University of Crete
Computer Science Department

AN ARTIFICIAL PRESENTER AVATAR WITH SOCIAL BEHAVIOUR, EMOTION-DRIVEN ANIMATION AND MINI GAMES

by

EFFIE KAROUZAKI

MASTER'S THESIS

Heraklion, November 2008

University Of Crete
Computer Science Department

AN ARTIFICIAL PRESENTER AVATAR WITH SOCIAL BEHAVIOUR, EMOTION-DRIVEN ANIMATION AND MINI GAMES

by
EFFIE KAROUZAKI

A thesis submitted in partial fulfillment of the
requirements for the degree of

Master of Science

Author: _____
Effie Karouzaki, Department Of Computer Science

Board of enquiry:

Supervisor _____
Anthony Savidis, Associate Professor

Member _____
Constantine Stephanidis, Professor

Member _____
Dimitris Plexousakis, Professor

Approved by: _____
Panos Trahanias, Professor
Chairman of the Graduate Studies Committee

Heraklion, November 2008

AN ARTIFICIAL PRESENTER AVATAR WITH SOCIAL BEHAVIOUR, EMOTION-DRIVEN ANIMATION AND MINI GAMES

EFFIE KAROUZAKI

Master's Thesis

University Of Crete
Computer Science Department

Abstract

This Thesis discusses the design and implementation of an artificial game presenter named *Amby* as a software avatar supporting social behaviour and emotion-driven animation control. The main concept is that during a game session, an independent ambient presence avatar monitors the game progress and player activities to perform typical social comments. Additionally, in synergy with the main game, the avatar introduces extra challenges to players called mini games, such as hangman and random card selection. It also displays standard game elements like players' inventory, lives and score while enabling dice rolling.

The avatar behaviour adopts the sense-think-act model which is typical for game characters, while internally it is implemented as an adaptive behaviour in which social responses are driven by game-oriented stimuli. In this context, the entire AI component has been implemented with two layers of decision making using an appropriate existing language.

Amby is rendered as an anthropomorphic character, currently displaying only its face with basic elements like eyes, ears, nose, hair and mouth. In this context, expressions like happiness, surprise, sadness and anger are supported, handled by an emotion-driven animation core, according to emotion state transitions driven by the basic AI component.

To address the previously mentioned challenges, *Amby* has been implemented in three different languages: (a) underlying services and communication with the master game, in C++; (b) decision rules, in DMSL; and (c) rendering, animation and mini games, in Delta.

ΕΝΑΣ ΤΕΧΝΗΤΟΣ ΧΑΡΑΚΤΗΡΑΣ ΠΑΡΟΥΣΙΑΣΤΗ ΜΕ ΚΟΙΝΩΝΙΚΗ ΣΥΜΠΕΡΙΦΟΡΑ, ΣΥΝΑΙΣΘΗΜΑΤΙΚΑ ΚΑΘΟΔΗΓΟΥΜΕΝΕΣ ΚΙΝΟΥΜΕΝΕΣ ΕΙΚΟΝΕΣ ΚΑΙ ΠΑΡΑΠΛΕΥΡΑ ΠΑΙΧΝΙΔΙΑ

ΕΥΦΡΟΣΥΝΗ ΚΑΡΟΥΖΑΚΗ

Μεταπτυχιακή Εργασία

Πανεπιστήμιο Κρήτης
Τμήμα Επιστήμης Υπολογιστών

Περίληψη

Η παρούσα εργασία μελετά το σχεδιασμό και την υλοποίηση ενός τεχνητού παρουσιαστή παιχνιδιών με το όνομα Amby, ως χαρακτήρα λογισμικού που υποστηρίζει κοινωνική συμπεριφορά και έλεγχο συναισθηματικά καθοδηγούμενων κινούμενων εικόνων. Η βασική ιδέα είναι κατά τη διάρκεια ενός παιχνιδιού, ένας ανεξάρτητος χαρακτήρας με διάχυτη παρουσία παρακολουθεί την εξέλιξη του παιχνιδιού και τις ενέργειες των παικτών για να κάνει τυπικά κοινωνικά σχόλια. Επιπροσθέτως, σε συνεργασία με το κυρίως παιχνίδι, ο χαρακτήρας παρέχει επιπλέον προκλήσεις για τους παίκτες, τα επονομαζόμενα παράπλευρα παιχνίδια, όπως είναι η κρεμάλα και η τυχαία επιλογή καρτών. Επίσης παρουσιάζει βασικά στοιχεία του παιχνιδιού όπως τα αντικείμενα των παικτών, τις ζωές τους και η βαθμολογία τους, ενώ επιτρέπει τη ρίψη ζαριών.

Η συμπεριφορά του χαρακτήρα υιοθετεί το μοντέλο «αισθάνομαι – σκέφτομαι – ενεργώ» που είναι τυπικό για χαρακτήρες παιχνιδιών, ενώ εσωτερικά είναι υλοποιημένο σαν προσαρμόσιμη συμπεριφορά στην οποία κοινωνικές αντιδράσεις καθορίζονται από ερεθίσματα που προέρχονται από το παιχνίδι. Σε αυτό το πλαίσιο, όλο το τμήμα της τεχνητής νοημοσύνης είναι υλοποιημένο σε δύο επίπεδα λήψης αποφάσεων χρησιμοποιώντας μια κατάλληλη προϋπάρχουσα γλώσσα.

Ο Amby προβάλλεται ως ένας ανθρωπόμορφος χαρακτήρας εμφανίζοντας επί του παρόντος βασικά χαρακτηριστικά όπως μάτια, μύτη, αυτιά, μαλλιά και στόμα. Σε αυτό το πλαίσιο, υποστηρίζεται η έκφραση συναισθημάτων όπως χαρά, λύπη, έκπληξη ή οργή τα οποία διαχειρίζονται από ένα σύστημα συναισθηματικά καθοδηγούμενων κινούμενων εικόνων, σύμφωνα με τις συναισθηματικές μεταβάσεις που καθοδηγούνται από το βασικό τμήμα της τεχνητής νοημοσύνης. Για να αντιμετωπιστούν οι προαναφερθείσες προκλήσεις, η υλοποίηση του Amby έγινε σε τρεις ξεχωριστές γλώσσες προγραμματισμού: (α) οι υποκείμενες υπηρεσίες και η επικοινωνία με το κυρίως παιχνίδι, σε C++, (β) κανόνες λήψης αποφάσεων, σε DMSL, και (γ) απεικόνιση, κινούμενες εικόνες και παράπλευρα παιχνίδια, σε Delta.

Ευχαριστίες **(Acknowledgements)**

Θα ήθελα να ευχαριστήσω τον επόπτη της μεταπτυχιακής μου εργασίας Αντώνιο Σαββίδη για την συνεχή καθοδήγηση και υποστήριξή του τα τελευταία τρία χρόνια στο πλαίσιο της συνεργασίας μας στο Εργαστήριο Αλληλεπίδρασης Ανθρώπου-Υπολογιστή, του Ινστιτούτου Πληροφορικής του Ιδρύματος Τεχνολογίας και Έρευνας και ειδικότερα στο πλαίσιο της εκπόνησης της μεταπτυχιακής μου εργασίας.

Θα ήθελα επίσης να ευχαριστήσω την οικογένειά μου και τους φίλους μου και τον Γιάννη για τη βοήθεια και τη στήριξη που μου παρείχαν όλα αυτά τα χρόνια, καθώς και τον Αντώνη τον Κατζουράκη για τη γραφιστική εργασία που χρειάστηκε.

Table of contents

List of figures

List of tables

1. Introduction

The concept of ambient intelligence fosters an environment where humans are surrounded by computing and networking technology embedded in their surroundings. It refers to an exciting new paradigm in which people are empowered through a digital environment that is aware of their presence and context, and is sensitive, adaptive and responsive to their needs, habits, gestures and emotions. In such an environment, people don't have to focus on the technologic part of their surroundings; still they can communicate with their everyday environment in an enhanced manner– for example when they ask their fridge to make a list of lacking goods (GENIO [4]). In this sense, computing technology is infiltrating into our everyday lives, by disappearing inside our homes, our cars, our clothes. Most of the research being performed is in the field of smart homes, where intelligent appliances and systems are being developed able to recognize human presence in a room or near an appliance and interact with them in a natural way. For example, consider a smart oven that can download recipes from the internet and make a cooking suggestion depending on the type of the meal and the time it needs to be cooked. Moreover, the oven could communicate with the fridge through a network to determine that all of the ingredients required for the recipe are available. There are also various home appliances enhanced with specially designed wired clothing that can be used to monitor peoples' medical condition, recording medical data about peoples' heart beat rate, blood pressure and weight. Such medical data could be used to automatically make a fitness and diet program for them to follow or, more importantly, they could be kept in log files or sent to the nearest hospital if something went wrong, or if the person living in the house needs to be under medical attention.

Another example of ambient intelligence usage is in the domain of entertainment. Systems that adapt the living room's lights and air-conditioning are promising to change the way people play games. Atmospheric lighting systems, air blowing machines, multiple screens displaying images and playing videos according to the game context, along with artificial avatars presenting and commenting the games are some parts of the picture of next generation gaming. Artificial commentators have been added to video games like soccer in order to convey excitement to audience and

make the game more interesting, by providing smart in game observations and comments, as well as statistic information that are very difficult to extract for the viewers. Such artificial characters can also have visual representations (for instance talking head reference) able to provide emotional feedback additionally to knowledgeable comments on video games. Most progress in commentating software has been done in soccer video games, in order to stand in for real commentators in football stadiums.

Focusing more on gaming, it can be observed that nowadays, people are gradually starting to get away from their monitors, and start playing in ambient environments which combine the benefits of human to human interaction with the latest computing technology. In this sense, several ambient electronic games have been developed targeting ambient environments. Moreover, there are various examples of more traditional board games enhanced with electronic technology, combining the benefits of a computer game (dynamic terrain structure, fancy sound effects, and ambient environment context) with human to human live interaction. Such setups are the best possible candidates for making the most out of ambient technology, especially when incorporated with an additional artificial avatar aware of the board game concept and able to interact with the players and their environment. Unfortunately, such characters have not yet infiltrated this field, probably due to the fact that classic board games don't dedicate additional players to observe and comment the other players' actions.

The purpose of this thesis is to design and develop AMBY, an ambient presence avatar able to provide socially-oriented game narration and knowledgeable comments, artificial emotion state through emotion-driven animations and a collection of additional side-games. Amby is intended to be attachable to a wide set of electronic board games, which embrace specific characteristics and can provide a collection of in-game events to Amby through a specified set of functions (API) provided by the latter. More specifically, the Amby character is designed to provide support for:

- Displaying and handling some of the most common board game concepts, like dice, score, life and inventory
- Playing media such as sound effects, music, video, images and text, and displaying various combinations of them.

- Using a text to speech synthesis tool for talking to players in order to boost the interaction between the players and the game.
- Making knowledgeable comments on game progress adapted to players' social profile, their progress in game and current game status. Comments can be chosen according to Amby's personality and emotional state combined with players' social profiles, such as age or profession. That means that Amby can develop a personality, judging his own behavior towards each player separately.
- Providing a set of external extensible AI rules that can be easily adapted to many game contexts and player profiles that contain usable information about them. The latter will be taken into account in the process in order for Amby to decide the most suitable comment for the current player to make.
- Enhancing the interaction between the players and the game by making facial expressions that convey the avatar's emotions.
- Using adaptable animated cues to comment game status and draw players' attention.
- Enhancing board game play by providing and supporting mini games. Mini games can be used for giving special bonus to the players, or as a means of drawing the players' attention from the main game in order to maintain their interest in it and boost the gaming experience. Of course, a mini game is a game itself and Amby is in charge of running, supervising and commenting it, as well as the player's progress in it.

Additionally, Amby is designed to be displayed in the same ambient environment as the board game table and specifically close to it (for example, on a close wall or on a screen over the players' heads) and exchange information with the game over the network. Amby has the ability to support multiple screens and interact with the game surroundings, such as changing the room's ambient lighting, or regulating the room's temperature to mach the game context.

In order to test and evaluate the system, the *on-board!* game platform [3] was used. The ***on-board!*** game platform comprises: (a) a board game engine suited to ambient setups, supporting configurable roundtable player positions, adaptive pluggable input infrastructure, extensible in-game dialogues and remote interface-feedback services;

(b) an integrated terrain editor providing a directed graph model for board terrains consisting of game-path vertices, while facilitating the authoring of terrain components' libraries (sub-graphs as bitmaps with connectivity information), game-board composition from path elements with game items, and game logic scripting in the Delta language. Finally, the Amby character was attached on a fully working board game entitled ***The Four Elements***, which has been built using the ***on-board!*** platform. In Figure 1, the supported configurations of Amby and the board game are displayed.

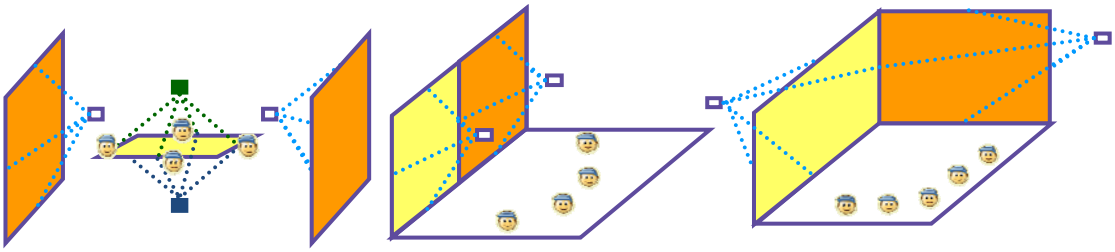

Figure 1 – Supported configurations for the Amby character and the Board game. Yellow indicates the board game screen while the orange color represents the Amby character.

2. Related work

Artificial intelligence characters are commonly used in software applications. They are mostly found in games as opponents, assistants, commentators or narrators, but they are also used for other applications (mostly interfaces) as presenters. In the field of artificial presenters, PPP Persona [10] is an artificial character, which receives data from the network and organizes them into paragraphs and sentences in order to present them to users. Tsukasa Noma and Norman I. Badler have developed a virtual human presenter based on the Jack^{TM} animated agent system [11]. Their agent takes as input some text with embedded commands about the presenter's body language. The system then makes him act as a presenter with presentation skills in real time 3D animation synchronized with speech output. However, the most challenging field for artificial character usage is computer games. Artificial characters are everywhere: opponents, assistants, allies, narrators and commentators, and all of them need to decide their actions by using an AI model.

Game narrators and commentators are the most relevant to the work presented in this thesis. For example, an expressive talking head narrator for interactive storytelling systems is reported in [12]. This project uses an expressive talking head system, represented by an avatar with facial expressions, who acts as an animated Narrator of the story to the attending spectators. The implemented system integrates modules for managing plot generation, user interaction, visualization and narration. In terms of game commentating, a lot of work has been done for sport games, especially soccer, where the game excitement has to be conveyed to the players. Soccer commentators are ROCCO from DFKI [14], [5], BYRNE from Sony CSL [6], MIKE from ETL [13], [5], Team Assistant 2006 [7] and Caspian Commentary System [20]. All of them use as input data from the soccer server (like, players' positions and orientation, ball position and velocity and play modes such as goal, throw-in, free kick and so on) to extract information about the game situation. Then, game analysis is performed in order to decide the comment that will be announced to the players based on events' importance. The BYRNE system differs from the others as it includes a 3D talking head with emotional feedback. It focuses mostly on facial expressions and claims to

support commenting on a wide range of games. However, the events that these systems are claimed to get as input are very specific, so it is assumed that although they might support sport game commenting, they are unable to comment computer games in a general manner. In terms of speech, Team Assistant and Caspian use a set of prerecorded statements for each game status, while the other systems use a speech synthesizer. In addition to speech, Team Assistant also uses sound effects, and that seems to make a lot of difference to the audience. Finally, Team Assistant's main power is claimed to lie in its ability to be extended using AngelScript plug-ins [9]. In fact, designers must develop plug-ins that contain all the artificial logic, since in general plug-ins take game input and makes output decisions about players' and ball's moves, the match mode, controlling the camera etc. In fact, Team Assistant is claimed to be extendable because every part of it is implemented with plug-ins.

All of these commentator systems cannot be easily expanded (even in Team Assistant one has to make the commenting logic from scratch to provide it as a plug-in). Also, none of these systems is capable of commenting board games, and none of them uses more than three means of comment representation (they only use text, voice -artificial or prerecorded-, while BYRNE additionally uses facial emotions and the Team Assistant uses sound effects).

This thesis presents a developed a system for commenting board games, which

- provides support for several characteristics commonly found in board games
- is able to make adaptable comments on players' progress in game by using players' profile data through an extensible set of rules. The comments are not only made via text drawing, but also through the use of several media such as sound effects, music, images, animations, video, speech and facial emotions.
- Uses facial emotions and adaptable animated cues for commenting game situations
- Supports mini games.

3. Overview of Amby features

The Amby character is able not only to provide support for a list of features that the majority of Board games have in common, but also to enhance board game play through adaptive game narration, emotion-driven animation and mini games.

3.1 Public Inventory and Score

Inventory is a common feature in both board games and computer games. It enables players to store the items they collect for later use, thus let them chose which item to use and when. Considering that most inventories support a maximum number of items, it is a real puzzle for the players to use the inventory in the most efficient way. Inventory items can have many usages, and some of them may be considered as "secret aces". That's why some games may require that each player keeps his own inventory secret (such as Cluedo), while other games encourage the players to show their items to everyone (like Monopoly). Amby supports keeping public inventories, since a public screen is used to display them all. Four inventories, one for each player, are supported, each one displayed next to the owner player's panel. Each player can control his own inventory to be shown or hidden at any time in game. Due to space limits, the inventory can display only the current selected item, along with its name, description, its position in inventory and the times of usages left for it. Because Amby has no input device attached, the players can navigate through their items via the board game interface when their inventory is open. Adding, removing and selecting item actions are also integrated through the Board game interface. When an event happens in the game, Amby is notified to update the status of the inventory, even if the inventory is hidden at the time. Game events like "drop item" and "lose item" result in removing the item from the inventory, while events like "get item" results in adding an item into inventory. When the player uses an item, the usages left are decreased (unless the item has unlimited usages), and when there are zero usages left, Amby is notified to remove the item from the inventory.

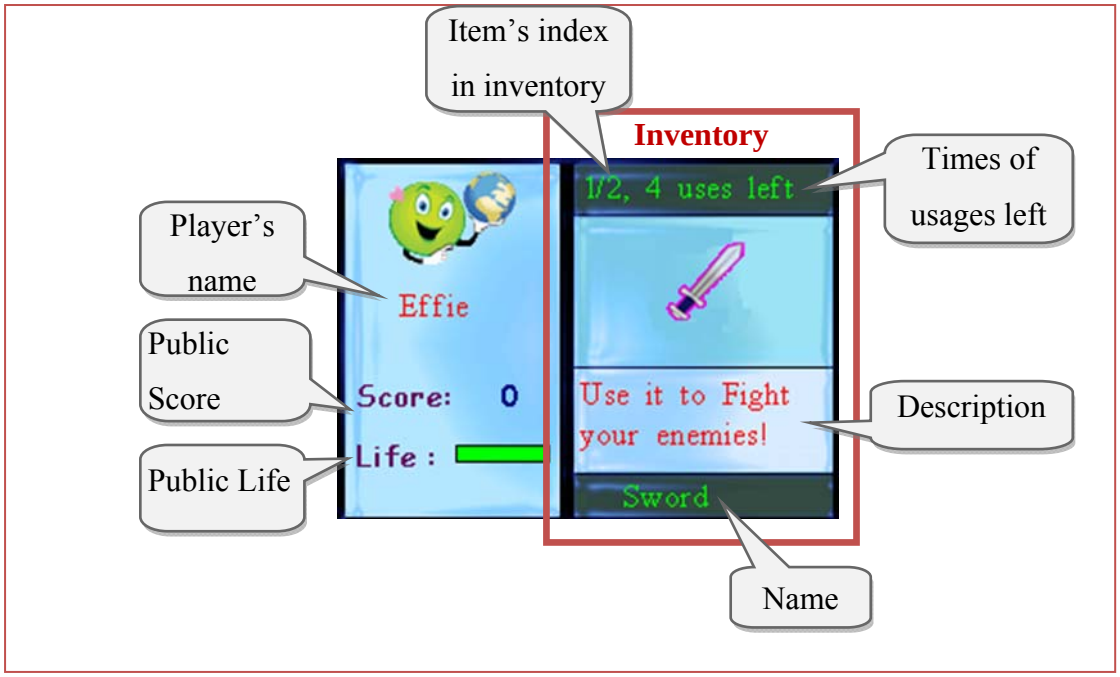

Figure 2 – Player's panel and Player's Inventory

3.2 Mini Games

A Mini game is defined as a small game within a bigger one, usually optional, which tends to be very different from the main game. It is a common tactic for game designers to adopt mini games in order to distract the players from the main storyline, so that they don't get bored. Because they are different from the main game, mini games can be used for special rewards and extra bonus in game, or they can just be challenges or puzzles that the players need to solve in order to proceed in the main game. Usually these mini games are builded from scratch for every game, but Amby suggests that they can be completely independent. The Board game only needs to delegate Amby to start and supervise a mini game, and need to get its result when finished. This concept was implemented through two mini games, Hangman and Card drawing. The board game just decides what mini game to play and when, and Amby takes charge for starting, supervising and ending it, returning a result string to the main game. The latter decides the consequences of the mini game results for the players (for example, whether to give a bonus or not) and the main game continues.

In addition to displaying and supervising the mini games, Amby takes charge of commenting each player's progress in them. Commenting is supported through the same mechanism as commenting the board game itself, with the only difference that the triggering events source from Amby himself and not from the main game. The exact mechanism is further analyzed in section 4.2. Commenting the mini games includes knowledgeable observations, emotion animations related to the players' moves and several multimedia such as sound effects, music support, text to speech synthesis, image displaying and animations, all decided through an extendable rule-based logic based on player's progress, game state and most importantly, on player's social profile.

3.3 Adaptive Social Comments

Amby was developed as an intelligent artificial character able to make *smart* comments on game and player's progress. The word *smart* suggests the character's adaptivity to the game situations, as well as to the target's player social profile information. Most game commenters use a list of phrases suitable for every game situation, and chose to say one of them randomly in order not to be boring. In the case of Amby, the player's background information and social profile combined with progress in the game are filtered and processed in order to decide a proper Amby reaction for each player's move. The latter characteristic transforms the Amby character from a mere event announcer to an agile game commenter. In addition, the mechanism used for commenting is boosted by the use of several multimedia which add up expressive power to Amby's comments and makes Amby suitable for use in ambient environments. More specifically, Amby's reactions can vary from simple text displaying to sound effects, images, animations, text to speech synthesis, as well as facial expressions conveying emotions. Amby can thus adopt several behaviors, and chose his actions according to the selected one. For example, some comments can be considered as suitable for an ironic behavior, while other can be more appropriate for a sympathetic behavior. The interesting thing is that Amby can adapt his conduct to every player separately according to their profiles. That means that Amby can treat a child with sympathy, while he selects to be more austere to a grownup person. That

gives Amby a *personality*, something that is very important for a commenter character to have.

3.4 Adaptive Animated Cues

Apart from commenting a board game, Amby has the ability to identify specific game situations, and provide several animated cues about them. In this context, animated cues are intended as the individual animations and facial expressions that Amby can perform depending on the board game state. When amby perceives a specific game state, he can initiate an emotion or an animation to convey his feelings. For example, if the game or a player is inactive for a long period of time, Amby notices it and starts acting in a specific way in order to motivate the players to continue playing. For example, he can display text complaining about the player's inactivity, or make a snoring sound or get angry because other players will get bored. Using animated cues gives a fancy effect to Amby, transforming Amby's screen into a restless scene that is targeted to motivate the players to play more passionately and stimulates the viewers' interest for the game.

3.5 Emotion Driven Expressions

Among the most challenging characteristics of Amby character is the ability to express himself not only by using several multimedia for commenting a board game, but also to convey his emotions through facial expressions. These expressions arise as evaluation results through the Amby's AI component (using C++ and DMSL language) or as specific commands given by the media service, and are implemented in the UI component, using the Delta [2] language and the ALADIN graphics library. In order to avoid frozen expressions, a mechanism is developed that provides for smooth emotion transitions as well as for restoring Amby's emotional state to happy. As one of Amby's main targets is to present and comment the board game he is attached on, it is important that he communicates his thoughts and feeling via facial expressions such as "Happy", "Sad", "Neutral", "Surprised" and "Hurt". That makes

Amby feel more real, giving the sense that someone is watching the game and cares about the players' progress.

3.6 Configurable Intelligence

One of our main targets when developing the Amby character was to make the character game- independent and easily adaptable, and therefore attachable to the majority of the board games with none or minimum amendments. Towards this objective, an extensible and configurable AI component has been developed using the DMSL [1] language. Because DMSL uses external files for describing the artificial logic rules, developers are free to intervene and change them according to their preferences. The changes made in rule and profile files do not affect the rest of the code. Rules and profile attributes can be added and removed, and the programmer can define which of these rules need to be evaluated and when. The DMSL logic rules describe the actions that Amby has to make in every different game situation, plus they can be used for filtering raw event data into meaningful variables. For example, the information that the player got a +10 or +11 score bonus can be tagged as a "small score gain" and be used as input variable for more generalized rules. In this way programmers do not need to change all DMSL rules each time game semantics change – they only need to define the filtering rules that will transform raw game data into usable information.

Except from filtered information, DMSL rules can take into account the player's progress in the game, the game status and their personal social profile – e.g. their name, age or profession. By combining all these variables, Amby rules can be used for describing very complex logic which enhances Amby character with personality and awareness about the events happened in the game. For example, the programmer can define Amby's behavior to be ironic towards a teenager, while he treats a child with sympathy. The comments made, the icons and the animations showed, the sounds played, even the expressions made by Amby's face are configurable and adaptable to the players through the DMSL logic rules. Programmers can add decisions and define new attitudes only by changing the rule files, without affecting

the rest of the code. We analyze the benefits of the DMSL language and explain how the Artificial Intelligence component works in the section 5 of this Master's Thesis.

3.7 Multilanguage Architecture

The Amby character provides several mechanisms in order to support receiving game events, processing them and making knowledgeable comments, using several multimedia and emotions to communicate with the players. In order to ensure the system's extendibility and reusability, Amby has been built in independent components, and for each one the most appropriate programming language has been chosen. The C++ language was used to develop Amby's services, that is the component which receives message notifications from every game service and dispatch the services requests to the Amby's User Interface.

In order to build an extensible code-independent AI module, the DMSL language was adopted [1]. DMSL is able to evaluate a set of rules taking into account the attributes set in a specific player's profile. That makes the decisions that result from the rule evaluation adaptive to the target player. Since the DMSL rule files are located outside the code, they can be easily changed by the developer, or extended by taking more attributes into account or even by making new rules from scratch. The game programmer can define which rules must be evaluated when a game event arrives, and also which of these events need some filtering in order to make their values meaningful for every game. When the decisions about which actions must be performed by Amby in order to comment an event are taken, Amby's AI subsystem needs to link those decisions to the UI component.

The UI component of the Amby character has been developed using the Delta [2] language. The Delta programming language is an imperative scripting language that encompasses (a) dynamically typed variables, (b) runtime classes, (c) functions as first-class values, (d) unnamed functions, (e) dynamic handling of actual arguments, (f) extensible operator semantics and (g) dynamic inheritance. The selection of the Delta language was based both on the variety of feature it provides as well as on the fact that its implementation is in C++, thus allowing easier integration with the

Amby's AI and Services components. Also, Delta allows easier usage of the ALADIN library which was adopted for displaying graphics. The linkage between the C++ and Delta is accomplished via wrappers, so that the C++ and Delta components are totally independent.

Figure 3 presents the three selected programming languages and the component for which each language was used.

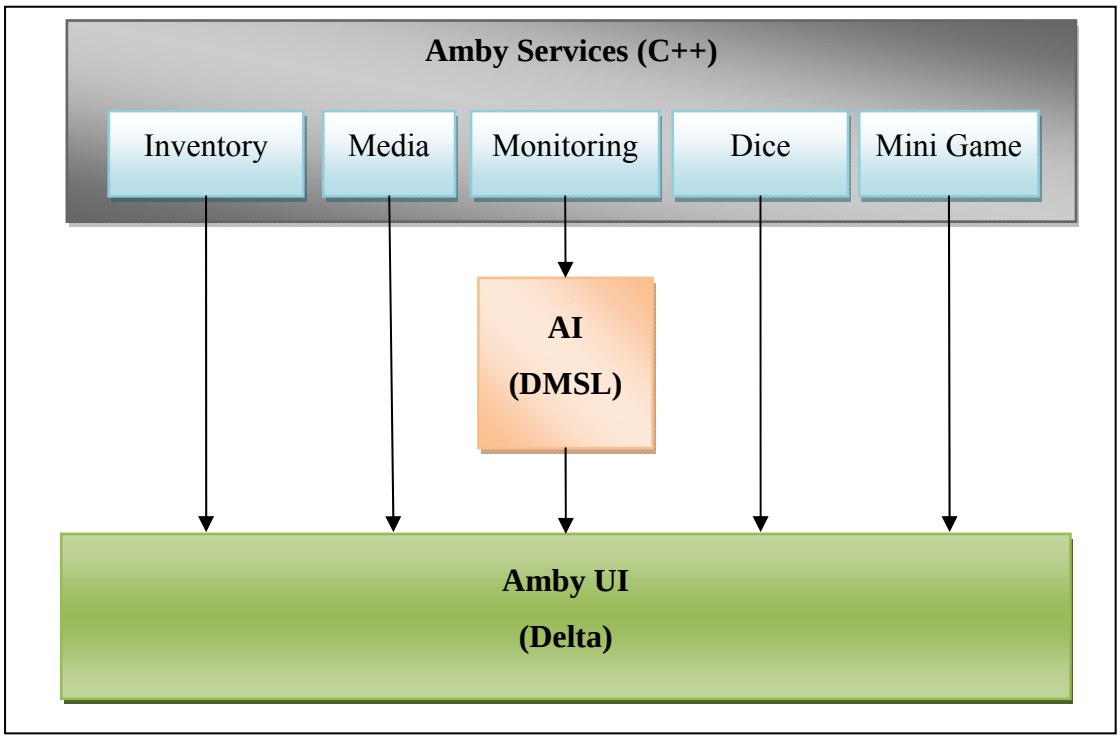

Figure 3 – Amby's Multilanguage Architecture

4. Software Architecture

4.1 Services

Amby offers a set of independent services for the board game maker to use. Each service represents one of the most common game characteristics and is packed in an independent dll file for the game programmer to use. By incorporating services, the board game designer authorizes the Amby character to display and comment these characteristics via Amby's avatar. The services provided are Inventory, Dice, Media, Mini Game and Game Monitoring. In order to make service handling scalable and reusable, a component was built containing an abstract Service class, a broker and a Service manager. All services derive from the parent Service class and have a unique id and a constructor function, while the Broker and the Service Manager component are not even aware of the services they handle. In order to link the services with the service handling component, an implementation specific class is needed, which will add the services to the broker. This class is called Service Initializer, and needs to be aware of both the services and the service handling component.

Figure 4 explains how Service handling works through a dependency graph. The Service Initializer component adds each service to the broker, providing the service's id and a pointer to its constructor function. The broker is the one receiving the network messages sent from the dlls. These messages use xml form to encode the service's name and a parameter value. The Broker checks if the service's name exists among the ones that the service initializer added, and if it does, it calls the constructor function with two parameters, the socket itself and the parameter specified in the xml message. This way the Broker not only constructs the services without knowing them, but also by passing the socket to them it enables direct communication between the specified service and the dll. If the connection breaks, a message will be addressed to the Broker again, and will be routed to the corresponding service. When a service is constructed, it adds itself to a service list hold by the Service Manager class.

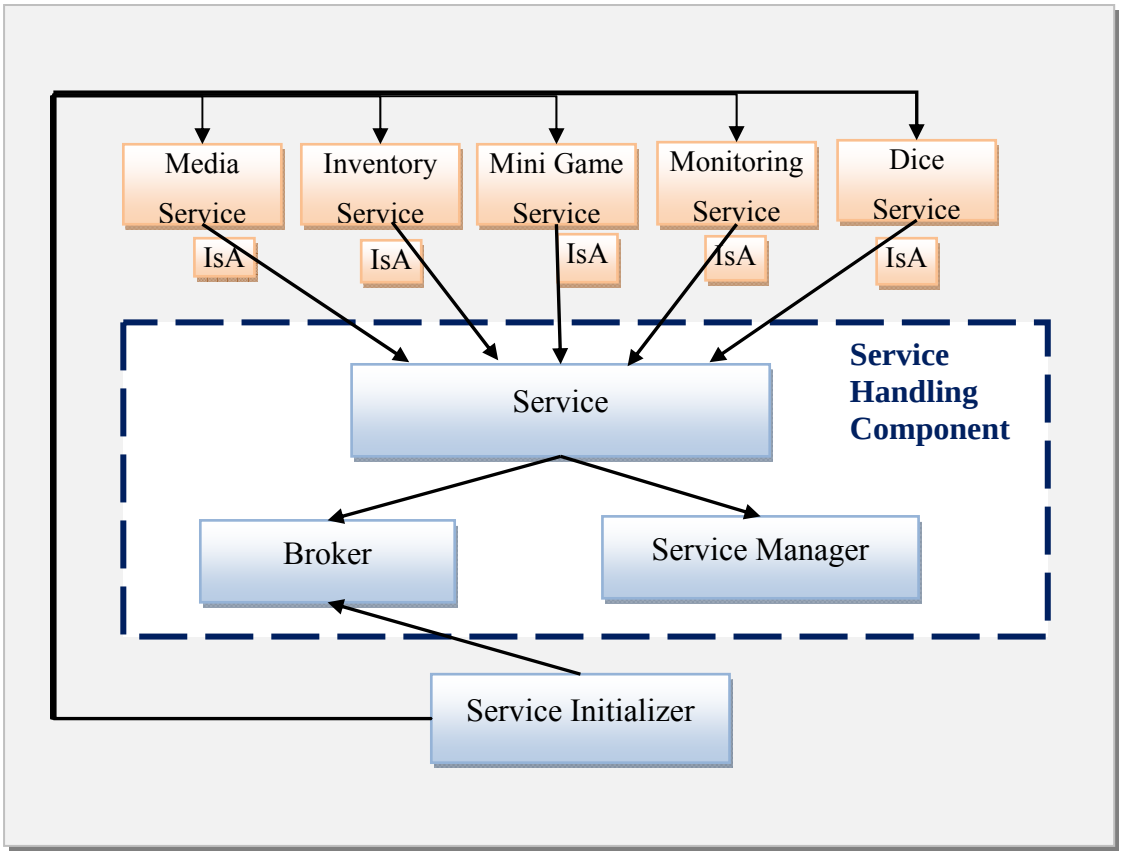

Figure 4 – Service handling mechanism. The Broker takes a message from the web containing a services id and a constructor function, then constructs and links the corresponding service to the sender dll.

The latter forces all services in the list to serve their pending messages tactically (on every few Amby's main loop). By following this structure, it was achieved to have a reusable and scalable service handling component that can handle independent services efficiently.

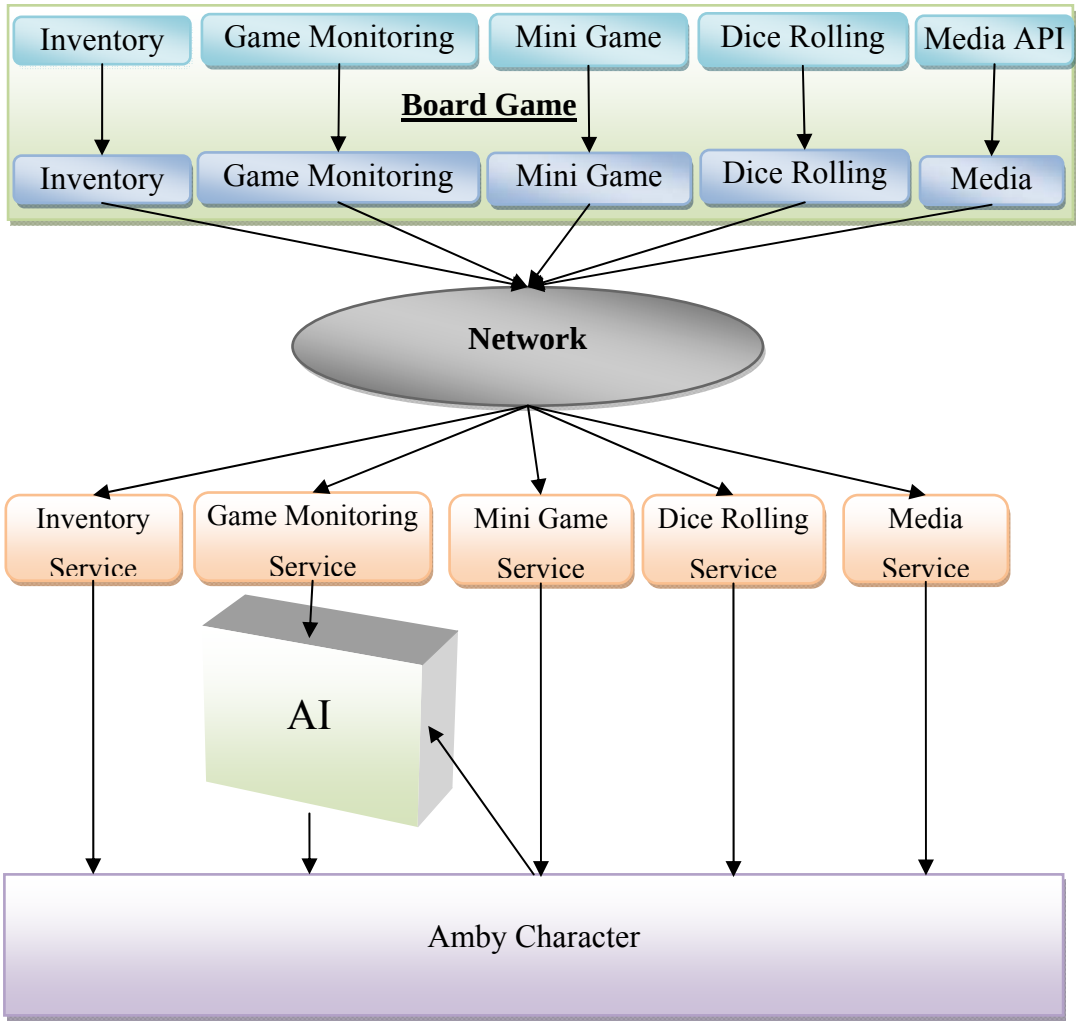

Figure 5 – Amby communication with the Dlls.

4.1.1 Inventory

As previously mentioned, in gaming, inventory refers to item storage available to a player or a character. It is used in the majority of games, whether they are board games or video/computer games. Inventory items usually have a name and an image, and they are accompanied by some text describing the items abilities and usages. Some items can be used only once, while others may be used infinitely. Players can carry a number of inventory items with them, and use them whenever they want.

Amby supports basic operations for the inventory, such as add, remove and select an item, while the players can navigate through their inventory when their inventory panel is open. Due to the lack of inputs devices, the board game is in charge of sending the "Show next" and "Show previous" commands to Amby. More

specifically, when the player selects to see the next item in the inventory, the board game sends a message to the Inventory Service, which uses the Inventory environment to propagate the notification. The Inventory Environment Delta Linkage then collects the event and sends it to the Amby's UI. There, the event dispatcher passes it to the Inventory component, which knows which is the next inventory item, and knows how to draw it on Amby's screen. The same procedure is followed when the player chooses to see the previous item in the inventory list, or chooses to show or close their inventory panel. When an item is added or removed from the inventory, the board game sends again the proper message to Amby's UI through the Inventory service. Additionally, it sends a notification event to the AI subsystem of Amby in order to comment the event. When the inventory panel is closed, items can be added, selected and removed from the inventory, but navigation is disabled. Add, selecting and removing events are the only actions sent to the inventory service, because it only keeps and displays the inventory of each player. Events such as lose or drop item are sent to monitoring service, because they need to be commented. For example, if a player drops an item, the board game sends a "remove item" event to the Inventory Service and a "Drop Item" notification to the Monitoring Service. When an item is used, the Inventory Service gets a message to update this item's usages, but it don't removes it automatically when the item is used up. It is up to the board game to send a notification to remove this item from the inventory list. The API provided by the Inventory service is shown in Figure 6. The functions "Is Visible" and "Get Selected" are used by the board game to get information about the inventory's status. The "Is Visible" functions return true if the inventory is visible, otherwise it returns false. The "Get Selected" function returns a string with the selected item's id.

```
class InventoryAPI {
public:
      virtual void Next         (void) = 0;
      virtual void Previous     (void) = 0;
      virtual void Remove       (const std::string& item) = 0;
      virtual void Show         (void) = 0;
      virtual void Hide         (void) = 0;
      virtual bool IsVisible    (void) const = 0;
      virtual void Select       (const std::string& item)  = 0;
      virtual const std::string GetSelected (void) const = 0;
      virtual void Add          (const std::string& item,
                                 const std::string& category,
                                 const std::string& description,
                                 unsigned timesOfUsage,
                                 const std::string& film
                                 ) = 0;
      virtual void UpdateUsages (const std::string& item,
                                 unsigned usages
                                 )= 0;
};
```

Figure 6 – Inventory API

4.1.2 Dice

Amby can be in charge of deciding and displaying the dice roll for each player. When the board game asks Amby to roll the dice for a player, Amby randomly chooses a number between one and six and displays the proper animation. Because the board game doesn't know when the animation ends, it keeps asking Amby about it and requests for the dice number only when rolling has finished. In this way, the board game only needs someone to decide a number between one and six and display it, and doesn't care about what the players see; it just asks for the resulting number on the animation's end. Thus, instead of dice animation, other "dice" representations can be made and supported, like a fortune wheel with six stages, or a scrolling wheel, or a wizard who magically presents a number between one and six. This would be very fancy and pleasant for the players. The API provided from the Dice service is shown in Figure 7. When a player needs to roll the dice, the board game sends a "Roll Dice" command to Amby's UI. Because the board game does not know when the dice animation finishes, it keeps asking Amby with the function "Roll Finished", which returns true if the roll is finished. Once the board game receives "true" (i.e. the rolling animation has come to an end), it sends a "Get Roll" command to get the dice number that was casted.

```
class DiceRollingAPI {
public:
      virtual void RollDice         (void)            = 0;
      virtual bool RollFinished     (void) const      = 0;
      virtual unsigned GetRoll      (void) const      = 0;
};
```

Figure 7 – Dice API

4.1.3 Media

Computer games are definitely enhanced by the use of multimedia. Cut scenes, sound effects and music tracks are widely used in order to develop an attractive atmosphere for the game. The story line is also conveyed by the use of multimedia. To address this need, Amby incorporates a mechanism for playing sound effects, music tracks and displaying images on game request. Additionally, the game developer can define Amby emotions to be shown, and provide text which Amby will reproduce using a text to speech mechanism. The media needs the same input arguments as Amby Actions. For example, to show an animation or an image, its id is required along with a vague position, but in order to show or read aloud some text, only the text literal is needed. For images and animations, information is necessary about their bounding boxes and their collision masks. That means that they cannot be dynamically displayed, but a preparation step need to be carried out using the Game Editor tool. Similarly, the animation definition files that describe animations need to be compiled into binary form in order to use them when showing animations. Therefore, all images and animations have to be prepared and loaded in special tables along with their corresponding data. Developers can use only their id names in order to use them. Sounds need to be registered in a configuration file with their ids along with the path where the sound file is located. The music files must also be registered in a configuration file similarly to the audio ones. Sound effects need to be in wma format, while playing a music file presupposes that it follows the ogg protocol. The above constraints are due to the use of the OpenAL library [18] for supporting audio files, and the Vorbis[1] library to support music tracks. The text to speech synthesis uses the

[1] http://xiph.org/vorbis/doc/Vorbis_I_spec.pdf

Flite library [19]. All these libraries were chosen because they offer a wide range of control functions for the sound, and they are open and free to use.

```
class MediaAPI {
public:
      virtual void Play (     const std::string& type,
                              const std::string& id
                         ) = 0;
};
```

Figure 8 – Media API

4.1.4 Mini Game

A Mini game is defined as a small game within a bigger one, usually optional, which tends to be very different from the main game. Mini games are often used to distract the players from the main storyline in order to keep them from losing their interest in the game. They are also good for offering special. They are commonly used as much in board games, as in computer games. For example, the "Go for Broke" board game uses a casino as mini game, and players have to play and lose all their money in order to win. Many board games also use the "Draw a card" mini game in order to make the game more interesting. A famous game example that uses cards as a mini game is "monopoly", in which commands and decisions are given not only via the game terrain, but also via drawing cards. In computer games, the use of mini games is also common; there even exist games that are entirely constructed of them (like "WarioWorld"). "Zelda" gives players puzzles to solve, "Donkey Kong" have mini game barrels which can be either hidden or not, "Mario World" uses bonus mini games and a slot machine and "Grand Theft Auto" is full of mini games for the player to choose and play.

In order to use a mini game in a board game concept, game designers were forced to build them from scratch for every different game. Amby supports a mechanism through which several mini games can be provided. Additionally, new mini games can be easily developed and added to Amby's mini game inventory. All mini games can be fully parameterized, in order to be flexible and adapt to any game concept with

minimal effort. In addition, Amby's mechanism for mini games includes sending game action notifications to the AI subsystem, so that the mini games can also be commented by Amby. This is achieved by sending event messages from the Amby's UI (which is in charge of the mini games). These messages will be collected by the Amby's AI component and be processed (see section 5.3) in order to extract the information needed for deciding suitable comments for each player. The decisions made are then sent to the Amby's UI environment to be executed (section 5.4). The commenting feature makes mini games a real enhancement, as it makes them more interesting for players to watch and play, and gives Amby the ability to change his comments or even his graphics to match the subject's player's profile information. Moreover, the board game does not need to know anything about how mini games work. When a player is about to play a mini game, the board game sends a "Play" request to Amby's UI and the latter takes charge of starting and supervising the requested game. As the board game has no information about what happens in the mini game, it keeps asking Amby if the game is finished using the "IsFinished" function. When this fucntion returns true, the board game asks for the mini game's result with the "GetResult" function (returns a string containing information about the game's outcome). Depending on the game, additional input from the board game might be necessary (because Amby is designed to be suitable for Ambient environments, no devices are attached in order to achieve total independence). In that case, the board game is in charge to provide the input needed to Amby, through the function "Output". For example, in the Hangman mini game Amby needs to get the letters suggested by the player. Because the board game needs to sent input information to Amby, it does not mean that it knows how the game is implemented. The mini game designer can interpret the string passed from the board game as input according to the internal mini game implementation, thus mini games remain independent from the board game (the latter may need to know the kind of mini game played, but it is no further interested about the mini game's implementation). The API provided by the Amby's character to the Board game regarding the mini games is described in Figure 9.

```
class MiniGameAPI {
public:
      enum Difficulty {
            EASY  = 0,
            MEDIUM      = 1,
            HARD  = 2,
            ANY         = 3
      };

      virtual void  Play (unsigned player,
                        const std::string& game,
                        Difficulty difficulty,
                        const std::string& category) = 0;
      virtual bool  IsPlaying  (void) const = 0;
      virtual bool  IsFinished (void) const = 0;
      virtual const std::string GetResult (void) const = 0;
      virtual void  Output     (const std::string& str) = 0;
};
```

Figure 9 – Mini Game API

4.1.5 Monitoring

The Monitoring Service enables the board game to feed Amby with every game event that the game designer considers necessary. Amby can then use these notifications to keep a history of all players' actions and extract information about the game state and players' progress, so that knowledgeable comments can be made about it. All the events that are sent to other services are not used for commenting purposes. Consequently, if an event occurs that needs to be both commented and displayed on Amby's screen, the board game needs to sent two messages to Amby, one to the corresponding service that will be sent to Amby's UI and another to the monitoring service in order to be commented. For example, if an item is added to a player's inventory, the board game need to send a message to the Inventory service and a message to the Monitoring service. These events may not be the same, since the services are independent from each other and may be interested in different kinds of information. For example, if a player loses an item, the board game sends a "lose item" event to the monitor service, but it sends a "remove item" event to the inventory service, because the inventory is not interested in the reason why the item is removed. The API provided by the monitoring service is shown in Figure 10.

```
class GameMonitoringAPI {
public:
virtual void NotifyStartGame        (unsigned numOfPlayers);
virtual void NotifyGotoNode         (unsigned player,
                                     const std::string& nodeName,
                                     const std::string& description,
                                     bool stopMovement);
virtual void NotifyScoreChange      (unsigned player, int score);
virtual void NotifyExtraTurns       (unsigned player, int turns);
virtual void NotifyInitialLife      (unsigned player, int life);
virtual void NotifyLifeChange       (unsigned player, intlife);
virtual void NotifyInitialInventoryCapacity (unsigned player,
                                             float capacity);
virtual void NotifyInventoryCapacityChange  (unsigned player,
                                             float capacity);
virtual void NotifyVariableChange(const std::string& variable,
                                  const std::string& value);
virtual void NotifyTerminal(unsigned player, bool win);
virtual void NotifyGetItem (unsigned player, const std::string& item,
                            const std::string& category,
                            const std::string& description,
                           unsigned timesOfUsage,bool playerInitiated);
virtual void NotifyLoseItem(unsigned player, const std::string& item);
virtual void NotifyDropItem(unsigned player, const std::string& item,
                            bool playerInitiated);
virtual void NotifyUseItem (unsigned player, const std::string& item,
                            bool playerInitiated);
virtual void NotifyNativeFunction(unsigned player,
                                  const std::string& function,
                                  const std::string& data);
virtual void NotifyDeltaFunction (unsigned player,
                                  const std::string& project,
                                  const std::string& script,
                                  const std::string& function,
                                  const std::string& data);
virtual void NotifyMiniGameStart (unsigned player,
                                  const std::string& game,
                                  unsigned difficulty,
                                  const std::string& category);
virtual void NotifyMiniGameEnd   (unsigned player,
                                  const std::string& game,
                                  unsigned difficulty,
                                  const std::string& category);
virtual void NotifyMultimedia    (unsigned player,
                                  const std::string& mediaType,
                                  const std::string& mediaId);
virtual void NotifyDiceRoll      (unsigned player, unsigned dice);
virtual void NotifyPossiblePaths (unsigned player, unsigned paths);
virtual void NotifyDoorFail (unsigned player, const std::string& door
                             const std::string reason);
virtual void NotifyDoorPass(unsigned player, const std::string& door);
virtual void NotifyCrossPointPass(unsigned player,
                                  const std::string& crossPoint);
virtual void NotifyTurnBegin     (unsigned player, unsigned turn);
virtual void NotifyTurnEnd       (unsigned player, unsigned turn);
virtual void NotifyScheduleEvents(const std::string& type,
                                  unsigned amount);
virtual void NotifyTogglePause   (bool paused);
};
```

Figure 10 – Monitoring API

4.2 Avatar AI

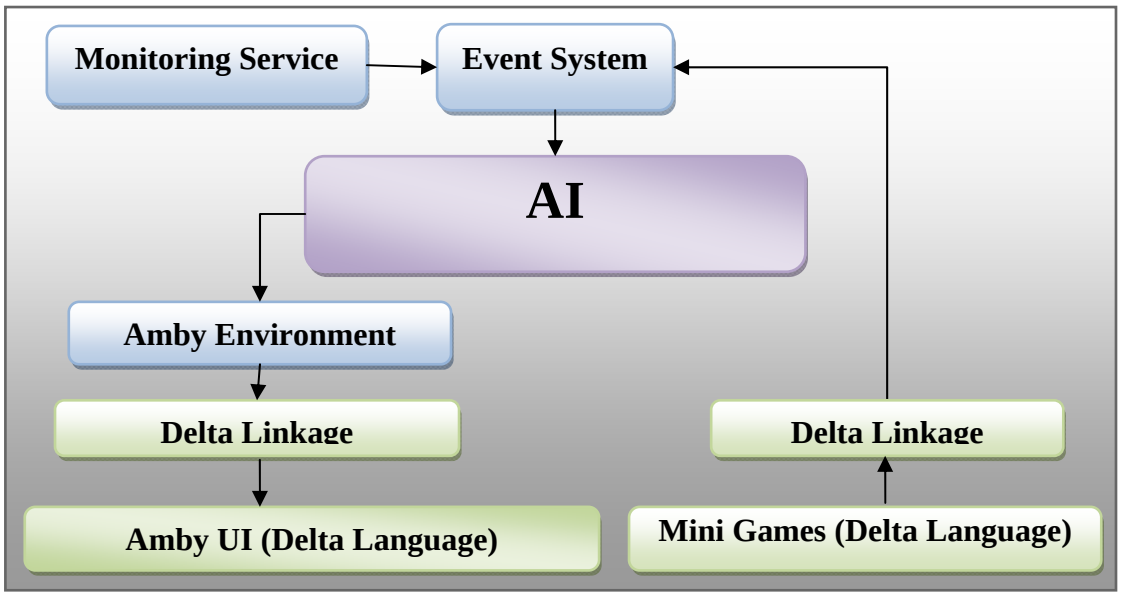

Figure 11 – AI system - Macro Architecture

The two major requirements for the development of the AI component of Amby were extensibility (any game designer can supply logic rules) and adaptivity (Amby's decisions made using AI logic must be socially adaptive). The underlying intention was to support automatic selection of the best-fit comments for a specific player given their personal profile and their progress in game at a given pint in time. In order to achieve such behavior, a set of rules was define containing information about game status, player progress and player's social profile. These rules are separated from the rest of the code in order to be extensible, and evaluated for each user, resulting in the activation of the appropriate commenting behavior each time a game event needs to be commented. The Decision-Making Specification Language – DMSL [1] was used to implement this component. DMSL is a domain specific rule-based language for making decisions on activation / deactivation of components during the run-time of a system. It is based on a compiler that loads and compiles files containing DMSL adaptation rules, and an interpreter that evaluates the compiled rules based on the profile of each player. The output of the decision making process is a set of activation and deactivation commands that can be interpreted by the system to perform the appropriate adaptation actions. Specifically for Amby, the DMSL compiler and interpreter were incorporated, providing a set of logic rules for social adaptation of

comments and per players' profiles that contain attributes describing their current progress in game together with attributes regarding their social profile. By compiling these rules per player, Amby gets a list of resulting decisions– i.e., a list of actions that the Amby avatar must perform in order to comment a game event properly, in a way adapted to the player's profile as well as the current game circumstances. An example of a DMSL file that contains rule based logic for a specific event is displayed in Figure 14. DMSL rules are based on an if-then-else structure, thus it is very easy for game developers to add their own DMSL rules, or modify the existing rules in order to match their game perfectly. A macro architecture sketch for the AI component is displayed in Figure 11.

4.3 Avatar UI

Amby is an ambient presence avatar that can be displayed on a screen above the board game table, or projected on a wall. The scene projected is a game scene. The Amby UI defines the artifacts to be displayed on the screen and uses a Game Engine in order to do the rendering. As displayed in the macro architecture of the Avatar's UI component (Figure 12), the UI gets input commands from all services except monitoring, as well as the actions commands that source from the AI component. The services that interact with the UI component directly, without being commented are:

- *Inventory Service*. Commands the UI component to handle and display the inventory items for each player.
- *Media Service*. Delegates Amby to display images, video and animations and to play sound effects and music. It also delegates Amby to show and read a text line to the players.
- *Dice Service*. It orders Amby to roll the dice. Amby's UI component shows a dice animation and when it finishes it notifies the board game about the number casted.
- *Mini Game Service*. The board game uses this service to delegate the Amby's UI component to start and supervise a mini game. In order to support commenting during a mini game, the UI component sends notification events to the AI component using the event system. These events are further

processed, and commenting actions are decided and sent back to the UI component as actions commands (the UI component don't have any information about who asked for these actions to be executed).

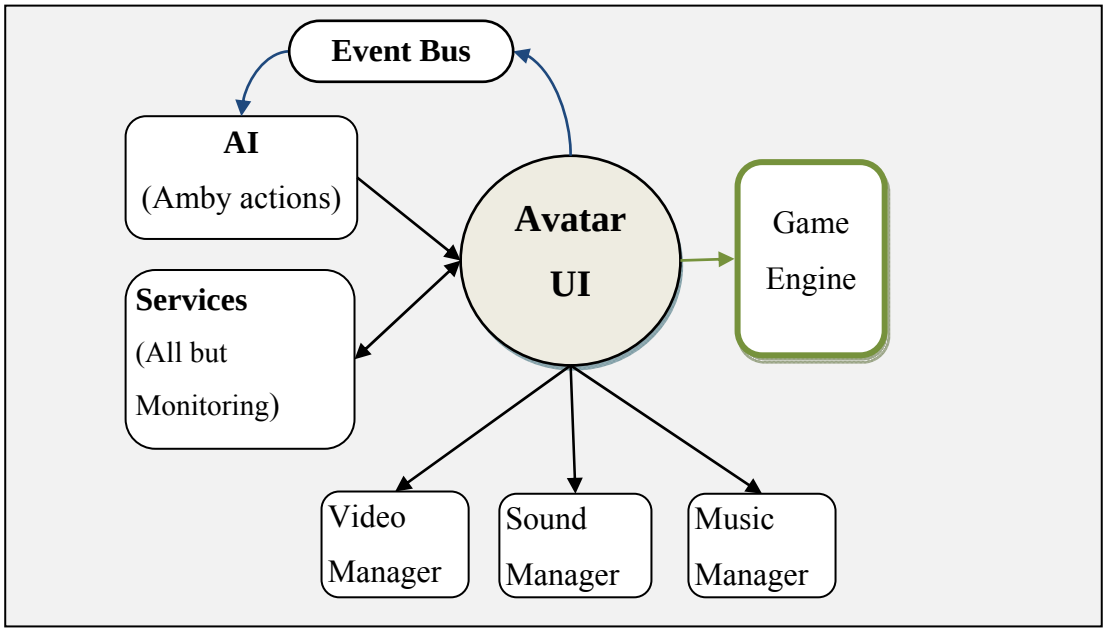

Figure 12 – Avatar UI macro architecture

5. Avatar AI

One of Amby's characteristics is the socially-oriented game narration. This feature is fundamentally different from static game narration, where comments to be presented for every event are choosen from a list. Therefore, Amby needs to have a brain. In this sense, a large Artificial Intelligence component was developed, which is able to collect and process information about the board game and the players, and use the knowledge extracted in order to make valuable comments. As in most AI systems, Amby's intelligence is based on a Sense - Think - Act model. In Amby's case, the sensing component receives notification messages from both the board game (Monitoring Service) and the Mini game (Amby's UI). The same mechanism is used in both cases to inform the AI component of in-game player actions. After sensing, the thinking component needs to elaborate the gathered data, filter them if needed and process them again combined with the target player's history to extract which of their profile attributes need to be added, changed or removed. Once the target player's profile is updated, the AI component uses the DMSL language to evaluate the logic Rules, which are located in a standard set of files written in DMSL language. When evaluation is finished, a list of activation statements is returned to the AI. Each statement refers to an action that the Amby's User interface environment needs to perform. These actions are adapted to the target player's social profile, taking into account the current game state and the player's progress. Subsequently, the actions are given to the acting component, which is responsible for passing them to the UI component. The UI component of the Amby character is developed in Delta language, thus the acting component has to link the actions to it. In the following section, a detailed description of each AI part is given.

5.1 Detailed Architecture

Monitoring Service receives data for every game event in xml format. After extracting all information needed from the xml message, monitoring constructs an event message notification containing these data and sends it to the AI subsystem. The AI subsystem consists of a) Data holders containing the current game state and history data per player that represent their progress in the game, b) a processing component

that elaborates game events combined with player action history to extract the attributes to be set in player's profile c) DMSL rule files and the DMSL component that performs the evaluation of these rules taking into account only the subject player's profile and results in a list of decisions that represents the actions that have to be fulfilled by Amby and d) a component that elaborates these decisions and pass them properly to Amby's environment. The Amby's environment can then convey these messages through the delta linkage subsystem to the Amby's UI. There, action messages are refined into images, sounds, music, animations and expressions, and are finally displayed on Amby's screen using the ALADIN graphics library [15]. In the following sections, Amby's AI is analyzed into three mandatory steps: Sense, that is the input data passed from the board game itself, Think, that includes history keeping, data processing and dmsl rule evaluation and analysis of the decision taken, and Act that concentrates on passing these rules to Amby's UI and their conversion into graphics, using the ALADIN [15] library through the DELTA [2] language.

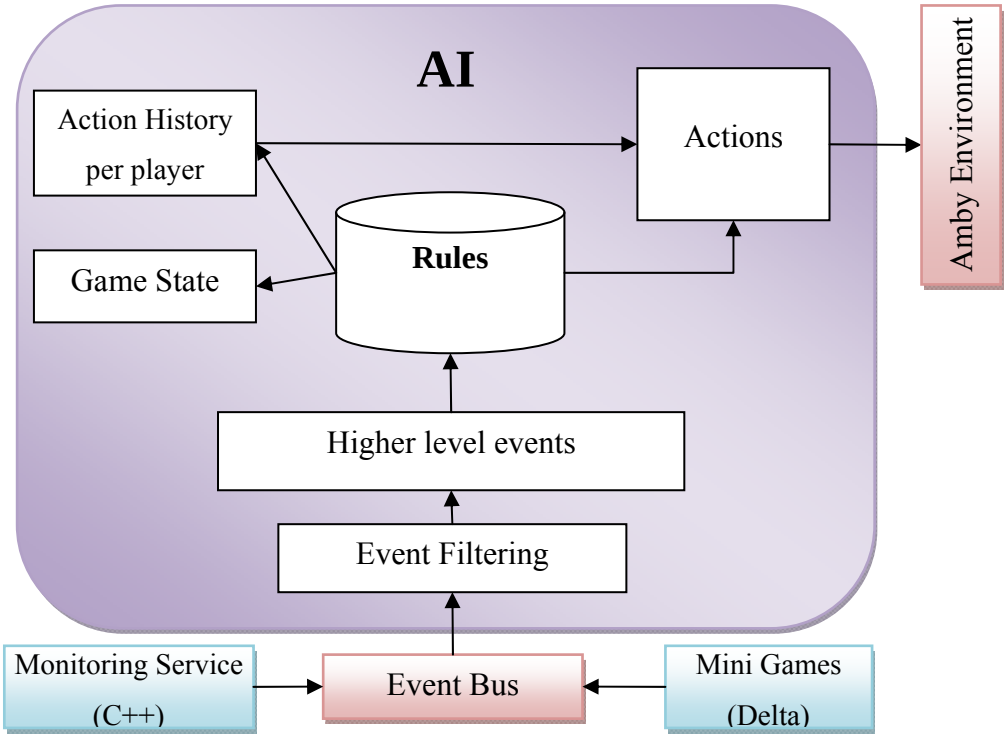

Figure 13 – Internal Architecture of the AI component. The sensing part receives messages from the event bus, while the acting part links the actions selected from the thinking part to Amby Environment.

5.2 Sense: Filtering Semantic Variables

Only two Amby components are able to send events to the AI subsystem, the Game monitoring component, which refers to all events sourcing from the board game itself, and the UI component that hands the Mini Games.

The Board Game sends through the game monitoring dll information messages to the corresponding service using the xml format. Since information can be extracted from the xml message about the type of the message arrived, the monitoring service can further extract all the expected parameters for each game event and pack them into an event notification message sent to the AI component. For example, a "Player Get Item" event must contain information about the player's id, the items id and it's category, it's description text, a number representing the number of times that the specific item can be used, and a Boolean flag indicating whether the player asked for it or not, while a "NotifyExtraTurns" event contains only the subject player's id and the number of turns gained. All the event types must contain information about the time at which they occurred. On the other side, the AI component contains several handler functions, each one specialized in the processing of a specific event. When such an event notification arrives to the AI subsystem, the corresponding handler is called in to handle it. The variables contained in events are game specific, as they come directly from the board game. These variables need to be transformed to a more generic form, so that AI rules do not need to be aware of the game itself. To do this, a filtering mechanism is needed to translate the specific variable's value into a more generic term. For example, a score gain of 100 points is very small if the average score gain is expected to be around 1000 point per round, but it is very good if the average score gain expectation is around 10 points per round. Filtering is performed through DMSL rules. DMSL was chosen for this purpose so the filtering rules are code-independent and easy to be extended by the game designer. The filtering process then needs to be automated. A specification file is used to define which variables need to be filtered, and which filtering rule is specified to do that. Then, for every incoming event, it is checked if any of its attributes needs to be filtered, and if that is the case, the delegated rule is evaluated. The result of the rule contains the filtered value necessary to set back to player's profile. After this procedure is completed, all profile

variables have non-game specific values, and can be used for drawing generic conclusions for the game state and the players' progress.

The procedure followed when Amby wants to comment on a mini game is similar. The difference is that the events arrive to the Event Bus source from Amby's UI subsystem instead of the board game. Consequently they do not pass over the network like monitoring events. By the time they reach the event bus though, their source is no longer relevant. The same mechanism is followed – the appropriate handlers are called which decide the attributes to be set in players' profile, along with their values filtered or not.

Amby also has a mechanism to identify when a player is inactive for a long period of time. The services system (including monitoring) checks periodically if new messages have been received from the game in order to serve them. In Monitoring, when such messages exist, Amby serves them and keeps the time at which the last message was served. If a long time has passed without receiving a message from the board game, Amby understands that the current player is inactive and sends an "Idle" event to the event bus. When a player is inactive during a mini game though, Amby cannot notice the inactivity because it keeps receiving messages from the board game asking if the mini game has finished. To solve that problem, the mini games are handled separately. The same idea is used here too: inside the Mini Game Service information is kept regarding the last time at which some game event occurred (i.e. which is the last time a letter was suggested by the player). So when a mini game is in progress (a "Play" request has arrived but no "Get Result" has arrived yet) and Amby notices that a long time has elapsed since the last interaction occurred, it sends an "Idle" event to the Event bus. When the AI component receives such an event, the procedure described above is followed, in order to evaluate the "Idle" rule and take the proper decisions.

5.3 Think: Deciding Adaptive Actions

In order to make the AI logic in Amby flexible and extensible, the DMSL language is used. Our decision is based on several important characteristics that this language

features. DMSL uses separate profile files for each player defining their personal attributes, such as name, age and profession. These profiles are further used in order to decide personalized actions for each player. Except from players' profiles, DMSL language use code-decoupled files in which all logic rules are defined. These logic rules follow an if-then-else structure, which is close to the way human think. This makes the rules easy readable and comprehensive, thus allows to various programmers to describe complex logic rules in an efficient way. These rules like profiles are described in text files outside the program's code, so changes in them do not affect any other software component, and thus no compile is required. This means that programmers can change the logic rules frequently and without affecting the rest of their code. Rules are then evaluated for each player separately, taking into account their personal profile, their game progress and the current game status. The outcome decisions are formed as activation and cancelation commands regarding arbitrary software components (in our case Amby actions). Therefore a wide range of contexts can be supported. Finally, rule evaluation in DMSL is extremely fast, making it an appropriate choice for describing AI logic in gaming.

An alternative choice for describing Amby's logic would be the well-spread language of Prologue. In Prologue, program logic is expressed in terms of relations, and execution is triggered by running queries over these relations. Because this is not the way in which people think, prologue rule files are much more difficult to be read and changed by a game developer, thus the system's extensibility is reduced. Moreover, the way prologue evaluates the rules (backtracking) makes it slow, and thus inappropriate for games with rapid action. Weighting the characteristics that the two languages (DMSL and Prologue) feature, we concluded that DMSL is more appropriate for our system, as we need logic rules to be evaluated as fast as possible and to be easily readable and changeable by any programmer. Moreover, DMSL supports separate rule evaluation for each player, taking into account their personal variables.

To take decisions about Amby's reaction to an incoming event, all rules that correspond to that incoming event are compiled, taking into account the target player's profile parameters. For this purpose, the player's profile needs to be filled with parameters that reflect the player's action history and the current game state, in addition to the player's personal information already contained in the profiles (age,

name, profession etc). Each event specifies a standard list of attributes to be set in the player's profile and their values. The latter will be decided at runtime as they depend on the current game state and the player's history of actions. Except that the variable set need to be adequate for an AI designer to use, they must also be fresh and represent facts happened recently in the game. In order to do that, Amby needs to "forget" all events happened before a specific amount of rounds. Therefore, all profile attributes that have not been updated in the last five rounds are removed from the player's profile. This mechanism encourages the AI programmer to use as many profile attributes as they want, because it guarantees that all of them are recently updated and thus the resulting actions will always be valid.

Every incoming event corresponds to a handler able to find and process the available data regarding this event. The outcome of this process is a set of attributes and their values which must be set to the player's profile. The set of variables for each event is predefined, so the handler function needs to go back to the player's history and decide the value each attribute must have. For example, for the "Roll Dice" event, the set of variables to be set for a player is a dice characterization (small, medium or big – can be used for generalization in cases where the possible throws differ from one to six), the actual dice casted (for example 3), if it is the first dice the player castes, how many same rolls have been casted in a row, the player's average dice so far, if all the rolls casted the same number and the amount of rolls they have casted in total. The latter variable is necessary in order to value the attributes referring to player's average dice and whether all the rolls casted by the player were the same. For example, if it is the second time that the dice is casted, a comment such as "your average dice rolls are good" has no sense. The same holds for a comment like "all the dices you rolled were 5" if the player has only thrown the dice once. After that, the player's history is updated, so that the new data will be available for use in the player's next round. By passing all the knowledge available to the player's profile, the rule designer is enabled to combine several variables in order to have a spherical perspective about what is happening to the game. Although the parameters are not extensible, the set of variables provided for each game event is adequate for the rule makers to use, as it represents the knowledge about this event. Rule makers can also check if an attribute exists in the player's profile, which gives them the ability to combine several variables depending on their existence. By combining them with player's actual

profile parameters, and having the ability to exploit all profile attributes (including attributes which have been set by other events), the DMSL rule designers have the ability to build an integrated set of both simple and more complicated rules for use in the Amby's AI system. For example, if a player fails to pass a door and their inventory contains a small amount of items, Amby can say something like "With such a small inventory how could you expect to have all items needed to pass this door?" Of course there can be many rules that need to be evaluated when an event occurs. One can define which rules are to be evaluated each time an event occurs. For this purpose, a configuration file is used, linking the basic attribute of each event to a list of rules that need to be evaluated. At runtime, the AI system checks this list and triggers the evaluation of all respective rules. The results of all these rules are gathered in a list which is further processed in order to extract Amby's actions.

Social player's data, such as their name, their age and their profession Available are also for the rule maker to use. Other variables may also exist describing a person's disability, such as blindness or deafness. In that case, the DMSL rules can involve a statement that check if a disability parameter is set and select accordingly the action that must be performed by the Amby character. For example, if a person has vision problems, Amby can adapt so that sound effects are selected instead of images and animations. The DMSL rules can use all the parameters existing in the profile (checking if a parameter exists is supported) not only for selecting the action to be done, but they can also use the value of an attribute to parameterize the text that amby is to display. That means that the player's name can be used in a statement, making the message targeted to them. The use of player's personal data combined with history information about their progress and information about the current game state, provide a wide range of capabilities for the rule maker, who can give Amby character a unique personality, evolving him into a pleasant companion for the players and a shrewd game commenter. To give an example of the adaptable comments made by Amby, consider a player who is trying to open a door and fails. A generic comment message would be "It's a shame you cannot pass the door". But an Amby's message can be enriched by player's name and age, and the door's name can be displayed too, like "***John***, it's a shame you cannot pass the ***Big Boss Door*** although you are***18*** years old!"

Except from displaying a mere text message on the screen, Amby can perform other actions too, such as displaying an image, playing an animation, play a sound effect or a music file (using respectively OpenAL [18] and Vorbis libraries) and can use a text to speech synthesis tool(using the Flite library [19]) in order to speak. The ability of reproducing media other than text, alone or in combination with each other, is a very important characteristic, since it boosts the commenting capabilities of Amby. Of course, all media must be carefully chosen according to each player's profile.

Also, as previously mentioned, Amby notices it when a player is idle for a long time and sends an "Idle" notification message to the AI system. The AI component then adds the "Idle" attribute to the current player's profile and triggers the evaluation of the "Idle" rule. The evaluation will result in adapted cues that will be sent to Amby's UI to perform. By letting the logic rules decide what to say and when, Amby can behave differently for every player who's idle. For example, if the game has only one player, a message such as "take your time" can be displayed, while if there are other players waiting, a message like "Hurry up, all the others are waiting for you" can be selected.

Finally, facial expressions can be used in order to convey Amby's emotions. This gives the ability to the rule designer to develop a personality for the Amby's avatar, making him react to user actions. Depending on his emotions and his comments, Amby can be sympathetic, or ironic, or he can adopt the behavior given to him by the AI designer. Amby's behavior can also be defined by a profile parameter for each player. For example, one can define that they want English humor for themselves, while other may want Amby to reflect their emotions. By considering the specified profile parameter Amby can be programmed to react differently for each player, saying different things and showing different emotions. It is a very important characteristic of Amby that he can impersonate a commenter with personality and emotions. It is also very important that any game designer is able to control Amby's logic and behavior only by changing some parameters and rules in the DMSL files, without having to change even a single line of code. That makes Amby a universal board game commenter that can automatically adapt to the game and its players. It is believed that the provided events and functions can cover the majority of board game needs, however, in the case that some rules turn out to be inadequate, the game

developers can add their own rules, which can be further used in other games, and so on. The results of the rules are activation statements. Each statement contains a string with an identified keyword followed by arguments separated with comma. The statements are:

a) ***Verbal***: It is used when Amby needs to say something. It is followed only by the text that Amby displays.

b) ***Image***: It is used when Amby needs to show an image. In that case, the rule designer has to provide the Image's unique id, the position on the screen where to show it, and the size of displaying. Positioning on the screen is defined by a string describing the vague positioning, like "top" or "center", and Amby follows a specific procedure in order to decide the exact positioning. Size parameter can take abstract values like "small" or "medium" to define the size of the image to be displayed. Depending on the value of the size parameter, Amby must decide on image's enlargement or shrinking in order to display it properly.

c) ***Animation***: By activating this command, the rule maker decides that an animation must be played on Amby's screen, and must provide the specific animation's id. Abstract positioning is also required in this case.

d) ***AmbyAnimation:*** Amby Animations is currently used to convey Amby's emotions. It is called Amby Animations because it commands Amby to play an animation on his own facial characteristics, in order to express his feelings. For this purpose, a string with the emotion's id is expected.

e) ***Audio:*** Amby is able to support playing sound effects. The rule designer is only expected to define the audio track's id.

f) ***Music:*** Similarly to Audio, the music command only needs a track's id to be provided by the rule maker.

g) ***Video:*** Asks Amby character to play a video file. It needs only the video id to be defined.

All selected actions are further passed to the acting part of the AI system in order to be sent to Amby's UI component.

Figure 14 presents an example of DMSL rule. As depicted in the figure, these rules are very simple and readable, and can be easily extended . Some of the player's parameters are used in order to personalize the decisions taken, while the rule maker

can extract valuable information about the game state and the player's progress by observing the values that the player's profile parameters have.

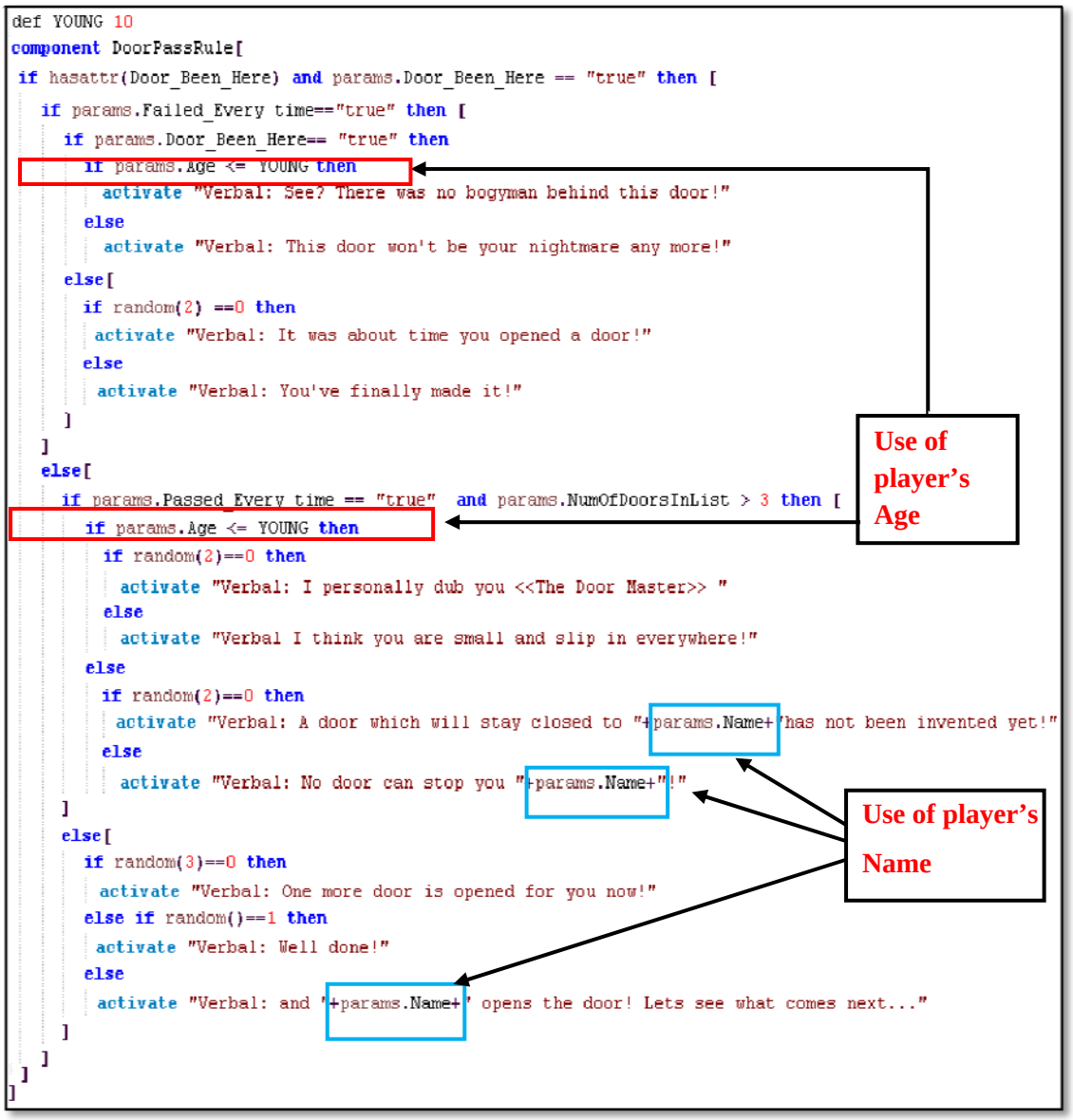

Figure 14 – Example of DMSL rule File. Notice how player socialized profile is used in combination with game progress attributes to compile an adaptive behavior for Amby.

5.4 Act: Linking to UI Actions

In order for Amby to actually perform the decided actions, an environment is required providing the actions functionality is provided, as well as a linkage step between the two languages, C++ and Delta. For the Amby AI subsystem, the corresponding components are the Amby actions environment and Amby the Actions Environment

Delta Linkage. Because the AI component needs to be independent and unaware of the language used for implementing the actions decided, the Actions Environment component has to provide all the necessary functionality. In the current implementation, the Actions Environment just posts an action request, being unaware of who will collect these requests and implement the actions. The component which collects these requests is the Actions Environment Delta Linkage. This is the component that actually links C++ with Delta. It needs to be aware of both the Amby Actions Environment and the Delta Virtual Machine.

Function calls are performed as follows: In Delta a dispatcher function is used to call another delta function depending on the first parameter of the former. The rest arguments are passed directly to the dispatched function. The dispatcher function is then passed to the C++ component as the handler function that the action functions need to call. So, when an action request is collected by the actions environment delta linkage component, the action's handler calls the dispatcher given by delta, passing as arguments the name of the action to be done, followed by the rest of the arguments. In this way, the Delta dispatcher knows from its first argument which delta function needs to be called, and calls it passing the rest of its arguments. This implementation provides independency among the components, making it possible to change the way – even the language in which the Amby's UI is implemented without affecting the AI component.

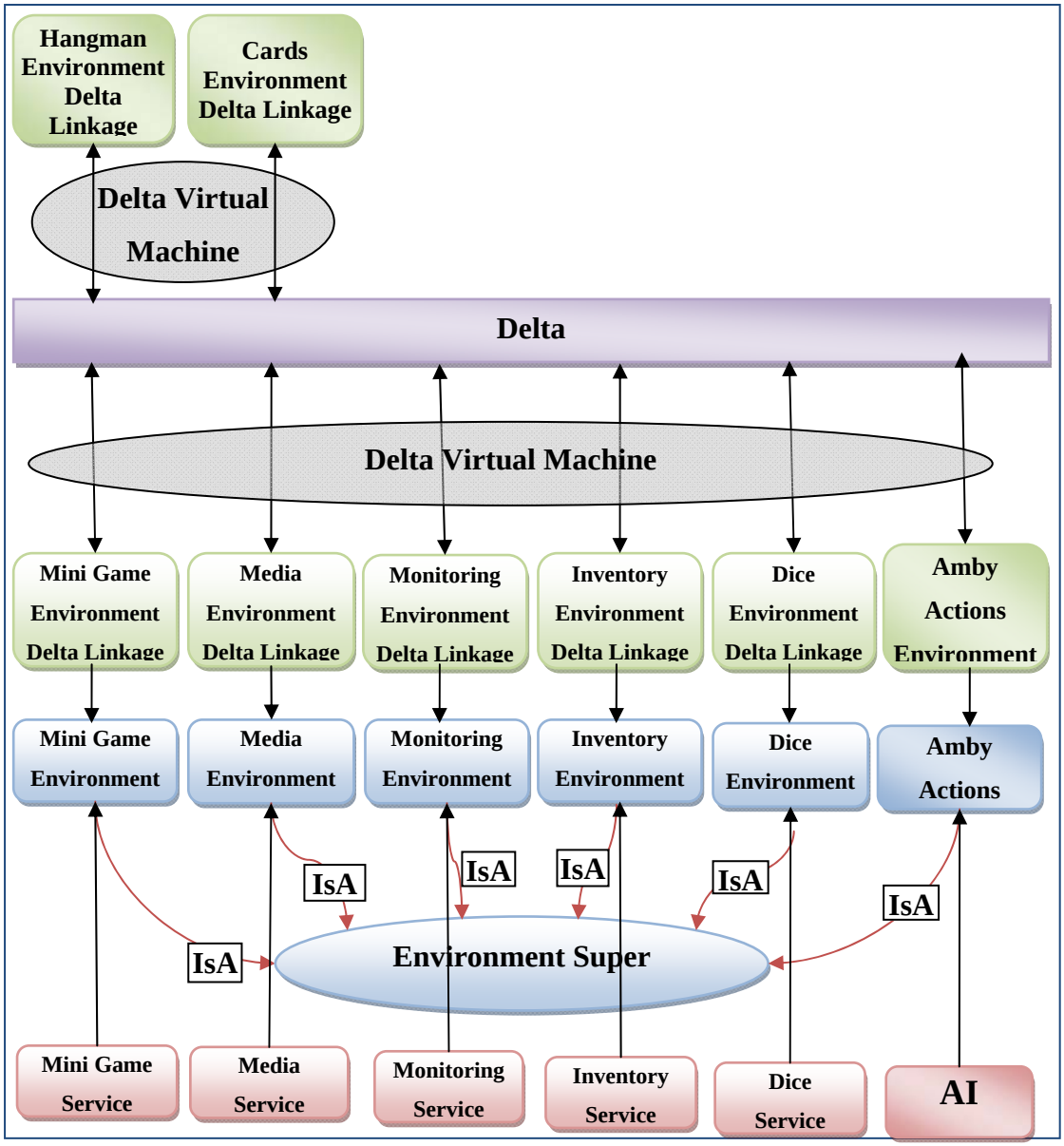

Figure 15 – Linkage architecture.

displayed on the terrain (Amby's characteristics), and their exact positioning can be adjusted by using the editor's visual tool.

This preparation procedure must be followed for every image and animation film to be displayed (an image is an animation for only one frame, for which no animation needs to be defined). Therefore, the images, the animations and the animation definition files have to be loaded at Amby's startup. Film tables are used for binding animation and image ids to their data, and animation tables are used in order to keep the animation definition files bound to an id name to be used when an animation has to be applied.

Upon Amby start-up and before the game loop starts running for the first time, an initialization handler in C++ is called which initializes Amby's data. After that, every Service (written in C++) must check if new messages from the board game have arrived in order to serve them every few game loops. This is accomplished by calling a C++ function at the end of every game loop which every once in a while forces the broker and the services to check for pending messages and serve them, if there are any. With this mechanism, it is ensured that Amby is properly initialized before the game loop starts, and that new game requests will be served every few game loops, upon checking if new messages exist to serve them. Serving the requests means propagating the messages to the Amby's UI for the Inventory, Media, Dice and Mini Game Services. For the Monitoring Service, serving a request means transforming it into an event message (though the event system), which the AI subsystem will receive in order to decide the proper adaptation actions. The AI component will then propagate the action decisions to the corresponding delta linkage component which will send the action requests to the Amby's UI. As mentioned in previous sections, the language selected for the UI development is the *Delta* language. The development was conducted using Sparrow ([16], [17]), an Integrated Development Environment (IDE) developed for *Delta* language. In the following sections describe the UI's architecture and explain how each UI component works and how the communication among Amby UI, Game Engine and Amby Services is performed.

6. Avatar UI

This section analyzes the Amby's Avatar User Interface and explains the actions that are performed in order for the UI to execute a game command. The user interface for the Amby character is a game scene displayed on a screen (Amby's screen). Therefore, everything which is displayed is handled as a game object. Amby is the main game character, images are sprites, and animations need their animation films, and so on. The avatar UI component is in charge not only for everything displayed on the screen (including the Amby's avatar), but also for the mechanisms used to execute the commands received from the supported Services. The Delta language was used to develop these mechanisms, and the Game Engine [15] (which uses the ALADIN [15] library for graphics), was used for handling and displaying the graphic elements. The Game engine is also in charge for executing the game's main loop. In order to give the game data in a suitable form to the game editor to read, a Game editor [15] tool was used.

Some of the Game Engine's basic elements are:

- ***Animation Film:*** Contains a bitmap with all the different states of an animation (frames). It also contains information about the bounding box of each frame.
- ***Sprite:*** Contains an animation film, a frame index (which frame to display), and a position (on the screen) to be displayed.
- ***Animation:*** Contains information about a sprite's movement and frame change.
- ***Animator:*** Contains a sprite and an animation, and applies the changes and the movements that are described in the animation onto the sprite.

Once a bitmap with the animation's frames is available, the Game Editor is used to extract information about their minimum bounding boxes and save them in an "expanded" file. The Game Editor is also used to transform the animation definition files (adf) into binary form, so that the Game Engine can read them. Animation definition files are files that contain the description of a specific animation, and bind that animation to a unique name. Moreover, a set of initial sprites can be defined to be

6.1 Detailed Architecture

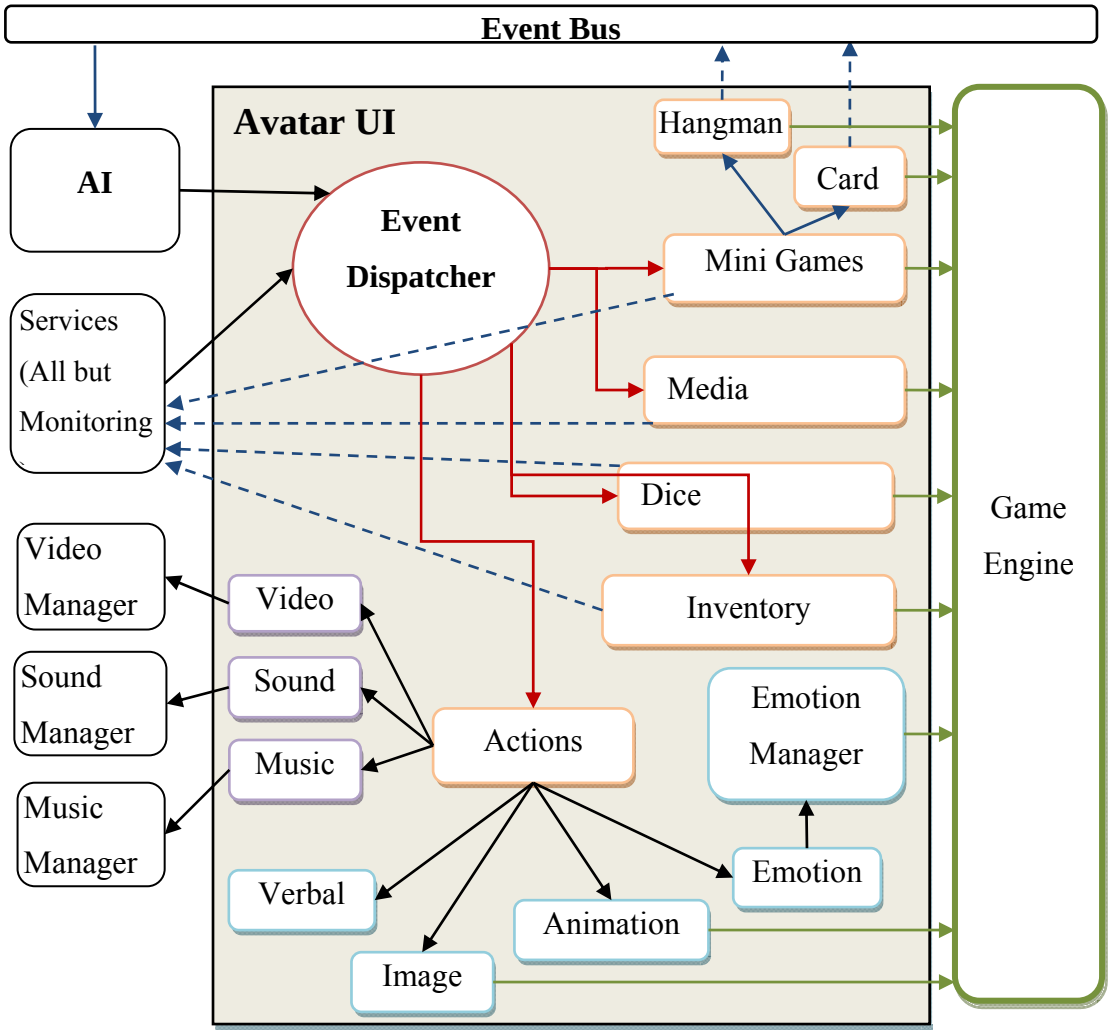

Figure 16 – Avatar UI architecture (detailed)

The Avatar UI component consists of the following subcomponents:

- An Event dispatcher, which takes the incoming commands from the services and dispatch them to the handling components
- An Inventory component which is charge of handling and displaying the events coming from the Inventory Service
- A Media component which is charge of handling and displaying the events coming from the Media Service
- A Dice component which is charge of handling and displaying the events coming from the Dice Service
- An Actions component which is charge of handling and displaying the actions coming from the Amby Actions Service. This component is actually

in charge of game commenting. Its subcomponents (Image, Sound, Video, Animation etc) are also used from the media component.

- A Mini Game component which is divided in two subcomponents, Hangman and Cards, each one dedicated to handling the corresponding mini game. It is generally in charge of all messages sourcing from the mini game Service.
- An Emotion Manager which implements the emotion scheduling mechanism and is in charge of handling and displaying Amby's facial emotions. It receives data from every component when an emotion needs to be shown.

The events that UI component receives as input come from the services Inventory, Media, Mini Game and Dice and from the AI component. The latter uses the UI subsystem in order to display the adaptive comments and the animated cues that have been decided using the rules. The actions that the UI component is called to perform are "Verbal", "Show Image", "Play Sound", "Play Music", "Play Video" and "Show Emotion". The actions that need to be rendered on the screen are supported by the Game Engine, while the sound, music, video and speech synthesis commands are handled by external libraries (which had to be exported to the Delta language first). The emotions are passed to a separate component, the *Emotion manager,* which contains logic about scheduling the animations and knows exactly which actions need to be done in order to display an animation correctly and how to return Amby's face to the "happy" state again. As far as the other services are concerned, the Inventory Service uses the Amby's UI system in order to handle the displaying part of the inventories of all players. This means that Amby decides the way inventory looks, while it provides simple functionality such as Show, Hide the inventory panel, Add, Remove or Select an inventory item and Update an items usages. The Dice service is in charge of displaying an animation that represents the dice roll and returning the random number that was selected back to the board game. It can support different dice implementations such as wheel, dice, magic box, etc., without any consequences for the board game. The Media Service needs Amby to do an action. The difference from the Actions component is that the incoming requests come directly from the board game, without any logic rules evaluated. They can be used in order to support storytelling in plot games, where a media has to be played when players reach a specific game point. Media service can also be used to provide ambient effects to the

game – play several sounds, background music which can change accordingly to the game concept, show images and animations depending on the game block that player steps in etc. Finally, the Mini game Service provides to the board game the ability to incorporate several mini games into the basic game plot without having to design them from scratch. Any input support for the mini games have to be provided by the board game though, as Amby is an ambient presence avatar and there may be configurations where no devices can be attached on it (for example the case in which Amby is projected on a play room's wall). Amby can completely undertake the mini games execution, and the AI component can be used in order to make adaptable comments on their progress too. Two mini games were developed in order to demonstrate how they are supported by the Amby's character. The first is a politically correct version of the well-known Hangman game , and the second represents the drawing card feature that is incorporated in the majority of board games. Both games are designed to be independent from the main game, although they can be configured to match the concept and the difficulty of the latter. The functions that are provided for each service to communicate with the UI component (with their arguments), along with the actions commands which arrive from the AI sub system, are presented in Table 1.

In order for the UI component to interact with the board game, it needs not only to take input from it, but also to return data to the services in order to be processed by the board game. These functions (that the UI component calls in order to return data to the board game) are presented in Table 2.

In order for the mini games to be commented, the UI component has to send notifications events to the events bus, to be collected and processed by the AI system. The commenting behavior which results from the AI rules is sent back to the UI component through command actions (sent by the AI). The messages that each mini game propagates to the event bus are displayed in Table 3.

Service	**Commands**	**Attributes**
AI	Verbal	Text
	Image	Id, position, size

	Animation	Id, position
	Sound	Id
	Music	Id
	Video	Id
	Emotion	Id
Inventory	Show	Player Id
	Hide	Player Id
	Add Item	Player Id, Item Id, Category, Description, Usages Left
	Remove Item	Player Id, Item Id
	Select Item	Player Id, Item Id
	Use Item	Player Id, Item Id, Usages Left
Dice	Roll Dice	-
Media	Verbal	Text
	Image	Id, position, size
	Animation	Id, position
	Sound	Id
	Music	Id
	Video	Id
	Emotion	Id
Mini Game	Play	Player Id, Game Id, Category, Difficulty

Table 1 – Commands that arrive to the UI component.

Service	**Commands**	**Attributes**
Inventory	Set Selected	Player Id, Item Id
Dice	Set Finished	Dice Result
Mini Games	Finish	(String) Game result (Card Action if Cards, “Hangman Finish” if Hangman)

Table 2 – Functions that are called from the UI component

Hangman	Begin	Player Id, Word selected, category, difficulty, errors allowed
	End	Player Id, win, errors left
	On Letter	Player Id, letter, belongs to word, errors left, letters to be found
Cards	Draw	Player Id, characterization (0 == Good, 1== Bad, 2= Unknown)

Table 3 – Functions that are called from the UI component

6.2 Facial Expressions

Amby is able to convey several emotions using facial expressions. An emotion is expressed via moving, changing and rotating Amby's characteristics according to the target emotion. For example, the "sad" emotion requires the mouth change to sad, the eyes to look down and the eyebrows to rotate in order to shape a ^. In order to perform the necessary changes, the system must remember the current emotion state of Amby. When an emotion arrives, a handler function is called based on the current emotion state and the desired one. Each handler function represents a transition between two Amby emotion states. In order to avoid having functions for every possible transition, a transition function is used from every emotion state to the neutral state and the from neutral state to every emotion. If a direct transition function between the current emotion and the desired one does not exist, then the transitive function between the current state and the neutral state is called, and the function which transits from the neutral to the desired emotion is scheduled after a time interval. In this way, when an emotion is added, only transition functions from the Neutral state to the new emotion and vice versa are required. If more transition functions are available, the system will automatically use them instead of using the Neutral state as medium. All transition functions are named in a specific way, so that the system can obtain the name of the function automatically given the Amby's current emotion state and the target emotion's name. Each transition function knows which actions need to be performed for each Amby's characteristic (mouth, eyes, eyelids, eyebrows and hair) in order to go from the current emotion to the targeted one. In order to avoid frozen expressions, a restoring mechanism was also developed

to bring Amby back to the happy state, using the opposite emotion transitions. The restoration process must be gradual in order to look nice, and can only last a couple of seconds. It uses the same approach to transitions described above – if a transition from the current emotion to happy does not exists, the neutral face is used as a medium.

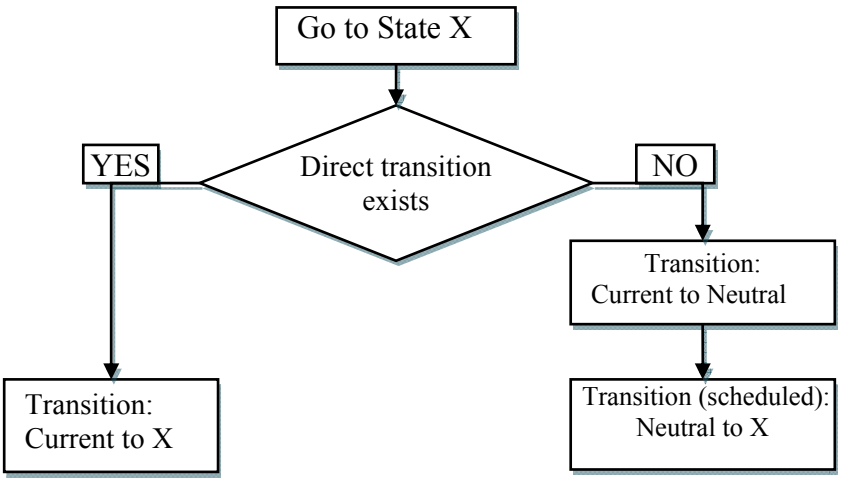

Figure 17 – Emotion transition diagram.

In Figure 17, the emotion transition diagram is showed. The X state is the target emotion that Amby needs to display, which can source either from an incoming game command or from the restoration logic (in the latter case, X is always Happy). Note that if the current state is the same as the X state, no action is done (Amby is already in that emotion state). From the moment that Amby receives a command to display an emotion until the moment that Amby returns to the "Happy" state again, a few seconds elapse. If a new emotion command arrives in these few seconds, Amby has to reconsider his Actions and cancel all pending transitions. If another emotion is being performed at that point in time (animators are playing), then the new emotion has to be scheduled to start immediately after the current emotion's end. Restoration transitions have to be scheduled again for the latest emotion (since previously pending transitions were canceled). If no emotion transition has started yet, then the new emotion starts execution immediately. Again, restoration transitions are scheduled in order for Amby to return to the "Happy" state again. Figure 18 displays the actions performed by the emotion mechanism if an emotion arrives before the previous emotion has finished playing.

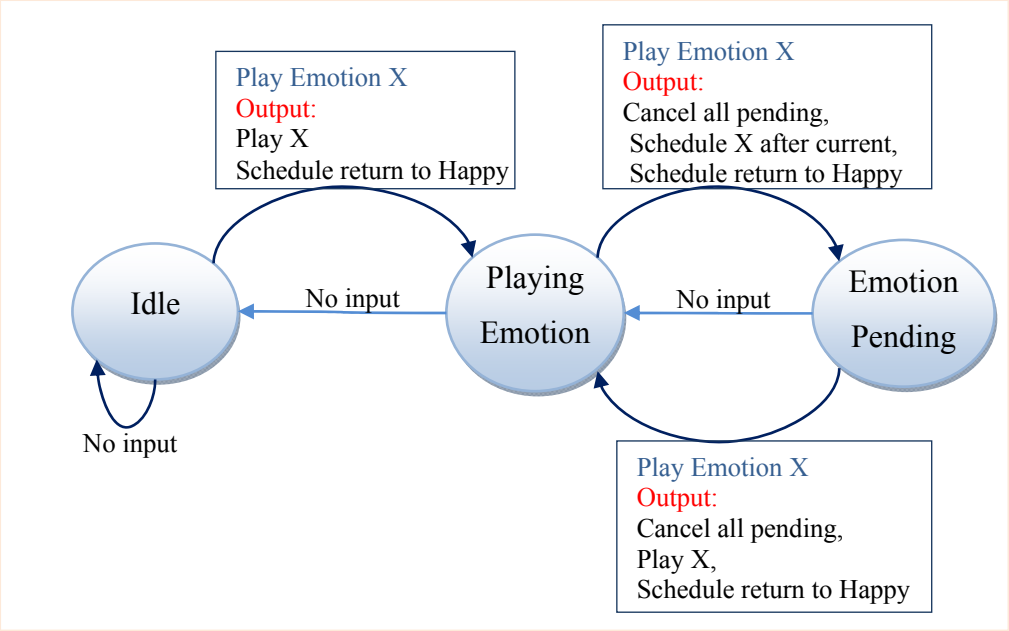

Figure 18 – Emotion state diagram.

The emotions which are currently supported by Amby and their descriptions are Sad, Neutral, Happy, Surprised and Hurt. In order to support them, eyes and mouth animations are provided in .adf (animation definition file) files. These are text files which the game engine knows how to read and interpret into animations. To give an example, an adf file is used to describe all mouth transitions for the mouth expressions. When an emotion is to be displayed, the proper animation is retrieved and applied to the mouth. The same process is followed for the defining the direction of Amby's eyes, but we make a custom animation is used for the eyebrows for better control of their rotation.

More emotions can be added by Delta developers in an easy way, and will be automatically supported by Amby's emotion mechanism. This makes the Amby's emotion mechanism extensible and easily reusable, because each emotion that game developers add can be further used from others to support their games and so on. To avoid frozen expressions, Amby's emotions are implemented in an animated way, and do not stay on Amby's face above a time threshold, that is a few seconds. In this way Amby's face is kept animating, like a real game presenter's face would be. To give an example of Amby's emotion driven animations, let's suppose that a player has been asked to play a hangman mini game. When he selects a letter that belongs to the hidden word, Amby smiles at him. On the contrary, when he selects a letter that does

not belong to the hidden word, Amby gets Sad. But when they make three mistakes in a row, or when they have less than three error tries left, Amby gets Hurt with every mistake, because he knows that the player have little chance to win the game. Some of Amby's emotions are displayed in Figure 19.

Figure 19 – Amby Emotions.

6.3 Adaptive Comments

As mentioned above, Amby is in charge of making knowledgeable comments adapted to the target player's profile. These comments result from rule evaluation, and can have the form of text, video, image, sound, animation, music or facial expressions. More specifically, the actions that Amby is requested to make in order to comment an event can be one of Verbal, image, sound, music, video, animation and emotion, or a combination of them (for example, show an image along with a sound effect). For each one of them (except from audio, video and music) the proper data to have to be provided to the Game engine in order to handle their rendering on the Amby's screen.

a) *Verbal*: It is used to produce Amby's verbal behavior. It takes as an argument only the text that Amby displays. With every verbal command, Amby will display the specified text. Although the Verbal command contains a single-line text to be drawn on screen, a mechanism has been developed that "breaks" the text into multiple lines in order to fit in the boundaries given. The text stays a few seconds on Amby's screen before it disappears. If a text is already displayed on the screen and a second verbal command arrives, the latter forces the first one to disappear. To draw the text on the screen, it is rendered as a dynamic film –a new transparent bitmap is constructed and the text is drawn on it. Then, the text od "cut" it to its bounding box and displayed on the screen at the desired position. Also, a box is drawn containing the text message. A triangle is used to make it look like a text bubble. In this way, Amby to "say" the text, just like in comic books the characters "say" their lines.

b) *Image*: It is used for Amby to show an image. In requires three arguments: the Image's unique id, its vague positioning on the screen and a generic displaying size. The image's id is used to locate the image to display. All images are displayed as films –they must have a resource id, a specification file and an expanded film which contains information about the image's bounding box and

collision mask. These data together with the film's bitmap are bound to a unique id through a resource file, and given to the game engine in order to be displayed.

To enable AI designer to control the position on Amby's screen at which an image or animation is displayed, a parameter is defined that fuzzily specifies this position. This parameter has a string value that corresponds to a range of points, and when Amby needs to display it a random point is chosen within the given range. A configuration file is provided defining point ranges, with special care not to exceed Amby screen's borders. For example, for the "TOP_LEFT" position, the range [0-256] for the x coordinate and [0-192] for the y coordinate are defines. The size parameter gives the ability to the game programmer to approximately define the image's size on Amby's screen. Just like positioning, a resource file is needed to bind each size-label with a range of widths and heights. Although this feature is provided by Amby, the current Amby's UI does not support changing an image's dimensions. This feature is among future work targets.

c) *Animation*: Upon receiving this command, Amby loads and shows an animation on the screen. It takes two parameters, the first one is the animation's id and the other one is the abstract positioning that the game programmer defines. Animations, like images, cannot be created dynamically. Given an animation id, Amby search it in a preloaded table, in which all the animation films are registered. This requires that an "Animation Definition File" (.adf) is registered with animations name, and also that an animation exists with the same name. Then the animation defined in the animation definition file is applied to the film with the same name.

d) *AmbyAnimation:* As Amby animations are the animations that Amby applies to his own characteristics in order to convey emotions (Facial Animations). All animations are pre-builded using either adf specifications or coded animations for each characteristic to be animated. Supported emotion ids are: "Sad", "Happy", "Neutral", "Surprised" and "Hurt". When any of these emotions is requested, a handler function is called based on the Amby's current emotion state and the emotion requested, which defines the animations to be performed.

e) *Audio:* Amby is able to play sound effects using the OpenAL sound library, which is a cross-platform 3D audio API appropriate for use with gaming applications. All Sounds need to be referred in a resource file, linking their id name with the audio file path. Only wav files are supported.

f) *Music:* The game designer might need at someplace to load and play a music file. Amby supports such functionality via Vorbis (Ogg Vorbis). Vorbis is a fully open, non-proprietary, patent-and-royalty-free, general-purpose compressed audio format for mid to high quality audio and music. It supports playing music files in ogg format (.ogg files). All music tracks need to be referred in a resource file, linking their id name with the corresponding file path.

g) *Video:* Although the "play video" command arrives to the Amby character, this functionality is not yet implemented, as support of an extra screen is necessary to play video. In future work, both multiple screen utilization and video supporting will be addressed.

The game engine is used in all the above cases (except from Audio, Music and Video) in order to perform the rendering on the screen. All animations, images and sprites are displayed using functionalities provided by the Game Engine.

6.4 Adaptive Cues

In terms of commenting a board game, Amby has the ability to identify specific game states, and make a comment about them using several methods, including adaptive animated cues. This term refers to the individual animations and facial expressions that Amby can perform depending on the board game state (not for commenting purposes). For example, if the game or a player is inactive for a long period of time, Amby notices it and starts acting in a specific way in order to motivate the players to continue playing. The motivation can be a text or a multimedia like image, video, sound, music, animation and emotion. The orders to display such cues the AI component (which decides what and when will be displayed) sends the proper message to the UI component, using the same mechanism as in commenting. The

commands are then executed through the Game Engine, following exactly the same procedure described in section 6.3, Adaptive Comments.

6.5 Hangman

One of the mini games that are currently supported by the Amby character is the game of Hangman. The traditional hangman is a paper and pencil guessing game for two or more players. One player thinks of a word and the other tries to guess it by suggesting letters. The word to guess is represented by a row of dashes, giving the number of letters. If the guessing player suggests a letter which occurs in the word, the other player writes it in all its correct positions. If the suggested letter does not occur in the word, the other player draws one element of the hangman diagram as a tally mark. The game is over when the guessing player completes the word, or guesses the whole word correctly, or the other player completes the diagram. This diagram is, in fact, designed to look like a hanging man. Although debates have arisen about the questionable taste of this picture, it is still in use today. A common alternative for teachers of young learners is to draw an apple tree with ten apples, erasing or crossing out the apples as the guesses are used up.

In the Amby's implementation of the Hangman game, an alternative scenario is adopted consisting of is a bridge, every piece of which represents a letter of the hidden word. The game ends when the player completes the bridge or when they exceed the number of errors allowed. Hangman words are loaded from a resource file when Hangman starts. They are separated in categories, and each category has words for several difficulty levels (easy, medium, hard etc). When game designers ask Amby to play a Hangman game, they can choose a category and a difficulty level for the word in order to fit in the game concept. If they don't want to restrict these parameters, they can use the "generic" category and random difficulty (if the given category or difficulty level does not exist, Amby automatically chooses the generic category and the medium difficulty). Then Amby randomly selects a word that combines these criteria, sets the error tolerance according to the difficulty level (10 misses are allowed in easy mode, 8 in medium and 6 in hard mode) and calculates the

bitmap sizes in order to fit the whole word in the Amby's screen. After that, Amby prepares for Hangman. The Amby's avatar is shrunk and moved to the upper right corner in order to make space for the letters and the bridge, the letters are gradually shown at the middle of the top's side on the screen and the bridge constructed of the hidden letters gradually appears above the players' avatars. When the player suggests a letter, Amby checks the hidden word to determine if there are any occurrences. If there are, then the suggested letter falls in its place (or places if there are multiple occurrences), but if there aren't the suggested letter falls through a blank part of the bridge to the ground and disappears. Each incorrect suggestion makes the bridge parts move more vividly and thus harder for the player to cross the bridge. After each suggestion, Amby makes a comment about the player's progress. If the suggestion was incorrect or if the suggested letter has been played before, Amby announces the number of misses left or that the letter is already selected respectively. The game ends either when all the pieces of the bridge are completed (the word is found) or when the player selects a wrong letter and all guesses are used-up.

Since Amby doesn't have any input devices attached, the suggested letters are received from the board game. In other words, the hangman player chooses a letter via an interface provided by the board game, and then the board game sends the letter to Amby. On game begin, game end and on each letter suggested, the hangman component sends a notification event to the Amby's AI subsystem using the event bus. The AI subsystem keeps the incoming game data and processes them in order to decide the attributes to be set in the player's profile and their values. After that, rule evaluation follows in order to decide the actions to be made by the Amby character to comment the game state and the player's progress in it. These decisions are then sent back to the Amby's UI as commands (Verbal, image, audio, music, animation and emotion). With this mechanism, Amby is able to use his own commenting mechanism to comment the mini games. Finally, when the game ends, hangman component must send two notification messages to the mini game service, one containing the result of the game – if the player won or lost, and the number of tries that had left (always 0 if the player lost) and the other containing a string with the game outcome. The latter will be used by the board game in order to decide the player's award or scold if needed, and it can contain simple Boolean information indicating whether the player

won or lost or more complicated information such as the player's progress. In every case, the board game and the Hangman component must use the same communication protocol in order to understand each other.

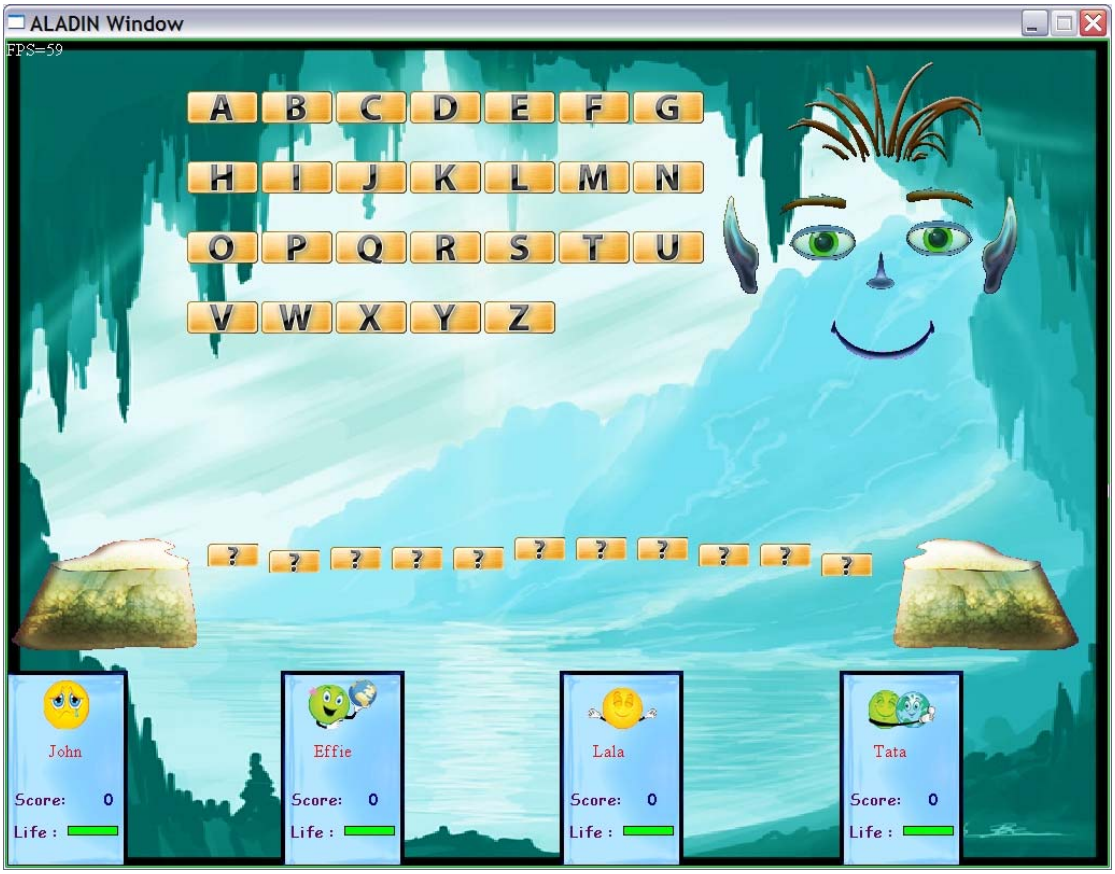

Figure 20 – Hangman game.

6.6 Cards

Drawing cards in one of the most common activities that players are asked to do when playing a board game. The cards can express several abilities or they can be used to instruct players to do something or to go somewhere (for example "play in the casino" card in "Go for Broke" or the "go to jail" card in "Monopoly").

In Amby integrates an electronic form of the cards mini game. In order to let the developer decide the cards' context, Amby loads the cards from a resource file the

first time that the board game asks for a card draw. Each card must have a description text and an action, while it can optionally contain:

- an image to display
- a sound or a music file to be played
- an animation film
- an emotion for the Amby avatar

On the card draw command, the Amby's avatar is shrunk and moved to the top right corner of the screen in order to make some space for the deck to appear. A "draw card" animation follows and the card's description is displayed inside a randomly selected card border. In parallel, Amby displays the image and the animation defined by the card, plays the music and the sound effects and makes the proper facial animations to display the defined emotions. All these media are optionally provided from the card designer, and when used, they significantly enhance the card mini game.

Amby is designed to be game independent, and therefore has no knowledge of a card's action semantics and interpretation; for Amby such an action is merely a string literal. Upon drawing a card, Amby retrieves the action string literal associated with the card drawn and provides it to the board game through the result of the mini game service. The board game is then responsible to interpret the given action and perform any further actions associated with that. Such actions can then be propagated back to Amby for further processing and commenting through their normal flow from the corresponding services. For example, if a player draws a card whose action is "ScoreChange(+10)", Amby takes that as a string literal and sends a message that the mini game result is "ScoreChange(+10)". The board game knows how to interpret this string, and uses the Monitoring Service to send a ScoreChange event to Amby with the value +10. The monitoring service will construct a notification message that will be passed to the AI component. There, the proper processing will follow, setting the proper attributes to the player's profile, evaluating the corresponding DMSL rules and retrieving commenting actions as evaluation results. These actions (for example a verbal comment and a proper sound effect) will be further sent to Amby's environment and linked to the Amby's UI, in order for Amby to execute them. In Figure 21 and Figure 22 we display how Amy's screen looks when a card is drawn.

Specifically in Figure 21, notice that the card is accompanied with an Amby's expression and an Image.

Figure 21 – Draw-a-card mini game.

Figure 22 – Draw-a-card mini game.

6.7 Inventory and Players

6.7.1 Players

Amby is an avatar suitable for use in an intelligent ambient environment context. This means that Amby can be displayed in several places, for example on a touch screen besides the board game table, on a separate screen above players' heads, projected on a room's wall etc. Therefore, all artifacts that are displayed inside the Amby screen must be clear and have well defined boundaries. Each player must have his own place on the screen to look in order to review his character's attributes. This place must have concentrated information about the player without taking up too much space, and must be specified from the beginning so that the player does not have to search for it on Amby's screen. In order to fulfill these preconditions, a panel is displayed for each player with well distinctive boundaries, inside which all the data regarding a player are displayed. More specifically, the player's avatar is displayed at the top of the panel; right below their name, score and lifes appear. The latter characteristic is displayed using a life bar which changes colors depending on player's lifes. This makes it easier for the players to read id at a single glance, since they do not have to read its value. Figure 23 shows an example of how a player's panel looks like. Figure 24 depicts four players with their inventories open (only the last player's inventory is hidden) on the Amby's screen. The player's name is loaded from a configuration file at game start, along with their avatar ids. Each panel must be stuck at a specific location on the screen. The bottom part of Amby's screen was chosen as a specific location for all players, since using the top of the screen would draw the player's attention away from the Amby character, and their positioning on Amby's side or on Amby's four corners would leave uncomfortable spaces difficult to be utilized efficiently. Moreover, some space next to the player's panel need to be dedicated to the player's inventory. As mentioned above, each inventory item can have an id, a category (which is the item's name shown to the players – for example "Sword"), a description (for example: "Fights your enemies with fierce"), an image and a number indicating how many times it can be used (0 means unlimited). Each player can review their inventory independently, and there is no need to wait for their turn to do that. In other words, all players must be able to have their inventory open and interact

with it simultaneously, the space it consumes must be dedicated for that purpose and must not be used to display other things for the game or for the Amby character (for example show animations or images). By keeping some space for each player inventory next to their avatars, the players are able to locate it easily, and potential collisions with the rest of Amby's screen are avoided, since the bottom of the screen is dedicated to showing players and inventories, and no other valuable features are displayed there. The only drawback is that the maximum number of players that can currently be supported is bounded to four. Figure 23 shows how a player's panel is structured, while Figure 2 illustrates how an open inventory looks like. Notice that because of space limits, only one item per time can be shown. This configuration was adopted in order to support also displaying the name, description and left usage times of the item which is displayed. These data couldn't be displayed in another way because there are no pointing devices attached to Amby. Figure 24 shows an Amby screen with four players represented. All but the last player have their inventories open. Notice that all information is mustered and organized inside strict boundaries, thus making it very easy for a player to review them at a glance.

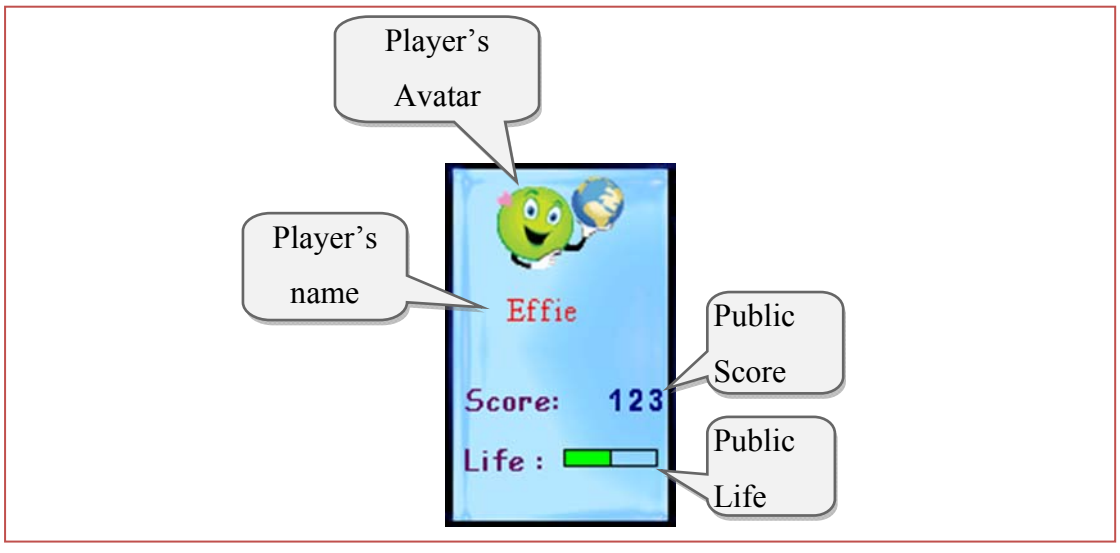

Figure 23 – Player's Panel.

Figure 24 – Player panels and their Inventories. For the last player the inventory is hidden.

7. Discussion Conclusions and Future Work

This thesis has focused on the development of an artificial game presenter avatar which can enhance electronic board games by making knowledgeable social adapted comments, providing support for inventory displaying, dice rolling and media reproduction. Amby lives in a restless environment where several player actions and the game state are projected in a fancy animated way. Moreover, Amby can be a pleasant companion to the players by cleverly commenting their progress in the game, as well as a nice game presenter who will stimulate viewers' interest.

The amby character provides a number of services, and namely Inventory, Dice, Media, Mini Games and Monitoring. Amby integrates an AI model which is responsible for its intelligent behavior by supporting the provision of knowledgeable comments, cues and emotions derives form logic rules. This AI component provides user-based adaptation of Amby behavior and is easily extensible. In this way, the Amby character can develop a personality and become a pleasant companion to the players and the viewers of the game. Amby's actions are projected on its screen. Specific mechanisms have been developed to handle and show players' data, emotions, inventories and multimedia, and all these are combined to synthesize a restless game scene. Finally, two mini games have been developed that are currently provided by the Amby character. They receive input from the board game and how can utilize the Amby's AI system in order to comment themselves efficiently.

Amby can currently be used for all board games that can be made by the ***on-board!*** platform [3], i.e. all games that have events similar to these described in Monitoring API (presented at Figure 10). The game programmer does not have to use all the services provided by Amby; services are independent from each other and can be used separately. If we need to modify our system in order to support games with different events, we would have to add the new event types to the event system, provide the corresponding event handlers to the AI component, and provide the DMSL rules that will be used in order to make adaptive comments for the new game characteristics. New services may also be added. The latter only needs the new Service to derive from our Service super class, and to be added in Service initializer in order to be initialized. The service provider must also

provide a dll file with the desirable API to the board game so that the latter can communicate with the new service provided. Once these extensions are done, our system can automatically link the new dll with the corresponding service and call the latter to check new incoming events from the board game periodically. If we want Amby to support comments about the new service added, we will need to add a new event into the event system and write a handler function for this event to be used by the AI component. DMSL rules to be evaluated when the new events occur are also required. In Figure 25, we present the modifications that need to be made in Amby's architecture in order to support different kind of games. Note that the main architecture structure remains the same.

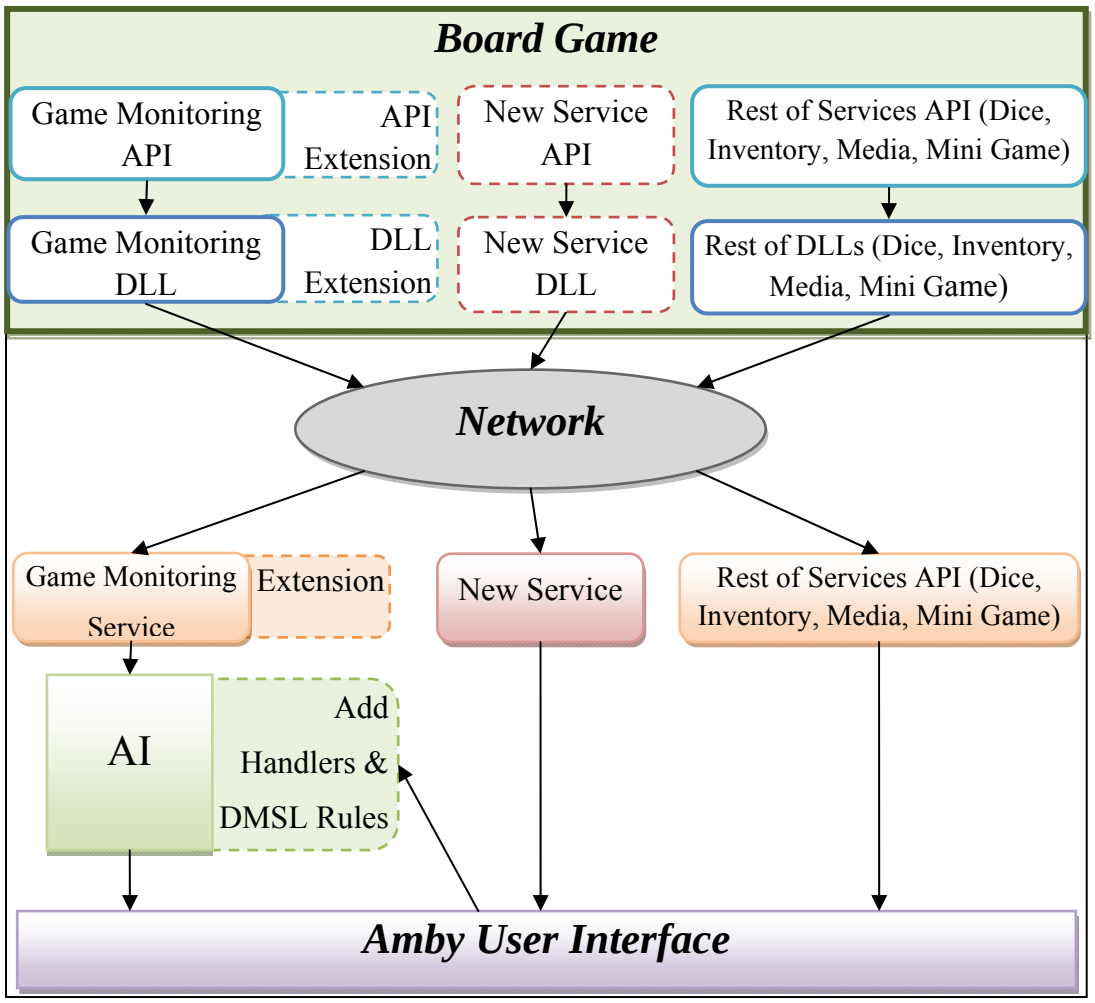

Figure 25 – Modifications in Amby AI in order to support new games.

Although Amby is a game independent fully functional character, some additions and optimizations can be proposed in order to improve it. The configuration that the

Amby avatar currently supports is a single screen, on which everything is projected, from players' avatars to mini games. A future work objective is to enable multiple screen utilization in order to distribute the information projected, support more players and enable video scenes. Moreover, handheld devices could be supported in order to display for each player the information they are interested in. In this way, each player will be able to view his avatar, life, score and inventory items privately in a personally-adapted way.

Another future objective is to support additional pluggable mini games developed for the Amby platform. For such mini games, developers should provide a) a Delta scripts implementing the mini game functionality b) a DMSL rule files containing the adaptable logic for Amby based on the specific game events and c) a dynamically linked library (dll) able to perform the AI event dispatching registration (C++) and the mini game dispatching registration (Delta), as well as create the C++ to Delta linkage for the specific game, registering additional Delta library functions. Towards this end, the AI subsystem has to export a part of its functionality, becoming able to interact with the libraries provided by the mini game developers. Additionally, the Mini Game component in the UI must be extended to a dispatcher component allowing the registration of external function to perform the mini game functionality. Finally, in case these external mini games demand more sophisticated input control than simple keystrokes, additional modifications are required from the Mini Game API and Service to be able to forward the input from the main board game to the mini game.

More facial expressions can also be supported. This is an easy extension because only the animation transitions from the new emotional state to the neutral state and vice versa is required. After that, the system will be able to automatically load them when needed, provided that they are properly named.

8. Bibliography

[1] Savidis, A., Antona, M. and Stephanidis, C.: A Decision-making Specification Language for Verifiable User-interface Adaptation Logic. International Journal of Software Engineering and Knowledge Engineering, pp. 1063-1094, 2005.

[2] Anthony Savidis: Dynamic Imperative Languages for Runtime Extensible Semantics and Polymorphic Meta-Programming. RISE 2005: 113-128, 2005.

[3] Yannis Lilis: Ambient Cross-Media Board Games: Game Engine, Terrain Editor, Extensible Interface And Adaptive Pluggable Input (Master Thesis).

[4] Alfonso Garate, Nati Herrasti, Antonio Lopez, "GENIO: An Ambient Intelligence application in home entertainment environment" October 2005.

[5] Elisabeth Andrè, Kim Binsted, Kumiko Tanaka-Ishii,Sean Luke, Gerd Herzog, and Thomas Rist, "Three RoboCup Simulation League Commentator Systems"

[6] Kim Binstend, Sony Computer Science Lab "A Talking Head Architecture for entertainment and experimentation"

[7] Amin Habibi Shahri, "An Introduction to a new commentator for RoboCup 3D Soccer Simulation"

[8] M. N. Sedaghat, N. Gholami, S. Iravanian, and M. Kangavari. Design and implementation of live commentary system in soccer simulation environment. In proceedings of RonoCup Symposium 2004 "Design and implementation of live commentary system in soccer simulation environment". In proceedings of RoboCup Symposium 2004, Lisboa, Portugal,2004

[9] A. H. Shahri, A. A. Monfared, and M. Elahi. "A deeper look at 3d soccer simulations". In roceedings of RoboCup 2007 Symposium, Atlanta, USA, 2007.

[10] Thomas Rist, Elisabeth Andr6, Jochen Muller “Adding Animated Presentation Agents to the Interface”

[11] Tsukasa Noma, Norman I. Badler “A Virtual Human Presenterr”

[12] Paula S. L. Rodrigues, César T. Pozzer, Bruno Feijó , Luiz Velho, Angelo E. M. Ciarlini, Antonio L. Furtado “An Expressive tlking head narrator for interactive storytelling system”, May 2005

[13] Sadasdasd K. Tanaka-Ishii, I. Noda, I. Frank, H. Nakashima, K. Hasida, and H. Matsubara. “Mike: An Automatic commentary system for soccer”. Paper presented at the 1998 international Conference on Multi-agent Systems, Paris, France, 199

[14] D. Voelz, E. Andr´e, G. Herzog, and T. Rist. “Rocco: A robocup soccer commentator system”. In M. Asada and H. Kitano (eds.), RoboCup-98: Robot Soccer World Cup II. Springer, pages 50–60, 1999.

[15] HCI Laboratory, Internal tools

[16] Bourdenas Themistoklis: Circular Meta-IDE for the Delta Language: Extensibility Layer for Delta, Debugger, Runtime adaptation, and Project Manager. Master’s Thesis, 2007.

[17] Geogalis Yannis: Circular Meta-IDE for the Delta Language: Dynamic Extensibility, Remote Deployment, Interactive Introspection, and Syntax Directed Editor. Master’s Thesis, 2007.

[18] Kabileshkumar G Cheetancheri. Ch OpenAL package. URL:http://chopenal.sourceforge.net/.

[19] Alan W. Black and Kevin A. Lenzo “Flite: a small fast run-time synthesis engine”

[20] Mahammad Nejad Sedaghat, Nina Gholami, Sina Iravanian, Mohammad Reza Kangavari “Design and Implementation of Live Commentary System in Soccer Simulation Environment”, RoboCup 2004: Robot Soccer World Cup VIII, volume 3276/2005 (Spring 2005)

APPENDICES

A - DMSL specifications

a. DMSL language grammar specification

The DMSL language grammar specification can be summarized below:

logic *::= { block | stereotype | definition }*

block *::= '**component**' **identifier** compound*

*compound ::= '**[**' { stmt } '**]**'*

stmt *::= ifst | command expr | compound*

ifst *::= '**if**' boolexpr '**then**' stmt ['**else**' stmt]*

*command ::= '**activate**' | '**cancel**' | '**evaluate**'*

expr *::= primary | boolexpr | arithexpr*

*primary ::= const | param | funccall | '+' expr | '-' expr | '**not**' expr | **identifier***

param *::= '**params**' '.' **identifier { '.' identifier }***

*funccall ::= libfunc '**(**' [expr { '**,**' expr }] '**)**'*

libfunc *::= '**isactive**' | '**tonumber**' | '**hasattr**' | '**random**'*

const *::= '**true**' | '**false**' | **number** | **identifier***

*boolexpr ::= expr boolop expr | expr '**in**' set*

arithexpr ::= expr arithop expr

arithop *::= '+' | '-' | '*****' | '/' | '**%**'*

boolop *::= '**or**' | '**and**' | '<' | '>' | '<=' | '>=' | '=' | '**!**='*

*stereotype ::= '**stereotype**' **identifier** '**:**' boolexpr*

define *::= '**def**' **identifier** expr*

Legend:

{ a }: zero or more occurrences of *a*

[a]: zero or one occurrence of *a*

a | b: occurrence of either *a* or *b*

‘**a**’: terminal character appearing in the DMSL file (language keyword or operator)

identifier: string with or without quotes (e.g. “string” or string)

number: a real or integer number (e.g. 1, -12, 34.5, -5.0)

Additional to the main language keywords and operators, the following can also be used (syntactic sugar):

1) ‘**adapt**’ for ‘**evaluate**’
2) ‘**==**’ for ‘=’
3) ‘**&&**’ for ‘**and**’
4) ‘||’ for ‘**or**’
5) ‘**TRUE**’ for ‘**true**’
6) ‘**FALSE**’ for ‘**false**’

Finally, there are three types of comments that are supported:

1) // comment until end of line
2) # comment until end of line
3) /* comment everything until */

b. DMSL language grammar semantics

In DMSL, type checking is performed both at compile time as well as at runtime. Most type checking issues are resolved upon compilation and reported with a compile error. Still, an expression containing a user or context parameter can only be evaluated at run time and therefore it is also possible to receive a runtime error regarding bad types in an expression or given to an operator.

The semantics of the language are pretty straightforward and can be summarized to the following:

- All identifiers are case sensitive and unique. In case of redeclaration of an identifier we have a compile error.
- The expression of the *if* statement should evaluate to a boolean expression, else we have an evaluation error.

- All command expressions should evaluate to a string expression, else we have a compile or evaluation error.
- The library functions *isactive, tonumber* and *hasattr* take one string argument. If these functions are given more than one argument we have a compile error, or if the argument isn't a string we have either a compile or an evaluation error.
- The expression of the stereotype declaration should evaluate to a boolean expression, else we have an evaluation error.

Operators

Operator Types (with precedence increasing from top to bottom)	**Operators (with precedence increasing from left to right where separated)**		
Logical Operators	or	and	not
In Set Operator	in		
Equality Operators	= !=		
Relational Operator	< > <= >=		
Arithmetic Operators	+ -	* / %	(unary) + -
Parenthesis	( )		

Table 4: DMSL Operators and their precedence

- The logical operators are used between boolean expressions and the constant boolean values *true* and *false*. The result of a logical operator is a boolean expression.
- The "in set" operator is a binary operator taking as its first argument an arbitrary expressions and as its second argument a set. The result of the operator is a boolean expression.
- The equality operators are used between expressions with the same type. The result of an equality operator is a boolean expression.
- The relational operators are used between arithmetic expressions. The result of a relational operator is a boolean expression.

- The arithmetic operators are used between arithmetic expressions. The result of an arithmetic operator is an arithmetic expression. Specifically for the plus (+) operator, it can be used if at least one of the operands is a string. The other operand can be a string, a boolean or a number. The result in this case is a string.
- The parenthesis operator has the highest priority and can be used to enforce priority upon any case.

- If an operator is given invalid operands we have a compile or an evaluation error.

Built-in library functions

- The *isactive* function checks if the given component is active. The result of this function is a boolean expression.
- The *tonumber* function transforms the given string argument to a number. The result of this function is an arithmetic expression.
- The *hasattr* function checks if the given attribute exists in the user profile. The result of this function is a boolean expression.
- The random function has two overloaded versions; the first taking an integer number and returning a random number in the rage of [0, number], and the second taking no arguments and returning a previously generated number through a call to the first version. The result of this function is an arithmetic expression.

c. Profile grammar specification

The user profile contains an arbitrary number of attribute-value pairs corresponding to the user attributes and the values they take. The profile grammar specification can be summarized below:

profile ::= { pair }
pair ::= attribute '=' value

attribute ::= ***identifier*** *{ '.'* ***identifier*** *}*

value ::= ***'true'*** | ***'false'*** | ***number*** | ***identifier*** | *set*

set ::= ***'{'*** *[* ***value*** *{ ','* ***value*** *}] '}'*

Legend:

{ a }: zero or more occurrences of *a*

[a]: zero or one occurrence of *a*

a | b: occurrence of either *a* or *b*

'**a**': terminal character appearing in the profile file (language keyword or operator)

identifier: string with or without quotes

number: a real or integer number

Syntactic sugar: '**TRUE**' for '**true**' and '**FALSE**' for '**false**'

Comments supported:

1) // comment until end of line
2) # comment until end of line
3) /* comment everything until */

B - Delta specifications

The Delta language syntax

```
code ::= { [ def ] }
def  ::= ( [ stmt ] ‘;’ | func )
func ::= ( funcprefix | methodprefix ) funcdef
funcprefix ::= ‘function’ [ id ]
methodprefix ::= ‘method’
funcdef ::= ‘(‘ [ id { ‘,’ id } ] ‘)’ block
block ::= ‘{’ code ‘}’
funcexpr ::= ‘(‘ func ‘)’
stmt ::= ( expr | whilest | forst | ifst | ‘break’ | ‘continue’ |
‘return’ [ expr ] | block | assertion |
‘const’ id ‘=’ expr | ‘try’ stmt ‘trap’ lval stmt |
‘throw’ expr )
whilest ::= while ‘(’ expr ‘)’ stmt | ‘do’ stmt ‘while’ ‘(‘ expr ‘)’
forst ::= ‘for’ ‘(’ exprlist ‘;’ expr ‘;’ exprlist ‘)’ stmt
ifst :: ‘if’ ‘(’ expr ‘)’ stmt [ ‘else’ stmt ]
exprlist ::= [ expr { ‘,’ expr } ]
expr ::= ( assign | primary | boolean | arith )
assign ::= ( lval ‘= expr | lval ‘+= expr | lval ‘-=’ expr |
’’ lval ‘*=’ expr | lval ‘/=’ expr )
lval ::= ( [ ‘local’ | ‘static’ | ‘global’ | ‘::’ ] id | member )
member ::= ( expr get id | expr subscr | expr get string )
get ::= ( ‘.’ | ‘..’)
subscr ::= ( ‘[’ expr ‘]’ | ‘[[’ expr ‘]]’ )
primary ::= ( lval | lval (‘++’|‘--') | (‘++’|‘--') lval | ‘lambda’ |
const | callable )
object | ‘self’ | ‘arguments’ | ‘-’ expr | ‘not’ expr )
callable ::= ( lval | ‘(’ expr ‘)’ | funcexpr | call )
call ::= callable ‘(’ [ actual { ‘,’ actual } ] ‘)’
actual ::= (pr | ||
ex‘’ expr ‘’ )const ::= ‘nil’ | ‘true’ | ‘false’ | number | string
boolean ::= expr boolop expr
arith ::= expr arithop expr
arithop ::= ‘+’ | ‘-‘ | ‘*’ | ‘/’ | ‘%’
boolop ::= ‘or’ | ‘and’ | ‘<’ | ‘>’ | ‘<=’ | ‘>=’ | ‘==’ | ‘!=’
object ::= ‘[’,’ slot} ] ‘]’
[ slot { ‘slot ::= ( ‘{’ expr {‘,’ expr } ‘:’ slotval ‘}’ | slotval )
  slotval ::= ( expr | methoddef )    methoddef  ::=  ‘(’
                                       ‘method’ funcdef‘)’
```

C – Code Examples (Delta)

C.1. Icon Creation

```
function CreateHair  ( x , y ) {
      if (isundefined(static hairfilm))
            hairfilm = filmset_getfilm(
                  filmsetmgr_getfilmset( HAIR_FILM_SET ),
                  HAIR_FILM
            );
      sh.util.CreateIcon(HAIR_NAME, true, "bg", x, y,
      hairfilm, 1, true, "centre", "bottom", 0, true);
}
```

C.2. Add Lighting Animator to a given Item. Type can be “lightsource” or “darkensource”.

```
function AddLightingAnimator ( icon, type ) {
      if (not icon[type]){
            icon[type] = [
                  {.animator:
animator_get("renderingvalueanimator") },
                  {.anim    : gallery_getanim(
                              gallerymgr_getgallery("Lighting"),
                              "Lighting") },
                  {.renderer  : renderer_create(type) }
            ];
            icon_addrenderer(icon, icon[type].renderer);
      }
      renderingvalueanimator_start (
            icon[type].animator,
            icon[type].anim,
            icon[type].renderer
      );
}
```

C.2. Animation Creation

```
function CreateBird ( x , y ) {
      if (isundefined(static birdFilm))
            birdFilm = filmset_getfilm(
                        filmsetmgr_getfilmset(BIRD_FILM_SET),
      BIRD_FILM  );

      local icon = sh.util.CreateIcon(BIRD_NAME, false, "bg", x,
                              y, birdFilm, 0,  true, "centre",
                              "centre", 5);

      local animator = animator_get("iconanimator");
      icon.animator = animator;

      if (isundefined(static  anim))
            anim = gallery_getanim(
                  gallerymgr_getgallery(BIRD_GALLERY),
                  BIRD_ANIM);
      icon.anim = anim;
      iconanimator_start (animator, anim, icon);
}
```

C.4. ***Emotion Transition***: From Sad to Happy

```
function Sad2Happy ( id ){
        sh.animScheduling.PERFORMING_EMOTION = true;
        /* Mouth:  Happy to Sad */
        local mouth__anim = gallery_getanim(  gallerymgr_getgallery("SadToHappy"),
                        "SadToHappy");
        iconanimator_start(animator_get("iconanimator"), mouth__anim,
                        sh.util.singleicontable["Mouth"]);
        /* Eyes: Look Straight */
        local eyes__anim  =  gallery_getanim(
                                gallerymgr_getgallery("FromDownToCenter"),
                                "FromDownToCenter");
        iconanimator_start(animator_get("iconanimator"), eyes__anim,
                                sh.util.singleicontable["LeftEye"] );
        iconanimator_start(animator_get("iconanimator"), eyes__anim,
                                sh.util.singleicontable [ "RightEye" ] );
        sh.animScheduling.CurrEmotion = "Happy";
}
```