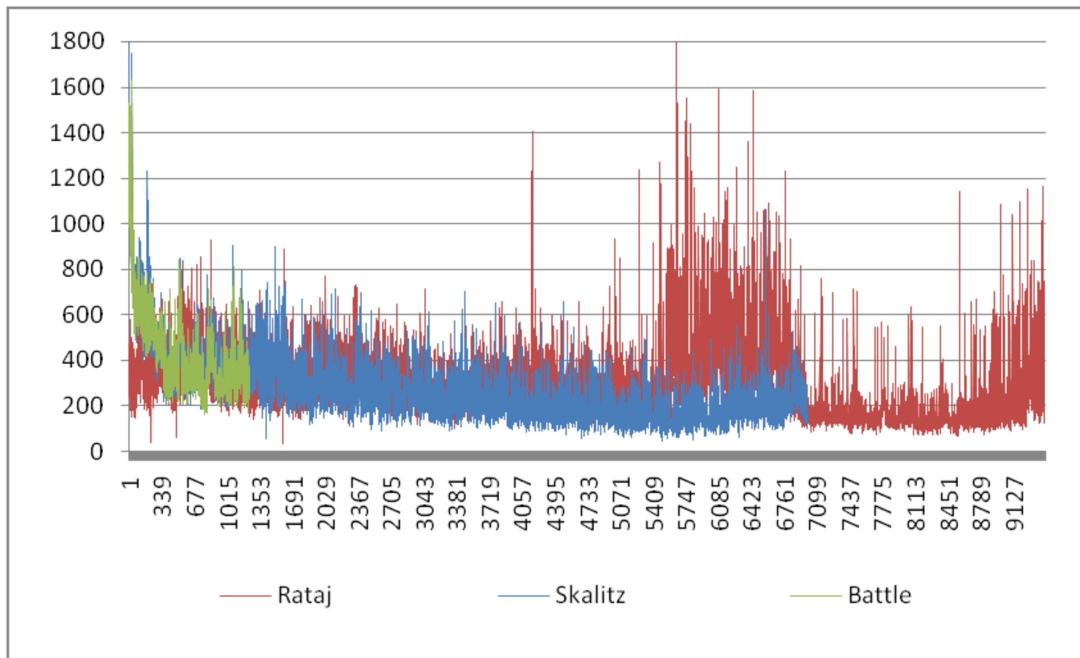

Figure 67: Node updates per frame

In (Figure 68) illustrates how many unique SBT are active over a period of up to 9000 frames. Every of these SBTs has at least one updated SBT node thus doing some decision making. The number of trees does not coincide with the number of NPCs but they are very close, thus the architecture can manage a fairly decent throughput of NPCs doing decision making. It is noteworthy that it is not required for every NPC to update every frame, thus spreading the load of NPC execution over several frames. Also, often there are several nodes updated at one NPC at once, to make use of cache coherencies and memory proximity on both node code and node data.

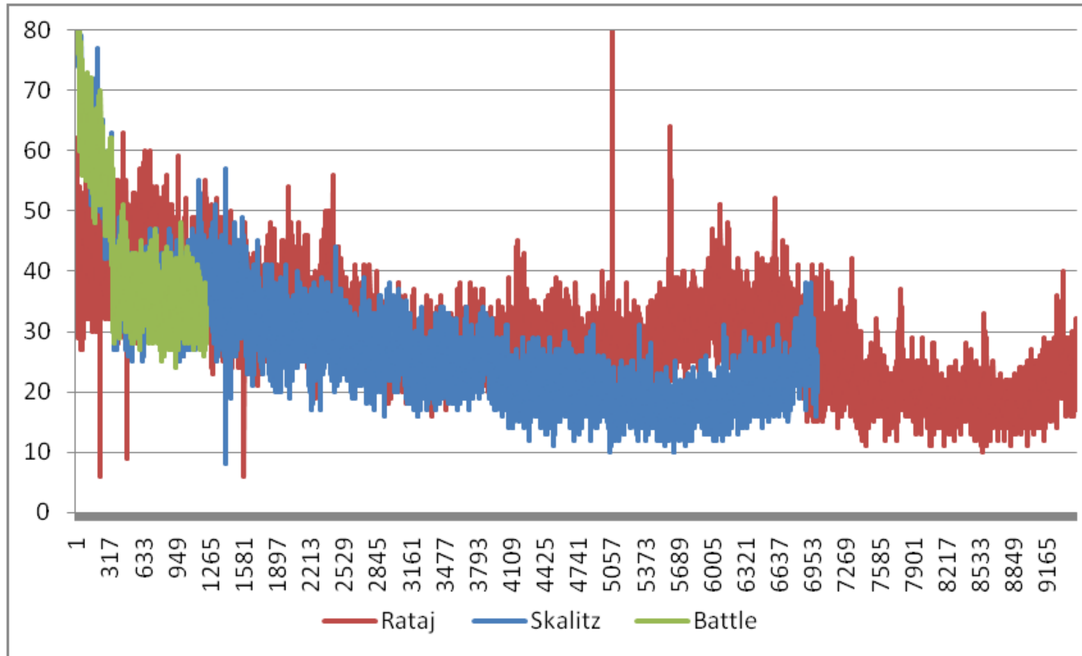

Figure 68: Active SBT which have at least one updated node

8.4.6 RKN queries

The RKN queries are one of the primary means on how SBT logic obtains relevant semantic information to make decision on how to approach the world. In principle, the SBT searches the RKNs edges via a query language specifying the search predicates. Therefore, the throughput of the RKN query system is key to having responsive NPC in a dynamical world. In (Figure 69) we show how many RKN queries are commonly executed in an active situation. It can be seen that in Battle the RKN is used a lot, mostly to acquire new targets and on evaluation of the battlefield. In the case of both Rataj and Skalitz, the RKN queries are very similar in nature, oscillating around 7 queries per frame.

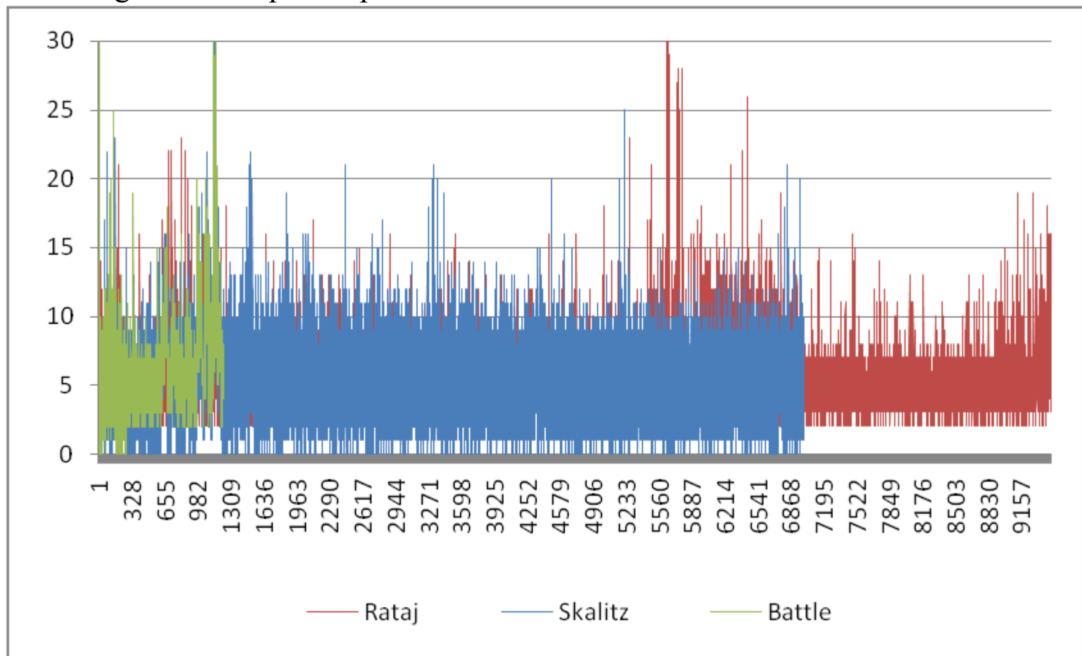

Figure 69: Unique solved RKN queries per frame

The amount of edges traversed (Figure 70) in every single frame indicates how complex the queries into the RKN are. It can be seen that often the edge traversal does not go above 50 edges per frame, however some peaks may occur if more complex searches are executed. For the purpose of dealing with such peaks, we provide the RKN query mechanism with a budget of how many edges can be traversed per search in a frame, thus distributing the load over several frames.

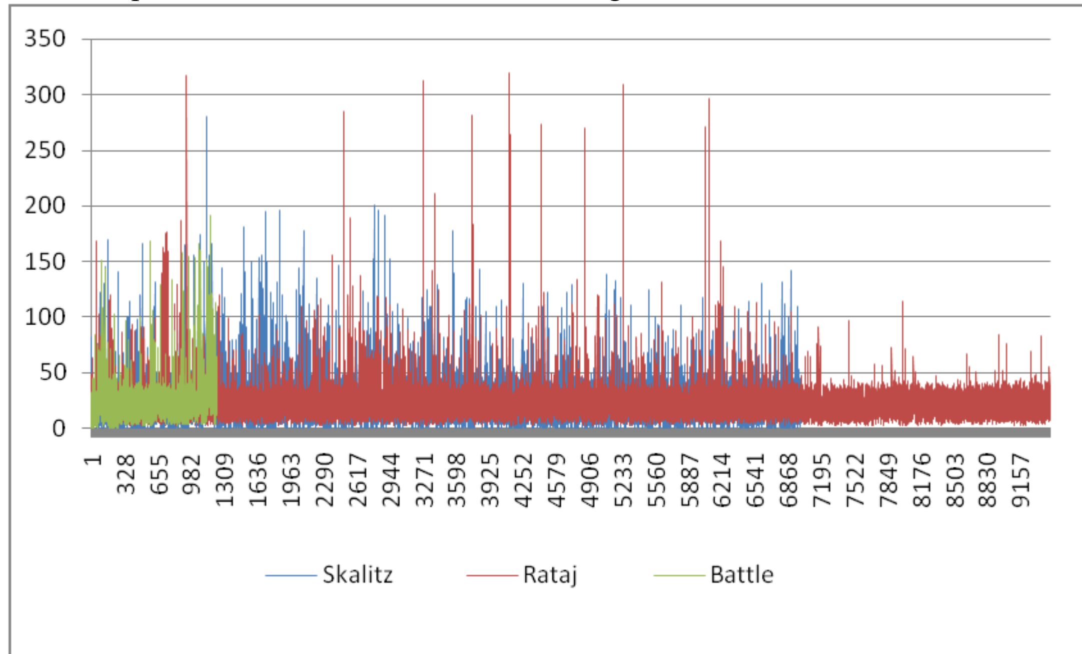

Figure 70: Traversed edges every frame by the RKN queries

9 Summary

Believable behaving Non-Player Characters are vital for conveying a believable virtual world with immersive gameplay. Maintaining the illusion of proper NPC's reasoning and cognition provides an intriguing environment a player may explore and enjoy. Our thesis was focused on providing tools and mechanisms to enhance the capacity of open world computer games to express believable behaviors for both ambient and story driven NPCs, so they may convey the illusion of deliberation and intelligence.

Our primary focus was to advance the illusion of Virtual Life in a complex environment by presenting complex behaviors with short and long term goals beyond a reactive approach. Our thesis was concerned with distributing intelligence into the game's environment to provide the virtual world with capacity to reason and deliberate about necessary reaction to player's actions in a localized and isolated manner. An intelligent and adaptable environment is as important as the individual reasoning of a singular NPC, since we want to maintain the illusion of a believable environment where actions have consequences and places have meaning. We also identified the need to manifest context specific behaviors in respect to object use, topological presence, and participation in a larger event or quest. Context specific behaving NPCs are key to maintaining the illusion of adaptable individual NPC's intelligence capable to live in a complex and changing world. Last but not least, we focused on providing designers with means to express semantic bindings to be able to manage logical and conceptual understanding of the virtual world surrounding an NPC. At the begin of our thesis, we voiced 5 goals:

Goal 1) *Believable Ambient Environment* to advance the illusion of Virtual Life within open world computer games.

Goal 2) *Emergent Situations* occur within the virtual world because of the world state, NPCs' decisions and player's actions.

Goal 3) *Story Driven Environment* is required to function properly for games with a storyline for the player to follow.

Goal 4) *Behavior Depth and Complexity* facilitates the necessity for complex human-like behavior to keep up the illusion of believable NPCs' behaviors.

Goal 5) *Large Scale Deployment* is necessary for manifesting the framework within a large scale open world game.

To address these goals, we have proposed 5 mechanism which in conjunction can satisfy the proposed goals:

1) *Stateful Behavior Tree Language* is based on combining the Behavior Tree annotation for NPC action selection and the Object-Oriented Programming paradigm. We decompose our SBT language for NPC action selection into nodes which are internally represented as Finite State Machines. We provide a scoped access to strongly typed variables with type inheritance support. We enhance the SBT capabilities with synchronization, messaging and decomposition of logic into synchronous and asynchronous actions.

2) *Three Tier Deliberation Architecture* denotes our decomposition of the NPC's DMM into isolated substructures (i.e. SubBrains) of a NPC's Brain, having different competences where each competence houses an SBT based action selection. Further, the Brain represents the high tier deliberation about long and midterm goals manifested as a Day Plan for every NPC.
3) *Intelligent Environment* represents the adaptive and reactive component of the virtual environment manifested as singular objects, areas, observers and mechanisms (i.e. quests). All these structures observe what is happening within the virtual world and act accordingly to achieve their agenda (i.e. design intent). These structures are responsible for changing the world's behavior and structure to adapt to the player's activities as well as following the designer's agenda.
4) *Smart constructs* are containers for context relevant behaviors to be used by SBT capable entities. Smart constructs provide SBT templates for Behavior Injection at host entities to provide them with a context relevant way to deal with using objects, behaving in areas or engaging in quests and situations.
5) *Semantic network* represents one of our key means to discover the world beyond the player's proximity, where NPCs utilize visual and sound perception. The semantic network denotes an oriented graph of annotated relations between entities within the world. We provide SBT capable entities with a query language to infer connections and dependencies present within network.

9.1 Stateful Behavior Tree Language

We use the SBT language (Mechanism 1) to provide an Action Selection mechanism for any entity within the virtual world which needs to deliberate about changes to itself or the world (i.e. all thinking entities, like NPCs, IE Objects, IE Areas etc.). The SBT language is the supporting mechanism for our Goals 1 – 4, where expressive AS is key to provide believable behaviors (Goal 1), handle emergent situations (Goal 2), have an adaptive story driven environment (Goal 3) and to produce complex behaviors (Goal 4).

We consider our language an advancement in respect to capacity to produce more complex behaviors with a wider set of language constructs (e.g. messaging, synchronization, interleaved actions) to express complex behavioral patterns. Without messaging, our NPCs could not communicate with each other, with Intelligent Environment's components or engine subsystems in an organized manner. Messages provide an easy to understand and simple to employ mechanism for delivering complex data from entity to entity in a well-orchestrated fashion.

Synchronization is key to manage complex execution patterns which rely on coordination. Without synchronization, our NPCs would rely on pure luck that coordinated actions would happen in synchrony.

Our action system is key for producing fluent execution of actions in a complex environment. Actions influencing each other allows us to create NPCs in a much more data driven way, where for example animations can dictate exact positioning for movement.

All this aggregates in a much more believable ambient environment, where NPCs live their day-by-day life. Management of emergent situations relies heavily on the capability to express complex execution patterns by using the SBT language. Quests and stories would be much more straightforward without the use of more complex coordination between quest participants at the lowest level of AS.

9.2 Three-Tier Deliberation Architecture

Our Three-Tier Deliberation Architecture is mainly focused on dealing with Goal 2–4. In principle, we decompose the structure of our NPCs' DMM into a management tier – Brain, a competence tier – SubBrain and an execution tier – SBT Action Selection.

The Brain's principal function of – creating and maintaining the NPC's Day Plan – is aimed at solving problems for Goal 3 (and partially Goal 1), where the NPC can follow its individual (daily) agenda. The questing component of the game can influence this agenda by introducing patches to the Day Plan, so the NPC can take part in the quest without distributing its day-by-day life. This allows us to influence the NPC in a decoupled and less intrusive way, than introducing the knowledge of all possible quests into its low-level AS (common in other games).

The SubBrain concept is primarily aimed ad splitting specific competences into a distinct subcomponents of a NPC's DMM. The main focus of such decomposition is the capacity to deal with emerging situations (Goal 2) which are beyond what can be handled in a day-by-day fashion. Such emergent situations can range from combat, trough quest specific AS, to custom management of in-game situations. This allows us to address emerging situations in a much more controlled way.

All three tiers combined can create an illusion of complex NPC's deliberation process with many facets (Goal 4). Various NPCs can be equipped with a different range and setup of SubBrains to allow designers to create unique experiences when engaging different NPCs.

9.3 Intelligent Environment

We have shown how an open world's environment can be enriched with intelligence to provide a more adaptive and reactive world in respect to NPCs' decisions and player actions. In principle, a game's Intelligent Environment is comprised of non-NPC entities like objects, areas, observers and mechanisms (i.e. quests) capable of deliberation and action selection. However, the actions executed by the IE components differ from actions an NPC chooses, since they are not manifested in the same fashion a NPC is. This concept is aimed at dealing with our Goals 2–4.

In respect to new and unexpected emerging situations (Goal 2), the IE represents the environment's counterpart to NPCs and the player. The IE observers and manages emerging situations in respect to the design agenda (i.e. how we want the game to behave). This allows for creating various constructs like a game's crime system or siege and battle management.

An Intelligent Environment is key to manifesting a well-orchestrated story (Goal 3), where we use the capacity to observe and manage the quest's progress. Intelligent Mechanisms provide proxies for quests to handle distributing commands and Day Plan patches to NPCs who should take part in a quest. When done in an adaptive fashion, the quest can change the story's course based on what currently happens.

The overall complexity and depth (Goal 4) of NPC behavior can be heavily influences by the IE's deliberation about what should or should not happen. Areas can watch for trespassers and send messages to inhabitants that someone has broken into their home. The decomposition of the overall world's behavior into the IE is a key component in maintaining a believable environment.

9.4 Smart Constructs

Smart constructs can be simplified as behavior containers where any SBT capable entity can request a behavior template to inject and contextually enhance its own deliberation. Since the behavior templates are identified by names, the required injection can be composed during run-time, based on a SBT deliberation. The conceptual mission of Smart constructs is to provide a context relevant behavior to be used at either an object, location or in a situation (i.e. quest).

Smart constructs are mainly aimed at supporting Goals 2, 4 and 5. From a behavior complexity (Goal 4) standpoint, Smart constructs provide the means to adequately enhance the present NPC AS based on its intent and internal agenda. Topological behavior requests are a powerful tool to make use of Behavior Injections based purely on the topological position, thus reflecting the realistic context. Having fun at a church differs vastly from having fun at a tavern. Since behavior requests for BIs can be recursive, an NPC can enhance its own AS almost to any necessary extent.

In respect to emergent situations (Goal 2), Smart Constructs provide a contextually valid way to handle such situations. A Smart Area representing the region of an emerging situation (e.g. a murder) can provide participating NPCs with specific and relevant behaviors to deal with that situation in a believable way.

The overall decomposition of behaviors into Smart Constructs allows to mitigate the immediate size of most NPCs' SBTs. It also forces the designers to use decomposition as their primary tool for shaping the virtual world.

9.5 Semantic Network

Since actual perception is computationally costly, most NPCs cannot have the luxury of employing it beyond the player's proximity. Beyond that, some information cannot be perceived by means of visual or sound perception. Therefore. we introduced the semantic network of directed relations between entities. These relations are abstracted as a directed multigraph where edges connect entities and carry relevant data in respect to the annotated relation.

This mechanism allows the designer to shape the world's dependencies and relations in a human understandable way (e.g. John likes Jane, John works at a farm). To support our Goals 1–5, we provided the SBT with a simple predicate based query language on how to inspect properties of the semantic network and search for dependencies between endpoints. Further, since the network is adaptive, it can be utilized to shape the way an NPC solves the approach to satisfying its goals (e.g. the NPC wants to have fun, thus discovers its closest place providing the »fun« behavior).

9.6 Evaluation and Integration

Part of our thesis is a quantitative, qualitative and integration evaluation of our architecture's deployment at Warhorse Studios for the big budget open world role

playing computer game Kingdom Come: Deliverance. Our evaluation has shown that our approach is superior in respect to use and expressiveness in comparison to other industry standardized approaches present in state-of-the art computer game engines (e.g. CryEngine by Crytek). We also have shown that our integration proves to be an effective component in managing the entire virtual worlds population of NPCs (600+), Smart constructs, Intelligent Environment's components and others (almost 80000 entities in total). Our architecture is currently deployed in the game's full production and is used by the scripting (16 scripters) and testing (16 testers) department. All of the game's quests are written utilizing our IE Mechanisms and every NPC and deliberating entity, including the player's NPC is managed by our SBT trees.

Therefore, we assume the architecture is beyond a theoretical proof-of-concept and is a fully production mature technology capable of supporting large scale simulations of virtual life in open world computer games.

10 Future work

Providing believable environment's in large scale OWG at a production level quality is a tedious task. In our future work, we intend to intend to focus on three major issues with our architecture.

First, we need to improve the tool chain used to produce and maintain the contents of the believable environment. One of our primary concerns is the capability to track what an NPC does and why has it decided to do it. In most cases, when something goes wrong (i.e. an NPC ends up at a different place doing something not desired by the designer), we need to be able to track and replay the deliberation that caused it.

Secondly, we intent do improve the low-level implementation of our mechanisms to be even more capable to handle large scale loads of NPC and other deliberation capable entities. We have to focus on computational and memory optimizations to be able to accommodate larger and more complex worlds.

Thirdly, we intend to introduce further low level AS mechanisms to be at the designer's disposal, ranging from low level FSMs, to code snippets in C# and other programming languages, to be plugged into either the SBT 's structure, or as standalone AS modules at the lowest DMM level.

Fourthly, we will focus on more concurrent execution of our architecture, to utilize more of the multicore capabilities of current hardware platforms. We intend to run individual NPCs in a concurrent fashion.

Fifthly, we aim at improving our high level DMM by going beyond the current mechanism of predefined plans for every NPC. We will focus on introducing classical planning in respect to long term goals (i.e. ranging beyond a day-by-day schedule). We also intend to capitalize more on an affordance based mid-range planning for day by day activities, where SA can satisfy daily needs in a more emergent fashion.

Lastly, we will work on improving our low-level action mechanisms, to provide more interleaved execution so our NPCs look even more realistic.

11 Conclusion

For the last five years, we have worked on improving the (virtual) Lives of countless Non-Player Characters, who were standing sadly in the rain, waiting for the player to come by and ask for a quest. Our mission was to give them an actual (virtual) Life with meaning and goals to aspire to. We wanted to populate their worlds with something else than just bloodthirsty creatures, whose only purpose is to die by the player's sword.

Our vision was to let the world work on its own, not caring about a player at all, just letting stories and situations unfold. We wanted players to ask themselves what do NPCs thinking about, what are their motivations, goals and purpose. We want players to investigate, stalk and observe, so they can appreciate the (virtual) Life that makes up the world surrounding their game.

The mechanisms and architecture presented in this thesis were developed to manifest this vision. Presently, all NPCs within the Kingdom Come: Deliverance's virtual world think about their actions utilizing our SBT language. They go beyond their individual deliberation and communicate and exchange information with others. They infer knowledge about their environment to follow their goals and adapt to what they find. They have their own agenda, they follow their desires and have a plan on how to spend the next day being happy in their own way.

We wanted to go beyond making only NPCs think, we wanted the world itself to cast an eye on the player and watch him as he watches the world. We filled the virtual world with Intelligent entities that watch and think for themselves, steering the world, stories and the player in various direction. The world is filled with things, places and stories that can provide those who are interested with means on how to use, act or participate. We want the world to make the NPCs smarter just by being (virtually) alive.

Figure 71: Virtual Life in KCD (©Warhorse Studios 2017)

However, the road out of the Uncanny Valley is still ahead and fairly steep.

Bibliography

Aahz. (2003). *Typing: Strong vs. Weak, Static vs. Dynamic.* Retrieved 4 4, 2017, from Artima Developers: http://www.artima.com/weblogs/viewpost.jsp?thread=7590

Agha, G. (1986). *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press.

Bartak, R., Brom, C., Cerny, M., & Gemrot, J. (2013). Planning and Reactive Agents in Dynamic Game Environments: An Experimental Study. In J. Filipe, & A. Fred (Ed.), *Proceedings of 5th International Conference on Agents and Artificial Intelligence (ICAART 2013)* (pp. 234-240). SciTePress.

Bartel, J. (2011). *Non-Preemptive Multitasking.* Retrieved 4 4, 2017, from ClassicCmp: http://www.classiccmp.org/cini/pdf/HT68K/HT68K%20TCJ30p37.pdf

Bazzy. (2012). *C++ concepts: PODType.* Retrieved 4 4, 2017, from CPP Reference: http://en.cppreference.com/w/cpp/concept/PODType

Bazzy. (2012). *Lambda expressions.* Retrieved 4 4, 2017, from C++ reference: http://en.cppreference.com/w/cpp/language/lambda

Beck, S. D. (2011). On the Congruence of Modularity and Code Coupling. *In Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering.*

Bergmann, M. (2008). *An Introduction to Many-Valued and Fuzzy Logic: Semantics, Algebras, and Derivation Systems.* Cambridge University Press.

Bertz, M. (2011, 01 17). *The Technology Behind The Elder Scrolls V: Skyrim.* Retrieved 4 4, 2017, from Game Informer: http://www.gameinformer.com/games/the_elder_scrolls_v_skyrim/b/xbox360/archive/2011/01/17/the-technology-behind-elder-scrolls-v-skyrim.aspx

Bohemia Interactive. (2006). *ARMA.*

Booch, G. (1994). *Object-oriented Analysis and Design.* Addison-Wesley.

Bovet, D. P., & Cesati, M. (2006). *Understanding the Linux Kernel.* O'Reilly Media.

Bratman, M. E. (1999). *Intention, Plans, and Practical Reason.* CSLI Publications.

Brom, C. (2005). Hierarchical Reactive Planning: Where is its limit? *Proceedings of MNAS - Modelling Natural Action Selection.* Edinburgh.

Brom, C., Gemrot, J., Bída, M., Burkert, O., S., P., & Bryson, J. (2006). POSH Tools for Game Agent Development by Students and Non-Programmers. *Proceedings of CGAMES 06*, (pp. 126-135). Dublin.

Brom, C., Lukavsky, J., Sery, O., Poch, T., & P.Safrata. (2006). Affordances and level-of-detail AI for virtual humans. *Proceedings of Game Set and Match*, (pp. 134–145).

Brom, C., Lukavský, J., Šerý, O., Poch, T., & Šafrata, P. (2006). Affordances and level-of-detail AI for virtual humans. *Proceedings of the Game Set and Match*, (pp. 134–145).

Brom, C., Poch, T., & Serý, O. (2010). AI Level of Detail for Really Large Worlds. In *Game Programming Gems 8* (pp. 213-231). Course Technology.

Brooks, A. (1986). A Robust Layer Control System for a Mobile Robot. *IEEE Journal of Robotics and Automation RA-2*, (pp. 14-23).

Bryson, J. J. (2001). *Intelligence by Design: Principles of Modularity and Coordination for Engineering Complex Adaptive Agents.* Massachusetts Institute of Technology.

Cardelli, L. (1991). Typeful programming. *Formal description of programming concepts* (pp. 431-507). Springer-Verlag.

Cerny, M. (2016). *Reducing Complexity of AI in Open-World Games by Combining Search-based and Reactive Techniques.* Charles University in Prague.

Cerny, M., Plch, T., Marko, M., Gemrot, J., Ondracek, P., & Brom, C. (2016). Using Behavior Objects to Manage Complexity in Virtual Worlds. *IEEE Transactions on Computational Intelligence and AI in Games.*

Černý, M., Plch, T., Marko, M., Ondracek, P., & Brom, C. (2014). Smart Areas: A Modular Approach to Simulation of Daily Life in an Open World Video Game. *Proceedings of 6th International Conference on Agents and Artificial Intelligence*, (pp. 703-708).

Champandard, A. J. (2004). *AI Game Development: Synthetic Creatures with Learning and Reactive Behaviors.* New Riders.

Champandard, A. J. (2007). *Understanding Behavior Trees.* Retrieved 4 4, 2015, from AI Game Dev: http://aigamedev.com/open/article/bt-overview/

Champandard, A. J. (2007.2, 12 28). *AI Game Dev.* Retrieved 4 4, 2017, from 10 Reasons the Age of Finite State Machines is Over: http://aigamedev.com/open/article/fsm-age-is-over/

Chandler, H. M. (2009). *The Game Production Handbook* (2 ed.). Hingham, Massachusetts: Infinity Science Press.

Clocksin, W. F., & Mellish, C. S. (2003). *Programming in Prolog.* Springer-Verlag.

Conway, M. E. (1963). Design of a Separable Transition-Diagram Compiler. *Communications of the ACM, Volume 6, Issue 7*, pp. 396–408.

Corkill, D. D. (1991). Blackboard Systems. *AI Expert*, (pp. 40–47).

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2001). *Introduction to Algorithms.* MIT Press;McGraw-Hill.

Craig, R. S. (2012). Applied Evolutionary Psychology. (p. 423). Oxford University Press.

Crockford, D. (2008). *JavaScript: The Good Parts.* O'Reilly Media.

Crytek. (2002). *CryEngine.*

Crytek. (2013). *Smart object system.* Retrieved 4 4, 2017, from CRYENGINE Manual: http://docs.cryengine.com/display/SDKDOC2/Smart+Object+System

Cubbi. (2013, 3 25). *Exceptions.* Retrieved 4 4, 2017, from CPP Reference: http://en.cppreference.com/w/cpp/error/exception

De Sevin, E., Chopinaud, C., & Mars, C. (2015). Zones To Create the Ambiance of Life. In *Game AI Pro 2* (pp. 89–100). CRC Press.

de Silva, L., Sardina, S., & Padgham, L. (2009). First principles planning in BDI systems. *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems.*

Entertainment, Blizzard. (2004). *World Of Warcraf.* Retrieved 4 4, 2017, from http://www.worldofwarcraft.com/

Epic Games. (1998). *Unreal Engine.*

Erol, K., Hendler, J., & Nau, D. S. (1996). Complexity results for htn planning. *Annals of Mathematics and Artificial Intelligence* (pp. 69–93). Springer.

Feathers, M. (2004). *Working Effectively with Legacy Code.* Prentice Hall.

Feigenbaum, E., McCorduck, P., & Nii, H. P. (1988). *The rise of the expert company.* Times Books New York.

Ferguson, S., & Terrion, J. (2014). *Communication in Everyday Life, Personal and Professional Contexts.* Oxford University Press.

Fikes, R. E., & Nilsson, N. J. (1971). STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, (pp. 189-208).

Fog, A. (2010). *Calling conventions for different C++ compilers and operating systems.* Retrieved 4 4, 2017, from Agner.org: http://www.agner.org/optimize/calling_conventions.pdf

Fowler, M. (2002). *Patterns of Enterprise Application Architecture.* Addison-Wesley Professional. Retrieved from martinfowler.com.

Fowler, M., Beck, K., Brant, J., Opdyke, W., & Roberts, D. (1999). *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Professional.

Freeman, E., Freeman, E., Kathy, S., & Bates, B. (2004). *Head First Design Patterns.* O'Reilly Media:.

Freeman, E., Sierra, K., & Bates, B. (2004). *Head First Design Patterns.* O'Reilly.

Fu, D., & Houlette, R. (2004). The ultimate guide to FSMs in games. In *AI game programming Wisdom* (pp. 283–302). Charles River Media.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley.

Geijtenbeek, T., van de Panne, M. A., & van der Stappen, F. (2013). Flexible Muscle-Based Locomotion for Bipedal Creatures. *ACM Transactions on Graphics. 32.* ACM SIGGRAPH.

Gemrot, J., Cerny, M., & Brom, C. (2014). Why you should empirically evaluate your AI tool: From SPOSH to yaPOSH. *Proceedings of 6th International Conference on Agents and Artificial Intelligence.*

Ghallab, M., Nau, D. S., & Traverso, P. (2004). *Automated Planning: Theory and Practice.*

Gibson, J. J. (1977). The Theory of Affordances. *Perceiving, Acting, and Knowing.*

Girault, A., Lee, B., & Lee, E. (1999). Hierarchical finite state machines with multiple concurrency models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (pp. 742 - 760). IEEE.

Graves, A., & Czarneck, C. (2000). Design patterns for behavior-based robotics. *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans*, *30*, pp. 36-41.

Green, R. D., MacDorman, K. F., Chin-Chang, H., & Vasudevan, S. (2008). Sensitivity to the proportions of faces that vary in human likeness. *Computers in Human Behavior* (pp. 2456-2474). Elsevier Ltd.

Gregory, J. (2014). *Game Engine Architecture* (2 ed.). A K Peters,CRC Press.

GSC Game World. (2007). *S.T.A.L.K.E.R.*

(2001-2017).Halo Games. *Series.* Bungie;Ensemble Studios;343 Industries;Creative Assembly.

Hart, J. (2005). *Windows System Programming.* Addison-Wesley.

Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics SSC4*, (pp. 100–107).

Hayes-Roth, F., Waterman, D., & Lenat, D. (1983). *Building Expert Systems.* Addison-Wesley.

Hewitt, C., Bishop, P., & Steiger, R. (1973). A Universal Modular Actor Formalism for Artificial Intelligence. *International Joint Conference on Artificial Intelligence.*

Humble, J., & Farley, D. (2011). *Continuous Delivery: reliable software releases through build, test, and deployment automation.* Pearson Education.

Hunt, A., & Thomas, D. (1999). *The Pragmatic Programmer: From Journeyman to Master.* Addison-Wesley Professional.

Ingebretson, P., & Rebuschatis, M. (2014). Concurrent interactions in The Sims 4. *Game Developers Conference.*

IO Interactive. (2014). *Hitman: Absolution.*

Irrational Games. (2013). *BioShock Infinite.*

Isla, D. (2005). Handling Complexity in the Halo 2 AI. *Game Developer's Conference 2005.*

ISO/IEC 9899:TC3. (2007). 6.2.1 Scopes of identifiers. In *WG14 N1256 (2007 updated version of the C99 standard).*

Jackson, K. (1977). Parallel processing and modular software construction. *Proceedings of the DoD Sponsored Workshop on Design and Implementation of Programming Languages* (pp. 436-443). Springer.

Joshi, P. (2014, 4 5). *What Is Metaprogramming? – Part 1/2.* Retrieved 4 4, 2017, from PERPETUAL ENIGMA: https://prateekvjoshi.com/2014/04/05/what-is-metaprogramming-part-12/

Kallman, M., & Thalmann, D. (1999). Modeling Objects for Interaction Tasks". Springer: 73–86. *Computer Animation and Simulation '98. Eurographics* (pp. 73-86). Vienna: Springer.

Kambhampati, S., Mali, A., & Srivastava, B. (1998). Hybrid planning for partially hierarchical domains. *In Proc. of the 15th Nat. Conf. on Artificial Intelligence* (pp. 882–888). AAAI Press.

Kaminka, G. A., & Frenkel, I. (2005). Flexible Teamwork in Behavior-Based Robots. *In Proceedings of the Twentieth National Conference on Artificial Intelligence.*

Kaminka, G. A., Yakir, A., Erusalimchik, D., & Cohen-Nov, N. (2007). Towards Collaborative Task and Team Maintenance. *In Proceedings of the Sixth International Joint Conference on Autonomous Agents and Multi-Agent Systems.*

Kapoor, N., & Bahl, N. (2016). Comparative Study of Forward and Backward Chaining in Artificial Intelligence. *International Journal Of Engineering And Computer Science.*

Kendall, E., Krishna, P. V., Pathak, C. V., & Suresh, C. B. (1998). Patterns of intelligent and mobile agents. *Proceedings of the second international conference on Autonomous agents*, (pp. 92-99).

Kent, S. L. (2001). *The ultimate history of video games: from Pong to Pokémon and beyond : the story behind the craze that touched our lives and changed the world.* Prima Publishing.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., & Griswold, W. G. (2001). An Overview of AspectJ. *Object-Oriented Programming: 15th European Conference*, (pp. 327-354).

Kildaegi. (2008). *Variable Shadowing.* Retrieved from https://en.wikipedia.org/wiki/Variable_shadowing

Knuth, D. E., & Moore, R. W. (1975). An Analysis of Alpha–Beta Pruning. In *Artificial Intelligence* (pp. 293–326).

Kolombo, M., & Barták, R. (2014). A Constraint-based Planner for Mars Express Orbiter. In A. Gelbukh, F. C. Espinoza, & S. N. Galicia-Haro (Ed.), *Nature-Inspired Computation and Machine Learning (13th Mexican International Conference on Artificial Intelligence* (pp. 451-463). Tuxtla Gutiérrez: Springer.

Kortuem, G., Kawsar, F., Fitton, D., & Sundramoor, V. (2010). Smart Objects as Building Blocks for the Internet of Things. *EEE Internet Computing 14* , (pp. 44-51).

Krauss, A. (2014, 9 14). *Programming Concepts: Type Introspection and Reflection.* Retrieved 4 4, 2017, from The Societea: https://thesocietea.org/2016/02/programming-concepts-type-introspection-and-reflection/

Lamport, L. (1978). Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 558-565.

Landin, P. J. (1964). The mechanical evaluation of expressions. *The Computer Journal*, pp. 308-320.

Laplante, P. (2007). *What Every Engineer Should Know About Software Engineering.* CRC Press.

Larman, C. (2005). *Applying UML and Patterns.* Prentice Hall.

Lewis, J., & Loftus, W. (2008). *Java Software Solutions Foundations of Programming Design* (6 ed.). Pearson Education Inc.

Liskov, B., & Zilles, S. (1974). Programming with abstract data types. *ACM SIGPLAN Notices.*

Liskov, B., & Zilles, S. (1974). Programming with abstract data types. *Proceedings of the ACM SIGPLAN symposium on Very high level languages*, (pp. 50-59).

Loyall, A. B. (1997). *Believable Agents: Building Interactive.* Carnegie Mellon University Pittsburgh, PA, USA.

MacDorman, K. F., & Chattopadhyay, D. (2016). Reducing consistency in human realism increases the uncanny valley effect; increasing category uncertainty does not. *Cognition. 165*, pp. 190-205. Elsevier Ltd.

McConnell, S. (2004). *Code Complete: A Practical Handbook of Software Construction.* Microsoft Press.

Mercury Steam. (2010). *Castlevania: Lords of Shadow.*

Meyer, B. (1991). Design by Contract. (D. Mandrioli, & B. Meye, Eds.) *Advances in Object-Oriented Software Engineering*, 1-50.

Mitchell, D. R. (1990). *Managing Complexity in Software Engineering.*

Monolith Productions. (2005). *F.E.A.R.*

Monolith Productions. (2009). *F.E.A.R. 2.*

Moore, G. E. (1965). Cramming more components onto integrated circuits. *Electronics*, 4.

Mori, M., MacDorman, K. F., & Kageki, N. (2012). The Uncanny Valley. *IEEE Robotics & Automation Magazine . Volume: 19, Issue: 2*, pp. 98-100. IEEE.

Musser, D. R., & Stepanov, A. A. (1989). Generic Programming. *Symbolic and Algebraic Computation: International symposium ISSAC 1988*, (pp. 13–25).

Orkin, J. (2005). Agent Architecture Considerations for Real-Time Planning in Games. *Artificial Intelligence and Interactive Digital Entertainment.*

Orkin, J. (2006). Three States and a Plan: The AI of F.E.A.R. *Game Developers Conference 2006.*

Parera, J. (2013). *Combat AI and animations in CASTLEVANIA: Lord of Shadows.* Retrieved from AIGameDev.com.

Parnas, D. L., Shore, J. E., & Weiss, D. (1976). Abstract types defined as classes of variables. *Proceedings of the 1976 conference on Data : Abstraction, definition and structure*, (pp. 149-154).

Paterson, T. (1983). *An Inside Look at MS-DOS.* Retrieved 4 4, 2017, from Paterson Technology: http://www.patersontech.com/dos/byte%E2%80%93inside-look.aspx

Persson, M. (2012). *Try-block.* Retrieved 4 4, 2017, from CPP Reference: http://en.cppreference.com/w/cpp/language/try_catch

Pitnerová, B. (2008). *Etologie volné pastvy ovcí.* Brno: Mendelova zemědělská a lesnická univerzita v Brně Agronomická fakulta Ústav výživy zvířat a pícninářství.

Plch, T. (2009). *Action selection for an animat.* Prague: Charles University.

Plch, T., Chomut, M., Brom, C., & Bartak, R. (2012). Inspect, Edit and Debug PDDL Documents: Simply and Efficiently with PDDL Studio. *22nd International Conference on Automated Planning and Scheduling - System Demonstrations and Exhibits.*

Plch, T., Marko, M., Ondracek, P., Cerny, M., Gemrot, J., & Brom, C. (2014). An AI System for Large Open Virtual World. *Proceedings of Tenth Annual AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, (pp. 44-51).

Pressman, R. S. (1999). *Software Engineering: A Practitioner's Approach.*

Pressman, R. S., & Maxim, B. (2014). *Software Engineering: A Practitioner's Approach.* McGraw-Hill Education.

Quinlan, J. R. (1987). Simplifying decision trees. *International Journal of Man-Machine Studies.*

Rasmussen, J. (2016, 4 27). *Are Behavior Trees a Thing of the Past?* Retrieved 4 4, 2017, from Gamasutra: http://www.gamasutra.com/blogs/JakobRasmussen/20160427/271188/Are_Behavior_Trees_a_Thing_of_the_Past.php

Reed, C., & Geisled, B. (2004). Jumping, Climbing, and Tactical Reasoning: How to Get More Out of a Navigation System. In *AI Game Programming Wisdom II* (pp. 141-150). Charles River Media.

Russell, S., & Norvig, P. (2009). *Artificial Intelligence: A Modern Approach.* Pearson.

Sardina, S., de Silva, L., & Lin, P. (2006). Hierarchical planning in BDI agent programming languages: a formal approach. *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems.*

Schurr, N., Marecki, J., Tambe, M., Scerri, P., Kasinadhuni, N., & Lewis, J. (2005). The Future of Disaster Response: Humans Working with Multiagent Teams using DEFACTO. *AI Technologies for Homeland Security.*

Schuytema, P., & Manyen, M. (2005). *Game Development with Lua.* Charles River Media.

Scott, M. L. (2006). *Programming language pragmatics* (2 ed.). Morgan Kaufmann.

Sellers, J. (2001). Pong. In *Arcade Fever: The Fan's Guide to The Golden Age of Video Games* (pp. 16-17). Running Press.

Sheppard, D. (2000). *Beginner's Introduction to Perl.* O'Reilly Media.

Shimon, E. (2011). *Graph Algorithms.* Cambridge University Press.

Shneiderman, B., & Plaisant, C. (2005). *Designing the user interface: Strategies.* Pearson Education.

Silberschatz, A., Gagne, G., & Galvin, P. B. (2008). *Operating System Concepts.* John Wiley & Sons.

Simpson, C. (2014, 07 17). *Behavior trees for AI: How they work*. Retrieved 4 4, 2017, from Gamasutra: http://www.gamasutra.com/blogs/ChrisSimpson/20140717/221339/Behavior_trees_for_AI_How_they_work.php

Softworks Bethesda. (1985). *Bethesda*. Retrieved 4 4, 2017, from https://bethesda.net/en/dashboard

Sung, M., Gleicher, M., & Chenney, S. (2004). Scalable behaviors for crowd simulation. *Computer Graphics Forum*, *23*, pp. 519–528.

Sutherland, J., & Sutherland, J. (2014). Scrum: The Art of Doing Twice the Work in Half the Time. Crown Business.

Svenk, G. (2003). *Object-oriented Programming: Using C++ for Engineering and Technology.* . Cengage Learning.

Swanson, E. B. (1976). The dimensions of maintenance. *Proceedings of the 2nd international conference on Software engineering*, (pp. 492 — 497). San Franciso.

Tanenbaum, A. S. (2014). *Modern Operating Systems.* Pearson.

Tanenbaum, A. S., & Woodhull, A. S. (2006). *Operating Systems Design and Implementation.* TBS.

Tecchia, F., Loscos, C., Conroy, R., & Chrysanthou. (2001). Agent Behaviour Simulator (ABS): A Platform for Urban Behaviour. *The First International Game Technology Conference and Idea Expo.*

Tinwell, A. (2014). *The Uncanny Valley in Games and Animation.* CRC Press.

Tinwella, A., Grimshawa, M., Nabib, D. A., & Williamsa, A. (2011). Facial expression of emotion and perception of the Uncanny Valley in virtual characters. *Computers in Human Behavior* (pp. 741–749). Elsevier Ltd.

Turner, C. W., Lewis, J. R., & Nielsen, J. (2006). Determining Usability Test Sample Size. In W. Karwowski (Ed.), *International Encyclopedia of Ergonomics and Human Factors* (2 ed., Vol. 3). CRC Press.

Turner, D. A. (2012). Some History of Functional Programming Languages. *International Symposium on Trends in Functional Programming.*

Ubisoft Montreal. (2014). *FarCry 4.*

Umarov, I., & Mozgovoy, M. (2012). Believable and Effective AI Agents in Virtual Worlds: Current State and Future Perspectives. In B. Dubbels (Ed.), *: International Journal of Gaming and Computer-Mediated Simulations*, *2*, p. 23.

Valve, C., & Gearbox, S. (1998-2007). *Half-Life series.*

Vehkala, M. (2012). *Crowds in Hitman: Absolution. AIGameDev.com* . Retrieved 4 4, 2017, from AI Game Dev: http://aigamedev.com/ultimate/video/hitmancrowds/

Ward, C., & Cowling, P. (2009). Monte Carlo Search Applied to Card Selection in Magic: The Gathering. *Proceedings of the 5th international conference on Computational Intelligence and Games.*

Warhorse Studios. (2017). *Kingdom Come: Delieverance.*

Weber, B. G., Mawhorter, P., Mateas, M., & Jhala, A. (2010). Reactive planning idioms for multi-scale game AI. *IEEE Conference on Computational Intelligence and Games*, (pp. 155-122).

Wegner, P., & Cardelli, L. (1985). On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, (pp. 471–523).

Weik, M. H. (2000). *Computer Science and Communications Dictionary.* Springer.

Wikipedia. (2017). *Buffs.* Retrieved 4 4, 2017, from Status effect: https://en.wikipedia.org/wiki/Status_effect#Buffs

Wikipedia. (2017). *Forward Declaration.* Retrieved 4 4, 2017, from Wikipedia: https://en.wikipedia.org/wiki/Forward_declaration

List of Figures

Figure 1: The Uncanny Valley diagram shows how human resentment is more intense with increase in human-like appearance (© MacDorman, 2005) 7

Figure 2: Soldier NPC from the Half Life game series (Valve & Gearbox, 1998-2007). On the left a low polygon model with few animations (e.g. walking, kneeling, jumping). On the right a high polygon character with a large animation repository (e.g. jumping over obstacles, leaning etc.) (© Valve Software) 8

Figure 3: Virtual life of a simple farmer in Kingdom Come: Deliverance (© Warhorse Studios 2017) 9

Figure 4: Dependency between Goals and Mechanisms 12

Figure 5: The DMM contains the Action Selection process which is queried about the next suitable action in respect to the constructed context of the environment's configuration. The produced action is committed to the environment afterwards. 15

Figure 6: Simple FSM for NPC with a limited set of states and transitions focusing on four principal areas of function – Idling, Combat, Search, Self-Preservation and Flight. NPC has been used primarily as combat opponents 17

Figure 7: A simple Behavior Tree concerned with choosing the proper action to go through a door which may be closed 18

Figure 8: A simple Behavior Tree representing the action selection of searching an item and grabbing it with either one or both hands. If the item is not found, it is searched on the floor, in the drawers and in the closet. The block leaf nodes represent action choices, the rounded leaf nodes represent sense actions (determine if something is true or not), boxes with question marks represent selectors and arrows represent sequences. (©Wikipedia) 31

Figure 9: The FSM of a language element (node) within the S BTL structure. The node starts at the None state and traverses over the available states until it reaches the Fail or Success state. 33

Figure 10: Event propagation for delivering the »Foo« event which originates at the Action node. The event is caught by the Try node and the respective Catch branch is activated. If the event would be something else than »Foo« or »Bar«, the propagation would continue until reaching the DMM. 42

Figure 11: Schema of structuring types based on aggregation of either EDT or already defined types 44

Figure 12: Variable Reference consists of a variable specifier (name), a selector to locate a specific instance within a complex variable form (e.g. array) and a member specifier, which denotes the exact data part of the accessed variable 47

Figure 13: Nesting of data scopes for a SBT and access to that data via variable references. Variables are stored in Data Scopes. These can be at any level of the SBT hierarchy (i.e. at any node). The access searches the data scope hierarchy until a variable is found. If not, it results in a failure to locate the variable. 48

Figure 14: Message Delivery System provides a subsystem to be used by the SBT to deliver messages containing data to another NPC and its respective SBT 51

Figure 15: Message states and their respective transitions in the process of delivering the message from Sender to Receiver. 52

Figure 16: Inbox hierarchy within the DMM. Fetching of messages is based on the hierarchical ordering, where the closest inboxes are checked first 53

Figure 17: Action Scopes within the Action Manager denote various facilities the Action can occupy – e.g. movement, vision, hands, full body etc. Actions which want to take control over a FS are in conflict which may be solved by terminating the already present action. All FSs for an action have to be satisfied to run the action properly 60

Figure 18: Action Lifecycle starting at the Initialization and ending at being Interrupted or Finish on it own. An action in conflict triggers a termination of a previous action, however it may be postponed due to the previous action being in an Interrupt Safe state. 60

Figure 19: Synchronous Execution of Actions which ends in an Idle state since every action has to end for the SBT to run a node triggering the next action. 61

Figure 20:Asynchronous Execution where two actions interleave. The Action1 is started, the AS waits for 7 seconds and Action2 is started, which interrupts Action1. Both actions would run for 10 seconds when not interrupted..................................62

Figure 21: Executing Asynchronous Actions with its event handler closures in a parallel manner within the NPC's DMM. The Action1 has two handlers Attach and Detach, where the Attach handler when executed more than once, terminates the animation. The NPC's AS runs in parallel with the handler code............................62

Figure 22: Simple approached for executing animations in a synchronized manner by entering a synchronization lock 63

Figure 23: Synchronizing Actions between 2 NPCs which use the synchronization lock as a mechanism to postpone the delivery of the startup event for an animation 64

Figure 24: Actions Chains are created by Actions which do not require termination from their predecessors but can work with the results of those actions or even influence them 65

Figure 25: Actions Animation and Movement when not in a chain lead to the animation moving the NPC from A to B, where the move when executed before the animation ends, searches the path from A to C. Forecasting provides the move with the end location of the animation. When a Movement is followed by a Sit, the sitting animation provides the necessary target position for the move to plan to via inverse influence. 66

Figure 26: The Animated Action provides the Movement action with a forecast on the end of the animation, allowing for movement to pathfind during the execution of the action taking the end location as the start position for the path. If these actions would not be chained, the movement would be planned from the current location of the NPC when the movement action was invoked. 67

Figure 27: Adaptive inverse influence by the animation to change the destination of the movement action to suit the necessary alignment requirements. The movement goes in the general direction of the Bench, which has two points from which the Sit animation can be executed (Front and Back). Since path planning may get the NPC from either direction, the Animation Action (Sit) monitors the progress and chooses

the end destination (i.e. either front or back) when movement gets close enough. After the move reaches the end location provided by the animation, the animation already knows which animation should be executed. 68

Figure 28: NPC Architecture divided into the Brain, SubBrains and AS mechanisms 71

Figure 29: Day Plan of a common peasant NPC. The Goals are annotated by simple strings which are evaluated within the Day-By-Day SubBrain................................72

Figure 30: Activities are organized into priority lanes where the activity with the highest priority at the current time is to be executed. 73

Figure 31: Start and end tolerances change the chosen activity based on how close a higher-level activity is to either the start or an end of another activity that would be executed prior or after it. 74

Figure 32: Every SubBs default state is the Inactive state, into which it may enter when an activation by the SubB or external source (e.g. other SubB, environment etc.) is requested. The SubB is Queued to run. While waiting in the Brain's queue, all conflicts are resolved. Before entering and after leaving the Running state, the SubB transitions over the Switch In and Switch Out respectively. A running SubB may enter a suspended state if the conflicting SubB is at the same priority level. 75

Figure 33: Injecting Behavior SBT into a running SBT is done at the Injection Point (IP) node. The BI manifests after the IP is executed. The IP continues to run until the injected SBT runs. After the injected SBT fails or succeeds, it is discarded. 82

Figure 34: The NPC wanders around with a published IP called »Fear« which can be used by the Hunted Church to force a behavior to act afraid in the area 83

Figure 35: How to specify an IE Object a) the Mug object's perception informs the AS via messages, b) the Mug object waits on pickup events to happen to break at random 85

Figure 36: NPCs enter an IE Area triggering Enter events which are handled by the area's AS. After leaving the area, they trigger the Leave event. The events are strictly ordered. The IE area has three AS mechanisms (Enter, Leave, Action Selection) where the Action Selection runs the internal AS of the area, Enter and Leave are triggered by NPCs. All the AS share the areas data and communication context to store variables and send messages. 86

Figure 37: The IE Mechanism providing a proxy for a Quest, which is a part of a quest chain. The AS within the IE Mechanism can »cancel« the quest if for example someone kills Jerry (see Scenario 4). The quest also informs the IE Mechanism about specifics (e.g. the event's timeout expires) so the IE Mechanism can react accordingly (e.g. tell all NPCs to get back to work). 87

Figure 38: Smart construct contains behaviors which can be provided for BI at host AS. The behavior is identified by a name tag and there may be a limited amount available to be injected at the same time. 89

Figure 39: Requesting a BI from a Smart triggers the Requested event which is handled within the Smart construct. When dropped, the injected behavior returns to the Smart and the Returned event is invoked and handled 90

Figure 40: When the NPC enters the tavern it only requests the »Fun« behavior, which further makes use of requesting behaviors from the mug and tavern. The NPC does not know of any other behaviors except for Fun. 91

Figure 41: The NPC walks along the planned path with a SNO (Door) being part of it. The SNO provides the »Traverse« behavior to get through the door using specific animations and handling issues like locked doors. 92

Figure 42: Overlapping SA are ambiguous in respect to a TR for BI. Non-overlapping areas can be used in a hierarchy, where »FOO« when not found at an area, can be searched at its enclosing (i.e. parent) area. 93

Figure 43: Exclusive strict ordering of SA provides the TR for a BI with a set of areas to ask in a given order from the most bottom to the topmost SA 94

Figure 44: The topological division of a part of the virtual world into a Land, City, Districts within the City, and Taverns and Homes within each District. There is also o Castle where only soldiers are. 95

Figure 45: The SBT of the NPC seeking Fun in the City 96

Figure 46: The Relation Knowledge Network annotates the semantic bindings between entities. John has a binding (i.e. edge) to Jane and Jill. John is also married to Jane but he likes Jill more. John also has two jobs and one house. John owns some tools which are SO who annotate themselves with their available behaviors. John also has a binding to himself, on how happy he is. 99

Figure 47: The RKN annotating the NPC's knowledge about the available shooting spots for both Bow and Crossbow. Some of them are at a Tower, some of them are in the open. 100

Figure 48: Depth First Search and Breadth First Search 101

Figure 49: The Subgraph from the RKN based on the Tag SubSelection of (A) and (C) excludes all other edges (B) from the search. 101

Figure 50: Adding bridge separated components to the RKN graph provides the search with an additional dimension to work with 103

Figure 51: Horizon of expanded edges which have not been evaluated by the Search Mechanism's predicate 105

Figure 52: Skalitz, Common day, frame time in millisecond measured over 10000 frames, standing in one place and looking around. Red line denotes 30ms.............119

Figure 53: Skalitz, Common day, frame time measured over 6800 frames, roaming around. Red line denotes 30ms. 120

Figure 54: City of Rataj, Common day, Frame time measured over 10000 frames, standing and looking around. Red line denotes 30ms. 120

Figure 55: Battle of 30 soldiers, Frame time measured over 1200 frames, taking part in the battl. Red line denotes 30ms 121

Figure 56: Long term game play running around in the country side, visiting villages and talking to NPCs, occasionally engaging in a fight. Red line denotes 30ms. 122

Figure 57: Inbox congestion for Skalitz, Rataj and Battle scenarios. The battle is short in comparision to other scenarios...123

Figure 58: Message pickup and processing for 6800 frames at the Skalitz scenario 124
Figure 59: Message pickup and processing for 1200 frames at the Battle scenario for 1200 frames 124
Figure 60: Message pickup and processing for 8700 frames at Rataj 124
Figure 61:Message count exchanged over the messaging system...........................125
Figure 62: Active AI entities in different scenarios. All these entities receive an update from our framework...126
Figure 63: NPC counts in various scenarios tracked over 9000 frames 127
Figure 64: Intelligent Environment Objects updated per frame 127
Figure 65: Intelligent Environment Areas updated per frame.................................128
Figure 66:: Total number of node instances aggregated over all SBTs...................128
Figure 67: Node updates per frame..129
Figure 68: Active SBT which have at least one updated node................................130
Figure 69: Unique solved RKN queries per frame 130
Figure 70: Traversed edges every frame by the RKN queries 131
Figure 71: Virtual Life in KCD (©Warhorse Studios 2017)..................................138

List of Tables

Table 1: Brian's bindings (edges) to others. The Target represent the entity the edge points to. Annotation represents the tag associated with that edge, Data represents the associated variables 106

Table 2:The results of the quantitative evaluation. The table displays mean (left),and .99 quantile and maximum times (right) 111

Table 3: Summarization of our experiments comparing the Behavior Trees and Smart Areas 113

Table 4: Subjective qualification of the tasks difficulty when creating new SBTs/SA or modifying existing ones. Scale is from 0 – 3, (easy – hard) 113

Table 5: First set of questions. Every question is stated in the upper part, where the reasoning behind it is in the lower part 116

Table 6: Second set of questions aimed at feedback in respect to design problems when creating the believable ambient environment 117

Table 7: Overall amount of entities and constructs within the KCD data 122

List of Abbreviations

RPG	Role Playing Game
FPS	First Person Shooter
RTS	Real Time Strategy
NPC	Non-Player Character
CGE	Computer Game Engine
OWG	Open World Game
KCD	Kingdom Come: Deliverance
FPS	Frames Per Second
CFG	Configuration
DMM	Decision Making Mechanism
AS	Action Selection
BDI	Belief-Desire-Intentions
BOD	Behavior Oriented Design
GOAP	Goal Oriented Action Planning
TRD	Transferable Design Reasoning
AI	Artificial Intelligence
PC	Personal Computer
AIF	Artificial Intelligence Framework
BOD	Behavior Oriented Design
OOD	Object Oriented Design
OOP	Object Oriented Programming
SBT	Stateful Behavior tree
BDI	Belief Desire Intentions
AM	Action Manager
PAA	Playing Animation Action
MA	Movement Action
HTN	Hierarchical Task Network
STRIPS	Stanford Research Institute Problem Solver
SubB	Sub Brain
CSubB	Combat Sub Brain
BI	Behavior Injection
SE	Smart Entity
IE	Intelligent Environment
SEnv	Smart Environment
SO	Smart Object
SA	Smart Area
WSA	World Smart Area
RKN	Relational Knowledge Network
IGC	Isolated Graph Components
LOD	Level of Detail
ADT	Advanced Data Types
PDT	Primitive Data Types
POD	Plain Old Type
IEVO	Intelligent Environment Virtual Observer
BSA	Brain-Sub Brain-Action Selection

List of Author's Publications

Publications relevant to this thesis

1. Cerny, M., Plch, T, Brom, C.: Beyond Smart Objects: Behavior-Oriented Programming for NPCs in Large Open Worlds. In Lengyel, Eric (eds.) Game Engine Gems 3. USA: CRC Press, 2016. pp. 267-280. ISBN 978-1-4987-5565-8.
2. Cerny, M., Plch, T.,Marko, M., Gemrot, J., Ondracek, P., Brom, C.: Using Behavior Objects to Manage Complexity in Virtual Worlds. In IEEE Transactions on Computational Intelligence and AI in Games, doi: 10.1109/TCIAIG.2016.2528499
3. Černý, M., Plch, T., Marko, M., Ondracek, P., Brom, C.: Smart Areas: A Modular Approach to Simulation of Daily Life in an Open World Video Game. In: Proceedings of 6th International Conference on Agents and Artificial Intelligence (ICAART 2014). 2014, pp. 703-708
4. Plch, T., Marko, M., Ondracek, P., Cerny, M., Gemrot, J., Brom, C.: Modular Behavior Trees: Lanuage for Fast AI in Open-World Video Games In: Proceedings of 21st European Conference on Artificial Intelligence (ECAI 2014), pp. 1209-1211.
5. Plch, T., Marko, M., Ondracek, P., Cerny, M., Gemrot, J., Brom, C.: An AI System for Large Open Virtual World In: Proceedings of Tenth Annual AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE 2014), pp. 44-51.
6. Plch T.: Action selection for an animat, 2009, Master thesis

Other publications

1. Plch, T., Chomut, M., Brom, C., Bartak, R.: Inspect, Edit and Debug PDDL Documents: Simply and Efficiently with PDDL Studio, In: 22nd International Conference on Automated Planning and Scheduling - System Demonstrations and Exhibits, (2012).
2. Plch, T.: Towards Believable Intelligent Virtual Agents with StateFull Hierarchical Reactive Planning. In: Safrankova, J. and Pavlu, J. (eds.) WDS 2011, Part I - Mathematics and Computer Sciences, Matfyzpress, Prague, pp. 119-124, ISBN 978-80-7378-184-2, 2011.
3. Gemrot, J., Brom, C., Plch, T.: A periphery of Pogamut: from bots to agents and back again. In: Agents for Games and Simulations II, LNCS 6525, Springer, pp. 19--37 (2011). The short version of this paper also appeared in Proc. Agents for Games and Simulations, AAMAS workshop pp. 1--19 (2010).
4. Plch, T., Brom, C.: Enhancements for reactive planning - tricks and hacks, In: Proceedings of SOFSEM (2010) Czech Republic.
5. Plch, T., Jedlicka, T., Brom, C.: Utilizing HLA for Connecting Virtual Worlds to Prototyping Tools, ITAT 2010, extende abstract. (2010).

6. Gemrot, J., Kadlec, R., Bida, M., Burkert, O., Pibil, R., Havlicek, J., Zemcak, L., Simlovic, J., Vansa, R., Stolba, M., Plch, T., Brom C. Pogamut 3 Can Assist Developers in Building AI (Not Only) for Their Videogame Agents. In: Agents for Games and Simulations, LNCS 5920, Springer, pp. 1--15. (2009) The short version of this paper also appeared in Proc. Agents for Games and Simulations, AAMAS workshop (2009) 144--148.
7. Brom, C., Bida, M., Gemrot, J., Kadlec, R., Plch, T. Emohawk: Searching for a "Good" Emergent Narrative. In: ICIDS 2009, Guimaraes, Portugal, LNCS 5915, Springer, p. 86-91.
8. Plch, T., Jedlicka, T., Brom, C.: HLA Proxy: Towards Connecting Agents To Virtual Environments by Means of High Level Architecture (HLA). In: Proc. of Cognitive Agents for Virtual Environments, LNCS 7764, Springer, Heidelberg, pp. 1--16, (2013).
9. Brom, C., Bida, M., Klima, M., Gemrot, J., Pibil, R., Plch, T., Kadlec, R.: Virtuální agenti In: Umelá inteligencia a kognitívna veda II, STU v Bratislavě, pp. 21-52 (2010).

Appendices

Appendix A – Attached Digital Content

- `/video/` – videos showing mechanisms presented in this thesis in the KCD game
- `/stats/` – measured data for deployment evaluation
- `thesis.pdf` – text of this thesis
- `readme.info` – description of attached digital content

Appendix B – Video commentary

- `Video 1` – The Day-By-Day life of NPCs in a testing level. The left side of the screen has working areas. At 0:05 the NPCs go Home (right side of the screen) or to work (left side of the screen). At 0:25 and 0:40, a woman on the right side goes to sleep. Men on the left side of the screen are working on the fields. At 1:00, the farmer on the bottom right corner decides to go home for a dinner. His wife (left bottom side) wakes up to prepare dinner. They sit at the dining table together.
- `Video 2` – The same situation as in `Video 1` from a different angle. We see the areas and the RKN of the world at the video's start.
- `Video 3` – Soldiers going to the Tavern for a beer. The tavern is a Smart Area and has limited Behavior Injection to simulate the limited places within the Tavern. One NPC at 0:05 can be seen to walk away from the Tavern since he did not acquire a »guest« BI. Sitting at tables is managed by the SA. At 0:07 we can see the waitress being called to the first occupied table. She also greets the guests. This is a BI provided by the Tavern to the waitress. Sitting at tables is managed by the SO Table and the SO Bench. The SO Table provides a BI which asks the SO Bench for a sitting BI. All other table activities are handled by BIs provided by the table to both the waitress and the quests. At 0:22, there is a synchronized behavior managed by the Table SO to pour beer to different NPCs. The table has the information who sits where and provides the behavior to the waitress how to synchronize with the NPCs sitting at the table. The synchronization part on the guests is part of the BI provided by the Table SO. At 0:39 the waitress has nothing else to do, so she cleans tables. The BI to go and clean the tables is from the Tavern SO, but how to clean the tables is provided by the Table SO. After she is done cleaning tables, she goes on at 0:52 to sweep the floor. This BI originates from the SA Tavern which knows where the broom is.
- `Video 4` – Shows the RKN network in the Skalitz village. The links are green lines between entities, where flying arrows denote the edge's direction and what tags annotate the semantic connection. At 0:04, we can see sheep being linked to their home meadow. At 0:06 we can see the hierarchy of links in Skalitz.
- `Video 5` – Shows the virtual life in Skalitz, at 0:18, we can see farmers working the nearby fields. At 0:26 the player commits an attack on NPC, which drops the hoe. All other NPCs follow suit and evaluate the situation. The woman when chased, runs way. The men will go and get help from a local guard. At 1:08 the player scares some sheep around their meadow. We can see how the sheep behave like a heard. At 1:29, the flock is separated and creates 2 separate flocks which function independently. At 1:41 a guard finally finds the player and confronts him about the crime committed earlier. After that the player is arrested.
- `Video 6` – Shows the life in Skalitz. At 0:07 we can see a woman buying groceries for dinner. At 0:28 we could see two women gossiping about

what happened lately. At 0:30 there is a local guard on duty walking around the village. At 0:34 a woman recognizes the player and greets him. At 0:40 we can see local men being at the Tavern.

- `Video 7` – Show how the environment reacts to crimes. At first 0:05 the player has drawn a weapon and the guard tells him to put it away, since it is a crime (in medieval times this was a capital offense). At 0:21 when the NPC is attacked it defends itself but does not engage since he has no weapon of his own or friends nearby. Another NPC runs for help. At 0:34 a guard arrives and directly engages the player since he saw him attacking an NPC.
- `Video 8` – Shows life in Skalitz. At 1:12 the player starts to follow NPCs to work. We can see that the NPCs actually went to their field to work on their farm. At 2:23 we can see two men talking daily news.
- `Video 9` – Shows use of a »door« Navigational Smart Object. The door connects two parts of a navigational mesh, where specific animations have to be played (i.e. open and close) to traverse over the border created by the door. At 0:08 we can see the door telling the first NPC that went through to close the door behind it, since it (i.e. the door) had no knowledge of the second NPC which wanted to go through. We can see that at 0:12 at the upper door, the door instructed the NPC (via the BI behavior provided by the NSO) to avoid closing the door (i.e. it played only the »open door« animation since there are more NPCs in line. However, the last NPC closes the door. Later, this process repeats in both direction. For example, at 0:54 at the lower door the returning NPC does not close the door since there is another one coming through.
- `Video 10` – Shows us the synchronization mechanisms between NPCs. In a simple labyrinth, NPCs wait for each other at 0:04 and go together to the edge of the maze. At the edge, only one NPC passes through and does not wait for its friends. Everybody goes to a different end location. This repeat and at 0:20 we can see the NPCs again waiting to be at the same spot together.
- `Video 11` – Shows the tool chain within the KCD editor. At 0:05 we can see the SBT visualized. Green nodes have succeeded and red nodes have failed. Blue nodes are still running. At 0:08 we can see a Behavior Injection into the running SBT. Yellow lines denote where the execution already was. At 0:17 we can see continuous evaluation of conditions which results in stopping branches and starting new ones which triggers BI at their leaf nodes. This shows the basic principles of visualization and Behavior Injection in practice.
- `Video 12` – Shows a combination of NSO behavior and SA BI into NPCs. The NSO provides the NPCs at first with a line to be in front of the door. Within that behavior, the NPC tries to open the door but fails, standing in front of the locked door for a while. After that, the NPCs asks the World Smart Area what to do, and the provided BI sends the NPC back to the end of the line.