DESIGN AND IMPLEMENTATION OF EMBODIED CONVERSATIONAL AGENTS

By

FERNANDO GERACI

A thesis submitted to the

School of Graduate Studies

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Master of Science

Graduate Program in Computer Science

Written under the direction of

Mubbasir Kapadia

And approved by

_____

_____

_____

_____

New Brunswick, New Jersey

May, 2019

ABSTRACT OF THE THESIS

DESIGN AND IMPLEMENTATION OF EMBODIED CONVERSATIONAL AGENTS

by Fernando Geraci

Thesis Director:

Mubbasir Kapadia

In less than a decade, virtual assistants had established themselves as very handy natively included components of the main commercial consumer systems available. These, expand both the applications and challenges of human-machine interaction, sparkling many solution to some well researched and new problems. Virtual agents are a manifest of how artificial intelligence is making its way into improving the regular consumer's experience, productivity, and how this, is gradually becoming an essential part of people's life.

However, there is are still big gaps in those available today, in terms of the quality of the actual human-machine interaction. These agents are not fully prepared to understand the nature of actual human communication and conversation. This process not only involves understanding natural language utterances and following commands, but true conversation is achieved by also capturing information from facial expressions and body language, to finally assign a semantic meaning, in context of it all. This introduces a new layer of complexity in terms of how perception is handled, signals are processed, interleaved and analyzed, for then elicit coherent and proper behaviors and responses.

This work will display the design and implementation, of a virtual modular intelligent agent. We expand the boundaries of simple unilateral communication, to a more robust, engaging, believable and meaningful interaction. This is achieved by enabling the agent to complement the basic speech input with voice emotion, facial expressions recognition, and smarter natural language parsing and generation. Finally, we demonstrate, with the presented implementation, how this agent raises the standard for human-machine communication, and how artificially intelligent agents can work in a multi-input model of complex signals in order to elicit meaningful and compelling behaviors in more natural interactions. Moreover, as opposed to other smart agents, this learns, remembers and pro-actively communicates with a complex set of coherent behaviors.

Acknowledgements

First and foremost, I would like to thank Dr. Mubbasir Kapadia, whose mentoring, expert advise and demanding but extremely personable approach have always taken me to my limits, and helped me keep pushing those boundaries further away.

Additionally, this work would have been impossible without the help and collaboration of Mr. Samuel Sohn and Mr. Xun Zhang. Their expertise, will to aid and openness has been an invaluable gift and inspiration.

Finally, I would like to thank my students, who primarily, have always showed me new ways to look at the old.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Introduction

When discussing artificial intelligence (AI), the average consumer public typically pictures products which are the embodied bridge between the academic research theory and the implementation of these techniques. The idea of mechanizing aspects of human behaviors, especially those attributable to our sentient nature, was one on which many authors, throughout history, have focused their research on. It would not be accurate to say that the field of Artificial Intelligence, and all its derived products, is one which have suddenly spawned and flourished just in the late stretch of the twentieth century. Furthermore, despite that the discipline as such was academically formalized in the 1950s, its foundations go many centuries back. However, just since the beginning of the 21st century, we, as a society, can perceive a sheer exponential growth of the field, through the introduction of unparalleled products and services. Commercially, these come accompanied by colorful terminology which is rapidly making its way into popular knowledge. Some are *Deep Learning*, *Machine Learning*, *Natural Language Processing*, *Computer Vision*, *Speech Recognition* and *Robotics*, just to name a few. These concepts have gradually, and at a rather hastily paced, been making their way into people's minds, mainly via the consumption of said goods. Nowadays, the so called *Smart* products, are normally enhanced and enabled by intelligent software systems. From the perspective of popular perception, being able to visualize a tangible AI-powered product, gives one the sense of being staring straight into the future. Everyday, we are benefiting from, and relying more and more on these products for our daily activities. One doesn't have to reach far out, to find an example. It suffices to

think of how smart-phones, which are almost indispensable nowadays for the vast majority, can be considered as a gateway to a vast pool of intelligent products. This work will focused exclusively on virtual assistants which in the past decade, since Apple's commercial inception of *Siri* in 2011, have been growing and reaching new heights of complexity and reliance. This was a critical milestone on the path of developing and enhancing the machine-human interaction experience. Throughout this path, we can find many systems which have been developed and subjected to, among others, the test presented by Turing, 1950. This path spans through many decades, some of the earliest computer models are Weizenbaum, 1966 *ELIZA* and Mark Kenneth Colby's *PARRY* (1972).

Just in the past seven years, we witnessed the rising of several of these products offered by different lead technology companies. For instance, IBM Watson (2013) - Ferrucci, 2012, Microsoft's *Cortana* (2014), *Ok Google* conversational search (2013) and Amazon's *Alexa* (2014), just to name a few. It is clear that many leading research enterprises are currently geared at further developing their commercial AI products due to an existing high demand for such products. Furthermore, all the aforementioned agents are included natively in each of the companies' main operating systems and flagship devices. This is not only a statement of how useful consumers find them, but also how aggressively, companies are collecting data to further improve their offering.

But it begs the question: what are the main factors which make a virtual assistant useful, attractive, and reliable? Certainly before 2011, there were virtual agents around to help with simple automation tasks, so what makes today's agents different? First, we will look at the Human-Machine Interaction factor. Being able to simply *talk* to a computing system, removed a big layer of complexity which was in the way for many average consumers, just as *touch* capabilities, bridged between less tech-savvy users and this formidable devices. People no longer need to have access to a laptop or a workstation, and to understand the concepts of chat rooms or terminal interactions. The interaction is now intuitive and readily available, either by active listening from the devices, a simple *voice command* or a push of a dedicated button. The second factor

is their usefulness. As opposed to the original *chat-bots* for support, or phone assistants, today's virtual assistants are no longer dedicated to a single task. Instead, they are now able to handle many of smaller, everyday tasks such as calendar handling, reminders, media playing, search and more. Moreover, this impacts directly on the reliability part of our question. The more people use these services, the more robust they become and better perceived by the public. Hence, availability, access, and usefulness are three factors which have allowed virtual intelligent agents to steadily and hastily grow in the industry encouraging research on pertinent fields.

There is one more factor which, as of today, is not fully addressed, engagement. Many of the previously listed products claim to have *conversational* capabilities, but what is really comprised in such a definition? A conversation is a bilateral communication process, based on the exchange of ideas, rather than simply the issuance of commands–which is most of the technology we have commercially today. However, true conversation goes beyond the one-sided interaction, with deeper understanding on the meaning of the message. Saying that communication is *the mere process of exchanging words* is limiting and incomplete. There are many factors which are encompassed in engaging and believable interaction. For instance, facial expressions, the emotional load of the speech and body language, etc.

Our focus is on addressing these features, and to combine and process said signals in order to generate a deep understanding of the message. That then allows the agent to elicit meaningful behaviors and responses. We will proceed to expose the implementation of Sara, a virtual intelligent assistant agent, capable of perceiving, understandings and acting in accordance to the conversation and its context.

In summary, the key blocks for keeping the fluency and reality of the communication with a smart agent are understanding and coherent behaviors. Understanding is a product of the analysis of the input of the agent. This is comprised of multi-modal information, facial expressions from live video stream, speech containing the tone information and the recognized text, and other sources of inputs such as touchscreen interactions. The output of the deliberation based on the input signals, will come in the form of behaviors, that is, the feedback of the smart agent. This includes speech,

facial expressions and body language, in the form of animations.

This work describes the end-to-end design and implementation of an embodied conversational agent. It presents the input-analysis-output pipeline, and the processing involve in each segment of the perception-deliberation and acting cycle. Finally, it demonstrates such implementation with demo examples and use-cases.

1.2 Problem Background

Many of the techniques and systems which are combined and synergized in this implementation have been heavily researched and developed throughout the past decades. The final product of this work is a product of the interpolation of said components into a single system. Intelligent conversational agents have been the focus of research for years now. Eliza Weizenbaum, 1966 and Parry Colby, 1972 are some of the first chat-bots ever created. Moving two decades forward, we can find ALICE Foundation, 2002, the Artificial Linguistic Internet Computer Entity. It is another type of chat-bot inspired by Eliza using Artificial Intelligence Markup Language files to store its knowledge. There have also been approaches in having a general-domain question-answering agent in a museum Leuski et al., 2006; Robinson et al., 2008. Leuski et al. Leuski et al., 2006 and Robinson et al. Robinson et al., 2008 describe a general question-answering agent used in a such a scenario, trained using utterances.

In terms of logical processing, techniques have been evolving significantly. For instance, a simple hard-coded rules approach, can benefit from an additional statistical model layer on top of them, as seen in Young et al., 2010. It is also possible to learn generation rules from a minimal set of authored rules or labels using the vector space framework as described in Banchs and Li, 2012. Furthermore, deep recurrent neural networks like Seq2Seq Sutskever, Vinyals, and Le, 2014; Serban et al., 2015 or Long Short-Term Memory (LSTMs) Hochreiter and Schmidhuber, 1997 provide a powerful way of creating general conversational agents. However, such systems usually require a superlative and difficult to gather amount of training data which can easily result in a knowledge acquisition bottleneck.

Many games, entertainment aimed implementations and educational solutions take advantage and present novel work when it comes to conversational agents.Morris Morris, 2002 provides an example of using conversational agents in games. Tarau and Figa Tarau and Figa, 2004 provides another example of an agent in an educational environment. In the latter, a Prolog database is used to store a variety of possible question/answer sets that the agent can pick from.

Regarding the analysis of human emotions, these can be measured from different aspects. One efficient and direct way to obtain emotional information is through facial expressions. The Facial Action Coding System (FACS) devised by Ekman and Friesen established a formal way to describe facial expressions. To improve the accuracy, speeches and body language analysis can also be applied. Emerich, Lupu, and Apatean proposed a emotion recognition method by analyzing speech and facial expressions. Their framework devised statistical models for classifying emotions with the input of image sequences and utterances. The result showed the correlation of facial expressions and speeches. Recent study by Chowanda et al. shows that correctly modeling emotions and large-scale personalities for virtual agents can greatly boost the user experience and engagement. Potard, Aylett, and Baude proposed an evaluation baseline for synthesized emotional speech and facial expression, and it allows the speech synthesis to add gradual changes to the perceived emotion both in terms of valence and activation.

Finally, believable agent embodiment plays a key role in engaging machine-human interactions for they allow humans to assimilate to their digital counterparts on a different level. There are frameworks such as SmartBody which address complex problems, such as lips synchronization Xu et al., 2013 and motion blending techniques Feng et al., 2012, and ADAPT Kapadia, Marshak, and Badler, 2014, a complete testbed of purposeful human characters in a rich virtual environment. The concept of meaningful embodied agents can be expressed in terms of their capacity to perform behaviors, as presented in parameterized behavior trees all, n.d.

Our goal is not to overlap with this research but to leverage it by combining all this elements into an engaging and meaningful multi-modal conversational agent.

1.3 Problem Scope

The Embodied Virtual Agent's model presented in this work, is in part, the product of the interpolation of many diverse techniques and systems which have been heavily developed and researched upon. The scope of this presentation is limited to explain such a design and how this methods converge to achieve the final goal of simulating an embodied conversational virtual agent. Although these systems will be presented and used in the implementation, these will be explained from the product's perspective. The breadth and depth of their exposition is limited by their interfaces, input and output required by the agent. Throughout the different chapters, the concepts of software engineering, signal input and output, computer vision, behavioral modeling and natural language analysis will be covered, among others. This is, above all things, a software implementation. The utilized framework will be explained in detail and how this modular architecture can be design to model an abstract representation of agent's the *Perception*, *Deliberation* and *Behaviors* overall cycle. In terms of signal handling, processing and analyzing, the main tools and how these are intertwined within the system is explained in from the perspective of their input and output. We will expose the main processing mechanisms and how these, ultimately, produce the necessary output, as input for their consuming modules. When it comes to the deliberation phase, on of the products of the previously processed signals is speech in the form of text. The models for handling their syntactic and semantic analysis and their implementation will be presented. The main Prolog analysis engine will be explained in detail. A simplified version of the used grammars are presented in the form of diagrams and rule sets. In addition, we will also explain how the tokenized parsed input is utilize and what role it plays in the agent state of beliefs. Because emotional analysis is a key factor of the implementation in terms of the conversational aspect of the agent, different modeling approaches will be explained, ultimately detailing the system used by this model. Finally, in terms of the agent's behavior modeling mechanics, we will present the concept of behavior trees, their different types and components, culminating on the implementation of an asynchronous interactive model, capable of handling

many complex responses.

1.4 Outline

Throughout this work, it is described a bottom-up, end to end, design and implementation of the embodied conversational agent's model. This is comprised of three main foundational blocks, which are the agent's *Perception*, *Deliberation* and *Behavioral* capabilities.

Chapter 2 presents the design of the agent model we use as the foundation on which we build upon the other components. The implementation and the framework are presented to explain how an agent can modularly and asynchronously manage a real time perception, deliberation and act continuous cycle. Along the framework, the assets pipeline is explained and how these are included in the implementation of the avatar. Some of this assets are animations, sound and models. We also present the concept of modularity, as defined by this work, which allows for scalability and independence of the interacting gears and components of the system.

We then proceed to brake the system down in its three fundamental building blocks, as mentioned before, these are: perception, deliberation and behaviors.

When explaining *Perception*, we will describe the main feeds of the agent and how these are acquired, processed, analyzed and finally combined to as a product for the agent's system of beliefs. The input is mainly comprised by audio and video. The processing and deriving products will be explained in detail in *Chapter 3*. Some of these are user utterances, their semantic meaning, sentiment on this speech and emotional facial recognition, among others.

Chapter 4 describes in detail the utilized mechanisms for parsing speech as input, validating and understanding its structure and utilizing this to update the agent's system of beliefs and deliberation process. It is also explained how the analyzed semantics will aid in creating and completing a knowledge base represented as a conceptual network. This structure is presented with examples and use-cases. Finally, an approach for generating language and the challenges this impose is explained along

with approaches for the agent to utilize such results.

The last foundational component of this model's design is presented in *Chapter 5*. This is the agent's Behaviors system. This is represented and implemented utilizing *Behavior Trees*. Their composition, types and examples are covered in detail. The agent's interactive nature is demonstrated via the utilization and design of asynchronous real-time capable tree structures. This section finalizes with a simplified implementation of such interactive trees and their diagrams.

Chapter 2

Architecture

2.1 Implementation and Foundational Components Overview

This section goes on to explain the basic architecture, design choices and components of the general framework used in this work. The overall implementation not only deals with the basic animation and physics of the virtual agent, but also with every component which enables it to perceive, understand and ultimately interact with its environment, other agents, and users.

The framework used for this implementation is called *NPC*. Every basic component was built within the Unity engine (Technologies, 2018), in its .NET C# scripting language option. Along the framework, libraries such as *Nemesysco* and *Affectiva* are used for different signal processing. We can enumerate components of the implementation in the following four categories:

- Framework Foundation: Main NPC Controller
- Basic Agent Components: Body, AI and Perception
- Modules: Customizable Interface Implementations
- Libraries: Vendor Provider Unity plugins.

In addition of defining basic members and properties of an agent, the foundation is the main part of the implementation, being the only section which executes the main simulation loop on a per-frame basis. No other component is directly consumed by client code nor constantly updated. This allows for a controller capable of managing every single aspect of the execution of an agent. In just a few words, on frame update, the controller will resolve the current state and data of the agent's perception,

body and deliberation system. Each of the aforementioned segments, count with a single update function which if enabled, will process all data relevant to their domain of knowledge and execution. These are the basic aforementioned agent components: *Perception*, *Body* and *Deliberation*. Each of these are implemented in their own class and executed individually, utilizing and updating a centralized pool of state within the main controller. We will also display all how settings can be uniquely customized on each individual agent. This is to ensure that either a general model can be used on, for example, crowds, or agents can be customized based on the simulation's need, like in the case of a conversational agent, which might not need to process social forces or in-simulation perception if only interacting with the user. Extending the basic blocks capabilities, modules can be added to the framework to add general or unique capabilities to a single, or multiple agents. These modules are recognized by the main controller, enabled and updated during the simulation. Some of the modules used in this implementation are specific for signal processing, natural language parsing and behavioral modeling. Finally, although not part of the main framework, vendor libraries are added on top or within modules. This projects utilizes such libraries for natural language parsing, conceptual knowledge base, image and computer vision analysis and voice processing and analysis.

We proceed to elaborate on each of this four categories and their implementation.

2.2 Agent Model Framework

The implementation was design primarily with scalability and customization in mind. It is true that no single implementation would be able to offer and cover every single aspect needed of all virtual agents in every simulation. The primary intention was to provide solid bones, a skeleton, which would enable basic agent services, and allow for adding reusable customized modules as needed.

Many state-of-the-art frameworks could potentially be disregarded due to and unfriendly implementation and learning approach.

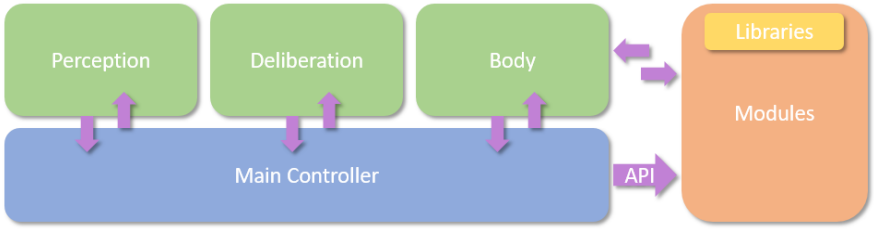


FIGURE 2.1: General Framework Structure

As opposed to other implementations, *NPC* provides an intuitive component which takes care of adding all dependencies and managing internally the complexities of for example: animations, inverse kinematics, navigation, audio and clips synchronization, social forces system and obstacle detection, among others. *NPC* centralizes all lits functionalities in the main controller and its modules. Also, unneeded features can be disabled, saving quite a big of computation, especially on high density crowd settings. Figure 2.2 display the primary settings which can be adjusted from the main controller component. Along this controller, there are a few other components which although added to the agent object in Unity, are not displayed. These components are: *NPCBody*, *NPCAI*, *NPCPerception*, a main *Collider*, a *Rigidbody*, the *NPCIK* controller and an *Animation Controller* state machine. Every public aspect of these components can be managed via the single displayed inspector interface, there is no need for the developer to access components individually in the editor interface. The controller, however, does expose properties

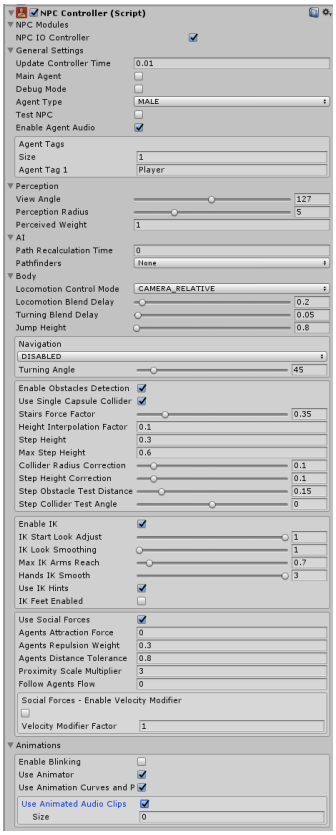


FIGURE 2.2: NPC Framework Component

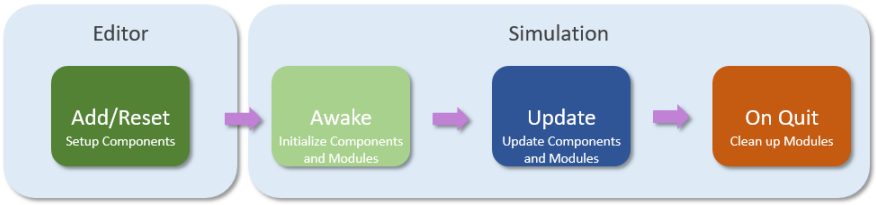


FIGURE 2.3: Global Framework Life Cycle

and methods which would allow consumer code to further interface with all the elements of the agent.

The first step on setting an agent up, in this case one which will ultimately be able to converse, is just to add the main controller as a component of the *GameObject*.

Data: Modules $\pi \in \Pi$
Data: AI α
Data: Perception ϕ
Data: Body β
if *(!initialized)* **then**
  $\Pi \leftarrow CollectModules()$
  $\alpha \leftarrow SetupAI()$
  $\phi \leftarrow SetupPerception()$
  $\beta \leftarrow SetupBody()$

Algorithm 1: Add or Reset

Algorithm 1 is executed every time the *NPCController* component is added to an agent. Once finished, all needed components will be added, and hidden from view, except for the controller settings display on figure 2.2. Having a single point of set up is not only efficient, but also narrows the scope of development and point of failure of the framework.

Data: Modules $\pi \in \Pi$
Data: AI α
Data: Perception ϕ
Data: Body β
$\phi.Initialize()$
$\beta.Initialize()$
$\alpha.Initialize()$
$nextUpdate \leftarrow (Now + UpdateDelta)$
$\Pi.Load()$

Algorithm 2: Awake

Algorithm 2 is execute when a simulation starts and before any component has been updated. This is really important since many components need to initialize themselves to ensure there are no missing dependencies or invalid state options. The most critical part of the initialization process, is to instantiate all fast data structures used during runtime, which cannot be serialized and transition between editor/simulation mode. Many components utilize these non-serializable data structures to optimize CPU lookup time during each frame. For example, ϕ (Perception) will initialize a *PerceivedEntities* map, adjust perception colliders and view angle. β (Body) will initialize main colliders, available gestures, affordances, IK controller reference, agent navigation navigation and steering mechanisms, the physics rigid-body and audio-animation clips synchronization objects, among others. Finally, the *NPCAI* module will initialize navigation modules, if available and most importantly, pre-loaded behavior trees for immediate execution.

Data: Modules $\pi \in \Pi$
Data: AI α
Data: Perception ϕ
Data: Body β
if *(nextUpdate* $>$ *Now)* **then**
  $nextUpdate \leftarrow UpdateDelta + Now$
  $\phi.Update()$
  $\alpha.Update()$
  $\beta.Update()$
  foreach $\pi \in \Pi$ **do**
    if *(*$\pi.Enabled$*)* **then**
      $\pi.Tick()$
  end

Algorithm 3: Update

Once the simulation started, as diplayed on 3, the agent or agents will be updated every frame. As previously mentioned, the only component which can update the framework is the *NPCController* driver. This will update every basic aspect of the model and each module, if enabled, which has been appended to that particular agent. It is important to notice the update order of the components. ϕ (Perception) is updated first, then α (AI), which will process environment or input changes, and finally β

(Body) for acting, steering, audio and animations. This is analogous to the Perception-Deliberation-Act cycle.

The implementation emulates a well established general agent model, the Perceive, Deliberate and Act cycle. On every frame, this cycle will update the components in that order. First, the environment (agents, objects and entities) will be considered, then, the agent's beliefs system will be updated accordingly, followed by deliberating on the current action, if needed, for finally acting upon this deliberation.

2.2.1 Perception, Deliberation and Action Model

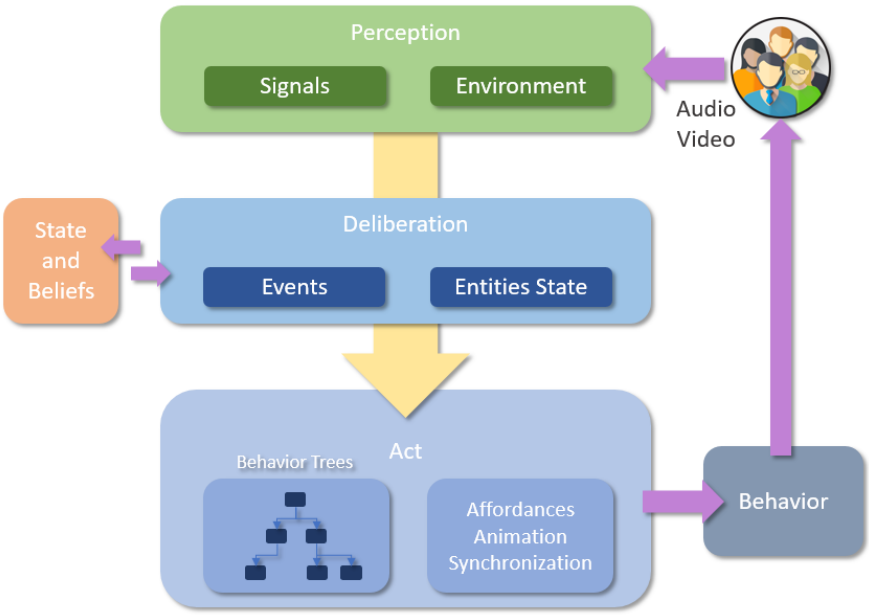


FIGURE 2.4: Agent Model

Those components previously described as the *basic* ones, are those which will execute specific tasks of the execution cycle. These tasks are separated and carefully sequentially executed, for output of a component, will be input of the other. We can see in figure 2.4, how each layer executes in order and consumes the previous layer's product as input. This section will briefly describe the main functions of each of these elements of the framework.

The first component in the chain is *NPCPerception*. Generally speaking, an agent

should be able to perceive its environment, and some external signal stimuli. The framework basic ϕ (Perception) component only deals with virtual line of sight and proximity of other entities - both agents and objects. When it comes to conversational agents, that input alone is not sufficient, we need also to be able to perceive signals coming from the user. These are in the form of audio and/or video. Special modules will extend the perception capabilities of the agent presented in this implementation to be able to read and process such signals. These will be explained in detail later. There are two aspects of this module which need to be highlighted. First, the fact that perception works on top of the engine's physics loop, which is independent from the main per-frame general Update. This means that entities are perceived asynchronously to the main component Update call from the *NPCController* component. This is because entities proximity are detected by a trigger on a sphere collider attached to the each agent. Once a trigger fired, we need to determine if the entity is within the agent field of view. Before determining this, it is really important to check if the agent was NOT previously detected. This can be quickly done in O(1) complexity with the previously mentioned hash maps which keep track of currently perceived entities. Finally, due to the cost of these physics and vector operations, it is important to exploit every possible optimization. The main optimization does occur in the main Update call for this component, and consist in shrinking the perception radius (not the view field). This will save the agent many unnecessary physic collision trigger detections, for if the agent is surrounded by other, say four agents, there is no need to expand the collision detection beyond the furthest agent. This is crucial in highly populated crows scenarios. The good news is that when it comes to conversational agents, this will probably be used in a single agent-to-user interactions, with no need to enable the perception cycle, saving a lot of physics computation which ultimately results in saving CPU cycles which will be most needed for other uses, such as natural language processing.

Next in the update sequence, comes the *Deliberation* component, called *NPCAI*. By default, the *NPC Framework* provides basic deliberation services, these are path resolution for navigation and behavior trees execution. Path resolution can be customized

with specific navigation implementations, or default to Unity's *NavMesh* library. Conversational agents will use modules to process signals, analyze and update state. In figure 2.2, we can see a Navigation component can be selected if available. These components are classes which implement a specific *NPCNavigation* interface, but generally speaking, the default Unity's path resolution mechanism is preferred due to optimizations and portability between projects. For the purpose of the conversational agent discussed in this implementation, no navigation is required, so any navigation capabilities can be safely disabled, again, optimizing computation cycles. The most important aspect of the AI component is the capacity of running, updating, pausing, starting, stopping and interrupting behaviors. These behaviors are presented in the form of trees and will be covered in detail in *Chapter 5*, but in short, every Update cycle of the component, will tick the currently executed behavior tree. At any given point in time, there could be either 0 or 1 nodes executing for each agent. Nodes which will result in actions are called *Affordance* nodes, and these affordances, are atomic actions which will make the avatar perform either a gesture, execute an audio-animation clip sequence, go to location, or perform any specific action. These affordances are C# attribute-tagged in the AI components as *NPCAffordances*. Each of these affordances are available for the behavior trees to use in their leaf action nodes. Although the *NPCAI* modules executes a single tree at a time, it can potentially queue many trees in FIFO order, and execute them sequentially. Finally, any running tree can be stopped, or paused. Pause will of course allow for resuming its execution, while stopping the tree doesn't. In *Chapter 5*, we explain how the system is capable of executing *Interactive Behavior Trees* which are the chosen solution for the conversational agent problem. A single tree, which evaluates many conditions in parallel and execute those which satisfy certain parameters. There is one more thing to highlight, which is the separation between executing affordances individually as opposed than in organized fashion, via a data structure such as a behavior tree. That is an important distinction, since the body component, discussed momentarily, does allow for atomic execution of affordances, but these are independent of any order or consequence of execution, as opposed to the trees. Finally, the *NPCBody* component will handle every operational aspect of

the agent in the simulation, this includes but not limited to: steering, social forces, IK targets, obstacle detection, path navigation, multi-layered animations execution, animation time and curves, user IO controls and audio-synchronization control. The fact that every aspect which has to do with animations, navigation - not path resolution - and inverse kinematics will be processed per-frame within this module, makes it to be the heaviest from the implementation and processing perspective. However, for single, one-way user interaction conversational agents, there are many features which we can disable and obviate. The main components which we will use from this module specifically are its animation and audio synchronization capabilities. This is achieved using a support data structure from the framework called *NPCANimatedAudio*. This element, encapsulates animations and audio clips, providing a way to annotate time-stamped execution events. For instance, while an audio is playing, assume a phrase, we can trigger sequentially phonemes animations to emulate lip-synchronized speech. The same is true for the inverse, a longer animation which needs to play an audio clip at a given point of the animation. An unlimited array of these can be defined via the inspector interface, as displayed in figure 2.2.

2.3 Basic Components

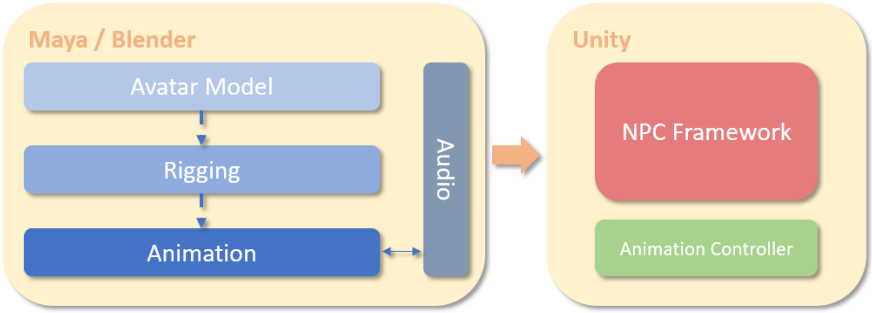


FIGURE 2.5: Overall Development Pipeline

Now that we count with a brief overview of the main components of the framework, it is time to define the objects on which this framework works on. The *Unity* engine

defines almost every component of a simulation scene a *GameObject*. These are components which usually comprehend a render mesh, to display the three dimensional model, an animator controller and the concept of a *transform*, which is the abstraction of an objects position, rotation and scale in the world. This section provides a brief description of the main components utilized for the simulation from the hard-assets perspective. These hard assets are going to result in the embodiment, animations and audio of the conversational agent. Ultimately, the animated avatar is the result of logic-driven assets. Figure 2.5 describes the development pipeline used for this implementation. There are two main blocks, the hard assets development portion - models, rigs, animations and audio - and the Unity simulation development part. Generally speaking, and once a primary model has been created, both parts of the pipeline work in parallel. While the simulation is being developed with an initial model and rough assets, artists and technical designers can work in creating new models, animations for these models, adapting rigs and creating audio. Within each of these phases, especially for the hard assets one, tasks usually work in sequential fashion. As mentioned, once an initial model and basic assets are in place, the end-to-end result is attainable by threading the assets to the simulation logic within the engine. By this time, designers and artists can re-iterate over the rig, add more animations and provide developers with version X assets. Developers should always keep their implementation independent from the assets. In other words, assets should not dictate logic, because otherwise, the whole reusability value of the framework will be lost. The only framework component which *must* be customized, are the animation-audio clip sequences. These will change throughout development until the final rigged model, animations and audio clips are in place.

We will describe separately those products and components used in this implementation.

2.3.1 Avatar

The main model which we use has been created and rigged in *Autodesk's Maya*, which is a highly professional, industry standard, modeling and animation tool. Creating this model is a highly technical and involved process. We will only enumerate its phases and the end result, focusing rather not on its development, but on what is needed to successfully bring a simulation-ready avatar into Unity, for the purpose of developing the simulation. The modeling global overall cycle consist of the following steps:

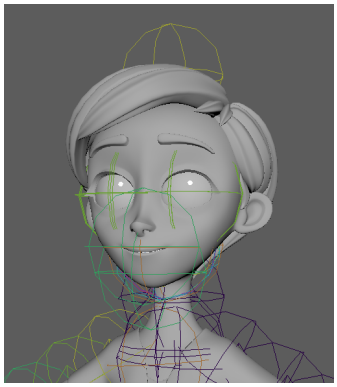


FIGURE 2.6: Facial Blendshapes

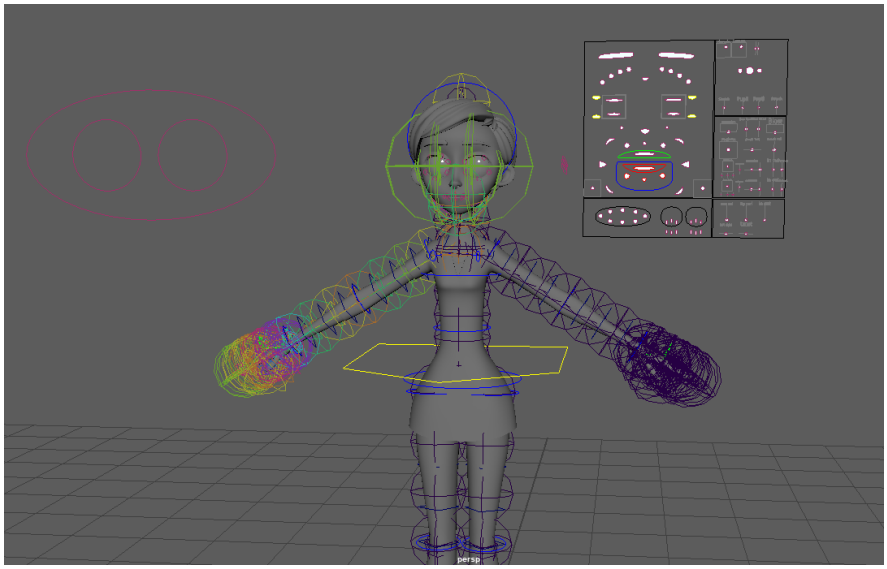


FIGURE 2.7: Model and Rig

- Modeling: This is when the complete mesh of the model is created. Several approaches could be used, but the ultimate result, is a single group of meshes which, in addition to textures, UV maps and materials will create our empty avatar. By empty, I mean that although the model is complete, bones and joints are yet to be added to the model.

- Skin and Rig: Once the model has been completed, we will have an A or T posed

character, as in figure 2.7. For this avatar to be a functional humanoid model, we need to add the basic joints and bones. This process will define all the character articulation points. This is paramount for animation for all rotation pivots are defined. In addition to the bones, we will also create the skin, which is how the weight of forward kinematics rotations will be distributed among the meshes' pixels, when animated.

- Blendshapes: Last but not least, once the avatar's model is completed, rigged and skinned, we proceed to create the blendshapes, which are the link between animation controllers and facial skin points. These controls allow for manipulation of any portion of the face mesh's triangles. The granularity depth depends on need. For this case, we will need the avatar to be able to not only perform phonemes, but also to demonstrate emotional states via gestures. For this, the technical design team must allow for brow, lips, cheeks and many other facial segments movement, as shown in figure 2.6.

- Export: Finally, once the model is fully functional, we can proceed to export/import it into Unity to thread it with the logical framework, as shown in figure 2.8. Once in Unity, we will attach the *NPCController* component and all custom required modules. In addition, we need to ensure the *Humanoid* rig has been selected, which means that the avatar is animation-state-machine ready. Unity's *Animator Controller* is a key element of any humanoid avatar in terms of multi-layered animations and inverse kinematics. From this point on, we can extend the basic framework to utilize other services such as audio and network access.

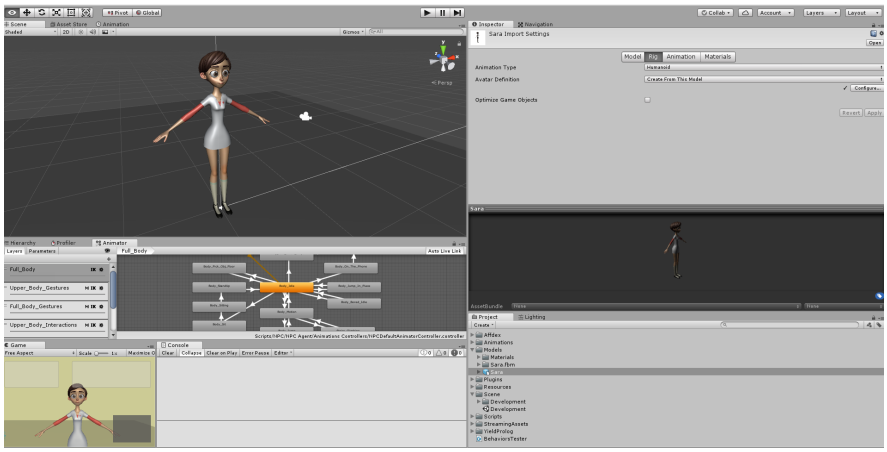


FIGURE 2.8: Exported Model

2.3.2 Animations and Audio

This assets are produced independently from the Unity development phase. As soon as the avatar is complete, we can import it directly into Unity and start working until receiving new these new assets with a library of either basic animations and audio, or dummy values which emulate what is going to be the final product. As mentioned, the *NPC* framework allows to synchronize this animations and audio clips into atomic units of execution. The result is an animation which can play any number of audio clips throughout its execution. The opposite is also true, we trigger audio clips at any given point of an animation in its normalized time. It is worth noting that this mechanic is yet another tool available in the framework and not the de facto system to be used for every single scenario. For the purpose of conversational agents, however, we can heavily rely upon this initially, to execute prerecorded sequences, until another, more sophisticated, logic-driven mechanisms is in place.

The *NPCAnimatedAudio* is the object utilized to execute these atomic sequences. This class is basically is collection of queues of animation and audio timestamps. When each unity will execute, depends on how the designer set it up in the editor. Once running, the *NPCBody* controller will be responsible for driving a coroutine which will iterate every frame, querying these queues and executing elements at the correct timestamped time. Algorithm 4 displays a coroutine which will run, not blocking the

```
Data: AnimatedAudio animatedAudio
Data: Animations animations
Data: Audio Clips audio
Data: NPC Controller npcController
if (randomizeAnimations) then
    animatedAudio.ShuffleAnimationClips()
animationQueue ← animatedAudio.Animations
audioQueue ← animatedAudio.AudioClips
startTime ← Now
runLength ← startTime + animatedAudio.Length
do
    if (audioQueue.Peek.ExecuteNow) then
        npcController.PlayAudio(audioQueue.Dequeue)
    if (animationQueue.Peek.ExecuteNow) then
        npcController.DoGesture(animationQueue.Dequeue)
while Now <= runLength;
```

Algorithm 4: Execute Animated Audio

main thread, until all audio and animation clips of an *NPCAnimatedAudio* instance have been executed. To alleviate the task of the designer, the *NPC* inspector allows him or her to design these clips utilizing their real execution time. Upon initialization of the simulation, these instances are **baked** into normalized time lines, on which other *metadata* is collected, such as total duration and proper execution order.

2.4 Modularity

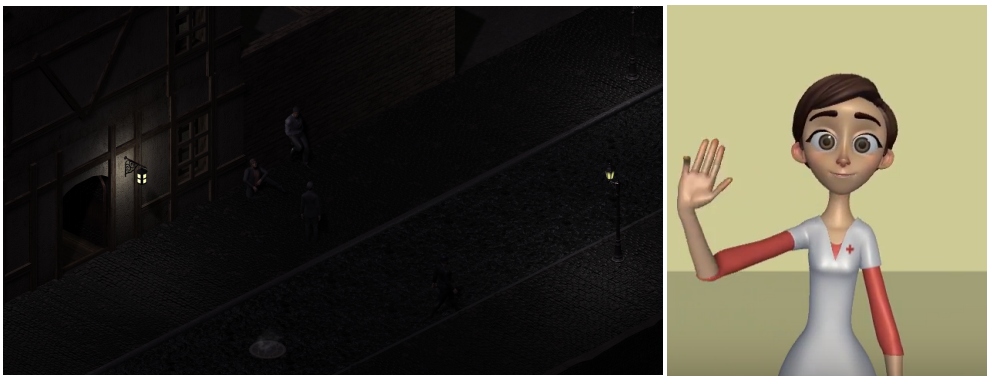
As explained before, the *NPC* framework supports the addition of custom modules which extend its capabilities. This is critical for any implementation, for these are very likely to have unique requirements which are not covered by the general services provided by the main controller. This modules can be added to any agent which contains the *NPCController* components. This controller will identify the added modules in the editor, and incorporate them to the main execution loop. Each module will be *ticked* every frame, or as frequent as the controller's update parameter specifies. This is of course, if the module is enabled or not. In addition, each module can choose to ignore the controller's update function, and work independently from it, still benefiting from all the aforementioned functionalities. This implementation utilizes many of these modules for natural language parsing, audio processing, *API* consumption and

behaviors modeling, among other aspects of the conversational agent. We proceed to briefly present the main interface which each custom component is to implement, with its basic definitions.

2.4.1 Interface

For a module to be recognized by the framework, it must implement the *INPCModule* interface. This interface, have the following contractual signatures:

- **InitializeModule**: To be called before the first frame is rendered in the simulation, but after all basic components of the driver framework have been initialized.
- **IsUpdateable**: Determines whether the module should be ticked on update or not.
- **TickModule**: Framework controller-driven function, which will be updated on general controller update cycle. Alternatively, each module can sub-class Unity's *MonoBehaviour* component, which allow for engine-driver update routines.
- **IsEnabled**: Determines whether a module should be initialized, updated, destroyed or cleaned up.
- **SetEnable** $\leftarrow$ *enabled* : *bool*: Set the *Enabled* value.
- **RemoveNPCModule**: Gracefully disassociates a module with the main controller on remove.
- **NPCModuleType : TYPE**: Meta-data module classifier to determine if this module is AI, Audio, Behaviors, and other predefined types.
- **NPCModuleTarget : TARGET**: Specifies if the module will affect an agent AI, Body, Perception components or other.
- **NPCModuleName**: Module display name.



(A) Isometric Adventure Game (B) Single Agent Simulation

FIGURE 2.9: Different simulations using *NPC Framework*

- **CleanupModule**: Called when a simulation terminates. Some capabilities, such as network connectivity, buffer allocation in memory and other services might require graceful termination.

As it can be perceived, the signatures definition are as general and simple as possible. There is only enough information for the controller to understand the type of module, its name, initialize, update, enable, remove and clean it up if needed.

Figures 2.9 displays two totally different simulations utilizing the *NPC* framework. Figure 2.9a, shows the framework used in an isometric action/adventure style game, while figure 2.9b uses the same implementation for a totally different single-agent simulation. Both implementations have two things in common; firstly, their foundation, lastly, that both customize the framework with especial modules. The game implements customized behaviors, game oriented AI logic and player IO controls. The second simulation, uses the aforementioned modules for behaviors, audio and API connectivity, among others.

Subsequent chapters, will discuss in detail the modules utilized to build up a conversational agent. These will extend Perception, Deliberation and Behavior capabilities of the agent.

Chapter 3

Perception

3.1 Multi-modal Sensing

In a general multi-agent simulation, the natively framework allows for an agent to perceive other characters, objects and behaviors in its natural field of view - modeled as a foveal cone of parameterizable angle and distance. The actions an agent can perceive are categorized as: (a) global: which impacts all background characters that see it, (b) targeted: meant for a specific character, and (c) neutral. For example, an author could mark a 'Warning Shot' as global, which when executed, will update every perceiving agent's hostility level of the source character. This is provided as input to the deliberation module to elicit appropriate responses in the characters.

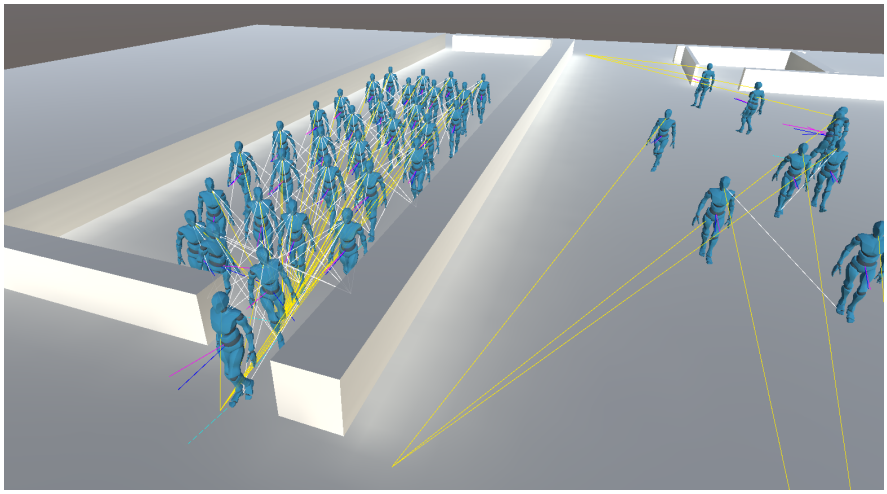


FIGURE 3.1: Multi Agent Perception

The perceived interactions are recorded in the agent's memory, then used in the deliberation phase to determine a reaction. The *conditions* that determine these responses are defined using a designable, and can also be tailored for unique agents,

using *modifiers*. Additionally, every agent will be aware of the *context* its is currently in if provided during design. Context is defined as a specific geographical place in the environment. The context can be used to enhance, reduce or neutralize behaviors' modifiers. As an example, an agent carrying a weapon in a combat training field would not increase perceived hostility levels on other characters. This should not hold true in a non-combat-controlled environment. This concepts are also available for design and are optional to every model, hence an author has no need to worry about the design complexity until is necessary.

Leveraging this capabilities and thanks to how the framework can be modularly extended, a conversational agent, as opposed to regular characters, perceives its environment in terms of audio feeds and imagery captured by the device's peripherals. Data is then forwarded for post-processing on an independent threads. The main processing modules we utilize are:

- Native Speech-to-Text solutions, defaulting to Google™ Cloud Speech API
- Affectiva™ Facial Expressions Recognition
- Nemesysco™ Voice Emotional Analysis
- Rasa Natural Language Understanding
- Proprietary Natural Language Parsing (NLP) System
- Proprietary Believe-Desire-Intention (BDI) Framework

Once all the data has been processed, the agent will either start, continue or interrupt a behavior. These are mostly conversational, as a response to the user's interaction. Input is collected continuously in the simulation loop, in parallel, consumer modules are constantly monitoring the messages queues, such that no signal message is left unprocessed on a first-come, first-serve manner. As an example, the agent can actively listen to the user as he or she speaks, interrupt, interject, express different emotions based on the message, respond with information or execute an action. Finally, a memory module keeps track of the agent's state and believes of the world. Memory

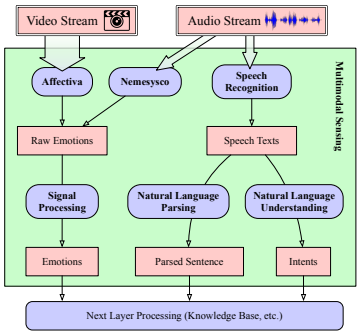


FIGURE 3.2: The pipeline of the multimodal sensing framework

is thus, a representation of the conversation flow actively consulted and updated on usage.

Figure 3.2 shows the overall multi-modal emotion and user intent sensing structure in our framework. This structure can detect and model emotions from various sources, parse speeches and analyze user intents. The output of the sensing unit will be passed to the next layer for higher level data processing, including knowledge base reasoning and the BDI model, etc.

Before being processed, signals by themselves are inherently meaningless. We will describe them all, starting with raw audio input, which is nothing but a simple sequence of sampled waves represented as bytes. We begin by discretizing said bytes into sequential atomic buffers. In this way, we ensure silence marks the beginning and the end of each potentially meaningful input unit.

Immediately after signals were processed, there are two main uses for the information created from this data. Initially, we parse and extract the meaning of said signals, this will be used to match already known concepts which the agent might be able to manage and respond to coherently. Secondly, we utilize this meaning to potentially fill up gaps and enrich the integrated agent's knowledge base. When the meaning has been extracted, a real-time system will make use of it and feed the behavior system. On the other hand, enriching and completing the knowledge base, although triggered right after each signal has been processed, is done on a separate thread of execution; this implies that the new concepts might not be available for the agent's next immediate response. We proceed to explain which and how signals are processed.

Once the buffer has been formed, we will attempt to produce the following products out of it: human natural speech, its semantics, and the characteristics of this in terms of its emotional load. This requires us to build a knowledge base and the corresponding representation and reasoning mechanisms.

The audio and video streams of the user are processed to construct a knowledge base which stores the constantly changing Sara's belief of the user, and the information obtained from the current (and past) conversations. The knowledge base can be then queried by the user by asking Sara questions, or can be leveraged by Sara directly in order to make a more informed response. Details of the implementation of signal processing, knowledge representation and reasoning are shown in Section *Knowledge*.

3.1.1 Active Listening

The aforementioned responsiveness in verbal communication corresponds to the most deliberate (and thereby the most delayed) means of communication from the agent to the user. In verbal communication, there is another type of responsiveness that is expected more frequently than the prior. What the previous responsiveness achieves is a delayed engagement. After the user is done speaking, the agent responds. A more active engagement requires responsiveness to the audio input as it is received. This means that while the user is speaking, the agent should show some indication that it is listening. This indication can take several forms, namely eye contact, nodding, and acknowledgements (e.g., "mm-hm"). To maintain eye contact, the agent should face the center of the camera by default. However, if there is additional information about the user's face through head pose estimation, the agent's gaze can be offset from the camera's center.

Active engagement and delayed engagement are mutually exclusive because they correspond to different phases in a conversation; i.e., the user's turn to speak and the agent's turn to speak respectively. The agent's behaviors are driven by these phases and changes between them. This means that the agent will always wait until its turn to speak. On the other hand, the user is free to disregard the phases and speak out of turn. The abrupt change in phase from the agent's turn to the user's turn will be

followed by the agent's active engagement, because the phase dictates that it should not be speaking, but listening. When conversations of this sort happen between humans, an abrupt change in phase is often accompanied by a short phrase to aid the transition; e.g., "Oh, sorry" or "Please continue." Our agent employs this behavior in order to raise the user's awareness of the phases, because contemporary virtual assistants have made people accustomed to virtual assistants' insensibility to conventions of conversation.

3.1.2 Emotions in Facial Expressions

Facial movements can be measured and described by FACS. Based on the data, facial expressions can be further analyzed and categorized into low-dimensional emotion classes. Our framework uses Affectiva^{TM} as the main source of the raw facial emotion data. Affectiva^{TM} can provide raw frame-based 8-dimension emotion data from the camera video stream in real time. As a drawback, it doesn't contain any temporal information about the emotions, which may fail under some situations when the muscles on the user's face move frequently and cause noises. It hinders accurate emotion analysis when the user is speaking, which is quite a common case in our implementation. Therefore, further signal processing is necessary to minimize the impact of the emotion noises.

3.1.3 Emotions in Voices

Although not as explicit as facial expression, people obviously show emotions in their voices. Voices provide two types of raw signals related with emotions: (i) sentiments based on the text, and (ii) tone and speed information based on the sound wave analysis. With semantics analysis, the former signal provides more figurative information about the emotional state of the user, while the latter signal provides a more abstract statistical conclusion on the sound of the voice, regardless of the actual words being spoken.

In our project, the sound data captured with the microphone is fed into multiple

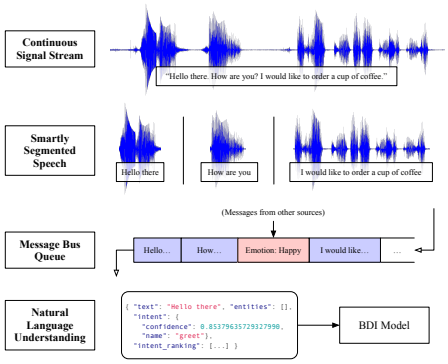


FIGURE 3.3: User intent classification procedure

processing submodules. The raw data is sent to different processing branches in parallel. One of them is our Motional server with Nemesysco^{TM} running for voice sound wave analysis, and the other ones are speech-to-text services based on the OS platform. When the speech-to-text result is returned, it will be further used as the inputs for sentiment analysis, knowledge base query, and intent detection. Nemesysco^{TM} can extract information related to emotions, but the result space is distinct from the traditional 8-dimension facial emotion information. To correlate the data, we adapted a mapping algorithm to merge the data from different emotion sources and remap them into a standard emotion space with 3 dimensions, which is an analogy to the HSV color space.

3.1.4 User Intent Detection

Voices also deliver more information than emotions, such as intents. Besides natural language parsing, we are beginning to utilize *Rasa* natural language understanding to extract user intents from the text recognized from speech. Depending on the scenarios where the agent is to be implemented, we trained different models accordingly.

In this implementation, the model is trained with a variety of user intents. The following list shows part of the intent labels that are related to the multi-modal emotion analysis and the BDI model.

- `greet` and `goodbye` show that the user wants to start and end the conversation, respectively.

- `affirmative` and `negative` show the user's acknowledgement and disagreement to the previous response from Sara.
- `state_mood` delivers the current mood of the user to Sara;
- `query_mood` is for when the user tries to ask for the current mood state of Sara;
- `desire` shows that the user want to request something.

Every spoken sentence of the user can be categorized into one of the intents. Moreover, we also train the model with related entities, and during runtime, Sara extracts them if there exist any in the sentence.

Figure 3.3 shows the procedure of the user intents classification. The voice sound signals are initially cut into pieces containing each spoken sentence. Meanwhile, each sentence is converted into text through platform-dependent speech recognition services. The returned texts are then enqueued into the message bus and waiting to be fed into NLU services. To optimize user experience, latencies are not expected. We minimized the latency between "end of the sentence in the speech" and "user intent classification result returned" to be less than 300 milliseconds, which includes two Internet communication sessions plus all the data processing time.

However, a naive NLU intent classification module doesn't fit every case in practice. For example, Rasa classifies intents without referring to the actual entities extracted from the sentence, and hence two sentences with a similar semantic structure but totally distinct entities (which implies different intents) might be incorrectly classified to the same intent. To resolve this type of conflicts between intents and entities, we implemented a post-processing algorithm to further categorize the intents in a higher level and hence improve the robustness of the NLU module.

3.1.5 Emotion-Certainty-Intensity Emotion Classification Model

ECI (Emotion, Certainty, Intensity) is a representation of the Lövheim cube of emotion, which was inspired by the HSV color model. The representation is a concise, unified alternative to emotion data, which in our framework, was processed and received in

the form of a 6-tuple corresponding to the probabilities of Ekman's six basic emotions: anger, disgust, fear, happiness, sadness, and surprise. In Lövheim's model, emotions are a blend of signals from three neurotransmitters: serotonin, dopamine, and noradrenaline. These signals are represented as orthogonal axes in three-dimensional space, making this model a precise analog to the RGB color model. Therefore, the equations for RGB to HSV conversions are applicable to the conversion of the Lövheim cube into the ECI model. In the ECI model, Emotion (the analogue to Hue) changes based on the proportions of the neurotransmitter signals. In increasing degree order, these emotions are disgust (with the highest proportion of serotonin), surprise, sadness (with the highest proportion of noradrenaline), anger, fear (with the highest proportion of dopamine), and happiness. Certainty (the analogue to Saturation) is a measure of how distinct the emotion is from general excitement, and Intensity (the analogue to Value) measures the intensity of the emotion. The ECI representation is an intuitive alternative to the Lövheim cube of emotion.

In order to convert emotion data from Affectiva^{TM} and Nemesysco^{TM} to ECI, we took the emotions with the highest likelihoods, normalized their probabilities, and multiplied them to vectors that corresponded to the points of maximum intensity for each emotion. The average of these vectors is the point in the Lövheim cube of emotion that we used for conversion into ECI.

Chapter 4

Deliberation

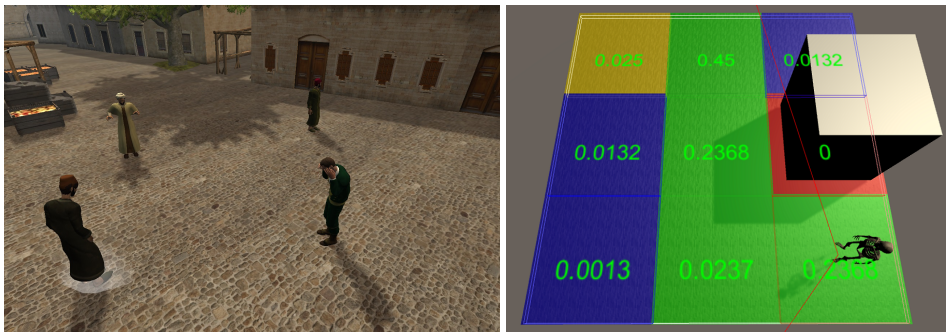
4.1 General Framework Capabilities

Let us first explain the generic capabilities offered by the framework for basic event and effect handling within traditional simulations. For a single agent, all the currently perceived entities, and their states, will be taken into consideration for selecting the next group of candidate events to be executed. The first step in this process is to filter all candidate events based on their conditions and modifiers. Conditions are evaluated against the state of the current agent, or the perceived character's state, and returns `TRUE` or `FALSE`. For example, let assume there exists an event 'GreetMainCharacter' with the following conditions: (1) `Target.IsForegroundPlayer`, (2) `Target.Friendliness` >0.5, (3) `Self.Idle`.

This mechanic can be used by designing a set of conditions and events, given the available affordances from agents and their traits. These *conditions* are not bound to any specific event until the author assigns them to it. At code level, a condition is an *Attribute* which get associated with either a function or a property, for then to be evaluated against a target value during runtime. Once a condition has been instantiated, it can be reused in any defined event, hence the overhead of these are not significant as the number of available Events increase. Assuming the agent's current state satisfies all needed conditions, then 'GreetMainCharacter' becomes available as a candidate event. Now, assume there is also another event which has been satisfied, for instance, 'Wander'. This, will also be included in the candidate Events queue. At the time of taking a decision which event is to be executed, we must consider mainly four things: (1) Is this agent currently selected by the User? (2) Is the character currently executing an event? (3) Is the first candidate event of higher priority of the currently executed

event? (4) Is the final chosen event the same one is being executed? Being a foreground character or taking part in a higher/equal priority than the top candidate one, can prohibit an agent to start executing a new event. Should two or more events share the same priority, the one with the higher number of constraints will be executed. Furthermore, if two or more share also the same number of constraints, then one will be picked randomly. Algorithm 4 is the highest-abstraction level routine for the event-deliberation system, to select potential candidates who satisfy its required conditions, based on their state. Every event may include modifiers that can persistently or non-persistently modify how its execution perceived by other agents. This enables a more flexible design which allows the author to further decide how a behavior will impact other agent's perceptive memory.

Additionally, the framework integrates modules for path finding, steering with built in social forces, and perceptive memory, among others which will be covered at the end of this section.



(A) Perceptive Memory (B) Path Resolution

Data: Predefined lexicon of Events $\mathbf{e} \in \mathbf{E}$

Data: Smart Objects $\mathbf{s} \in \mathbf{S}$

Data: Evaluating agent a_c

Output: Queue of valid events, $\mathbf{E}_v$

$\mathbf{E}_v \leftarrow \emptyset$;

$e_t \leftarrow \emptyset$;

foreach $\mathbf{e} \in \mathbf{E}$ **do**

  $\mathbf{P} \leftarrow$ `GetValidEventParticipants`$(\mathbf{S})$;

  foreach $\mathbf{p} \in \mathbf{P}$ **do**

    if $\mathbf{C}(\mathbf{e_p}) = \mathit{true}$ **then**

      if $\mathbf{E}_v = \emptyset$ **then**
        $\mathbf{E}_v = \mathbf{E}_v \cup \mathbf{e_p}$

      else

        if $\mathbf{p}(\mathbf{e_p}) > \mathbf{p}(\mathbf{top}(\mathbf{E}_v)$ **then**
          $\mathbf{E}_v = \mathbf{E}_v \cup \mathbf{e_p}$

        else if $\mathbf{p}(\mathbf{e_p}) = \mathbf{top}(\mathbf{E}_v)$ **then**

          $\mathbf{E}_v = \mathbf{E}_v \cup \mathbf{ep}$;

          if `ResolvePriority`$(\mathbf{e_p}, \mathbf{top}(\mathbf{E}_v)) = \mathit{true}$ **then**
            $\mathbf{E}_v = \mathbf{E}_v \cup \mathbf{e_p}$

        end

        else
          `InsertInOrder`$(\mathbf{e_p}, \mathbf{E}_v)$

        end

      end

  end

end

Algorithm 4: Event Deliberation Process

Act. Finally, the events that are chosen by the deliberation phase are used to invoke appropriate reactions in the background characters, implemented as PBT's.

4.2 Understanding Signals

Although this implementation does not utilize the previously explained mechanic offered by the framework, the deliberation process in this single-agent conversational simulation is still resolved within a domain of constraints, viable responses and adequate behaviors which are of course continuously influenced by the user's input. Responsiveness is a critical aspect of human communication. When a person is communicating through words, tone, facial expressions, etc., he or she expects to receive a prompt response from an engaged participant in the conversation. For each independent mode of communication, our framework prioritizes immediate feedback to provide this engagement to the user. To achieve the closest possible level of real-time responsiveness, we divide the processing engine in two blocks. The first part is on the client application, the second one, resides on a remote server not only capable of processing signals, but also capture data, which could eventually be used for analytical and learning purposes. Starting on the device, basic I/O peripherals will capture in real-time video and voice. Facial recognition is processed on the device almost seamlessly. As mentioned before, *Affectiva* utilizes a system of 40 points to capture the current user's expression, which allows us to determine certain possible expressions. The second client-side processing mechanism is *YieldProlog*, a library also installed with the application which translates regular *Prolog* constructs into executable *C-sharp* code, compatible with the application's base language. Finally we jump onto the last portion, which is emotional voice analysis. This is not processed on the device. *Nemesysco* receives raw buffers of audio, which were indeed processed to 8-bit mono signals in the device, on a remote machine. These machine maintains a persistent TCP connection with the device, and once a buffer has been completed and a signal successfully analyzed, a call-back function receives with a 2 to 3 seconds delay an analysis which represents the probable emotional state of the users based on his voice levels. This feature is still being tested, with a success rate of only 40 percent. The software was originally developed for telephone communication of clients to call centers, hence we are adapting a tool designed for long-lasting communications, and

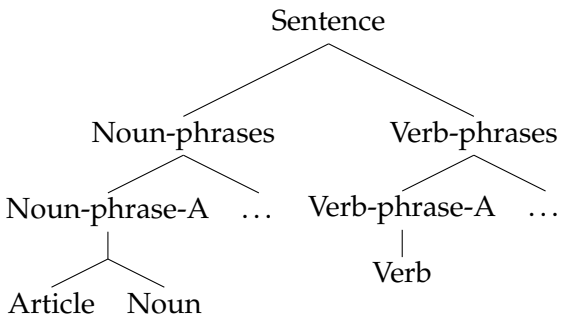


FIGURE 4.2: A parsing tree with different levels of grammar

fine-tweaking it to analyze short bursts of voice audio buffers. We implemented a noise and feedback control logical gate into the application, still, obtaining accurate results still proves elusive in most situations. In this sections we will explain how the previously discussed signals are comprehended by the agent and which information is taken into consideration at the time of deliberating a coherent response.

4.3 Natural Language Processing

The knowledge of a conversational agent is formed by two steps: first "consuming" natural language and then building the extracted concepts.

Natural speech, consist of obtaining the speech from the signal, and more importantly, understanding the semantics of said speech. We try to take advantage of each Operating System's native capabilities when available for NLP. For example, iOS, Android, and Windows offer extremely robust solutions to this well-explored problem. When unavailable, or on fail, we default to *Google Cloud Speech API*, which although very mature, the network overhead for the service call is significantly greater than the one from its native counterparts. Once speech has been extracted from the audio buffer, we feed the speech utterance into the parsing engine. This will be parsed with a syntax tree approach. We use this method to primarily identify and extract the following elements: (1) proper sentence formation, (2) subject, (2) verb, (3) subject characteristics, (4), simple expressions, and (5) wh- questions.

To achieve this analysis, the agent counts with a predetermined context-sensitive

$$
\begin{aligned}
S &\rightarrow NP|VP \\
S &\rightarrow AP|NP|VP \\
S &\rightarrow NP|VP|NP \\
S &\rightarrow \ldots \\
NP &\rightarrow article|noun \\
NP &\rightarrow noun \\
NP &\rightarrow \ldots \\
VP &\rightarrow verb \\
VP &\rightarrow \ldots \\
AP &\rightarrow determiner|adjective \\
AP &\rightarrow \ldots
\end{aligned}
$$

FIGURE 4.3: Sample Context Sensitive Grammar

grammar capable of identifying structure and content within the sentence. The grammar is used to conduct a tree-like analysis throughout the sentence. As its parts are identified, the content enumerated above is extracted to be returned for further processing. Once a sentences parts have been identified, we proceed to the knowledge base provides two main services: a repository of meta information for speech recognition, and a conceptual network of concepts. Figure 4.2 is a simple example of a possible parsing tree, derived from its grammar definition.

The grammar is of the form shown in figure 4.3.

Capturing the elements semantic elements of the speech occurs in parallel to the sentences being analyzed for proper formation. A properly formed sentence will follow certain rules of the spoken language, allowing the system to populate a collection with meta information of the utterance and the conceptual meaning of it. Firstly, the obtained meta-data is essential to set up the context of the speech. This information is primarily used for validating the rest of the components in the sentence during analysis. Some examples, are the tense and the number. Secondly, we extract the components we previously mentioned: subjects, actions, modifiers, expressions, and interjections, among others.

4.4 Knowledge-base Design and the Conceptual Network

The knowledge base the agent is compromised of rules, as explained above, a network of facts, dynamic assumptions and a highly granular classification of terms which allows her to understand certain constructs of the human speech. In addition, every time speech is received, this is analyzed for new terminology, and if a gap is detected, a recursive system will start querying a different online data source to include this permanently in Sara's knowledge base. We proceed to explain the conceptual network. Finally, Sara's state of beliefs, or dynamic assumptions, is dynamically updated based on the user's emotions and responses.

Initially, when referring to concepts non related to speech, we distinguish between root elements, *part of* relationships and entities. Root elements are those which we identify as top-of-the-tree classification nodes. For example: What is a cat? a cat is an animal, more particularly, a feline, and a mammal. But what is a mammal? A mammal is indeed an animal, so we could say that a cat, is a feline, which is a mammal, which is an animal. In this simple example, we identify the animal as the root of the conceptual network. But so far, we only count on a *path* rather than a tree. Now, if we include a lion, snow leopard, a lynx and the concepts of wild and domestic, we could form a graph of relationships as is shown in figure 4.4.

We can see how this forms what initially would look like a tree. However, a tree does not offer the flexibility that a graph, or network, would. Where would be placed canines and dogs? Fortunately, we do not need to worry about duplication, since we could easily add a new node, under Mammal, which would then derived in Domestic, and append it to its children, along with the cat. But it is worth noting that following a tree search would not work in our case because if we wouldn't keep internal references to parent nodes, we wouldn't be able to distinguish between domestic felines and canines. Next, let's see how this information is useful.

We can now query our knowledge base to ask: what other animals, other than dogs can be domesticated? Starting from the dog, we go one level up to Domestic, and query all its children. From there, the answer: cats. Switching gears on our example,

we could very much apply the same network to concepts such as pistons, and ask: What is a piston? It would be easy to answer that a piston is a part of a motor which is a part of a vehicle, not necessarily a car. This distinction was made when after querying for the piston, we determined that not only cars have motors, but many other means of transportation have them as well.

Finally, another advantage of this conceptual graph is the ability to share properties, hierarchically, among items from same categories. We could tag "Domestic": "Human-friendly". Now we span the property along the class, and when querying: which animals are human-friendly? the result would be the set of Domestic animals.

Thanks to the built-in processing rules, we can identify which parts of the utterance are nouns, and potentially identify those which Sara doesn't know. Once a gap is discovered, a thread independent process is triggered to go fill this gap. There are two main sources being checked, at the time of this paper. The first one is Google Cloud NLP, for entities. The second one, is Microsoft Concept Graph research base. Initially, we interpolate the definitions provided by both knowledge bases. Google's NLP will possibly identify the entity, and so will Microsoft. The big difference between these two, is the probabilistic factor associated with Microsoft definitions. Each concept is marked with a probability. Based on this, we will select the top k results, and keep querying until we reach a point where we hit a root entity, of the k iterations where completed. Once the data has been fully gathered, the server-side end of the implementation, will insert and update the conceptual graph, for then, write it back to the client-side, the agent, for future use.

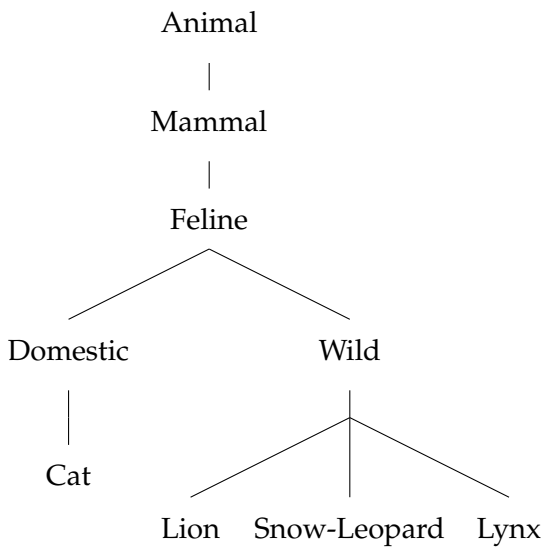


FIGURE 4.4: The tree structure of relationships

4.5 Planning

As an example of the framework's modularity capabilities, a partial-order planner module (*NPCPlanner*) was introduced to the main framework's assets for creating behaviors, conditioned actions and their goals and ultimately compute a plan to achieve a specific action. This is particularly useful when in need for creating more compelling and non-deterministic behaviors which could be found in an predefined actions search space. One of the main advantages of allowing for dynamic partial-order planning of behaviors is scalability. Provided a set of actions with associated goals and conditions, the planner has a larger ground of options when dynamically creating behaviors in real time only limited by the available actions. Therefore, increasing responses based on goals can be simply achieve by designing and creating new action nodes assets without the need of defining static behavior trees or massive interactive ones.

Lets assume we are implementing basic locomotion goals such as go to a given location. This can be easily achieve by implementing a single *Go/ To* affordance. Now lets assume we would like to set some constraints on this affordance, such as that the agent cannot be sitting down before executing the aforementioned action. Additionally, we create yet another affordance for either standing up or sitting down. This

affordance has the condition that the agent must not be navigating while executing it. Just by defining these two behaviors with simple conditions, a goal can be fed into the planner, such as go to a specific location, and this, will attempt to generate a tree which given the current agent state, will achieve said goal. The fact that the plan is partially ordered, means that some actions might not necessarily need to be executed in a specific order at some point in the plan for the goal to be achieve, which allows for a trail and error tree with multiple branches on which only one needs to succeed. For this particular trivial example, the plan will result in two execution paths:

1. Go To Location
2. Stand Up -> Go To Location

Assume we simply generated these two branches under a *Selector* tree. The first one executes, but because our hypothetical agent is sitting down, fails. Then we can execute the second branch, which will indeed satisfy our goal. Finally, because a partial plan will be computed, there might be execution paths which contain actions which can be executed utilizing the same mechanism, a selector, hence only one needs to succeed for the plan to potentially achieve completion.

Initially the mentioned planner could be added as a component of some individual agent, hence each plan will take into consideration, and as parameters for its actions, that particular agent's state, such as locomotion mode and target entities if interacting with something. Depending on the type of affordance, these are assigned dynamically and ultimately completed for execution in realtime by topologically sorting a partial plan.

More complex scenario could arise for interactive conversational agents, on which most of their executions are likely targeted to coherent responses given a user's input. This implementation is yet to include the planner for a conversational agent, nevertheless, the required actions could be designed, conditioned and added to the actions search space in the future. So far, the planner has been used to resolve locomotion and "search and find" goals in other projects which utilize the framework.

4.5.1 Planner Implementation

The planner utilizes the following structures:

- Causal Link: Triple ordered relationship between two action elements and a condition, of the form: <action A, C, action B> on which action A satisfies a condition C for action B to be executed.
- Constraint: An ordered pair of actions, A and B, which represents the fact that action A must be always executed before action B.
- Action Node: The node data structure which wraps an affordance, a list of conditions and a goal. If the given conditions are satisfied, the affordance can be executed to achieve the desired state expressed by its goal.
- Condition: Structure which represents a desired state of locomotion or traits for agents, or interaction state for objects.
- Goal: Also referred as post-conditions, this is the type of entity, agent or object, and ta

This structures are implemented within the class which also implements the *INPC-Module* interface, which highlights the fact that any type of object can indeed be utilize as a module. Then the following functions will participate in computing the final plan, if found:

- Public
  - GeneratePlan <- Goal Action
- Private
  - AddConstraint <- Action A, Action B : Adds a new constraint to the set if two given Actions, A and B are not already there or do not contradict an existing constraint.
  - GetSatisfyingAction <- Condition : Finds and Action associated with a given condition.

- FindSatisfyingAction <- Condition : Finds an Action which can satisfy a given condition in the action search space.
- IsSatisfied <- Condition : Checks is a given condition has already been satisfied in a Causal Link or by the agent's current state.
- OrderConstraints <- Causal Link, Action : Topologically sorts a graph of constraints represented as a list of linked lists of constraints.

Finally, the GeneratePlan algorithm, as seen in Chandra, n.d., generates a plan starts by receiving a desired action goal which the agent should execute. This action, or tree with actions can potentially be constrained with conditions by designed. Upon finding said conditions, these are added to an open set which must be fully satisfied for a plan to be generated successfully. The algorithm declares empty sets of actions, constraints and causal links. There two initial actions which will create a 0-state constraint. The meaning is that the current state - represented as an action - must precede the goal action. Finally, the algorithm will execute a loop which dequeues the next condition to be satisfied. Whether this has been fulfilled or not, an action which satisfies that condition will be yield, and during the current iteration, the logic could potentially create new constraints, causal links and will always organize the plan to ensure no action is violating an already created constraint. Should a plan is found, this will be returned in the form of a Behavior tree on which ordered actions are executed in *Sequence* control nodes, and partially-ordered ones are executed as children of a *Selector*, which will guarantee that if a partially ordered action was successfully executed, the next one will be omitted for the plan to move on during play.

Chapter 5

Action

5.1 Behaviors Overview

Almost every interactive commercial games, academic or professional simulations, will most likely have a primary artificial intelligence component. This module usually not only manages characters, but also game mechanics, decision making, narrative guidance, and many other components which vary based on the nature for each simulation. When it comes to create believable, rich, immersive and engaging interactive or scripted simulations and games, agents behaviors are paramount. How immersive the experience will be, is absolutely proportional to the level of effort, design placed on the characters behaviors and how these are ultimately executed.

FIGURE 5.1: Witcher 3 City

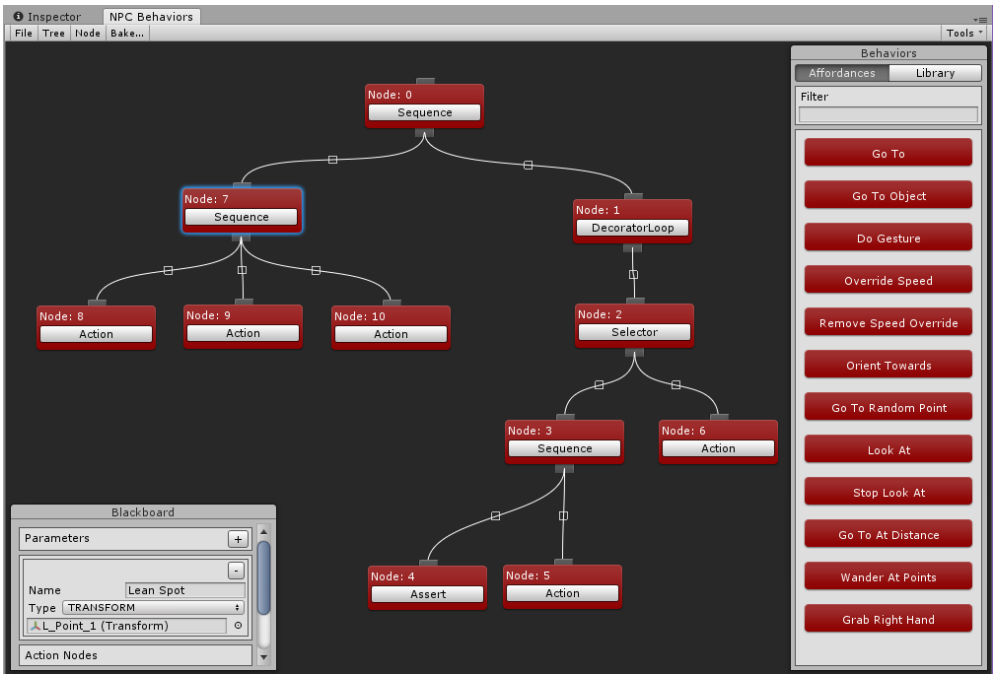


FIGURE 5.2: NPC Framework Behaviors Editor

There are many ways developers can implement these intelligence modules. Originally, agents behaviors were nothing but a set of stored patterns, which would usually have an effect on movement and combat. Nowadays, we can find vibrant and lively cities scattered throughout games. These are not only busy, full of simulated life, but also coherent. They have generic crowds, specialty groups and individual key characters which ultimately, will shape and define how immersive each scenario is. Games like *The Witcher* 5.1, *Assassin's Creed* and many others, have exploited and mastered techniques which would allow their developers to successfully create these scenarios for players to truly feel inside the simulation. Furthermore, systems within engines are just developed to allow designers to rapidly build these models, avoid bottlenecks in the development cycle and to still be able to produce a high quality product.

When it comes to single-agent interaction applications, such as this one, this mechanic is of utmost importance from the user's perspective. The intention is to have a system which is not only responsive computationally, but also engaging, diverse and make a user feel like if involved in almost in a real interaction. In this implementation, we have a developed a quick design system which allows us easily create, test and adapt

new behaviors to our agent. This component is part of the *NPC Framework*. Any agent which utilizes the *NPC Controller* can benefit of using this behaviors component. This editor is nothing but an interface to the actual implementation of all the nodes and controllers which ultimately will execute the tree at runtime.
In this chapter, we introduced the basic structure and definition of behavior trees, advantages, limitations and how to implement a system which hierarchically executes this structures.

5.1.1 Behavior Trees

We will start by defining behavior trees. These are hierarchical structures of nodes, represented as *n-ary* trees, which as opposed to finite state machines, they offer control capabilities within its subtrees. Their leaf-nodes are ultimately the structures which encapsulates agents's *affordances* behavior-oriented executions. Inner nodes, are called *control nodes*, and will define how their substructure (subtree under that node) will be updated, or *ticked*. We will start by listing the three main type of control nodes:

- Sequences: Will execute all of its children sequentially and succeed if and only if all of its children do.
- Selectors: Will execute all of its children sequentially, and succeed if at least one was successfully executed
- Decorators: Will handle the reported status from a subtree

Sequence and *Selectors* nodes can be viewed as conjunctive and disjunctive logical structures, respectively. Behavior trees are executed in in-order traversal, and on each iteration, each node is updated in order to inquiry the current state of the tree. A *Sequence* node subtree will only succeed if all of its children executed successfully. Selectors will execute every child until at least one returns success. Finally, *Decorators* are used to manipulate the outcome of a subtree based on its status at completion. These are generally placed on top of the a subtree root such as a *Sequence*. These statuses can be:

- Success: Overall, the tree executed with no errors and the expected behavior was properly ran.
- Running: The tree is still running some leaf node.
- Failure: The expected behavior couldn't be executed for some reason.

When updated, the tree will ultimately return one of these three possible states. A *successful* root node, implies that the entire tree has been completely executed, *running* means that some state is yet to terminate, still executing, and *failure* is of course, the opposite of a successful execution. Now, to expand a tree's complexity of execution, each of the aforementioned categories can be further expanded into different types. All of the three main categories presented before can be implemented in different ways, to allow richer flexibility and more complex behaviors. Some of these are:

- Sequence/Selector Parallel: All children execute simultaneously, for selectors, once all finished, if at least one was successful, the tree is successful, for sequences, if at least one child failed, the entire tree failed.
- Sequence/Selector Raced: All subtrees are updated simultaneously, the result for the tree will be the result of the first node which finished executing.
- Sequence/Selector Chance: Potentially executes a child node with probability *1-p*.
- Decorator Loop: Repeats a subtree as long as it successfully executes. One can think this as a *while(success)* statement.
- Decorator Negate: Flips the result of a tree and reports the opposite to its parent node.
- Decorator Force: Regardless of the actual result of its subtree, this decorator will always yield a specific result.

From the implementation stand point, all of these structures can inherit many basic functionalities from a parent abstract Node class, which means that many different types of controls nodes can potentially be created as long as these subclass said class.

In terms of the leaves, the following are those which are regularly used:

- Action: Executes an agent affordance
- Assertion: Evaluates a specific condition on the actual agent or perceived entities
- Wait: Simply holds the tree execution returning *RUNNING* status for a given period of time

Figure 5.3 is a simple behavior tree which utilizes many of the aforementioned components. This tree consist of a *Decorator Loop* which will execute a *Selector* permanently. This *Selector* will attempt to perform two tasks, the first one is an *Action* and the second one is a *Wait* leaf. The *Action* leaf runs an affordance which will make the agent select a random point in the map, and walk to it. Once completed, the agent will wait for a period of time, and then, the tree will start executing again thanks to the decorator. It is important to note that if the action node, for some fails, then the wait node will trigger and most certainly succeed, yielding to an overall successful execution of the tree - this is why we are in this case utilizing a *Selector* over a *Sequence* node, due to the unpredictability of the choices from the first node.

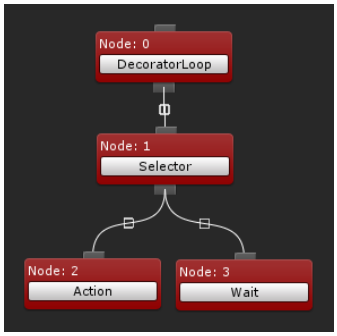


FIGURE 5.3: Basic Behavior Tree

Parameterized Behavior Trees

The next aspect to be discussed is how we can parameterize behavior trees. This will determine which data will be available for each leaf node to utilize on execution. For example, a most basic case would be how many milliseconds will a *Wait* node will hold the trees execution. Which type of parameters and how these will be determined

highly depend on how we want to use them, and whether we need global, local, dynamic or static parameterization. For example, a simple "Wandering Loop", as the one presented on figure 5.3, might not need to dynamically determine the available set of points available as destinations, but instead, have these fed to it statically on the design phase, and then utilize simple logic to randomize the next point in the array. This is not only simpler to implement, but also more computationally efficient. This is very important to keep in mind when designing behavior trees.

The traditional way to parameterize a behavior tree, is by utilizing a single data structure called a *Blackboard*. This consists of a global map of key-value pairs shared among all nodes, for these to retrieve specific values when needed. A big disadvantage of this approach, is the fact that if many nodes happen to be accessing these parameters simultaneously, a race condition might arise, turning the value in a volatile one. In addition, a single blackboard does not scale nicely. It becomes really hard really fast to keep track of all needed values as the simulation or game grows in size, and specially, complexity. Next, if the designer opts for limiting the number of parameters trees will share, the complexity will also be affected, so basically there is little balance between scalability and complexity resulting in compromised quality and flexibility.

A second approach, as exposed by all, n.d. is to feed those parameters which the tree will consume from the root of the tree, for them to cascade down as the tree executes. This approach removes the factor that each tree depends on the same data structure and all parameters are shared decentralizing them from a single point of dependency. This is an improvement from the first option since it completely removes the chance of a race condition to occur over certain parameters. However, we can point the following disadvantages: lack of flexibility to determine certain values during runtime, trees must be statically created which means that if two or more trees are concatenated, certain parameters might not be appropriate for other instances. Lastly, we would still need logic to determine how to instantiate the parameters which will be passed to the root of the primary tree, leaving no room for elasticizing values during runtime. Nevertheless, this model is extremely powerful and heavily used throughout several applications and modern games to control agents AI, characters' responses

and other components of the simulation.

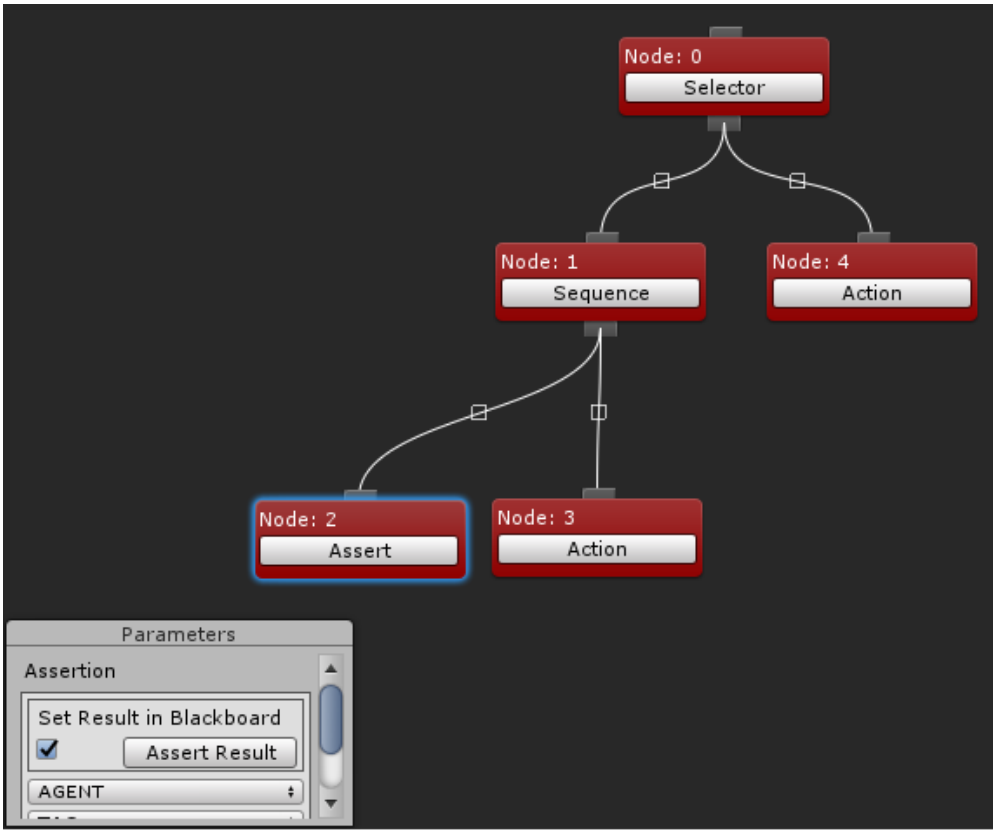


FIGURE 5.4: Look At If Tree

A third mechanism to parameterize trees, and the one designed for this implementation, is a mix between the two former and latter options. We utilize smaller atomic trees, parameterized by their own blackboard instances (not shared), with the added capability of resolving values dynamically in a two-way data binding system. Figure 5.4 is a simple example of how a value can be determined at runtime while asserting a condition. If a value satisfies said condition, that object will be passed up to the subtrees' *Blackboard*, for the next node to immediately consume.

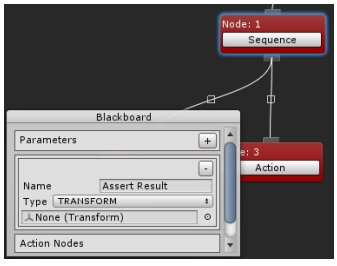


FIGURE 5.5: Subtree's Blackboard

This allows for different trees to be joined without the need of validating parameters for each execution unity will have its own defined static or dynamic subset. It also increases a tree flexibility for values are not needed to be known before hand so any assertion can be satisfied and a tree executed at any point and by many sources. To implement such a mechanism, we will cascade parameters from a Blackboard attached to a root node, say for example a Sequence, down to its children. Should another blackboard is attached down the down, we will skip these, to allow the lower Blackboard now to cascade its parameters to its children nodes. Another important fact, is that two sibling trees would not share a blackboard, even if their leaves are at the same level. If needed to, a new master Blackboard is to be attached to the root of both tree, remove the previous lower ones, and add all the parameters which would satisfy all action nodes. Finally, in this implementation, parameters are validated during design (editor) time, such that a parameter which does not correspond to the required type for the leaf, wouldn't be an option for selection. For example, if an action node, such as *Go To Point* requires a boolean and vector values, a *transform* parameter being cascade from the upper Blackboard will not be an option for this leaf.

Interactive Behavior Trees

This sections will explain how to expand a simple behavior tree into a more complex structure which would allow simulated agents to not just follow a single behavior pattern, but to be able to adapt to different situations, specially when interacting with a human controlled agent, or like in this case, the actual user through peripherals. So far we have explained the basic controls which would allow to form single-execution-path trees. These trees, although potentially complex, are usually not flexible enough to adapt to different situations. They are most likely execute single patterns over and over, which ultimately might result detrimental to the simulation immersive factor and increase response predictability. There are many ways we can enrich these trees and tackle the linearity problem. One such way is by implementing *interactive* behavior trees. These are a type of behavior tree which on every update, execute evaluation functions to determine if the current subtree must be executed or another branch

should take precedence.

Just with the simple set of resources covered thus far, we could create a tree which would adapt to different circumstances. These will be the base of our interactive agent. However, we will start by looking at this from a more generic and familiar example, such as a regular NPC which would be performing a generic task, then look at a specific agent once approached by it. Finally, if some agent in the scene turns hostile, the character will flee. A most common scenario in open world games. This same technique can be used to handle bigger aspects of a simulation such as story arcs, triggered events in games and of course, interactive agents. Before proceeding with an example of such implementation, it is worth noting that these approach is just but a tool in a wide array of resources which can potentially enrich agents' behaviors. Another simple but effective way to do so, is to, as mentioned before, extend the abstract implementation of the *Node* class to add chance and randomizer nodes to our repertoire.

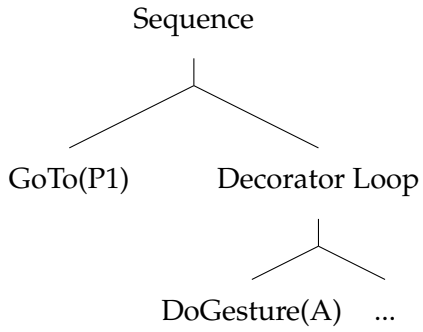


FIGURE 5.6: Simple Tree - **T1**

Tree 5.6 is a basic structure which will make an agent go to a position *P1*, and then repeat a set of actions over and over. This won't do much for a simulation but to create a static - non-interactive - NPC doing some trivial background behaviors. We can easily append another tree to it to enhance this behavior.

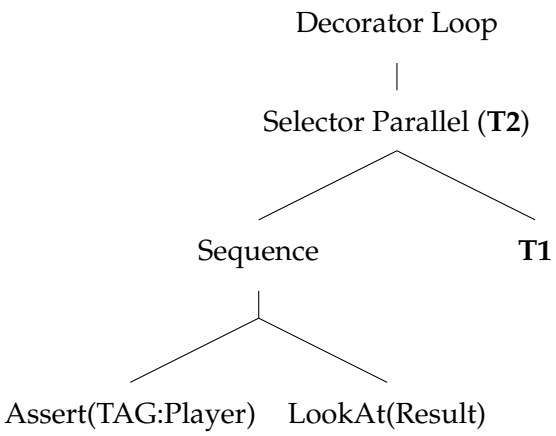


FIGURE 5.7: Compound Tree - **T2**

The second tree 5.7, wraps the first simple tree 5.6 and creates a behavior which will run continuously, and most importantly, execute two subtrees simultaneously. This will result on the agent to execute the actions in **T1**, while at the same time, constantly attempting to run the leftmost subtree under the *Selector Parallel* control node. it is very important to utilize a Selector in this case, because it is not guaranteed that an agent with the "Player" tag will be around all the time, therefore, most of the time, the subtree under the *Sequence* portion will fail. By using a *Selector*, the tree will move past this failed first attempt and execute its second subtree. A very important fact to note is that because this is a *Parallel* node, even after a subtree returns failure, this will be reseted and the root node will attempt to run it again. This is an example of the simplest of interactions between an NPC agent, and a user or player character. Although not fully interactive, just this subtle change, increases the level of immersion users might experience.

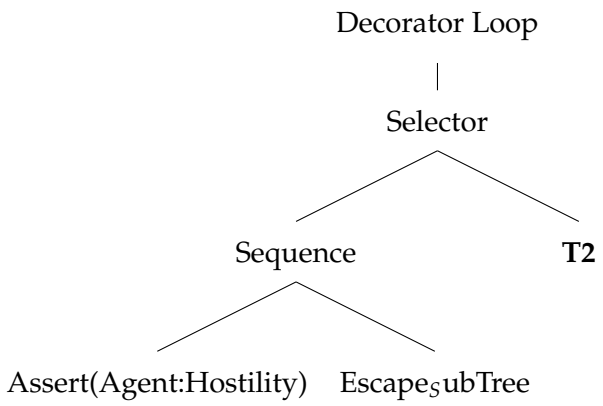


FIGURE 5.8: Compound Tree 3 - **T3**

Finally, tree 5.8 exploits the same technique used before, becoming yet another wrapper, which captures *T2*, also encapsulating *T1*. In this case, we are not using a *Parallel* type of control, since we do not want the second subtree, *T2*, to execute if the first one succeeded. A simple *Selector* will do. Also notice how we utilize the *Sequence* control node to create atomic actions, which will either execute, or not at all. As opposed to *Selectors*, on which a the control can succeeded if at least one of its subtree did, a *Sequence* will fail in all cases with exception of the one on which all its children have succeeded.

5.2 Behavior Design

Conversational agents need to be fully responsive, coherent and interactive. Its level of responsiveness will highly impact the user's engagement and immersive experience. This aspect is proportional to how capable our agent is to process input signals and semantically analyze these on the fly. It is for this reason that when designing a main interactive behavior tree, it is essential to favor breadth over height on our main tree. That is to say, we should always think in utilizing as many parallel processes as possible for our main execution tree. Nevertheless, it is also important not to neglect depth, for this will have a direct impact in the second aspect, coherency. For example, assume we have a parallel control node which executes two simple routines: voice commands parsing, and greeting the user. Commands must be parsed immediately, therefore

taking precedence and preempting any other type of interactions, for ultimately, this implementation's agent, is primarily a virtual assistant, capable of conversing. To fulfill this goal, it should be clear that the main control node, the root of our tree, must be a loop-decorated Selector (not parallel) with two main children: a left most - first in the order of execution – Selector, with many Sequence children underneath, and the right-most a Sequence Parallel for the conversation tree, as displayed on tree 5.9.

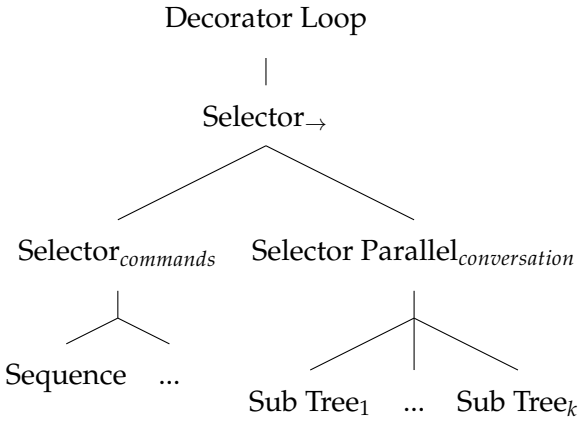


FIGURE 5.9: Interactive Agent Tree 1 - Foundation

This model, allows our agent to handle any pre-defined request over any conversation which might occur should no request is in the line on every update. By introducing *Sequences* under the first child, we know for sure that these can only succeed if and only if a command has been issued and satisfied by any of the subtrees. If this is the case, the *Sequence* subtree will make the first child *Selector* to succeed, notifying the root accordingly. One the root reports success to its parent decorator, the loop starts again on the next update. But what if we actually want to block the tree from updating while a certain animation sequence or processing is occurring in the background? for example, data fetching from a web-service, or some heavy computational task. The first approach would be to introduce a *Wait* node after every subtree, guaranteeing the tree to keep updating this node until the wait period is over. This is, however, extremely impractical and ineffective from an implementation standpoint, for knowing the exact times some tasks might take before hand is not scalable nor portable. To solve this problem, we can introduce a preemptive node before the original first

Selector which will assert for a blocking flag.

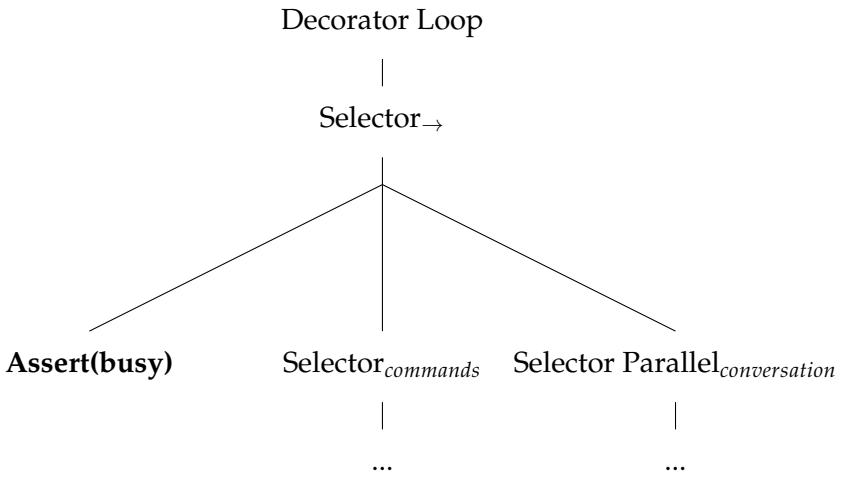


FIGURE 5.10: Interactive Agent Tree 2 - Blocking Control

Tree 5.10 provides an elegant and scalable mechanism which will block the main tree from being updated while the *busy* flag is set on. This same mechanic could be applied to any subtree which we don't want to execute given a certain condition is met, but overall, having a general *Assert* blocking node is always a good idea to ensure a single behavior sequence is played at any given time, without another one interrupting it, nor having to deal with specific animation lenght calculations.

5.2.1 Utility Based Nodes

Thus far, we have covered a model which might suit most basic requirements in terms of flexibility, scalability and ease of implementation. Nevertheless, it is worth pointing out that although behavior trees by themselves are great structures to execute a certain pattern of behaviors, these might not be as scalable when it comes to decision making and adaptability. Of course, one could spend a lifetime developing a tree capable of handling an impressively large amount of different circumstances solely utilizing *assertions*, but in reality, as any other monolithic structure, there is a trade-off between complexity, maintainability and scalability. Once a complex enough tree is in place, it proves to be a real challenge for a newcomer developer or designer, to modify or adapt a large group of nested nodes. Furthermore, there is yet one more characteristic

to be discussed when crafting believable behaviors, which is paramount to any truly immersive interaction. This is of course, adaptability. How could a tree be crafted in such a way that this would execute differently depending on a specific character's state? This implementation proposes to leverage the already more flexible dynamic trees by introducing components of *Utility Based AI*. The aforementioned AI model is based on the fact that decisions have a measurable outcome. The metric utilized is a value known as the *utility* of said choice. In summary, we can assign different values to a certain action, in our case, subtrees. The highest scoring one will be the one to execute. This definition begs the question: how could statically assigned values can result on a more dynamic and adaptive set of behaviors? This is true. Values are usually statically assigned in the most simple implementations, but this doesn't exclude the fact that these could be evaluated in function of the state of the agent. For example, an angry or hostile agent would weight utility u_i by a factor of trait t_j, hence the resulting utility value of the execution of an action would be the product of $U = u_i * t_j$. Furthermore, if comparing multiple subtrees with different options, we can simply determine which one would yield a more desired outcome by comparing their total *utility* values given by $\sum_{a=1} U_a * t = U_{st}$ for each subtree st. Dynamism is then a collateral product of fluctuating traits on our agents. Should we need to add yet more uncertainty to which direction our agent will choose, we could further multiply each subtree's value by a probability P_{st} of that path to be executed. So ultimately, our deliberation module will simply choose $max(P_{st} * U_{st})$ among those which can actually be execute, should any *assertion* needs to be satisfied first.

5.2.2 Implementation

To implement this model, first, we need to define the agent's traits. Secondly, we will simply need to modify our implementation by allowing action nodes to carry a list of traits which each actually affects, for not all actions will necessarily have direct bearing in all traits.

Starting with traits, we should have enough which would allow our implementation to achieve the desired complexity level for the simulation in question. Specifically

to this implementation, we distinguish two different types of measurable attributes: character traits and emotions. The first are static, while the second are dynamic, therefore they change throughout different interactions. Normally, character traits are initialized just once, this is, when an agent is being created. The second group's values are persisted on each run of the simulation, and slightly and randomly modified every time the simulation re-opens. As previously seen, the following is a viable example of general character traits and emotions:

Traits and Emotions			
Trait	Value	Emotions	Threshold
Friendliness	rand(6..10)	Joy	rand(3..10)
Charisma	rand(1..10)	Angry	rand(0..3)
Courageousness	rand(5..10)	Hostility	0
Respectfulness	rand(7..10)	Sadness	rand(0..4)
Bad Tempered	rand(3..6)	Fear	0
Arrogance	rand(1..6)	Disgust	0
Clumsiness	rand(1..4)	Annoyance	0

TABLE 5.1: Subset of Agent's traits and emotions

When defining an *Action Node*, we associate to it an instance of a *NPC Affordance*. This affordance not only describes the type of action, but could also be loaded with specific modifiers which are applied when execution said action.

Finally, the behavior tree that drives the behavior module is designed to be responsive to the user's communication across all modes. Responsiveness can be thought of as the ability to immediately respond to a stimulus. This includes situations where a different stimulus of the same or different type is already being responded to. In order to achieve basic responsiveness to one stimulus using behavior trees, a looping control node must check frequently to see if the stimulus has been signaled. This assumes that

the response to the stimulus is atomic, meaning that it cannot be interrupted by another stimulus of the same type. If the response is not atomic, the following structure must be used instead. For basic responsiveness to two stimuli with mutually exclusive responses, a concurrency control node must use the aforementioned structure for both stimuli in parallel. However, when the stimulus of higher priority is signaled, the response to the lower priority stimulus should be halted immediately. An example of this mutual exclusivity is the use of verbal responses by multiple stimuli, because an agent cannot speak two utterances at the same time. The behavior tree shown in Figure 5.11 is a simplification of the behavior tree used in the behavior module. However, it illustrates aspects of the aforementioned constructs for responsiveness to stimuli and the complexity of a behavior tree that has interplay between multi-modal stimuli. The structure of the behavior tree parallels the dependence of the behavior module on the sensing and knowledge modules. The right subtree of the root node uses the responsiveness structure for a single stimulus to listen for the end of a user's speech. This subtree is what maintains the agent's adherence to the phases of conversation. When the user begins to speak again, the subtree is interrupted and waits until the user stops speaking before attempting to deliver a response. The left subtree is responsible for the agent's verbal and nonverbal acknowledgements of the user's emotions (which are explained in the Emotion Response section below). It has a higher priority than the right subtree so that when a verbal acknowledgement is elicited, the left subtree is able to interrupt the right subtree.

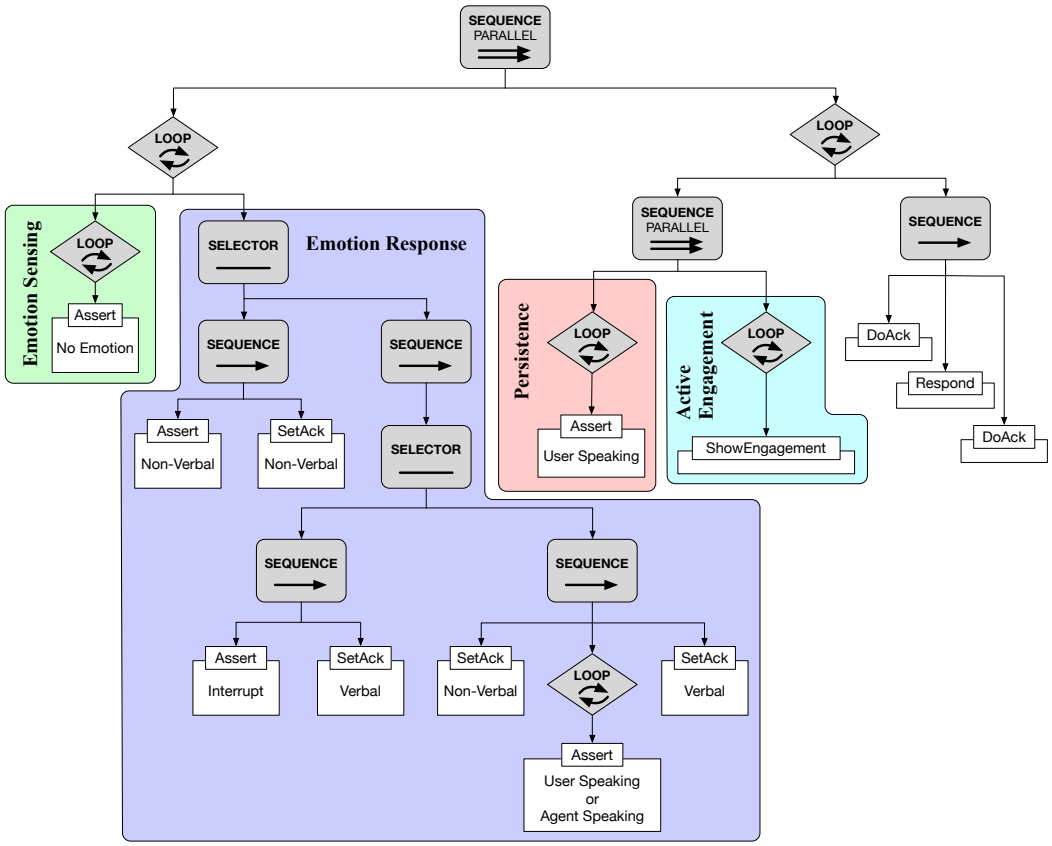


FIGURE 5.11: Simplified Behavior Tree Control Structure for Multimodal Behavior Synthesis

Chapter 6

Conclusion and Future Work

6.1 Conclusion

A video reel is provided for all the projects which have used the framework so far. The video displays three main cases categories: crowds, individual agents and behaviors.

NPC Framework Reel - URL: https://goo.gl/bVMTGG

The open source code for the entire framework is also available in Gitlab:

NPC Framework Source - URL: https://gitlab.com/fgeraci/NPC

When creating a new agent-driven project, be it a simulation, game or general application. there are many components which weight heavily on the development cycle due to the effort and time these require. Some of these are hard assets like models, animations and media. Once created, these need to be integrated with a system which allows for flexible and timely design. These systems, known as engines or components or engines, must be scalable, robust and intuitive. That is the goal of this implementation, to provide a non opinionated bed of tools which would allow all-agent components to be integrated into any project seamlessly, offering basic capabilities but at the same time not restraining customization and design.

The core of this framework was utilized and is being used by academic and non academic sources. Although the main goal was to create a commercial product to extend the Unity's engine capabilities, the decision was made to open the source up from its repository, to allow not only developers, but also students to understand how to integrate every agent main components:

- Avatars
- Media

- Animations
- Animation Controllers and State Machines
- Collision Detection
- Physics-based Dynamic Steering
- Behavior Modeling

These parts then come together by dragging and dropping a single component onto the humanoid prefabricated agent object. Among other assets, included with the framework, is a multi-layered multi-sub-state animator controller to handle every bones sector component of the agent agent based on the gesture or animation to be performed, utilizing provided masks and most importantly, code to granularly control these and ultimately, extend. The framework is open for pull requests, bug tickets and fixes, and it support will be continuous, since it is currently being used in the development of a graphical-adventure like game.

Additionally, and most importantly, not only the agent described in this work is being currently developed by a private company as a conversational virtual assistant, but the same framework was used by the world-recognized firm of architects, Zaha Hadid for a real world laser projected show against the Karlsruhe Castle in Germany of a crowd simulation walking throughout the premises. The latter was presented on 2017. Since then, the framework has evolved significantly to add intuitive UI based design of behaviors, dynamic paratemerization, better social-forces-based steering physics, expanded affordances and the option of modularly add any type of custom affordances, among other aspects. Therefore, the project will be actively supported beyond this work.

The utilization of such an individual management component is critical when implementing projects which otherwise, would require a substantive amount of effort just to get some of the aforementioned elements synchronized. Furthermore, the fact that the code is externally maintained and free, allows for versioning while developing other components and regular updates from external sources.

Throughout this work, it has been demonstrated that an agent-centric system to handle not only single avatars, but multiple ones in groups, can be developed by implementing each component following the natural division of capabilities of human beings. As explained, these are the ability to perceive, to deliberate based on the information and state of the world around and to act accordingly. Finally, all these sections re controlled by a central component which conversely, serves as the interface of said agent to the world for development and design.

Finally, the intention of this work is to offer simulation designers, researches and students a reliable platform to support their work, abstract complexities or offer engineering resources for study and to extend.

6.2 Case Studies

Although referenced above, here is a detailed list of the publications, implementations and projects on which the framework has been used thus far:

- AAMAS 2018 - Autonomous Agents and MultiAgent Systems
- ICIDS 2015 - Interactive Digital Storytelling
- Zaha Hadid Architects - Karlsruhe Castle, Germany
- Motional.AI - Sara, Virtual Assistant
- In Memoriam - Game

The AAMAS publication, explains how conversational agents can be implemented with all the aforementioned techniques from this work. The second paper the details the implementation of the framework's components which handle the perceptive memory capability of agent; how they are affected by events, remember interactions with other agents and ultimately deliberate on how to behave based on the surrounding context.

The last three projects use the framework as their primary agents engine for crowds simulation, a virtual assistant and a multi-agent interactive game, respectively. Both

the Zaha Hadid and the Motional.AI projects were privately funded endeavors, while the third one is currently under development to support the frameworks development in term of capabilities, debugging and features.

A demo of each project can be seen in the provided video reel.

6.3 Future Work

The framework has two aspect for development: stabilization and feature development. The source has been opened for public utilization and contribution under an MIT license since November 1st, 2018. Support will be continuous until reaching a stable v1.0 release version. As of today, there have been feature requests and bug reports which will be addressed on a timely manner. Once every components has been debugged, features will be included and ultimately, active support will be provided as Unity versions change. The original version was introduced with Unity 2015, and has been updated to the latest release recently. Along these lines, many aspects have been tweaked which resulted in backward compatibility issues, therefore different LTS and development versions will be created.

In terms of features, the partial-planning example module will be finished and finally, mixed behavior trees with dialog response options with effects will be implemented. This last feature is currently under development, since it will be used in the afore mentioned game, which utilizes the framework for their agents integration and interactions.

Comprehensive documentation will be added to the repository, in the form of a Wiki, along a formal API reference for each of the agent, and framework components described throughout the chapters, body, perception and AI. So far only video-documented examples of implementations are provided with the framework. Although useful, these must be complemented with detailed written procedures.

Appendix A

Abbreviations

ECA	**E***mbodied***C***onversational***A***gent*
NPCF	**NPCF***ramework*
AI	**A***rtificial***I***ntelligence*
NLP	**N***atural***L***anguage***P***rocessing*
NLU	**N***atural***L***anguage***U***nderstanding*

Bibliography

all, Alexander Shoulson et. (n.d.). “Parameterizing Behavior Trees”. In: *Motion in Games, Edinburgh, UK, November, 2011*. URL: https://doi.org/10.1007/978-3-642-25090-3_13.

Banchs, Rafael E and Haizhou Li (2012). “IRIS: a chat-oriented dialogue system based on the vector space model”. In: *Proceedings of ACL 2012 System Demonstrations*, pp. 37–42.

Chandra, Bhuvana (n.d.). “Real-time dynamic partial order planning for memory reconstruction in autonomous virtual agents”. In: URL: https://doi.org/doi:10.7282/T3FR002M.

Chowanda, Andry et al. (2016). “Computational Models of Emotion, Personality, and Social Relationships for Interactions in Games”. In: *Proceedings of the 15th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2016)*, pp. 1343–1344.

Colby (1972). *PARRY*. URL: https://www.chatbots.org/chatbot/parry/.

Ekman, Paul and Wallace V. Friesen (1978). *Facial Action Coding System: A Technique for the Measurement of Facial Movement*. Consulting Psychologists Press.

Emerich, Simina, Eugen Lupu, and Anca Apatean (2009). “Emotions Recognition by Speech and Facial Expressions Analysis”. In: *2009 17th European Signal Processing Conference*, pp. 1617–1621.

Feng, Andrew W. et al. (2012). “An Analysis of Motion Blending Techniques”. In: *The Fifth International Conference on Motion in Games*. Rennes, France.

Ferrucci, D. A. (2012). “Introduction to "This is Watson"”. In: *IBM J. Res. Dev.* 56.3, pp. 235–249. ISSN: 0018-8646. DOI: 10.1147/JRD.2012.2184356. URL: http://dx.doi.org/10.1147/JRD.2012.2184356.

Foundation, A.L.I.C.E AI (2002). *ALICE*. URL: http://www.alicebot.org/.

Hochreiter, Sepp and Jürgen Schmidhuber (1997). "Long short-term memory". In: *Neural Computation*, 9(8):1735–1780.

Kapadia, Mubbasir, Nathan Marshak, and Norman I. Badler (2014). "ADAPT: The Agent Development and Prototyping Testbed". In: *IEEE Transactions on Visualization and Computer Graphics* 99.PrePrints, p. 1. ISSN: 1077-2626. DOI: http://doi.ieeecomputersociety.org/10.1109/TVCG.2014.251.

Leuski, A. et al. (2006). "Building effective question answering characters". In: *Proceedings of the 7th SIGdial Workshop on Discourse and Dialogue*, pp. 18–27.

Morris, T.W. (2002). "Conversational Agents for Game-Like Virtual Environments". In: *AAAI Spring Symposium on Artificial Intelligence and Interactive Entertainment*, pp. 82–86. URL: http://www.qrg.northwestern.edu/Resources/aigames.org/papers2002/TMorris02.pdf.

Potard, Blaise, Matthew P. Aylett, and David A. Baude (2016). "Cross Modal Evaluation of High Quality Emotional Speech Synthesis with the Virtual Human Toolkit". In: *Intelligent Virtual Agents: 16th International Conference, IVA 2016, Los Angeles, CA, USA, September 20–23, 2016, Proceedings*. Ed. by David Traum et al. Cham: Springer International Publishing, pp. 190–197. ISBN: 978-3-319-47665-0. DOI: 10.1007/978-3-319-47665-0_17. URL: https://doi.org/10.1007/978-3-319-47665-0_17.

Robinson, S. et al. (2008). "What would you ask a conversational agent? Observations of human-agent dialogues in a museum setting". In: *LREC 2008 Proceedings*.

Serban, Iulian V et al. (2015). "Building End-To-End Dialogue Systems Using Generative Hierarchical Neural Network Models". In: *Proc. of AAAI*. URL: https://arxiv.org/pdf/1507.04808.pdf.

Sutskever, Ilya, Oriol Vinyals, and Quoc V Le (2014). "Sequence to sequence learning with neural networks". In: *Advances in neural information processing systems (NIPS)*, pp. 3104–3112.

Tarau, P. and E. Figa (2004). "Knowledge-Based Conversational Agents and Virtual Storytelling". In: *ACM Symposium on Applied Computing*, pp. 39–44. URL: http://www.cse.unt.edu/~tarau/research/2003/vschat.pdf.

Technologies, Unity (2018). *Unity Game Engine*. http://unity3d.com/unity/. [Unity Game Engine].

Turing, A. M. (1950). "Computing Machinery and Intelligence". In: *Mind* 59.236, pp. 433–460. URL: http://www.jstor.org/stable/2251299.

Weizenbaum, Joseph (1966). "ELIZA&Mdash;a Computer Program for the Study of Natural Language Communication Between Man and Machine". In: *Commun. ACM* 9.1, pp. 36–45. DOI: 10.1145/365153.365168. URL: http://doi.acm.org/10.1145/365153.365168.

Xu, Yuyu et al. (2013). "A Practical and Configurable Lip Sync Method for Games". In: *ACM SIGGRAPH Motion in Games*. Dublin, Ireland. URL: http://ict.usc.edu/pubs/A%20Practical%20and%20Configurable%20Lip%20Sync%20Method%20for%20Games.pdf.

Young, Steve et al. (2010). "The hidden information state model: A practical framework for pomdp-based spoken dialogue management." In: *Comput. Speech Lang*, 24(2):150–174.