The Embodied Conversational Agent Toolkit: A new modularization approach.

R. J. van der Werf

Master of Science Thesis in
Human Media Interaction

Human Media Interaction (HMI)
Department of Electrical Engineering,
Mathematics and
Computer Science
University of Twente
The Netherlands

ArticuLab
School of Communication /
School of Engineering
Northwestern University
United States of America

June 2008

Preface

To obtain a master's degree in Human Media Interaction a final research project has to be carried out. This research can be carried out both externally and internally. Before my final project I already carried out an internship of 14 weeks at the Institute for Creative Technologies (ICT) in Marina Del Rey, California. This internship abroad was very valuable; therefore I decided to also look abroad for my final research project. After I attended a presentation by Justine Cassell, I saw a lot of similarities with previous work I carried out during my master. After coming in contact with Justine Cassell I decided to do my research project under her supervision at the Northwestern University's Articulab in Evanston, Illinois, USA.

At the Articulab there were two possible projects for me to work on. The first was along the lines of my previous internship which involved creating rapport with virtual humans. The second possibility was to work on a project for rapid prototyping of Embodied Conversational Agents (ECAs) which involved working with Panda3D and the graphical representation of ECAs. Since graphics and virtual reality have played an important role during my studies, the second possibility seemed like an ideal opportunity for me to further focus on these areas.

I spent little over six months working on my research project at the Articulab. I soon found out that being part of this research lab involved a lot more then only working on my own project for six months. Every now and then the lab had to demonstrate their work to people, ranging from official demonstrations for faculty to presentations on 'bring your daughter to work day'.

After my return to the Netherlands I also spent a lot more time on this project. The larger part of this time was used for the implementation. To resolve certain issues with the software I also paid a visit to the University of Bielefeld.

Looking back at the complete project I can say that I've had great experience doing research. From reading a lot of research papers to writing one. Both of which I didn't really experience before. It has certainly made me think of pursuing a career in research in the future, which I didn't consider before the start of this project.

First of all I would like to thank Justine Cassell and Dirk Heylen for providing me with this opportunity, for supporting me and keeping faith in me. In addition to Dirk Heylen I would like to thank Zsofi Ruttkay and Anton Nijholt for their time and feedback. Special thanks to Nathan Cantelmo, Jessica Cu, Francisco Iacobelli, Paul Tepper and John Borland for their help on this project. A thank also goes to the rest of my colleagues at the Articulab which made my visit to the Northwestern University a very pleasant one. Moreover I would like to thank Joris Janssen for both his personal and professional support during my time abroad and back home. From the University of Bielefeld I would like to thank Ipke Wachsmuth for giving his consent to use the Articulated Communicator Engine (ACE). Especially I want to thank Stefan Kopp for his guidance and support on ACE and for letting me visit the University of Bielefeld. Thanks also goes to Klaus Brügmann, Sebastian Ullrich and Helmut Prendinger for their information about MPML3D.

Finally I would like to thank my mother for supporting me and giving me a place to live after my return from the USA. Last but certainly not least I want to thank my girlfriend for supporting me and dragging me through times were I was about to loose faith.

Abstract

This thesis shows a new modularization approach for Embodied Conversational Agents (ECAs). This approach is titled the Embodied Conversational Agent Toolkit (ECAT). ECAT builds upon the SAIBA framework, which proposed a three stage modularization of ECAs. ECAT focuses on the third stage, called behavior realization. The process of behavior realization can be summarized as: turning high-level behavior specifications into audiovisual rendering of an ECA. Internally this boils down to firstly converting high-level specifications into low-level specifications and secondly rendering these low-level specifications.

In between these two tasks is exactly where ECAT proposes a split. ECAT defines *compilation* as being the process of converting high-level specifications into low-level specifications and *translation* as the rendering of these low-level specifications. In addition to these two stages one preliminary stage is introduced called *interpretation*. *Interpretation* is meant to bridge between the wide variety of applications which are generating behaviors on the one hand and on the other hand the stage of *compilation*.

An ECA using ECAT uses one component for each stage: one *Interpreter*, one *Compiler* and one *Translator*. These three components are separated by two TCP/IP socket interfaces. This keeps the different components language and platform independent of each other. Both interfaces use XML languages for communication. The first interface currently uses the Multimodal Utterance Representation Markup Language (MURML). In addition to MURML this interface has future support for the Behavior Markup Language (BML), which is also used in the SAIBA framework. The second interface uses a custom markup language.

Proof-of-concept prototype components have been implemented for each of the three stages. One functional pipeline, including three components, is based on an existing ECA called NUMACK. Parts of this ECA have been reimplemented and modularized according to the three stages of ECAT. The performance of the ECAT version of NUMACK is similar to the original version. This shows that ECAs can be successfully implemented using ECAT. The proof-of-concept components can serve as an example for future components. The ultimate goal is to create a repository of components which can be shared and reused among researchers. Since ECAT supports and builds upon BML as an interface, a collection of ECAT components will also help the SAIBA framework to grow to a wider accepted standard.

Table of Contents

Preface ii

Abstract iii

Table of Contents iv

List of Figures vi

List of Tables vii

1 Introduction 1

1.1 Introduction 1

1.2 Embodied Conversational Agent systems 2

1.2.1 NUMACK 2

1.2.2 Sam / Alex 3

1.3 A toolkit for rapidly creating ECAs 3

1.4 Approach and outline 8

2 Existing ECA approaches 10

2.1 General 10

2.2 Behavior generation systems 11

2.3 Behavior realization 13

2.4 Summary of modularization of ECA's 16

3 The Embodied Conversational Agent Toolkit (ECAT) 19

3.1 (Additional) Modularization 19

3.2 Architecture Design 20

3.3 High-level behaviors in a standard format 21

3.3.1 MURML 22

3.3.2 BML 25

3.4 Low-level behaviors in a standard format 26

3.4.1 The requirements 26

3.4.2 Choosing a format 26

3.5 The three stages of ECAT 29

3.5.1 Interpretation 29

3.5.2 Compilation 31

3.5.3 Translation 32

4 A proof of concept prototype 35

4.1 NUMACK's Brain Interpreter 35

4.1.1 The Input 35

4.1.2 Global system description 35

4.1.3 The architecture 36

4.1.4 Separation between Brain and Body 37

4.1.5 The Interpreter 38

4.1.6 The output 38

4.2 ECAT Compiler 39

4.2.1 Articulated Communicator Engine (ACE) 39
4.2.2 Compiler Architecture 41

4.3 Open Inventor Translator 45
4.3.1 The input 45
4.3.2 The translator 46
4.3.2 The output 48

5 Additional ECAT components (under development) 49

5.1 WOZ Interpreter 49

5.2 Ogre3D translator 49

5.3 TGS Open Inventor translator 50

5.4 Panda3D translator 50

6 Evaluation 53

7 Conclusion 54

8 Recommendations 55

References 56

Appendices 61

Appendix A: Original NUMACK Skeleton 61

Appendix B: Partial Scenegraph of the NUMACK Skeleton 67

Appendix C: Partial Max.xml 71

Appendix D: Class diagram of the Compiler 73

List of Figures

Figure 1 General structure for a multimodal behavior generating system [12] 1
Figure 2 NUMACK and Sam (/ Alex) 3
Figure 3 The split in behavior realization proposed by the ECA Toolkit 4
Figure 4 possible high-level behavior specifications 4
Figure 5 simplified low-level description of wave gesture 5
Figure 6 Utility functions for joint rotations and audio playback 7
Figure 7 The three modules of the Embodied Conversational Agent Toolkit (ECAT) 8
Figure 8 ECAT with relation to the three stages of SAIBA 10
Figure 9 Architecture of the MPML3D framework [51] 11
Figure 10 Example of an MPML3D behavoir 12
Figure 11 Architecture of Behavior Expression Animation Toolkit (BEAT) [16] 12
Figure 12 Architecture of the NVBGenerator [44] 13
Figure 13 Example MURML utterance referring to a gesture from a gesticon 13
Figure 14 REA architecture [13] 14
Figure 15 High and Low level commands in Pantomime [17] 14
Figure 16 Rapport Agent architecture, using Smartbody for behavior realization [22] 15
Figure 17 Gesture Engine architecture, used in the Greta agent [26] 16
Figure 18 Existing ECA approaches w.r.t. the scope of ECAT 18
Figure 19 ECAT Architecture 20
Figure 20 Gesture phases 21
Figure 21 Example MURML specification [35] 22
Figure 22 the three body planes [35] 23
Figure 23 Core BML which's elements can be extended or new elements can be added. [69] 25
Figure 24 Example BML description [69] 26
Figure 25 joint rotation specified in COL 27
Figure 26 audio playback specified in COL 28
Figure 27 Rotation format of the Compiler Output Language (COL) 28
Figure 28 DTD for the Compiler Output Language (COL) 29
Figure 29 Four types of possible input behaviors mapped with example output 31
Figure 30 Start of sample interaction with NUMACK 35
Figure 31 NUMACK Architecture [63] 36
Figure 32 Microplanning architecture [14,40,65] 37
Figure 33 The BEAT architecture [16] 38
Figure 34 Example MURML Utterance from NUMACK's Brain 39
Figure 35 The states of execution of a chunk [41] 40
Figure 36 ECAT-Compiler architecture, shown in relation to ACE and the outer ECAT components 42
Figure 37 Scenegraph representation of original 'geometry data' versus new 'joint data' 45
Figure 38 COL Example 46
Figure 39 Class diagram of the Open Inventor Translator 46
Figure 40 Examiner Viewer showing the scenegraph 47
Figure 41 The result, showing a 'go left'-gesture. 48
Figure 42 WOZ-Panel for Sam / Alex [22] 49
Figure 43 Class diagram of the Panda3D Translator 50
Figure 44 Panda3D model controlled by 'ECAControl' 51
Figure 45 Panda3D model controlled by 'ECAControl', corrected 51

List of Tables

Table 1 Classes of High-level behaviors 5
Table 2 Classes of Low-level behaviors 6
Table 3 Typical elements needed for “Hi → Lo” 7
Table 4 Symbolic values which can be used for gesture descriptions in MURML [35] 24
Table 5 Interpreter functionality 29
Table 6 Compiler functionality 31
Table 7 Translator functionality 32
Table 8 ACE-classes shown from base class to most specialistic class 44
Table 9 Orientation of coordinate axes for zero rotation of the left-elbow joint 52

1 Introduction

1.1 Introduction

Embodied Conversational Agents (ECAs), also know as Virtual Humans (VHs), have been around for over a decade. ECAs are in essence agents. According to Wooldridge [70] an agent is: an encapsulated computer system that is situated in some environment and that is capable of flexible, autonomous action in that environment in order to meet its design objectives. According to Russell and Norvig [58] an agent is: “anything that perceives its environment through sensors and acts upon that environment through effectors”. Within this definition humans can also be seen as agents. ECAs are in fact agents in the form of Virtual Humans which are able to converse with humans using multimodal input and output. Input is most commonly gathered using sensors like, microphones and cameras, while output modalities are similar to those used in human to human conversation, like speech and gesture. ECAs have proven to be useful for different applications such as in training scenarios [28,34], for studying human behavior [14] as well as being an intuitive interface for Human Computer Interaction [18].
More recent, in addition to the development of these applications, researchers also have focused on working towards a modular architecture and interface standards, which would make it possible to more easily share and reuse each other’s work [23]. However a lot of approaches from the past years have introduced almost just as many markup languages for specifying behaviors of ECAs (Among which APML, RRL, PAR, MPML, etc. [12,20,54,56]). Efforts working towards interface standards from Vilhjalmsson and Marsella [46] and more recent from Kopp et. al. [38] have introduced the SAIBA framework. They show a three stage framework which lays down a general structure for every multimodal behavior generation system (see Figure 1).

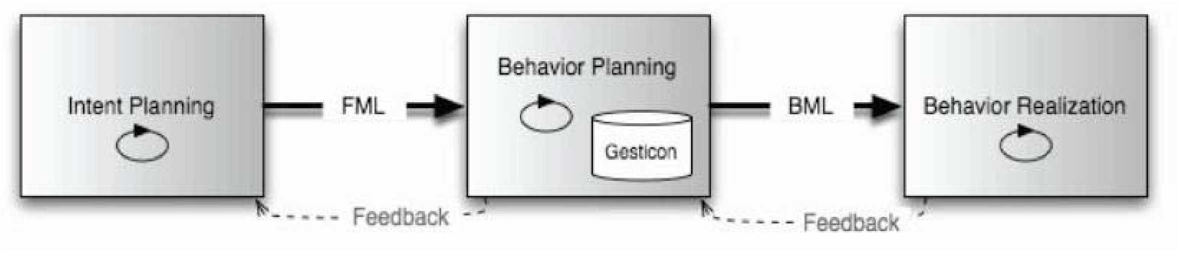


Figure 1 General structure for a multimodal behavior generating system [12]

A common issue with many ECA systems is the fact that the behavior realization module is often implemented as one monolithic component or as a multiple components which are so tightly coupled that it’s often hard to untangle them. NUMACK [65], a direction giving ECA, uses a behavior realization component know as the Articulated Communicator Engine (ACE) [41], which renders an ECA using a behavior description as input. The behavior realization component used at the Institute for Creative Technologies (ICT) and the Information Sciences Institute (ISI) is called Smartbody [66]. Smartbody is also capable of rendering ECAs using behavior specifications. The animation engine mentioned in the MPML3D framework [51] has even more responsibilities in addition to rendering an ECA using behavior specifications. All these systems rely on a graphics engine, or better said a rendering or game engine[1]. NUMACK for instance uses Open Inventor [1], the Virtual Humans using Smartbody use

[1] Rendering not only involves graphics, but also audio, therefore the term game engine is also used for: the engine responsible for the audiovisual rendering.

Unreal Tournament [2] and the MPML3D framework uses a self-made engine which uses OpenGL [3] and OpenAL [4]. Some of these engines require expensive licenses which makes them unsuited for a lot of researchers. To be able to share one's behavior realization component with other researchers this component has to be decoupled from the rendering engine.
The field of research concerned with ECAs is composed of a diverse group of interested parties. Among computer graphics experts this includes a large number of linguists, psychologists, and artificial intelligence researchers who may have little or no professional interest in the field of computer graphics per se. Most of these researchers would like nothing more than for their virtual humans to be expressive and easy to control, without retaining a dedicated computer science/graphics researcher on staff. Ideally one wants to focus on their own research without having to be bothered with the rest of the agent. A psycholinguistic researcher may be interested in how people respond to backchannel behavior [24] in the form of a subtle head nod. However this very same researcher does not want to be bothered with the animation of these head nods.
Up until now most systems have focused on the representation of the knowledge structure of multimodal behavior, like the markup languages mentioned above. Most often the different sub-modules of these systems are tailored for the specific system which makes them not directly usable as a module for a different system.
Other systems have focused mainly on the behavior realization component (ACE [41], Gesture Engine [25], Pantomime [17], SmartBody [66]). Most often these systems are to a lesser extent designed to support a wide variety of rendering engines. Also these systems are less suitable to be used as a component to be used with a different behavior generation component.
This thesis describes a framework called the ECA Toolkit. ECAT can be seen as a behavior realization system consisting of multiple (open-source) components which can be shared among researchers. A proof of concept prototype has been implemented using parts of existing ECAs. This prototype will also be discussed in this thesis and can serve as a base for future implementations.

1.2 Embodied Conversational Agent systems

The implementation described in this thesis uses (parts of) two existing Embodied Conversational Agent (ECA) systems which are used in the ArticuLab [5]. A short description of these systems will be given here. The architectures and the modules used for ECAT will be discussed in more detail in chapter 3, 4 and 5.

1.2.1 NUMACK

Northwestern University Multimodal Autonomous Conversational Kiosk (NUMACK) [65] is an ECA capable of giving directions on Northwestern's campus using speech, gestures and facial expressions. Humans can interact with the system using head movements and speech. NUMACK is capable of generating coordinated speech and gesture; these are realized using automatically synthesized speech and a kinematic body model.

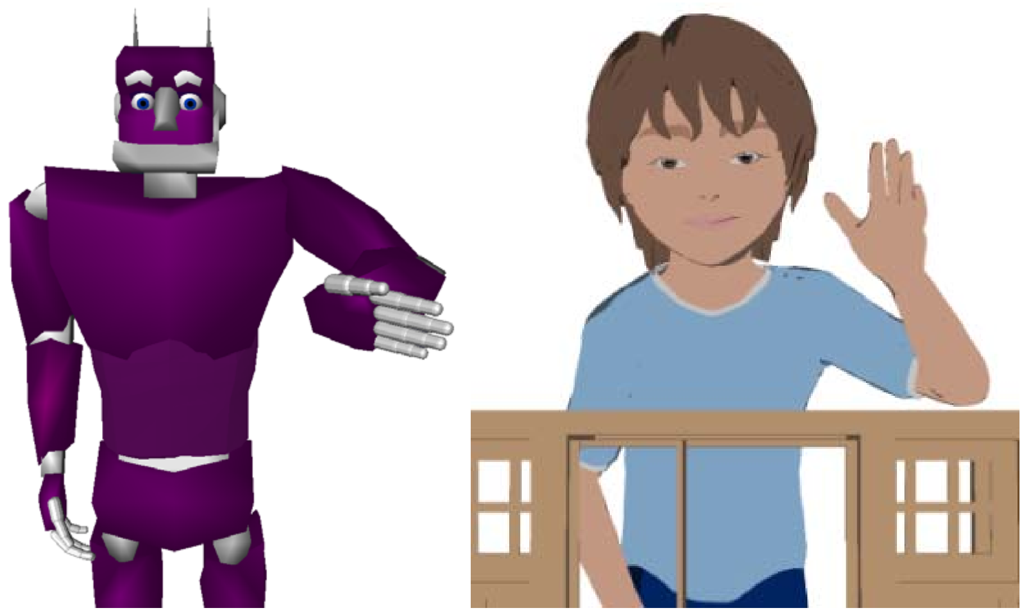


Figure 2 NUMACK and Sam (/ Alex)

1.2.2 Sam / Alex

Sam and Alex are ECAs which interact with children; both projects are sub-projects of the Virtual Peer (VP) project [64]. The VP project uses an ECA, controlled by a Wizard of OZ (WOZ) interface [47], to engage in collaborative storytelling with children, these ECAs are animated using Adobe Flash. The project aims to promote development of language skills of children. Alex's physical appearance is designed to be gender and race ambiguous; however both verbal and nonverbal behaviors are based on models of African American children. Sam's physical appearance is designed to be gender ambiguous. The original Sam was developed at MIT [60] and was an autonomous ECA, the current version is also used to collaborate with children with autism. Especially for research on children with autism, this project also focuses on story authoring. More specific it will enable children to control, build and finally interact with the VP.

1.3 A toolkit for rapidly creating ECAs

Recent research in the Virtual Peer project [30] (also see 1.2) raised the need for additional animations for Alex. With the current implementation, in Adobe Flash, it's quite cumbersome to change existing animations or to update existing ones. For this reason it was decided to create a new behavior realization component for the VP project.
Previously, bridging ECAs with different behavior realization systems required massive refactoring due to a lack of separation between high-level behavioral specifications and low-level behavior realization directives. As a result, it became standard practice to "throw the baby out with the bathwater" and replace the entire system rather than rebuilding large portions of it from scratch. However, due to a pervasive desire for code reuse, portions of the older systems would often be reused in the new systems. The larger issue, then, became the reuse of components that were not originally designed to be highly modular in the first place. Ultimately, this ongoing cycle produced an ever-worsening situation whereby ECA developers spent as much time decoupling and refactoring various subsystems as they did designing and implementing new research ideas.

The Embodied Conversational Agent Toolkit (ECAT) project was started to tackle this issue. It aims to minimize time spend on decoupling and refactoring ECA components, hereby facilitating rapid integration of (new) components and the construction of ECAs as a whole.
The SAIBA framework (see 1.1) already proposes three stages which generally lay down the structure of an ECA. The third stage of behavior realization also includes the audiovisual rendering of an ECA, which can be seen as the end result or the output. Systems such as ACE [41] and Smartbody [66] are often integrated with a rendering engine to form the complete behavior realization component. ECAT makes a clear separation within the process of behavior realization. It decouples on the one hand: the conversion of high-level behavior specifications into low-level directives for animation and on the other hand: the rendering of these low-level directives (see Figure 3).

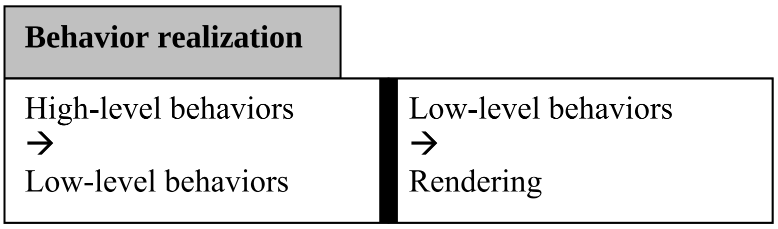

Behavior realization
High-level behaviors → Low-level behaviors
Low-level behaviors → Rendering

Figure 3 The split in behavior realization proposed by the ECA Toolkit

Before going into both subtasks of behavior realization, the meaning of high-level and low-level behaviors will be discussed first.

Figure 4 shows an example of what such high-level specifications might look like.

-Say("Hello my name is Harry") & Gesture("wave")
-Say("Take <emphasis>this</emphasis> ball") & Gesture(grab("ball"))
-Say("And who are you?") & LookAt("Person2") & Gesture("slouchRight")
-Say("What is your name?")
-Say("That's funny") & ShowEmotion("Smile")
-Gesture("Headnod")
-Say("Hello there, how are you doing?") & Gesture("wave") & ShowEmotion("Smile")
-Say("And next you make a right.") & Gesture("Make_a_right")

Figure 4 Possible high-level behavior specifications

The behaviors shown above are typically the result of the behavior planning stage of the SAIBA framework. Similar to the SAIBA framework Tepper et. al. [65] define three subtasks in the generation of coordinated language and gesture. They refer to these tasks as 1) Content planning, 2) micro planning and 3) surface realization. These three tasks are very similar to the three stages of the SAIBA framework. In fact the three tasks or stages could be seen as: 1) figuring out what to communicate, 2) how to communicate it and 3) communicate it [65].
An example would be:

1) **Greet**,
2) **Say("Hello my name is Harry") & Gesture("wave")** and
3) **'speaking and gesturing'**.

High-level behavior specifications will henceforth be referred to as the behavior specifications resulting from stage 2: behavior planning, which can be used as input for stage 3: behavior realization. It's important to note that timing information of different co-occurring behaviors (such as speech and gesture) is also specified in a high level specification. This can vary from behaviors being specified to be executed parallel (like in Figure 4) or in sequence, but it's also

possible to have more fine grained timing information. Gestures can for instance be specified to start when a certain word is uttered.
High-level behaviors can be divided into four classes, as is shown in Table 1. Every behavior in Figure 4 can be divided into one of these four classes.

Classes of High-level behaviors:	**Examples:**
Gestures	Head, body and limb gestures
Gaze	Look at someone/something
Text + paraverbals	Text to be spoken with optional paraverbal information such as emphasis, intonation and tone.
Facial expressions	Happy, sad, angry, surprise, disgust

Table 1 Classes of High-level behaviors

On the other hand, low-level behaviors are defined on a much lower level of abstraction as the corresponding high-level behavior. The wave gesture mentioned in Figure 4 may for instance take two seconds to complete. Figure 5 shows a simplified list of low-level descriptions corresponding to: **Say(…) & Gesture("wave")** from Figure 4.

```
At t=0.0s
        -Play(hello.wav)
At t=0.2s
        -Rotate(l_elbow,y,30) //Rotate arm up
        -Rotate(l_elbow,z,90) //Rotate arm around to make palm face forward
At t=0.4s
        -Rotate(l_elbow,y,60) //Rotate arm up
At t=0.6s
        -Rotate(l_elbow,y,115) //Rotate arm up
At t=0.8s
        -Rotate(l_elbow,x,40) //Rotate arm to the right
At t=1.0s
        -Rotate(l_elbow,x,60) //Rotate arm to the right
        -Rotate(l_wrist,x,15) //Rotate wrist to the right
At t=1.2s
        -Rotate(l_elbow,x,40) //Rotate arm back to the left
        -Rotate(l_wrist,x,0) //Rotate wrist back to default orientation
At t=1.4s
        -Rotate(l_elbow,x,0) //Rotate arm back to default orientation
At t=1.6s
        -Rotate(l_elbow,x,-40) //Rotate arm to the left
At t=1.8s
        -Rotate(l_elbow,x,-60) //Rotate arm to the left
        -Rotate(l_wrist,x,-15) //Rotate wrist to the left
At t=2.0s
        -Rotate(l_elbow,x,-40) //Rotate arm back to the right
        -Rotate(l_wrist,x,0) //Rotate wrist back to default orientation
```

Figure 5 Simplified low-level description of wave gesture

The description above shows a very simple wave gesture specified at 5 frames per second (fps). 5 fps is very low; normally animations will be played at 20 fps or more. This will lead to an even lengthier description of the very same animation. Also crafted animations often contain rotations of multiple joints (not only elbow and wrist) and often will be specified with rotation matrices. Figure 4 specified that speech and gestures were to be executed in parallel,

therefore audio playback is started at t=0.0s. At t=0.0s the gesture is also started. In a real world example the gesture would oftentimes be timed to start before the speech. This way the most effortful part of the gesture will co-occur with the speech which is affiliated with the gesture. The specifications in Figure 4 and Figure 5 merely serve to demonstrate what high and low-level behaviors might look like. More complex examples, with more fine-grained timing, will be shown in chapter 3.
Low-level behavior descriptions can be divided into three classes as shown in Table 2.

Classes of Low-level behaviors:	**Examples:**
Joint rotations	***Rotate(l_elbow,x,90)***
Audio playback directives	***Play(hello.wav)***
Facial deformations	***DeformFaceMuscle(r_orbicularis_oculi, 0.1)***

Table 2 Classes of Low-level behaviors

An important difference between high and low-level descriptions is that a high-level description defines a complete behavior, while a low-level description defines only the characteristics of this behavior for one frame.

As has been mentioned above the third stage of behavior realization can be split into two steps:

1. Hi → Lo
   Converting high-level specification into low-level ones
2. Lo → Rendering
   Rendering these low-level specifications

The discussion above has shown what is meant by high-level and what is meant by low-level behaviors. The first step inside the stage of behavior realization involves conversion of the former to the latter.
High-level behaviors specify a complete behavior and low-level behaviors specify only the characteristics per frame. This means the process: “Hi → Lo” involves filling in the blanks. This is oftentimes done by using a gesture database or a gesture lexicon, which will henceforth be referred to as gesticon. A gesticon contains a number of (low-level) gesture specifications, where the specific joint rotations are specified for every keyframe. Such specifications are most often exported from animation packages such as: Maya or 3D Studio Max.
A keyframe representation of a gesture only gives the low-level specification at certain, important, frames. To be able to create low-level specifications for every frame a smooth transition has to be made between keyframes. In addition to a smooth transition between subsequent keyframes, subsequent gestures also need to be blended smoothly. Blending between subsequent gestures is not as trivial as it may seem, because the end pose of one gesture may be very different from the start pose of the next gesture.
Some gestures may also involve pointing or grabbing. Gestures of this kind are typically parameterized, with for instance (the location of) the object which is pointed to. This very same kind of gestures often requires inverse kinematics (IK). IK is applied to limbs to provide for instance the complete configuration of the arm (rotation of shoulder, elbow, etc.) when the desired location of the wrist is known. IK and a gesticon can be used to map gestures, gaze behaviors and also facial expressions to a corresponding low-level representation. For humans the realization of motion is often referred to as motor control. This term is often adopted in virtual human systems.

Generation of speech however is a completely different ball game. In fact an audio playback directive like: **play(hello.wav)** could be seen as higher level of abstraction as the specification: **say("hello my name is Joe")**. However, for a computer it's easier to play a wave-file than to speak. In the process of "Hi → Lo" the high-level text is used by a text-to-speech (TTS) synthesis system to generate the speech. The TTS system also provides phoneme timings which can be used for cross-modal synchrony.
Table 3 shows a summary of elements which are typically needed for "Hi → Lo", which have been mentioned above.

Motor Control:
Gesticon
Animation blending
Inverse Kinematics (IK)
Speech production:
Text-to-speech (TTS) synthesis

Table 3 Typical elements needed for "Hi → Lo"

The second step (see Figure 3) within the process of behavior realization referred to as "Lo → Rendering" will be discussed in the following.

The separation of Hi → Lo and Lo → Rendering can be particularly useful because most of the presently used engines[2] for ECA representation have utility functions to play audio or to apply joint rotations (see Figure 6) to the ECA's body model. Thus these engines could well be used to turn low-level behavior descriptions into an audio-visual representation.

```
Panda3D [6]:
        jointNode = actor.controlJoint(None,"modelRoot","l_elbow")
        jointNode = setHpr(h,p,r);

        helloSound = loader.loadSfx("hello.wav")
        helloSound.play()

Ogre3D [7] (with FMOD Soundmanager):
        Bone b = skeleton.getBone("l_elbow");
        b.setOrientation(w,x,y,z);

        helloSound = soundMgr->CreateSound(String("hello.wav"));
        int channel;
        channel = soundMgr->PlaySound(helloSound,sceneNode, &channel);
```

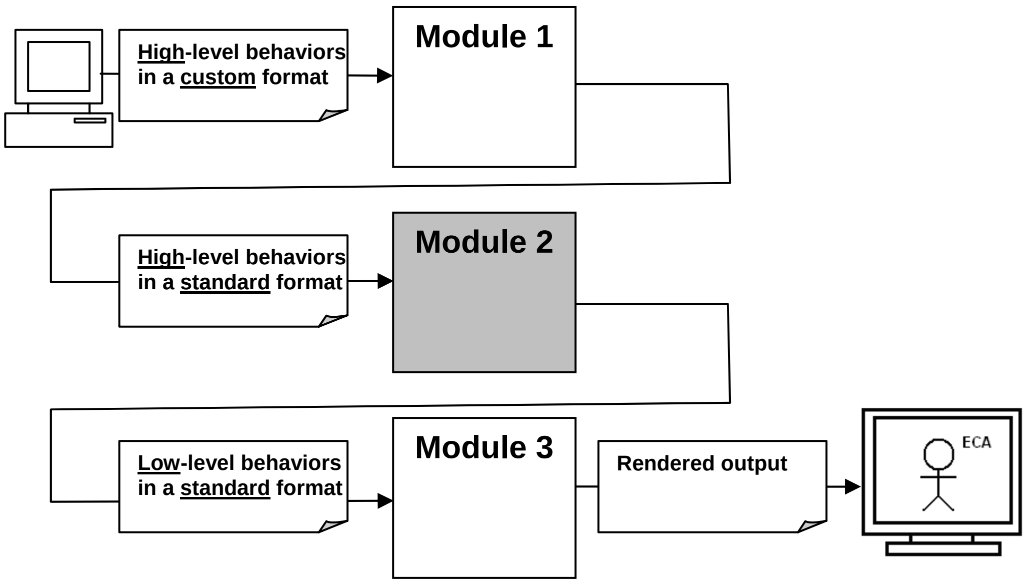
Figure 6 Utility functions for joint rotations and audio playback

With utility functions listed in Figure 6 the above-mentioned task "Lo → Rendering" is in fact a relative simple task which can be divided into two simple steps. The first being: parsing low-level behaviors specified in a predefined format. The second being: using utility functions to render these behaviors.
On the other hand, the complete task of turning high-level descriptions into rendering (behavior realization) is much more complex. The complexity mainly resides in the task: "Hi → Lo" which has been discussed above.
The complete task of behavior realization has been focused on by recent research [41,51,66]. Since these behavior realization systems are integrated with game engines they internally use

2 This could be any engine ranging from Ogre3D, Panda3D, AgentFX . It could also be a self tailored engine using a graphics library like OpenGL or Direct3D with similar functionality.

utility functions like the ones listed in Figure 6. The parameters for these functions correspond to the ones found in low-level specifications. Therefore a behavior realization system needs some form or internal representation for these parameters. Because this internal representation is needed it's a logical next step to create two different modules: firstly creating the low-level specification and secondly rendering this low-level specification.

The Embodied Conversational Agent Toolkit (ECAT) proposed in this thesis builds upon this modular separation of the process of behavior realization. The toolkit consists of three components (see Figure 7), most often the core component (responsible for "Hi → Lo") can be kept static (marked grey in Figure 7), while the outer components are more likely to be replaced by alternatives. The two outer components are responsible for relative simple tasks which makes them easier to replace by new or other parts. The first of these two outer parts is responsible for turning a behavior specification into a standard format and the second is responsible for: "Lo → Rendering".


High-level behaviors in a custom format
Module 1
High-level behaviors in a standard format
Module 2
Low-level behaviors in a standard format
Module 3
Rendered output
ECA
Figure 7 The three modules of the Embodied Conversational Agent Toolkit (ECAT)

1.4 Approach and outline

The main focus of this thesis will be the Embodied Conversational Agent Toolkit (ECAT) which has been introduced in the previous section. When adopted by enough researchers a repository of modules can be built and used to easily share and reuse each other's work. This repository will mainly contain modules for the two outer components, but can also contain modules which can be used as core component. An example of the first component would be: "A BML to MURML converter"[3]. Systems like Smartbody or ACE can be used as a base for the core component. The third component could be "A renderer of low-level behaviors using Panda3D".
Using this approach ECAT aims to facilitate rapid integration, sharing and prototyping of different ECA components, for either developing new ECA systems or updating parts of existing ones.
The work presented in this thesis shows my work during my final master project. The project was started due to the need for a new system for the Alex agent (see beginning of section 1.3).

[3] These are both markup languages which will be discussed in greater detail in the remainder of this thesis

My work started from the first outline, which was already present. This outline included the three modules shown in Figure 7. I started my approach by first studying existing ECA architectures (chapter 2). The results of this study were then used to create a more detailed design of the toolkit and its components (chapter 3). Using this design the end goal was to implement a proof of concept prototype (chapter 4). The prototype components can serve as a base for future development.

ECAT is destined to be the toolkit to be used for future ECAs in the Articulab [5]. For the proof of concept it was chosen to reuse parts of existing systems. For this reason the larger part of my work involved studying and reusing existing software (section 4.1 and 4.2.1). To let it function within ECAT it also involved modifying this software (section 4.1.5, 4.2.2, 4.3, 5.1). To a lesser extend it also involved creating new software components completely from scratch (section 5.2-5.4).
Because the larger part of ECAT builds upon existing software (4.2.1) and existing standards (section 3.3.1 and 3.3.2) the discussion in this thesis will also contain content about work not directly carried out by me during this project. Understanding this work was an important task to eventually being able to reuse parts of it; to get a working proof of concept prototype. In a similar manner the discussion about ECAT cannot be a complete one, without including parts about (re)used work (section 3.3, 4.2.1).
The remaining chapters of this thesis: evaluate the proof of concept prototype (chapter 6), conclude on the complete work (chapter 7) and give recommendations for future work (chapter 8).

2 Existing ECA approaches

2.1 General

For over five years research on modular ECA design has found a growing support [23]. Efforts working towards interface standards from Vilhjalmsson and Marsella [46] have proposed the interface languages FML and BML. More recent work from Kopp et. al [38]. has introduced the Situation, Agent, Intention, Behavior, Animation (SAIBA) framework, which uses these interface languages. In the authors' words, SAIBA was designed to provide "a powerful, unifying model of representations for multimodal generation."[38]. To frame the generation process in terms of a general structure, the authors describe a three-stage behavior production model (also see Figure 1) consisting of:

(1) communicative intent planning,
(2) multimodal behavior planning,
(3) behavior realization

As the SAIBA framework contains three stages, two interfaces are needed in order to bridge the entire system. The SAIBA authors are primarily concerned with producing a pair of standard data markup languages for these interfaces. Of these two markup formats, early efforts have been focused on the second bridge, connecting behavior planning and behavior realization systems. This markup format has been named BML (for the Behavior Markup Language).

In a similar way Poggi et. al. make a separation of mind and body in their design of Greta [55]. In this sense the mind can be compared to stage 1 (and in partially stage 2) of the SAIBA framework, where the body can be compared to the remainder of framework.

Since the SAIBA framework lays down a general structure for the design of in fact any ECA, the term *behavior realization* will be adopted and used for the final stage. The distinction between the first two stages isn't relevant to the discussion in this thesis; therefore both stages combined will be referred to as *behavior generation*.

The ECA toolkit (ECAT) which has been introduced in the previous chapter and which will be discussed in the remainder of this thesis ties into the very end of the process of *behavior generation* and is responsible for *behavior realization* of ECAs (see Figure 8). With this in mind the next section will focus on *behavior generation* systems, which illustrates where ECAT can tie into existing systems. The final section of this chapter will discuss *behavior realization* systems which could be used at the heart of ECAT, since they share a common goal. They all aim to realize generated behaviors.

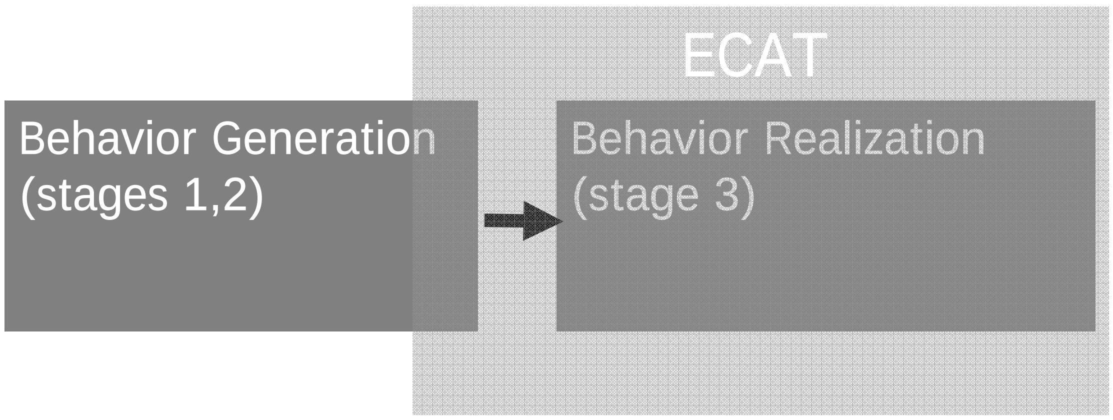

ECAT
Behavior Generation (stages 1,2)
Behavior Realization (stage 3)

Figure 8 ECAT with relation to the three stages of SAIBA

2.2 Behavior generation systems

The MPML3D authoring language provides a means for abstracting elements of the content creation process for interactive agent systems [51]. This approach is motivated by a desire to support digital content creators in the production of highly interactive and appealing content with a minimum of effort. MPML3D is the successor to MPML (the Multimodal Presentation Markup Language) [31,56], adding support for interactivity and greatly simplifying the language design to make it more accessible to non-experts. The architecture of the MPML3D framework (see Figure 9) is divided into three parts: user layer, developer layer and animation engine.

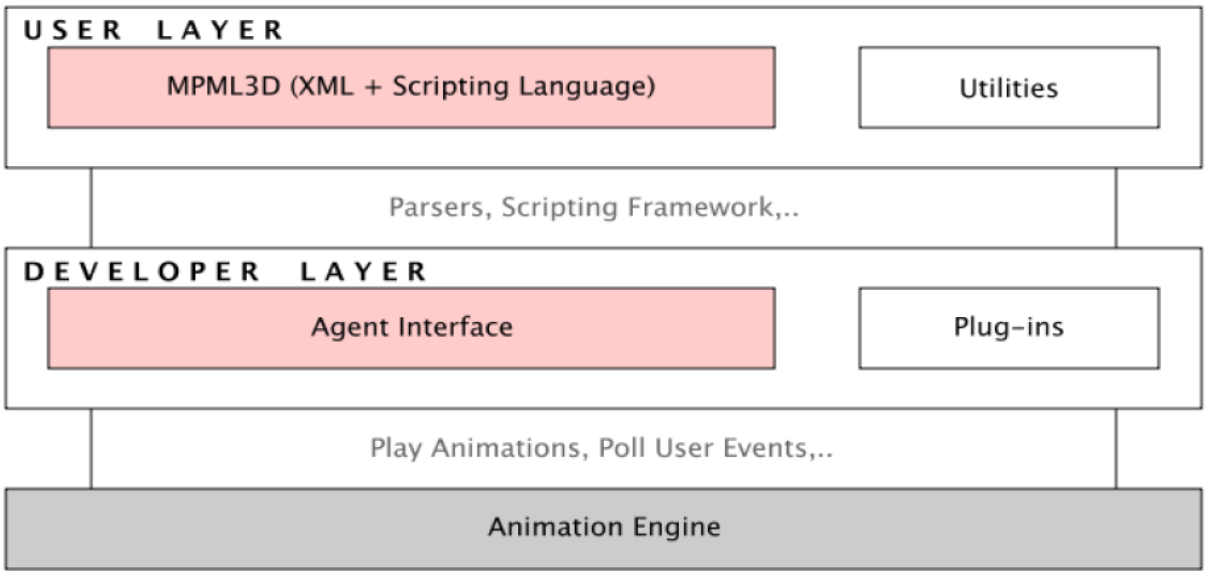

USER LAYER
MPML3D (XML + Scripting Language)
Utilities
Parsers, Scripting Framework,..
DEVELOPER LAYER
Agent Interface
Plug-ins
Play Animations, Poll User Events,..
Animation Engine

Figure 9 Architecture of the MPML3D framework [51]

The content creation for interactive presentations resides in the user layer. This is where the complete interactive scenario is stored. Behaviors (like the one in Figure 10) can be set to be executed when a user interacts or when another agent does a certain action. The behavior in Figure 10 is started when a user presses a key. Another possibility would be to trigger the start of the task when another agent finishes his introduction.
The developer layer is an intermediate layer which uncouples the user layer from particular implementations [51]. At runtime this layer is responsible for parsing the content selected from the complete interactive scenario. Also the agent's states are defined in this layer, which correspond to their available actions.
The animation engine is responsible for handling the interaction with the user and the animation of the interactive scenario.
Basically user interactions (animation engine) can trigger new tasks (the user layer), which causes the actions inside the task to be issued (developer layer), which on its turn sends a rendering request to the animation layer.
The animation engine, together with the developer layer, can be seen as a behavior realization system, because these two parts 'realize' the behaviors generated in the user layer. On the other hand, the user layer where the MPML3D content is created or generated can be seen as a behavior generation system.
Current research and development at the National Institute of Informatics (NII) in Tokyo is focused on a successor of MPML3D, which is called MPML-SL. It is based on MPML3D and

uses Second Life[4], which is an online virtual world. In this world people can control avatars to interact with each other. With the new MPML-SL it is also possible to interact with agents driven by an interactive scenario.

```
<Task name="introTask" priority="0" startOn="startKeyStroke">
  <Sequential>
         <Action class="gesture">
                   <Property name="type">BowVeryPolite</Property>
         </Action>
         <Parallel>
            <Action class="speak">
                   <Property name="text">Hi, my name is Naomi Watanabe.</Property>
            </Action>
            <Action class="focus">
                   <Property name="target">User</Property>
                   <Property name="angle">-5.0</Property>
            </Action>
         </Parallel>
  </Sequential>
</Task>
```

Figure 10 Example of an MPML3D behavoir

The Behavior Expression Animation Toolkit (BEAT) [16] can also be considered to be a behavior generation system. In the author's own words, behavior generation is the core component in the BEAT architecture (see Figure 11). BEAT aims to assist animators by adding appropriate nonverbal and paraverbal behaviors to plain-text segments of dialogue that it receives as input. In order to perform this task, BEAT produces behaviors that are first synchronized with synthesized speech and then stored internally as abstract behavior specifications. In its final processing stage, BEAT translates these internal behavior representations into a customizable format that may then be used by a particular behavior realization engine (such as ACE, Gesture Engine, or Pantomime, all discussed in the next section). Thus, BEAT is best described as a multimodal behavior generation system which can be tailored for usage by a number of different realization systems by means of a specific translator module (see Figure 11).

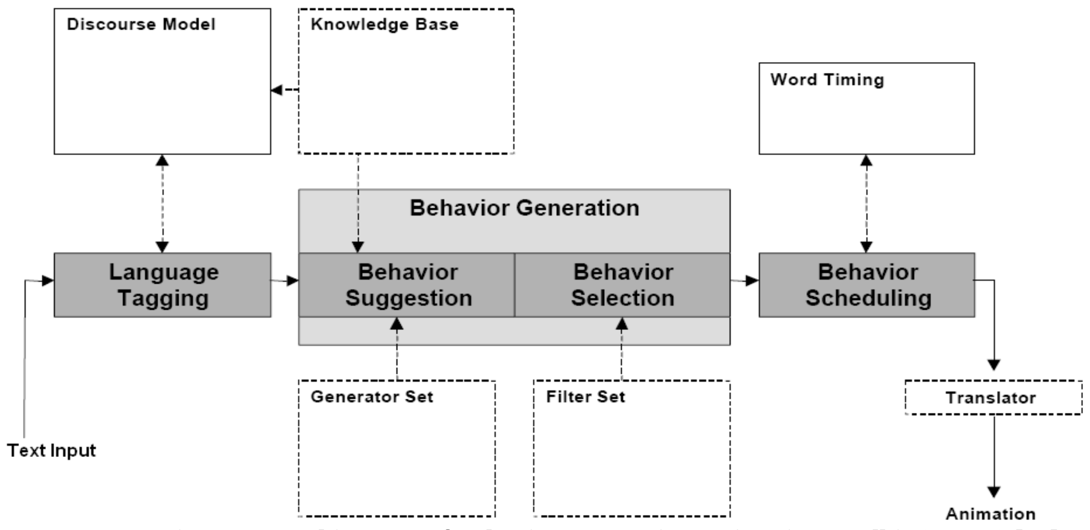


Figure 11 Architecture of Behavior Expression Animation Toolkit (BEAT) [16]

[4] See http://secondlife.com/

The NVBGenerator, proposed by Lee and Marsella [44], is also able to generate nonverbal behaviors using text and also the agent’s emotional state as input. The communicative and expressive intent of the agent is specified using the Function Markup Language (FML), which is the first interface language from the SAIBA framework. By using a natural language parser and a set of nonverbal behavior rules, appropriate behaviors are generated in BML. By using both FML and BML (see Figure 12) as interface languages the NVBGenerator can be seen as a SAIBA compliant behavior planning system and more general a behavior generation system.

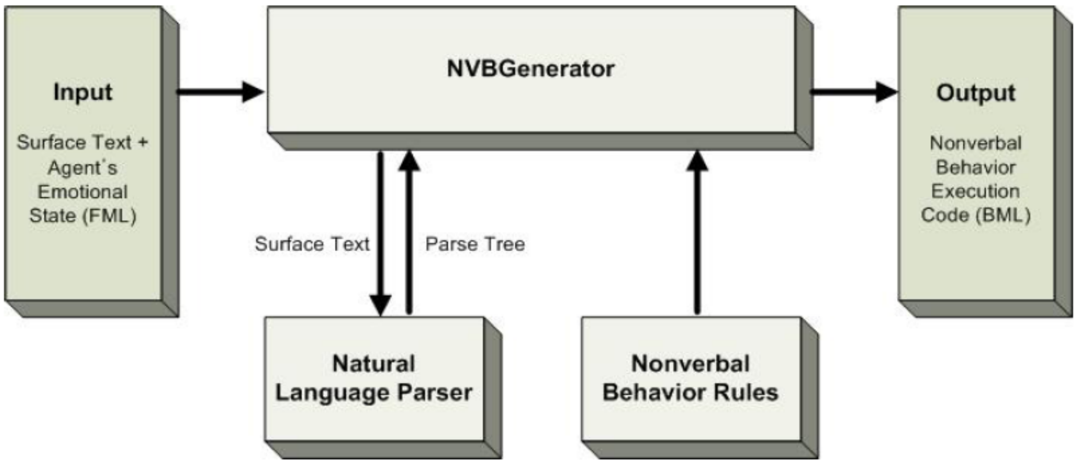


Figure 12 Architecture of the NVBGenerator [44]

2.3 Behavior realization

As has been mentioned in the previous section the developer layer and animation engine in the MPML3D [51] framework are responsible for the behavior realization of the generated behaviors from the user layer. This section will discuss additional systems which can also be seen as behavior realization systems.

The first of these systems, the Articulated Communicator Engine (ACE) [41], is a state-of-the-art system able to produce synchronized verbal, paraverbal and nonverbal behaviors for an ECA, given a behavioral specification. For input, ACE uses the Multimodal Utterance Representation Markup Language (MURML) [35]. Gestures in MURML can be specified in terms of keyframe animations but also on a higher level of abstraction in a format based on HamNoSys, a sign language notation system [57]. It is also possible to refer to canned gestures, stored in a gesticon. The gestures in the gesticon are also defined in MURML. The figure below shows an example utterance specified in MURML. It shows text: “Hello my name is Harry” which can serve as input for Text-to-speech (TTS). It also shows a gesture referred to by its communicative function: “signal_greeting”. The gesture is timed to certain points in the speech, which are marked by time-tags. A more detailed description about MURML will follow in chapter 3.

```
<utterance>
        <specification>
        Hello <time id=”t1”/>my name is Harry. <time id=”t2”/>
        </specification>
        <behaviorspec id="gesture_1">
                <gesture>
                        <affiliate onset="t1" end="t2" focus="this"/>
                        <function name="signal_greeting">
                        </function>
                </gesture>
        </behaviorspec>
</utterance>
```

Figure 13 Example MURML utterance referring to a gesture from a gesticon

Developed at the University of Bielefeld, ACE is currently used in multiple virtual human systems, including Max [37] and NUMACK [65]. ACE is directly integrated with the rendering engine, which does not make it directly possible to reuse it as a graphics engine independent behavior realization system. NUMACK for instance uses Open Inventor [1] for its visualization.
The implementation uses subclassing to extend and combine the functionality of both Open Inventor and ACE. Therefore it depends on a number of C++ libraries of the ACE system and it's also responsible for all the necessary initialization steps which are needed by ACE. Using Open Inventor and ACE in this way makes it hard to replace Open Inventor while keeping ACE.
The second of the behavior realization systems discussed in this section is Pantomime. This is an older character animation system designed for natural gesture generation [17]. Like ACE, the Pantomime system aims to combine multiple animations techniques into a coherent framework. In order to facilitate integration with external rendering engines, Pantomime provides an API and detailed instructions how it can be used for a specific implementation. Pantomime is used in ECA systems like REA (Real Estate Agent) [13] and GrandChair [61] and also the original version of Sam: Sam the Castlemate [60]. In fact it grew from earlier animation engines which were used for REA.

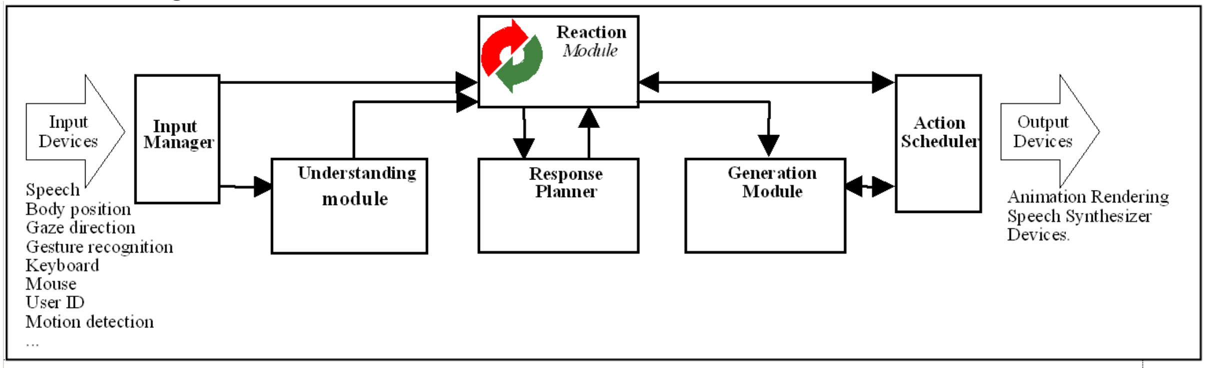


Figure 14 REA architecture [13]

According to Chang [17] Pantomime is situated just past the Action Scheduling of REA, namely the output device referred to by animation rendering (see Figure 14).
Just like the discussion in chapter 1 of this thesis, the design of Pantomime also uses high and low level commands (see Figure 15).

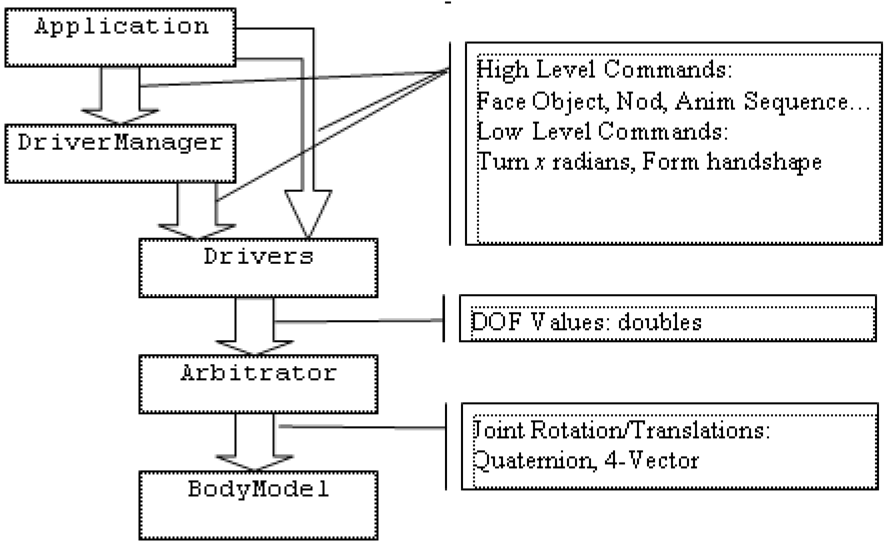


Figure 15 High and Low level commands in Pantomime [17]

In Pantomime Chang uses the term 'drivers' for the software equivalent of human motor control systems. In other words: drivers are responsible for the process of converting high-level commands into the corresponding low level ones. Pantomime uses drivers for: gaze control, inverse kinematics, keyframing and motion-capture playback.
To be portable between graphics engines Pantomime separates the motor control system from the (audio-)visual rendering carried out by the graphics system. The BodyModel (see Figure 15) is the class which can be used to interact with the graphics system. Pantomime comes with two different BodyModels, a DummyBodyModel, which outputs rotations to a text file, and a VRMLBodyModel which can interact with TGS OpenInventor[5]. Additional BodyModels to connect joint handles to actual joints in the graphics system can be created by subclassing the BodyModel base class. In a similar way it should also be possible to transfer joint rotations over the network to transfer the rendering to a separate process.
A more recent system, directly utilizing the aforementioned SAIBA framework, is SmartBody [66]. This is a character animation system capable of using controllers for keyframe interpolation, motion capture or procedural animation. The system itself generates a set of joint rotations (see Figure 16) and translations which can be transferred over the network to a rendering engine. The separation from the application and the rendering engine makes SmartBody portable to multiple rendering engines [66]. SmartBody has successfully been used in numerous virtual human projects at both the Information Sciences Institute (ISI) and the Institute for Creative Technologies (ICT) among which SASO-ST , SASO-EN [33], ELECT [27], Virtual Rapport [22], Virtual Patient [34] and Hassan [68]. Although SmartBody is portable to multiple rendering engines, all these projects use the Unreal [2] engine.

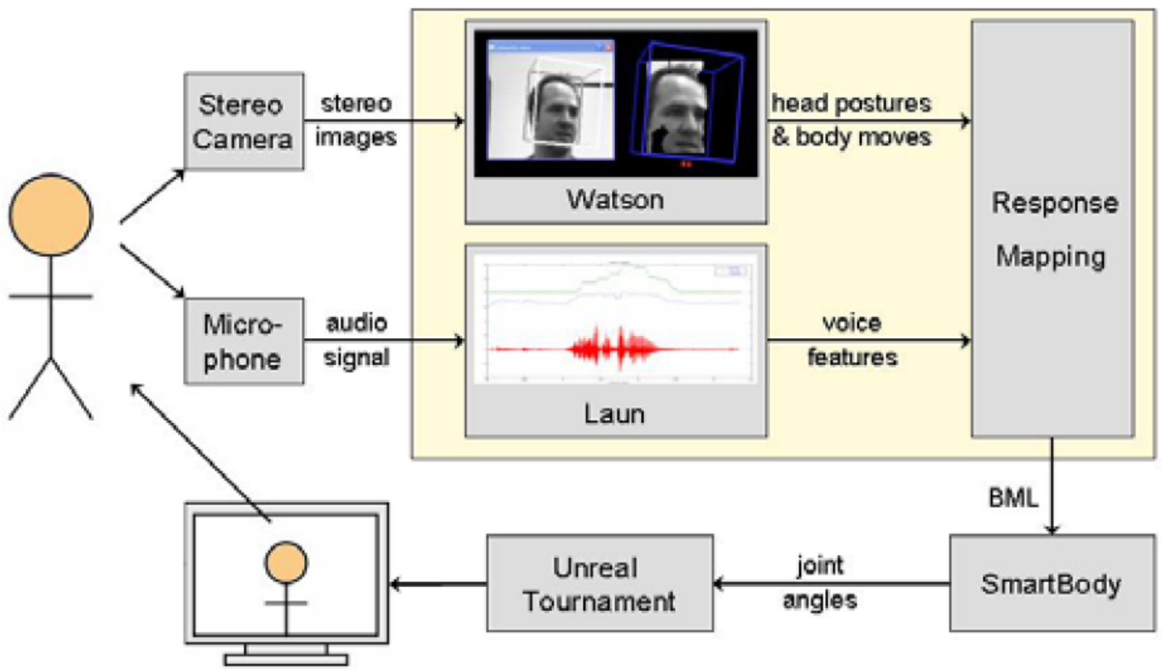


Figure 16 Rapport Agent architecture, using Smartbody for behavior realization [22]

Lastly, Hartmann et al. [25,26] describe an animation system called Gesture Engine that allows gestures to be specified in a format based on HamNoSys, in a similar way as in MURML. The Gesture Engine is used by their Greta agent which uses the Affective Presentation Markup Language (APML [20]) as input. Just like FML (see previous section),

[5] TGS Open Inventor is a licensed version of Open Inventor also available for Java. The Open Inventor used by NUMACK is a publicly available version for c++.

APML is meant to describe behaviors in terms of their communicative functions or at the meaning level. The MURML example in Figure 13 also shows an example of a behavior marked up with a communicative function (signal_greeting). Gesture Engine uses APML and a Gesture Library (see Figure 17) just like the Articulated Communicator Engine (ACE) uses the MURML specification and a gesticon to map communicative intent to gestures. The MotorPlanner is then used to calculate joint angles and timing of key frames and the Interpolator generates in between frames to create the complete animation [25]. As Figure 17 shows, this animation is then stored into an animation file. The animation format used by Gesture Engine uses Facial and Body Animation Parameters (FAPs/BAPs) of the MPEG-4 standard [53]. BAPs correspond to one degree of freedom of a joint and FAPs to translation of feature points in the mesh of a face model. There are for instance, 8 different FAPs to control the movement of the eyebrows [53]. BAPs and FAPs are encoded in a bitstream and can be played by a FAP/BAP player, which decodes the stream and animates the face and body model. In the Greta agent this is carried out by a custom player using OpenGL for visualization.

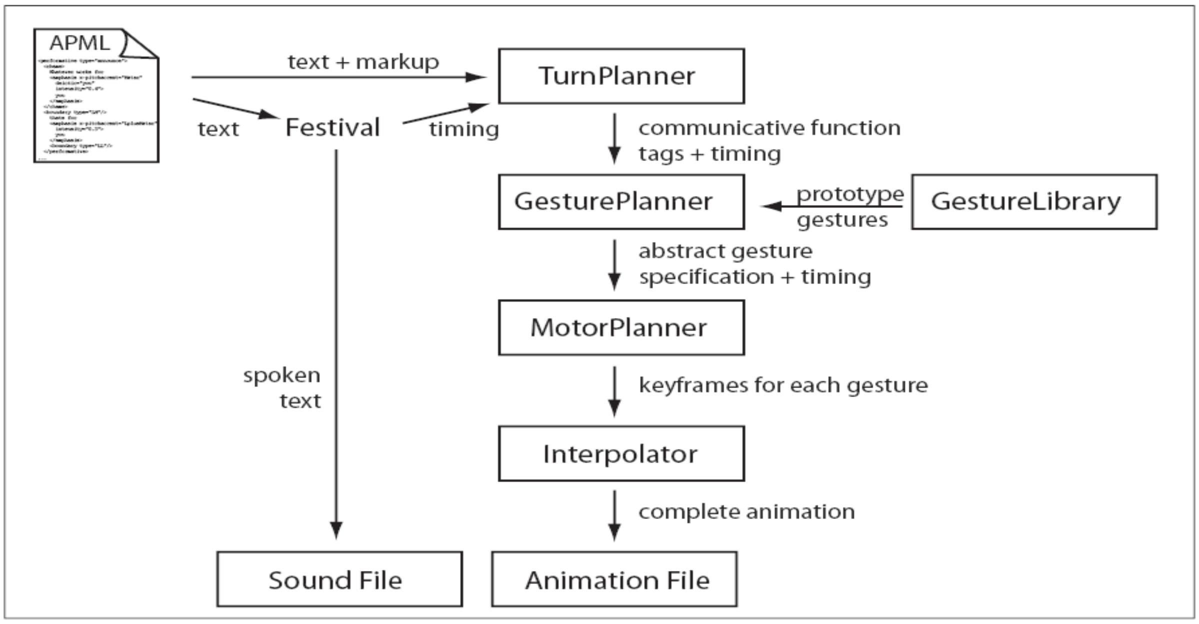


Figure 17 Gesture Engine architecture, used in the Greta agent [26]

2.4 Summary of modularization of ECA's

This chapter has shown a range of different markup languages, systems and approaches related to the field of ECAs. Section 2.1 started with the three stages of the SAIBA framework, which were divided into two stages for the sake of the discussion in this thesis.
The first stage, containing stage 1 and 2 of the SAIBA framework, has been called behavior generation and the second (stage 3 of the SAIBA framework) has been called behavior realization.
Figure 8 shows the scope of ECAT with relation to these both stages. Figure 18 on the end of this chapter is similar to Figure 9, with the difference that all the markup languages and systems discussed in section 2.2 and 2.3 are included into one schematic overview.
The figure is organized from top to bottom, starting with the simplest input at the top: plain text. Further down the line DPML (Discourse Plan Markup Language) [55] is shown, which specifies a discourse plan used to generate APML. Both APML and FML are shown on the same level, because both are used to specify input on the 'meaning level'. BEAT,

NVBGenerator and MPML3D user layer are shown next, all these systems use textual input (enriched with corresponding communicative functions or not) and generate high-level behavior descriptions. Next in line are the behavior realization systems Pantomime, ACE, SmartBody and the MPML3D Developer Layer, which all take high-level descriptions and generate low-level specifications to be used by the graphics systems. The graphics systems are the last components in the pipeline. Greta's body is drawn larger because it's responsible for turning text+communicative functions into low-level descriptions, which in fact is similar to what is carried out by two separate components in the other cases. It also bears mentioning that Figure 18 is a schematic overview which cannot be 100% correct without getting overly complex. MURML for instance can also be used to specify gestures related to communicative functions just like FML, therefore it should have been drawn starting at the same level as APML but since it can also specify behavior on a similar level as BML it also has to be drawn ending at the same level as BML. The figure also shows three red components, which are not connected to any other components, but they very well could be, as will be shown in subsequent chapters. Now the scope of ECAT is defined, the next chapter will focus on how ECAT can be useful and what's the purpose of the three different modules (see section 1.3).

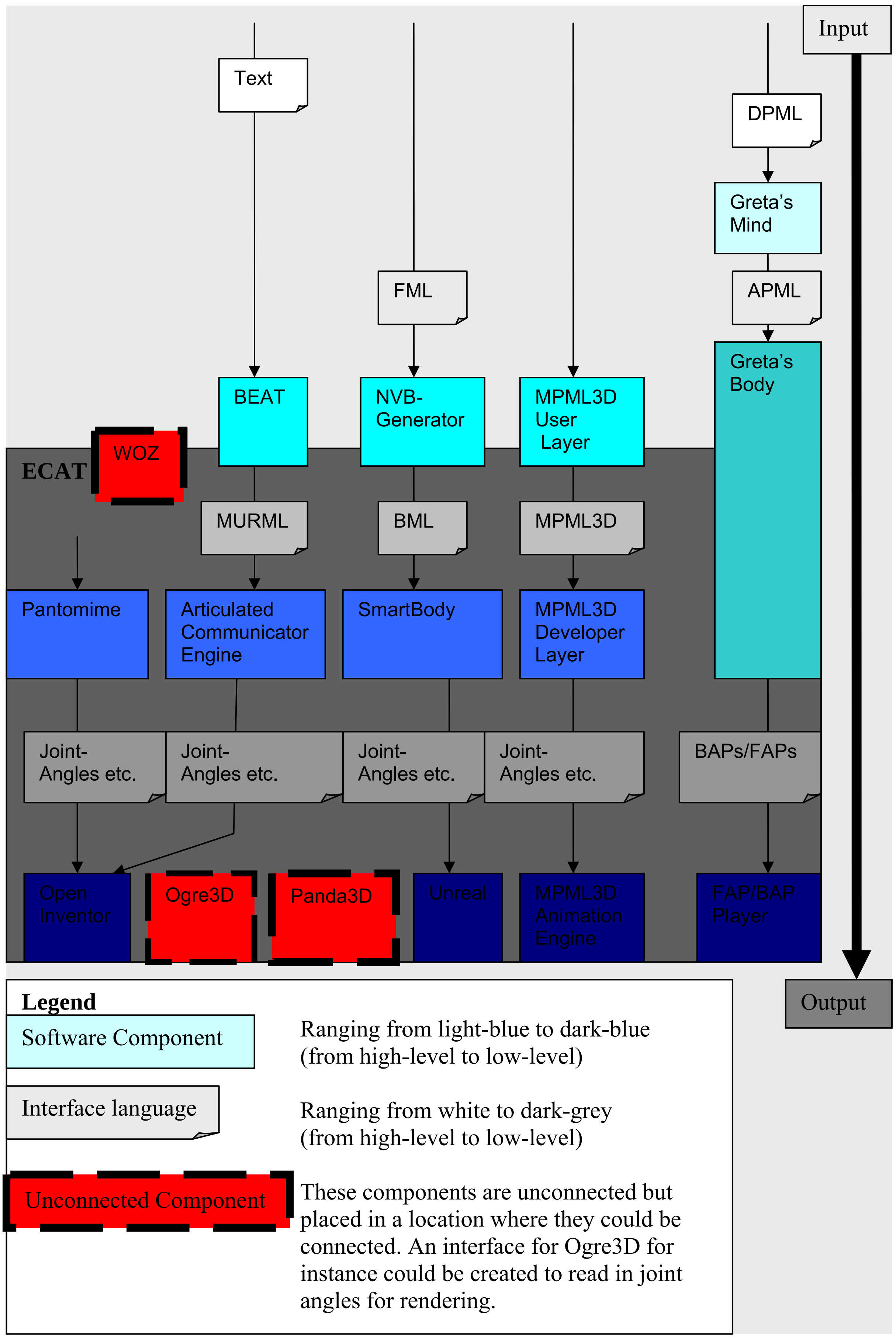


Figure 18 Existing ECA approaches w.r.t. the scope of ECAT

3 The Embodied Conversational Agent Toolkit (ECAT)

The first chapter introduced the modularization approach used by the ECA Toolkit (ECAT), see Figure 7. In the second chapter related systems have been discussed to illustrate the scope of ECAT, see Figure 18.
The first section of this chapter will start with a discussion of situations where ECAT can be useful and in which situations it might not be useful. The second section will show the architecture of ECAT. The remaining sections will discuss the interface languages and the three different modules of ECAT. As has been stated in section 1.3, one important goal of ECAT is to create a repository of these three different modules to be shared among researchers. The next two chapters will focus on specific implementations of each of these modules. The discussion in this chapter will be more general, about the architecture and the tasks and goals of each of the modules of ECAT, without going into details how this could be implemented.

3.1 (Additional) Modularization

Researchers may choose to use ECAT (modules) because they would like to use a state-of-the-art behavior realization system, such as ACE or SmartBody, for their own ECA. On the other hand, some researchers may already have a state-of-the-art system. Therefore they may not be interested to use any of the functionality of ECAT. For instance a research lab which uses BML, SmartBody and Unreal for behavior realization may have little or no interest in using ACE which uses MURML as input and Open Inventor. The Unreal engine however requires a quite expensive license, which could justify changing from Unreal to a cheaper solution. Such as for instance: Open Inventor, Panda3D or Ogre3D. Another reason why people might not want to adapt to specific ECAT modules is that their own behavior realization approach makes it easier to tweak the low-level behaviors resulting from the high-level specifications. Finding their system easier to tweak oftentimes stems from the fact that they are more familiar with their own work, which makes it easier to make small changes to it. It could also be the case that their behavior realization approach is just ‘better than the rest’ or just ‘fine as it is’. But even in this case a researcher could benefit from a public repository to gain useful insight for future improvements.
By working towards common interface standards, sharing and combining each others work will get easier and more efficient. For example, if somebody working on facial expressions uses ECAT and someone working on hand gestures also uses ECAT, both researchers can benefit from each other by implementing and combining their behavioral models without having to tweak their agents. Another example would be when one wants to test the effects of the agent appearance on learning. With a modular separation one can forget about the details about gestures and the technicalities involved in the graphics engine.
Some of the previous examples are also possible when researchers use just the SAIBA framework and not ECAT per se. In fact, the SAIBA framework already has proposed two interface standards which can prove to be an important step towards common interface standards. However it takes time before a collection of compliant systems has been developed. In the future ECAT systems can be part of this collection, thereby helping to work towards common standards. Once standards are established and accepted by a large party, sharing each others work can become a reality.
In addition to SAIBA, ECAT defines one more interface. This interface splits the process of behavior realization into two parts. The first part is responsible for the conversion of high-

level behavior to low-level specifications and the second part is responsible for rendering using these low-level specifications as input.
This separation gives new possibilities, for instance when trying out a CAVE setting, one could easily test multiple graphics engines to see which one works best. Also when moving from an expensive engine such as the Unreal engine, one could test low-cost different graphics engines before deciding to switch to a specific engine.
In general having a repository of alternative modules gives the possibility to benchmark modules, so more informed decisions can be made when choosing for a certain approach. It also allows researchers to focus on the research at hand because they do not have to focus on the complete system. The complete system can for a larger part be built from existing modules. With a repository of modules, conforming to interface standards, researchers can also more easily share their work with others.

3.2 Architecture Design

Up until section 3.1 questions like: why, where and when to use ECAT have been addressed. The remainder of this chapter will focus on the design of ECAT, the architecture, its comprised modules and what they serve for.
Figure 7, Figure 8 and Figure 18 all show ECAT and its scope. Figure 8 just shows the scope: partly overlapping behavior generation and completely overlapping behavior realization. Figure 18 is similar but shows a number of systems related to behavior generation and realization within this very same scope. Figure 7 shows the kind of input and output of the three modules of ECAT. Figure 19, which is shown below, can be seen as a combination of Figure 7, Figure 8 and Figure 18.

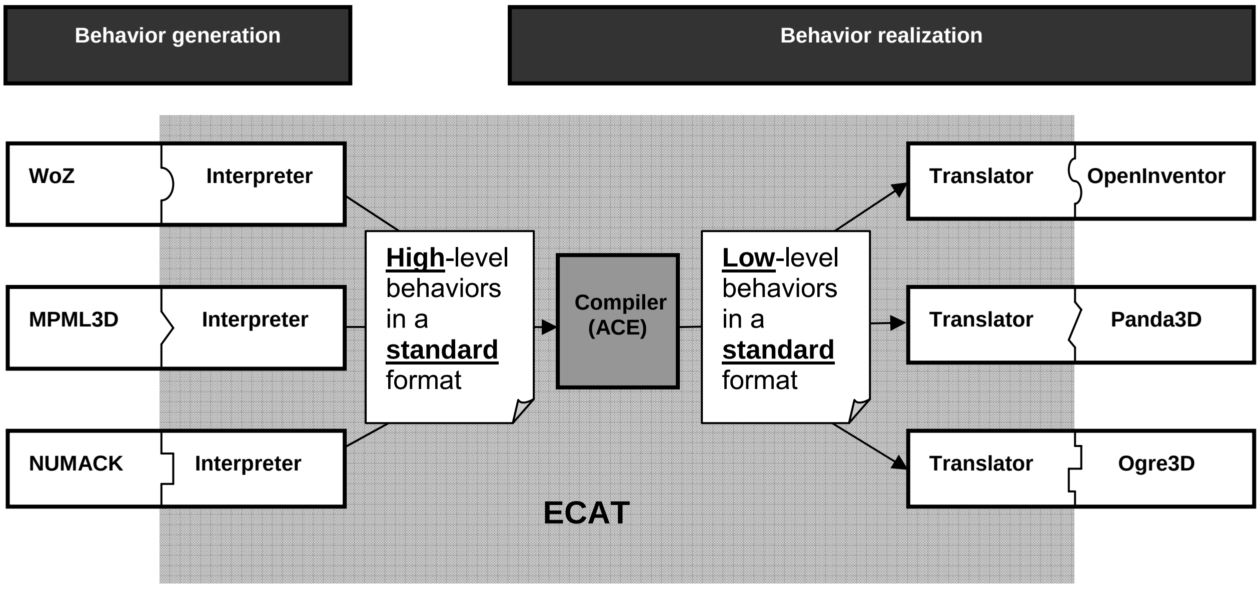


Figure 19 ECAT Architecture

While Figure 7 shows module 1, 2 and 3, Figure 19 shows the names: Interpreter, Compiler and Translator. These names will be used for the three different modules of ECAT. To keep the three realization modules platform- and language-independent, all inter-module communication is performed via a socket connection.
Within ECAT the three modules correspond to three stages: *interpretation*, *compilation* and finally *translation*. The first chapters of this thesis in combination with Figure 19 should

already give a good idea of the scope of ECAT and also the nature of the three stages. The upcoming subsections will discuss each of the stages in greater detail.
Before moving on to this discussion, it bears mentioning that ideally there is going to be a repository of modules for each of the different stages, as has been mentioned in section 1.3. The most complex work is carried out during *compilation*. The process of *compilation* is not connected to the 'outside world' but it's encapsulated between *interpretation* and *translation*. Therefore it's likely that the Compiler component (shown grey in both Figure 7 and Figure 19) can serve a multitude of ECA applications while both Interpreter(s) and Translator(s) are more tailored towards a specific implementation. For this reason an ideal repository of modules will only contain one or a few Compilers and a wide variety of Interpreters and Translators.
Before moving on to the design of the different stages, the interface data will be discussed, in Figure 7 and Figure 19 referred to as: '**High**-level behaviors in a **standard** format' and as '**Low**-level behaviors in a **standard** format'. The nature of both high and low-level behaviors has been previously discussed in section 1.3.

3.3 High-level behaviors in a standard format

The choice for the format to be used by ECAT for specifying high-level behaviors is largely motivated by the choice which system to use at the core of the Compiler[6]. Section 2.3 mentioned several candidate behavior realization systems. The Compiler used for the proof-of-concept version of ECAT uses the Articulated Communicator Engine (ACE) at its core. The implementation of this Compiler will be discussed in chapter 4.2. The motivation to choose for ACE will be described in chapter 3.5.2.
Since ACE, uses the Multimodal Utterance Representation Markup Language (MURML) as input; the input language for the Compiler will also be MURML. It is however important to note that efforts to add native support for the Behavior Markup Language are currently underway at the University of Bielefeld. This means that the Compiler will also support BML as input in the future. The fact that BML is going to be supported in the future was an important point in the decision to use ACE. Otherwise ECAT wouldn't really be moving towards a common standard (SAIBA+BML) but rather moving away from it. Moving towards (and supporting) a common standard is key to the success of ECAT, as has been illustrated in section 3.1.

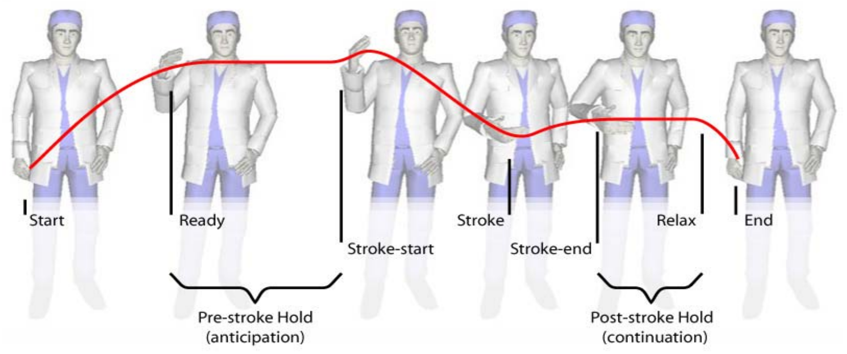


Figure 20 Gesture phases[7]

Because both MURML and BML will be supported in the future, both will be discussed in the following subsections. Both languages use gesture phrases as a mechanism of timing. Kendon

[6] We are talking about 'The Compiler' here, although there can be more than one. Even if more Compilers are being developed they should conform to the same interface standards.
[7] Image from the specification of the Behavior Markup Language, http://wiki.mindmakers.org/projects:BML:main

[32] refers to the part where the gesture shows ‘peaking in effort’ as the ‘stroke’. This has since then been widely adopted. More specifically, McNeill [49] speaks of a gesture phrases consisting of the following phases: preparation, pre-stroke hold, stroke, post-stroke hold and retraction. Except for the stroke phase, all the phases are optional. Figure 20 shows an example gesture, with the different phases in it.

3.3.1 MURML

The Multimodal Utterance Representation Markup Language (MURML) [35,42] is the language natively supported by the Articulated Communicator Engine (ACE), in this sense it is used in ECAs like NUMACK [65] and Max [37]. The language allows speech and gesture to be specified along with cross-modal timing information.
According to Kranstedt, Kopp and Waschmuth [42] the communicative intent of a gesture can be sufficiently described in terms of the spatio-temporal features of its stroke. In the markup language they describe (MURML) this is exactly how a gesture can be described, namely using features of its stroke. Therefore MURML does not specify gestural behavior during preparation, hold and retraction phases.
Based on previous research, Kopp [35] states the following (which he refers to as segmentation hypothesis): “Continuous speech and gesture are co-produced in successive segments each expressing a single idea unit”. He also notes that speech is organized over intonation phrases, phrases which are separated by significant pauses. Chunks are defined as pairs of intonation phrases and a coexpressive gesture phrase. Inter- or intrachunk timing of paraverbal and nonverbal behaviors can be specified in terms of affiliate onset and end times. The affiliate of a behavior is the word or subphrase which is affiliated with the specific behavior.

```
<definition>
 <utterance>
        <specification>
        And now take <time id=“t1”/> this <time id=“t2”/> bar <time id=“t3”' chunkborder=“true”/>
        and make it <time id=“t4”/> this big. <time id=“t5”/>
        </specification>
        <focus onset=”t1” end=”t2” accent=”H*”/>
        <behaviorspec id=”gesture_1”>
                <gesture id=”pointing_to”>
                        <affiliate onset=”t1” end=“t3”/>
                        <param name=”refloc” value=”$Loc-Bar_1”/>
                </gesture>
        </behaviorspec>
        <behaviorspec id=”gesture_2”>
                <gesture>
                        <affiliate onset=”t4” end=“t5”/>
                        <constraints>
                         <symmetrical dominant=”right_arm” symmetry=”SymMS”>
                          <parallel>
                           <static slot="HandShape" value="BSflat (FBround all o) (ThCpart o)"/>
                           <static slot="PalmOrientation" value="DirL"/>
                           <static slot="ExtFingerOrientation" value="DirA"/>
                           <dynamic slot="HandLocation">
                                <dynamicElement type="linear">
                                 <value type="start" name="LocShoulder LocCenterRight LocNorm"/>
                                 <value type="direction" name="DirR"/>
                                 <value type="distance" name="125.0"/>
                                </dynamicElement>
                           </dynamic>
                          </parallel>
                         </symmetrical>
                        </constraints>
                </gesture>
        </behaviorspec>
 </utterance>
</definition>
```

Figure 21 Example MURML specification [35]

The Figure above shows an example MURML specification which shows:

a) Chunking of speech (chunkborder="true")

b) Gesture timing specified in terms of affiliate onset time.

c) Stroke of the gesture specified in terms of its spatio-temporal features.

Point a and b have been explained above, however point c needs some more explanation. Gestures in MURML can be defined (like in Figure 21), in terms of HandShape, PalmOrientation, Extended FingerOrientation and HandLocation which stem from HamNoSys [57], which is a notation system for German sign language.

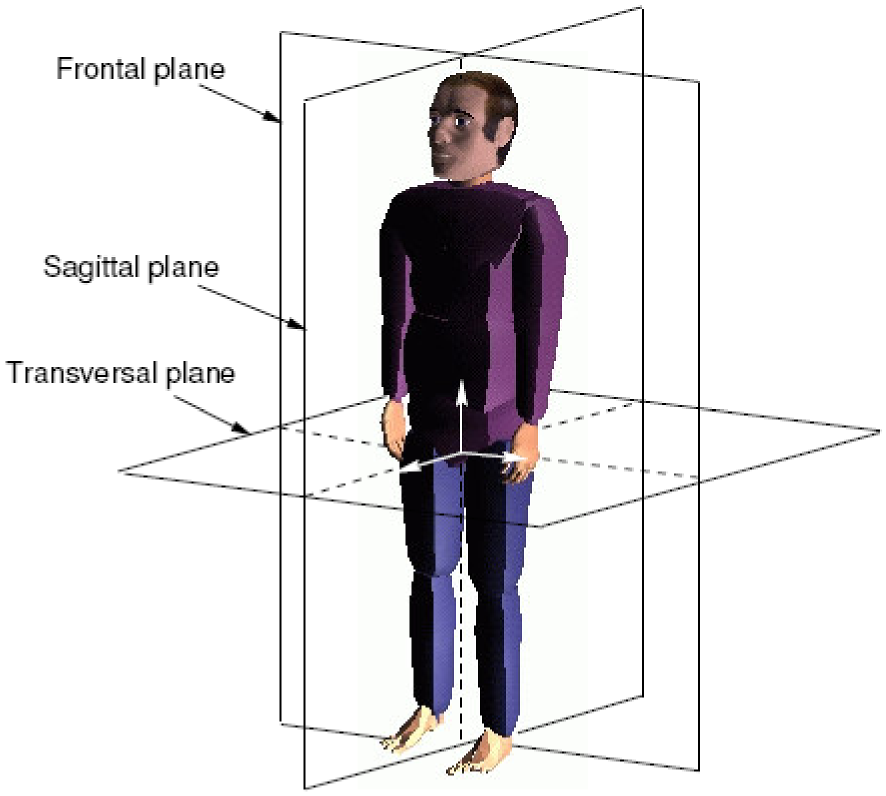


Figure 22 The three body planes [35]

Each of the four components used for gesture definitions can be described numerically or symbolically (like in Table 4). The location of the hand can be described using three symbols referring to its position in each of the three body planes, which are shown in Figure 22. The three body planes: Frontal, Sagittal and Transversal shown in the figure correspond to planes commonly used in anatomy.

Symbol	**Value**
HandLocation = *LocFrontal LocTransversal LocSagittal*	
LocFrontal	LocAboveHead, LocHead, LocForehead, LocEyes, LocNose, LocMouth, LocChin, LocLowerChin, LocNeck, LocShoulder, LocUpperChest, LocLowerChest, LocStomach, LocBelowStomach, LocHip, LocBelowHip
LocTransversal	LocCCenter, LocBackof, LocCCenterRight, ``LocCCenterLeft, LocCenterRight. ``LocCenterLeft, LocPeripheryRight, ``LocPeripheryLeft, LocExtremePeripheryRight,

	LocExtremePeripheryLeft, LocLeft, LocRight
LocSagittal	LocContact, LocNear, LocNorm, LocFar, LocStreched
Handshape = *Basicsymbol [(Thumbsymbol)] Fingersymbol)*	
Basicsymbol	BSfist, BSflat, BSffinger
Thumbsymbol	ThExt, ThAcr, ThCpart [*Degree*]
Fingersymbol	FBstr *Finger*, FBangle *Finger* [*Degree*], FBround *Finger* [*Degree*], FBroll *Finger*, FBbent *Finger*, FBstiff *Finger*
Finger	ll, p, m, r, l
Degree	c, no, o, wo
ExtendedFingerOrientation = *Direction*	
PalmOrientation = *Direction* | *RelativeRotation*	
Direction	DirL, DirLU, DirLD, DirR, DirRU, DirRD, DirU, DirD, DirA, DirAL, DirAR, DirAU, DirAD, DirT, DirTL, DirTR DirTU, DirTD
RelativeRotation	PalmL, PalmLU, PalmLD, PalmR, PalmRU, PalmRD, PalmU, PalmD, PalmA, PalmAL, PalmAR, PalmT, PalmTL, PalmTR, PalmAU, PalmAD, PalmTU, PalmTD

Table 4 Symbolic values which can be used for gesture descriptions in MURML [35]

In addition to the symbols used for describing the gestures Figure 21 also shows two types of 'movement constraints' (according to Kopp [35]), namely dynamic and static. A static constraint defines a posture which is to be held for a certain period, a dynamic constraint defines a movement. Figure 21 shows such a movement, which in this case describes a linear movement directed to the right and covering a distance of 125. In addition to linear movement, Kopp also defines curve shaped movement which has additional parameters describing the movement (see [35]). Additional tags like parallel and symmetrical can further describe the nature of the gesture (see [35] for a complete set of tags and their possible values). In this case gesture_2 describes a gesture which is symmetrical with respect to the sagittal plane and where the left hand mirrors (the dominant) right hand. The dynamic element species that the right hand should move 125 to the right, in this case the left hand would move 125 to the left, looking at the affiliate speech: "this big", the gesture_2 describes the size which the bar is to be made ("…take this bar and make it this big").
Specifying gestures using their overt form (like gesture_2) is a distinct feature of MURML, however gestures can also be specified using low-level keyframes or refer to a communicative function or a gesture from a gesticon. Being able to specify gestures using their overt form enables users to easily add new gestures. Normally adding new gestures would involve first creating an animation for the model using 3d software and second export this animation so it can be used from the gesticon.
The first gesture mentioned in Figure 21, uses an identifier ("pointing_to") which can be used by the system to look up the gesture in the gesticon. Gestures of this sort can be parameterized to maintain a certain sense of flexibility. Gesture_1 for instance uses a parameter called ref_loc which makes the gesture point to the desired location.

A discussion how inter-/intrachunk and cross-modal timing, which is specified in a MURML specification, is dealt with will be given in section 4.2.1, when ACE is discussed.

3.3.2 BML

The inchoate Behavior Markup Language (BML) which can be seen as successor of (among others) MURML, is a growing and still evolving language. BML was formed as one of the interface languages of the Social Performance framework [46] which led to the more recent SAIBA framework [38]. Since the beginning of the SAIBA framework more and more research projects has began using BML. One of these projects involves the Articulated Communicator Engine (ACE) which, as stated before, is the core component of the ECAT-Compiler (see 4.2). On going work at the University of Bielefeld is focused on making the ACE system BML compliant. The current version which is used for ECAT is however not supporting BML, for now this means the ECAT-Compiler expects its input to marked up in MURML. In the future this could also be BML. For now developers of BML-compliant systems have to develop their own interpreter which can convert their (subset of) BML behaviors into corresponding MURML utterances.
Since the future support of BML in ACE (and therefore ECAT) was one of the factors driving the decision to use ACE as core component of the ECAT-Compiler, a brief discussion of the Behavior Markup Language will be given here.

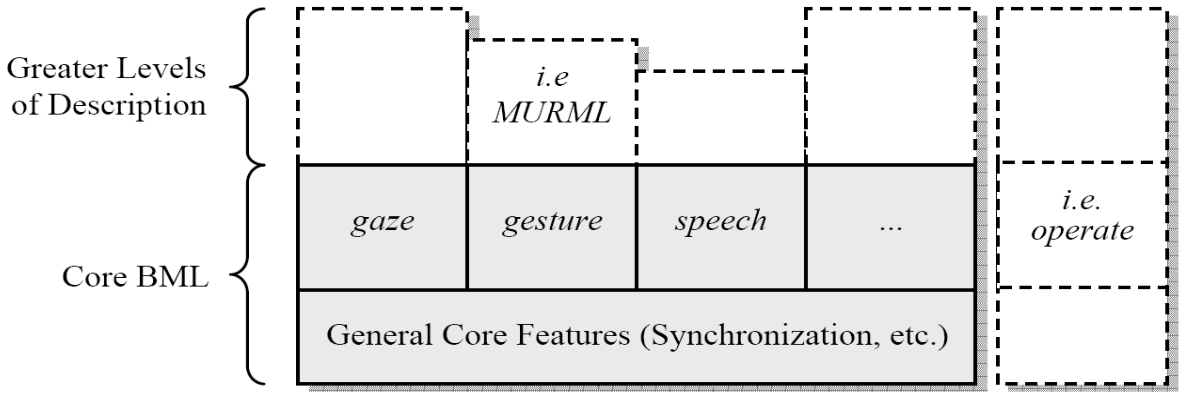

Greater Levels of Description
Core BML
i.e MURML
gaze
gesture
speech
...
i.e. operate
General Core Features (Synchronization, etc.)

Figure 23 Core BML which's elements can be extended or new elements can be added. [69]

Recent developments [69] have added levels of descriptions to the originally proposed [38] Behavior Markup Language. BML now has a core level of description (level 0) and supports extension with higher levels of description (see Figure 23). In the SAIBA framework, BML interfaces between behavior planning and behavior realization (see Figure 1). Core BML defines a number of behavior elements[8]: speech, gesture, gaze, head, body, torso, face, legs, lips and wait. The core level allows high-level descriptions for each of these elements. Additional descriptions levels can add more fine-grained behavior specifications for these elements. When additional elements might be needed in a behavior specification, these can be added by using a tag specifying a namespace corresponding to the name of the system which is expanding the core level. (For instance the CADIA system which defines a behavior called operate, see Figure 23 and Figure 24).

```
<bml>
        <speech id=”s1” type=”application/ssml+xml”>
                <text>This is an <mark name=”wb3”> example</text>
        </speech>
        <head id=”h1” type=”NOD” stroke=”s1:start”/>
        <gesture id=”g1” stroke=”s1:wb3” relax=”s1:end” type=”BEAT”>
                <description level=”1” type=”MURML”>...
```

[8] A more in depth and up to date specification can be found at http://wiki.mindmakers.org/projects:BML:main

```
                    </description>
            </gesture>
            <gaze id="z1" target="PERSON1" stroke="g1:stroke-0.1"/>
            <body id="p1" posture="RELAXED" start="after(s1:end)"/>
            <cadia:operate target="SWITCH1" stroke="p1:ready"/>
    </bml>
```

Figure 24 Example BML description [69]

The BML description from Figure 24 shows a level 1 description of type MURML, this means that the core BML can be expanded by MURML-like gestures (see previous section about MURML), which would allow for gestures to be specified using their morphological features. This also means that ACE, when supporting core BML in the future, can still use its unique features which rely on descriptions specified in MURML. Vilhjamsson et al. [49] also mention the fact that BML compliant systems should be able to realize behaviors defined in core BML, when only a subset of behaviors is supported this should be reported back to the behavior planning system with appropriate feedback messages (see [8]).
Unlike in MURML, where behaviors can be timed according to speech, BML supports seven points which can be used for synchronization (see Figure 20), this is in addition to the sort of timing functionality supported in MURML, which is also shown in Figure 24, with stroke = "s1:wb3". In addition to the synchronization point an offset can be added (like: stroke-0.1) or the predicates before() and after() can be used for a behavior that is supposed to start before or after a certain synchronization point, not specifying what exact point in time. As stated before BML is still evolving, more recent development is discussed by Vilhjamsson et al. [69] and an up to date specification and discussion can be found at [8].

3.4 Low-level behaviors in a standard format

3.4.1 The requirements

One important target of ECAT is keeping both ECAT-Interpreters and ECAT-Translators as simple and lean as possible. Having an overly complex interface language, would therefore result in more complexity in ECAT-Translator design and implementation. With this in mind the requirements for the output language of the Compiler were set to be:

1) Support for specification of joint-rotations
2) Support for specification of audio playback
3) Support for specification of facial deformations
4) Format which is easy to parse and read, therefore easy to debug

The fourth requirement is important for the initial proof-of-concept prototype version of ECAT. In future version this may not be as important. For initial testing and debugging it's more convenient to have an easy to read and easy to debug format.

3.4.2 Choosing a format

A growing number of systems use MPEG-4 [53] for facial and body animation. A layman may associate MPEG-4 with a video codec like DivX. However MPEG-4 is much more than only a video codec. Part of the MPEG-4 standard is the FBA object (Face & Body Animation Object). The FBA object defines facial animation parameters (FAPs), body animation parameters (BAPs), facial definition parameters (FDPs) and body definition parameters (BDPs). FAPs correspond to feature points on the mesh of a face defined by the MPEG-4 standard. The FAP-value's are specified in terms of FAP units (FAPUs). FAPUs are computed from distances between key features of the face, such as ‘mouth width’. This way faces of different sizes can be animated properly. The FAP-value determines the translation of a certain facial feature point. BAPs correspond to a degree of freedom of a joint. The BAP-value determines the amount of rotation for the specific joint. If a default body setup is used BDPs and FDPs are not needed, but these can be used to configure the behavior for different body models. While there are a lot of systems using MPEG-4, such as Gesture Engine (see section 2.3), there are also a lot of systems not using MPEG-4 for facial and body animation, such as ACE+Open Inventor, SmartBody+Unreal, the MPML3D framework (also see section 2.3). Within MPEG-4, FAP and BAP-data is encoded in an MPEG-4 stream, which needs to be decoded by an MPEG-4 player for animation. Using MPEG-4 within ECAT would mean including and MPEG-4 encoder in the Compiler and MPEG-4 decoders in every Translator. The need for an MPEG-4 encoder in every Translator would add extra complexity. However implementations of Translators should be kept as easy as possible, because this will facilitate the development of new Translators. Therefore it was decided not to use MPEG-4 FAPs and BAPs in the intial proof-of-concept prototype.

With the requirements from 3.4.1 and the discussion above in mind a number of existing XML-based animation system markup formats were reviewed.
None of the reviewed languages fulfilled all requirements. Languages weren’t reviewed any further as soon as it was evident that the particular language wasn’t going to pass all requirements. Therefore the discussion in this section will list, why the reviewed languages weren’t selected.
The Character Markup Language (CML) [11], the Avatar Markup Language (AML) [43] and the Virtual Human Markup Language (VHML) [45] all use FAPs and BAPs from the MPEG-4 standard for representing low-level animation primitives. As has been stated above, it was chosen not to use MPEG-4 for the initial implementation. For this reason these three languages were not suited as output language for the Compiler.
The XSTEP language [29] didn’t fulfill requirement two and three, it does not yet support facial animation and audio playback directives.
In fact the language closest to ‘perfect’ might be MURML [35,42], which is also used as input language for the Compiler. However MURML does not provide means for audio playback directives.
Since none of the reviewed languages fully fulfilled the requirements it was chosen to create a custom XML format as output language for the Compiler. To keep the initial proof-of-concept Translator(s) as simple as possible it was decided to create a new format instead of extending one of the other XML languages. The format will henceforth be referred to as the Compiler Output Language (COL).

```
<joint name=”l_elbow”        m00=”1.0” m01=”0.0” m02=”0.0”
                             m10=”0.0” m11=”1.0” m12=”0.0”
                             m20=”0.0” m01=”0.0” m22=”1.0” />
```

Figure 25 Joint rotation specified in COL

The current version[9] of COL supports specification of joint transformations using a 3x3 rotation matrix (Figure 25) and audio playback directives (Figure 26), which Translators are expected to ‘realize’ when received. Therefore, no timing[10] will be specified. Joint

[9] NOTE: the current ECAT-Compiler/ACE ←→ ECAT-Translator implementation is not fully functional; it does not yet include support for facial animation.
[10] It remains to be seen if this will lead to a correctly synchronized and working system as a whole. See the discussion in chapters 6-8.

transformations are specified in a 3x3 matrix which is common in the world of computer graphics.

```
<audio name=”utterance0_chunk0.aiff”/>
```

Figure 26 Audio playback specified in COL

Figure 25 shows a 3x3 matrix specifying a rotation. A matrix, however, is not the only way to represent a rotation. Two other possible representations are: quaternion and axis angle. The first is based on the latter, where an angle and a rotation axis are specified.
It is also possible to specify joint rotations as rotations around x, y and z (like in Figure 5). In this way a rotation cannot be uniquely defined, since the order of rotations is also important. Rotations specified in this manner are often called Cardan angles. Three rotation angles and the order of XYZ are specified. Euler angles work in a similar way, also with three rotation angles and a set of axes to rotate around, for instance: ZXZ. The difference between Cardan and Euler angles lies in the fact that Euler angles rotate around the same axis twice. Both Euler and Cardan angles may lead to a gimbal lock when certain rotations are interpolated. A gimbal lock occurs when the one coordinate axis is rotated on top of on of the other axes, which effectively means one degree of freedom is lost. By using rotation matrices, quaternions or axis angles this is avoided.
In most systems rotations are stored as either quaternions or matrices. It was chosen to use a matrix notation in COL instead of quaternions to ease the debugging process (see section 3.4.1).

$$R_x(\theta)= \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & \sin\theta \\ 0 & -\sin\theta & \cos\theta \end{bmatrix}$$

$$R_y(\theta)= \begin{bmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{bmatrix}$$

$$R_z(\theta)= \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$M = [\, x\ y\ z \,]\, [R]$$

Figure 27 Rotation format of the Compiler Output Language (COL)

Figure 27 shows three example rotation matrices of rotations around the three principal axes: X, Y and Z. It also shows that a resulting matrix is obtained by pre-multiplying a row vector with the rotation matrix. This is very important, since some other systems use post-multiplication with a column vector. In such a case the rotation matrices need to be transposed to get to the same results. In computer graphics complete transformations (rotation, scale and translation) are often represented with a 4x4 matrix. When dealing with a row vector (like in COL) the translation would be placed in the first three elements of the fourth row. Some people might be more familiar with having the translation in the first three elements of the fourth column. This is a subtle but important difference. It depends on the fact that column vectors + post-multiplication or row vectors + pre-multiplication (like in COL) are used.

Figure 28 shows the DTD of the current COL version. As has been stated above it has support for audio playback directives and joint rotations. An example will be given (see Figure 38) in the next chapter when the proof of concept prototype will be discussed.

```
<!DOCTYPE ecat [
        <!ELEMENT ecat (audio?,joints?)>
        <!ELEMENT audio EMPTY>
        <!ATTLIST audio name #REQUIRED>
        <!ELEMENT joints (joint+)>
        <!ELEMENT joint EMPTY>
        <!ATTLIST joint name #REQUIRED>
        <!ATTLIST joint m00 “1.0”>
        <!ATTLIST joint m01 “0.0”>
        <!ATTLIST joint m02 “0.0”>
        <!ATTLIST joint m03 “0.0”>
        <!ATTLIST joint m10 “0.0”>
        <!ATTLIST joint m11 “1.0”>
        <!ATTLIST joint m12 “0.0”>
        <!ATTLIST joint m20 “0.0”>
        <!ATTLIST joint m21 “0.0”>
        <!ATTLIST joint m22 “1.0”>
]>
```

Figure 28 DTD for the Compiler Output Language (COL)

3.5 The three stages of ECAT

Interpretation involves converting a high-level behavior specifications specified in a custom format into a high-level behavior specifications in MURML (see section 3.3).
Compilation involves converting high-level behavior specifications specified in a standard format into low-level behavior specifications specified in COL (see section 3.4)
Translation involves converting low-level behavior specifications specified in COL into audiovisual rendering of an Embodied Conversational Agent (ECA).
The next three subsections will discuss the purpose and functionality of the three different components of the ECAT pipeline. These sections can be seen as general design guidelines for specific implementations, such as the ones discussed in chapter 4 and 5.

3.5.1 Interpretation

Typically included in an Interpreter	Typically not included in an Interpreter
Socket Client (to connect to the Compiler)	Internal representation of joint rotations
UI for user input (i.e.: a WOZ-gui or a microphone and/or camera)	
(Internal) representation of (high-level) behaviors	

Table 5 Interpreter functionality

An *Interpreter*, while not strictly a part of the behavior realization process, is a vital part of the ECAT pipeline. Its primary purpose is to convert high-level behavior specifications into a standard format (see section 3.3), readable by the *Compiler*.
Under the current ECAT architecture, the *Interpreter* component is attached to (and integrated with) the very end of a behavior generation system. As a result, an *Interpreter* is considered to be *implementation dependent* (or application specific), in that it relies on the software libraries available to the behavior generation system.
The job of a regular interpreter is to change what someone else is saying in another language. Every source language needs another interpreter. If the target language is for instance English we would need four different interpreters to change: “*Hallo, myn namme is John*”, “*Hallo,*

mijn naam is John", "*Bonjour, je m'appelle John*", "*Hallo, meine Name ist John*" into: "*Hello, my name is John*".
Within ECAT an *Interpreter* serves as an interpreter between the different languages 'spoken' by the numerous behavior generation applications and the ECAT-Compiler which only 'speaks' one specific language[11].
As has been discussed in section 3.3.1 the primary language 'spoken' by the Compiler is MURML. While the number of official natural languages is a finite number, the number of interpretable languages for ECAT is infinite. It's up to the developers of an *Interpreter* to map every possible multimodal utterance (a behavior) of their system onto a valid MURML utterance. For example the specifications given in Figure 4 could be valid 'input' to an *Interpreter*. We are talking about 'input' here since the input language of an *Interpreter* is not necessarily input from the outside world, but its most often created inside the *Interpreter*. The MURML utterances used for NUMACK for instance are a result of a complex Natural Language Understanding (NLU) and route-planning system, as will be described in section 4.1 In case of an *Interpreter* for a WOZ-panel, see section 5.1, input can also come from a 'buttonListener'. A developer of such an *Interpreter* just needs to make sure that for every button a valid MURML utterance is created.
As can be seen in Figure 29 not all languages have the same features. The shown utterances in BML and MURML (see next page) for instance show the possibility to time the gesture with relation to the stroke of the gesture (see Figure 20). The other input examples shown in Figure 29 do not include such fine grained timing information. This does not necessarily mean that this input is un-interpretable; it just means that the developers need to decide the best way to map their input utterance to a valid MURML utterance. *Interpretation* is sometimes subjective, developers need to map their 'input' as they see fit. This means, just like in the regular sense of interpretation, that two people which interpret the same 'input' may not come up with exactly the same interpretation.

Example input:
MPML3D:
```
	<Sequential>
		<Action class="gesture">
			<Property name="type">Wave</Property>
		</Action>
		<Parallel>
			<Action class="speak">
				<Property name="text">Hello, my name is John.</Property>
			</Action>
		</Parallel>
	</Sequential>
```
BML:
```
	<bml>
		<speech id="s1" type="application/ssml+xml">
			<text>Hello, <mark name="wb3">my name is John.</text>
		</speech>
		<gesture id="g1" stroke="s1:wb3" relax="s1:end" type="SIGNAL"/>
	</bml>
```
Custom format:
```
	-Say("Hello,  my name is John") & Gesture("wave")
```
GUI button:

--

[11] Actually two languages, namely BML and MURML, as described in more detail in section 3.3.

Example output:
MURML:
```
        <utterance>
                <specification>
                Hello <time id="t1"/>my name is John <time id="t2"/>
                </specification>
                <behaviorspec id="gesture_1">
                        <gesture>
                                <affiliate onset="t1" end="t2" focus="this"/>
                                <function name="signal_greeting">
                                </function>
                        </gesture>
                </behaviorspec>
        </utterance>
```

Figure 29 Four types of possible input behaviors mapped with example output.

The task of an *Interpreter* can be as simple as to map a number of specifications in one language to specifications in MURML. Oftentimes this will suffice, since the application at hand only offers a limited number of specifications in the source language (For instance: the number of buttons on a WOZ is limited).
An *Interpreter* can also be more general, being able to convert every possible specification of one language into a valid specification in MURML.
Once a valid MURML utterance is formed it can be send to the Compiler using the Socket Client which also needs to be integrated with the *Interpreter* (see Table 5).
Although *Interpreters* are application-specific, portions can be reused in the development of a new *Interpreter* using either the same programming language or the same behavior description format. As such, it is anticipated that the availability of sample *Interpreters* will greatly simplify the development of new *Interpreters* in the future.
Also, by integrating the *Interpreter* with a behavior generation system, ECA developers are able to create and modify *Interpreters* in a programming language they are familiar with, further simplifying the development process.

3.5.2 Compilation

Typically included in a Compiler	Typically not included in a Compiler
XML Parser (to parse MURML/BML)	Audio playback functionality
Socket Server (for Translators and Interpreters)	Functionality to render/animate a body model
Gesticon	
Speech synthesis (Text-to-speech)	
Gesture synthesis (resulting in: 1) joint rotations to be used by an articulated figure and 2) facial deformations)	
Internal representation of joint rotations	
Inverse Kinematics	
Animation blending	

Table 6 Compiler functionality

ECAT's core component, the *Compiler*, expects high-level behavior descriptions, sent by an Interpreter. Its output involves low-level joint rotations, audio playback directives and facial deformations which will be input for a *Translator*.
The task of the *Compiler* is to convert high-level behavior descriptions into low-level joint rotations, facial deformations and audio playback directives for speech. As a part of this

process, the *Compiler* is tasked with synthesis of synchronized[12] speech and gesture, and maintaining a record of an agent's current state at all times. It's also responsible for a smooth transition (blending) between subsequent gestures. In theory, with specific additions and alterations each of the systems mentioned in section 2.3 could be used to fulfill the task of *compilation*. Because a number of projects within the Articulab employ the Articulated Communicator Engine (ACE) [41] for behavior processing of this sort, the decision was made to reuse this component in the *Compiler*.
ACE is a state-of-the-art system able to produce synchronized verbal, paraverbal and nonverbal behaviors for an ECA, given a behavioral specification as input. Developed at the University of Bielefeld, ACE is currently used in multiple virtual human systems, including Max [37] and NUMACK [65]. Because ACE is able to convert high-level behaviors into low-level output directives, it neatly and effectively performs the task of *compilation*. To let ACE function within ECAT, certain alterations and additions had to be made. These changes were necessary because ACE wasn't designed for the ECAT architecture, its original design can be utilized by implementing a behavior realization system which inherits from a certain ACE class (written in C++). The most important reason for this to be a problem is that it needs to be possible to implement *Translators* (and *Interpreters*) in any language. Therefore it was chosen to implement a wrapper around the basic ACE functionality which makes it possible to use socket interface for communication between, *Interpreters*, *Compiler* and *Translators*, as mentioned before in beginning of section 3.2. The details of the *Compiler* implementation including the integration of ACE, will be discussed in chapter 4.2.

3.5.3 Translation

Typically included in a Translator	**Typically not included in a Translator**
XML Parser (to parse COL)	Gesticon
Socket Client (to connect to the Compiler)	Speech synthesis (Text-to-speech)
Audio playback functionality	Gesture synthesis
Functionality to control joints of an articulated figure	Inverse Kinematics
Functionality to render/animate a body model	
Functionality to apply facial deformations	

Table 7 Translator functionality

The last component in the ECAT pipeline is the *Translator*. Like the *Interpreter*, the *Translator* is designed to be a flexible, easily replaceable component. Notably, *translating* is fundamentally different from *compiling* and *interpreting* as it does not convert its input language (COL) into another language. The task of a *Translator* is to parse the incoming COL data and render the audiovisual representation of the ECA. To do this, a *Translator* component must be integrated with an underlying graphics engine[13]. Accordingly, ECAT requires each different graphics engine to have a different *Translator*.
While it's also possible to use a 2D representation of an ECA for rendering, just like the current version of Sam / Alex (see section 1.2.1), ECAT only supports graphics engines with support for articulated body models. Body models like these contain joints, which can be used for animation. The majority of the systems use graphics engines capable of applying joint rotations to such a body model (see Figure 6).

[12] Synchrony, which is specified in the input needs to be maintained in the output, for more details see chapter 4 and 5.
[13] A graphics engine such as Ogre3D, Panda3D or AgentFX may be used for behavior realization. Most of these systems do not have the ability to render multimodal output by receiving an XML stream. Therefore, integration with a parser is needed.

Figure 25 shows a COL-snapshot describing a rotation of the left elbow, using a representation of a transformation matrix. Figure 6 shows which function calls could be used in Panda3D [6] to rotate the elbow and how something similar could be solved in Ogre3D [7]. In Panda3D the needed arguments are heading, pitch and roll, while Ogre3D needs 4 arguments describing a quaternion. The heading, pitch, roll and a corresponding quaternion can all be obtained from a transformation matrix.
Both engines (Panda3D and Ogre3D) have specific function calls and also depend on different libraries. Therefore two different *Translators* are needed. For this reason a *Translator* can be seen, just like *Interpreters*, as being implementation dependent (or engine-specific). A lot of engines can only be used for graphics (such as Ogre3D), this means the functionality for audio playback, socket connections and xml parsing needs to be added to get a fully functional *Translator*. The implementation of this functionality can then be reused in a different *Translator* which is programmed in the same programming language. For instance: both Open Inventor [1] and Ogre3D lack the aforementioned functionality, but they both have a C++ API. Thus, regarding this functionality, the same implementation can be used by both a *Translator* for Ogre3D and Open Inventor. When more *Translators* come available the ability to reuse large parts of a *Translator* will greatly facilitate the development of future *Translators*.
Also, unlike *Interpretation*, *Translation* represents a break with the SAIBA approach, regarding the design of behavior realization systems. Where the SAIBA framework frames the process of behavior realization in terms of converting high-level behavior specifications into rendered output, a *Translator* allows ECAT to further divide this process into two distinct stages – the conversion of high-level behaviors to low-level directives (*compilation*) and the realization of low-level directives as rendered output (*translation*) (as has been previously discussed in section 1.3). By modularizing the behavior realization process in this manner, swapping rendering engines becomes a relatively simple matter of remapping the compiled low-level directives from one engine's API to the next. Thus, the *Translator* is critical for making rapid integration of behavior generation and rendering systems a reality.
As input, a *Translator* accepts a custom output format, sent by the Compiler, containing low-level animation and audio specifications marked up in COL (see section 3.4.). Notably, the *Translator* does not require a gesture database (*gesticon*), inverse kinematics (IK) or any other features related to the process of gesture production or specification, as these items are already present in the *Compiler* (see Table 6 and Table 7).
A *Translator* does require the animation of the body model of the ECA. It could be possible that this body model contains more or less detail as the body model which is used internally by the Compiler. The H-Anim [54] standard defines different level of articulation (LOA) which corresponds to the number of joints defined in a humanoid figure. The term level of detail (LOD) is sometimes used for the same concept. Using this terminology, it could be possible to have a mismatch between the LOA of the body model used in the *Compiler* and the body model used in a *Translator*. Most research relating to LOD has been carried out from within the field of Crowd Simulation [48,52]. None of these research projects focus on what Ruttkay and Noot [59] refer to as 'body morphology limitations'. An example of such a limitation would be an arm without fingers, or an even bigger limitation an arm without a wrist or elbow. Currently specific ECAT-Translators can choose their own solution for a LOA mismatch between Compiler and Translator. A straight forward solution would be to only animate the joints which can be mapped and ignore the transformation of the other joints. In some situations this can lead to very weird animations. For example a wave animation where most effort is coming from elbow movement will not look like a wave at all when the arm contains no elbow-joint. A solution for this would be to have a separate wave animation for models missing elbow-joints, where most effort is coming from the shoulder. However, the

current *Compiler* implementation, discussed in the next chapter, contains no functionality to handle such issues. Therefore a plausible solution has to be found by a *Translator*.

4 A proof of concept prototype

The previous chapter has introduced a new three step modularization approach. This chapter will follow up on this, describing implemented modules using this approach. Parts of the aforementioned ECA system: NUMACK have been partially reimplemented to serve as modules for ECAT.

4.1 NUMACK's Brain Interpreter

The larger part of this section will discuss a part of an already existing ECA, called NUMACK.

4.1.1 The Input

The input to NUMACK's Brain is given by a human by speech and head gestures. The scenario which the system is used for is as follows: a user can interact with an ECA called NUMACK and he/she can ask him directions around the Northwestern University campus. NUMACK will ask the user for a starting location and a destination. Once the user has told these to the system, the system, plans the route (see below for more details) and explains it step by step. After each (or a couple of) step(s) NUMACK will wait for acknowledgment by the user, which can be given by saying something like: "okay" or just by a simple head nod.

NUMACK: Hello there <GESTURE: WAVE>
NUMACK: My name is NUMACK
NUMACK: I can give you directions around Northwestern campus
NUMACK: Where would you like to go?
USER: the Ford Building
NUMACK: Where are you starting from?
USER: from the Allen Center
NUMACK: ok you would like to go from the Allen Center to the Ford Building
NUMACK: leave the Allen Center by west exit
NUMACK: after that <GESTURE:GO_STRAIGHT>go straight across a parking lot
NUMACK: then there will be a building with a <GESTURE: V_CURVE_RIGHT> on your right
…

Figure 30 Start of sample interaction with NUMACK

4.1.2 Global system description

The NUMACK (Northwestern University Multimodal Autonomous Conversational Kiosk) [65] system builds upon earlier efforts which have lead to the MACK (Media lab Autonomous Conversational Kiosk) [15] agent. NUMACK is an ECA which is capable of giving directions around the Northwestern University campus. Users can interact with the system using multiple modalities, using both speech and head gestures. The agent on its turn uses synthesized speech, gestures and facial animation to convey information to the user.
In a typical interaction NUMACK starts with a brief introduction followed by the question: "where do you like to go?" The user can reply, using verbal input, with a certain destination on campus. After having asked a starting point, NUMACK will generate a route, including descriptions which will be described step by step to the user. After NUMACK has described one or more steps of the route he will wait for feedback from the user to proceed. The user

can acknowledge by either a head nod or by saying something like: “okay”. It is also possible to ask for more elaboration on the previous step.
During the route description NUMACK will mention landmarks along the route and he will use gestures to depict either the form and/or location of these landmarks. Landmarks range from a lagoon, a parking lot, to buildings on campus. The steps of the route can be described from either a route perspective (perspective of someone walking the route) or survey perspective (like a birds-eye view) [63].

4.1.3 The architecture

The architecture of NUMACK is given in Figure 31. A brief description of the modules will be given here, for a more detailed discussion of the architecture see [63].

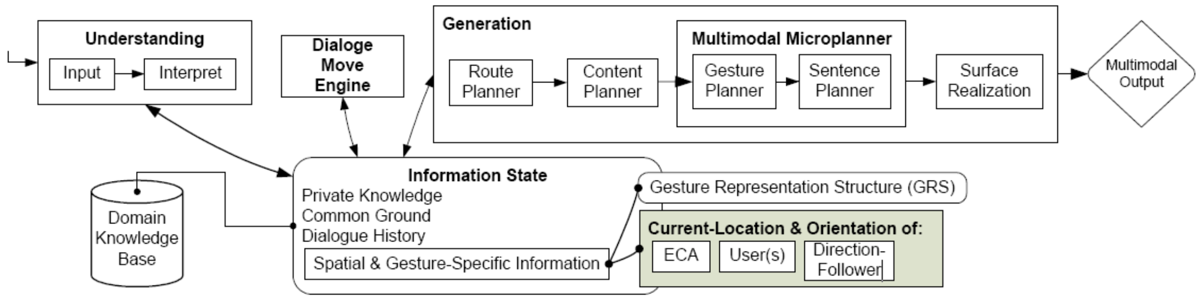


Figure 31 NUMACK Architecture [63]

The understanding module is where the Natural Language Understanding (NLU) of the user’s input takes place.
The information state module keeps track of the dialogue history and both private knowledge of the system and shared knowledge of user and system. This module is also used to keep track of what point of the route is reached by description. During route descriptions a data structure called: Gesture Representation Structure (GRS) is used to link the location of NUMACK’s hands and the represented landmarks.
The generation module resembles a pipeline architecture which is commonly used in Natural Language Generation (NLG), Tepper et. al. [65] refer to their pipeline as NLGG (for NLG with Gestures).
For route and content planning the system uses an internal representation of a map of the campus, including waypoints. These waypoints are linked to landmarks and building entrances. This representation can be used for route planning and is also used by the content planner to determine which landmarks to refer to during direction giving.
The architecture of the micro-planning module is shown in Figure 32. The gesture and sentence planner need additional information from the information state and knowledge base which Kopp et al. refer to as Image Description Features (IDFs) [14,40].

IDFs can be seen as the spatial and visual features of both morphology of gestures and the entities they can refer to. For every landmark salient (form) features have been determined, for instance the tallness of a church, the flatness of a lagoon and the curvature of a dome. Morphology of gestures on the other hand can be defined in terms of sign language. Using the salient features of landmarks (IDFs) together with the Gesture Form Feature Entries (see Figure 32) gestures corresponding to landmarks, referred to in the content plan, can be created. These gestures are not canned gestures from a predefined gesticon but are created on the fly using form features described in terms of sign language. After these dynamic gestures have been created they are added to the multimodal lexicon, which is used by SPUD.

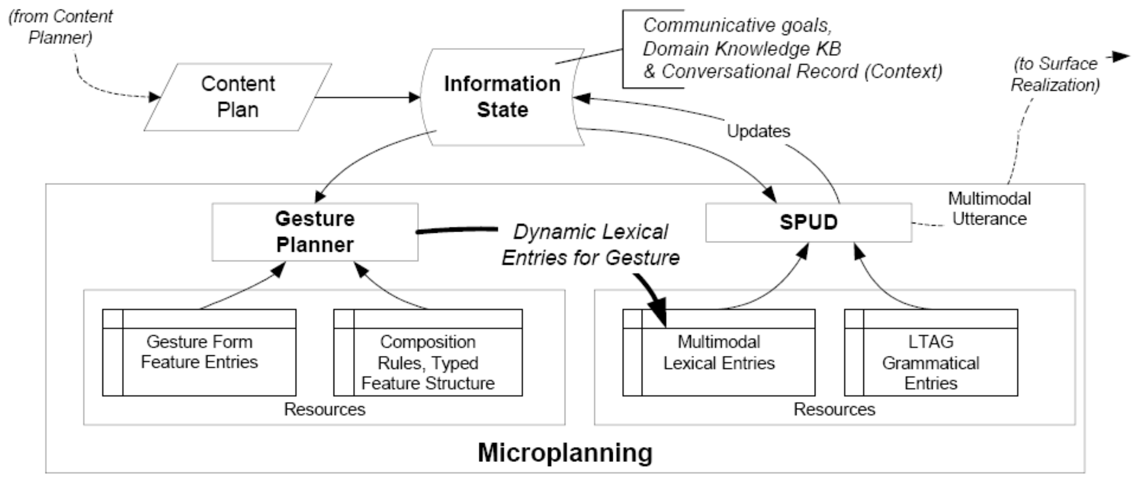

(from Content Planner)
Content Plan
Information State
Communicative goals, Domain Knowledge KB & Conversational Record (Context)
(to Surface Realization)
Updates
Gesture Planner
Dynamic Lexical Entries for Gesture
SPUD
Multimodal Utterance
Gesture Form Feature Entries
Composition Rules, Typed Feature Structure
Resources
Multimodal Lexical Entries
LTAG Grammatical Entries
Resources
Microplanning

Figure 32 Microplanning architecture [14,40,65]

The SPUD (Sentence Planning Using Descriptions) system [62] can be used for NLG. The NUMACK system uses a version of SPUD lite, a prolog (a logic programming language) version. Cassell et al. [14] extended SPUD so it can be used for multimodal generation. A similar approach (see [14] for details) is used in the NUMACK system to specify multimodal utterances including the dynamically generated gestures. These utterances are then passed on to the final step inside the generation pipeline, namely: surface realization.

4.1.4 Separation between Brain and Body

The *Interpreter* developed for the NUMACK system, is labeled NUMACK's Brain Interpreter, thus far neither Brain nor Body has been mentioned in the description of the NUMACK system. The separation of what will be referred to as Brain and Body lies inside the surface realization component (see Figure 32), the complete description which has been given up to this point can be seen as Brain, it manages: the interaction with the user and the planning and generation of specifications of multimodal utterances. Before the Brain passes the multimodal utterance specification to the Body, the specification is enriched with appropriate nonverbal and paraverbal behaviors. This is done by the Behavior Expression Animation Toolkit (BEAT) [26]. Within the NUMACK system, BEAT (see Figure 33) is used as a module to add additional behaviors to the utterance generated by the micro-planner. The module uses only the text of the utterance to add behaviors like, intonation, gaze and eye blinks. The language tagging component annotates (using XML-tags) the textual input with linguistic and contextual input. Using this annotated text all possible behaviors are suggested, conflicting behaviors are detected by the Behavior Selection process and a final set of behaviors is selected. The Behavior Scheduling component converts the final annotated text into a set of instructions which can be used for animation in an animation system. This set of instructions is then turned into a different XML format, the Multimodal Utterance Representation Markup Language (MURML) (see section 3.3.1), to be read by NUMACK's body.

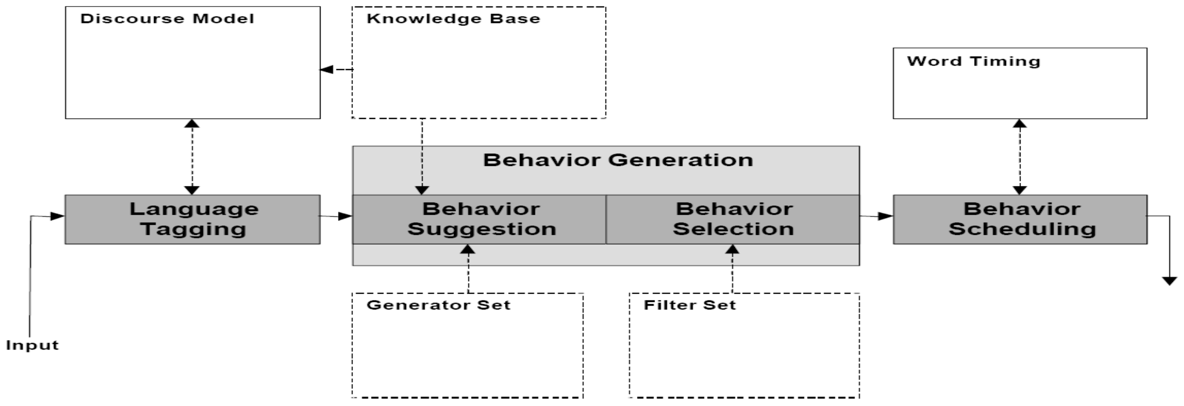


Figure 33 The BEAT architecture [16]

The Body's responsibility is the audiovisual rendering of the multimodal utterances generated by the Brain. The original NUMACK system uses the Articulated Communicate Engine (ACE) to bridge between the generation and the multimodal output, which in this case is rendered by the Festival Text To Speech (TTS) system for audio, and Open Inventor for the visual part. The ACE system will be discussed in more detail in section 4.3 about the ECAT-Compiler while the Open Inventor part of the original system will come back in the discussion of the Open Inventor *Translator* in section 4.4.1.

4.1.5 The Interpreter

The *Interpreter* for NUMACK hooks into the Brain interfacing this component with the *Compiler*. Since the Brain already uses MURML as output language no conversion is necessary to make it compliant with the input format of the *Compiler*. The original NUMACK Brain is implemented in JAVA, with prolog bindings to incorporate both the content and micro planner. Originally the NUMACK Brain component was implemented as an agent within a larger multi-agent architecture, using a specific JAVA library. In such a system a process can be seen as an agent. Since ECAT aims to have as little dependencies as possible, with relation to communication between the different components, it was decided to get rid of the reliance on this multi-agent library. Instead of this agent communication system a simple JAVA socket was implemented to send the MURML utterances, outputted by the Brain, to the ECAT-Compiler.

4.1.6 The output

Before NUMACK's utterances are sent to the *Compiler*, they are marked up using MURML (see 3.3.1), inside NUMACK's Brain. Figure 34 shows an example.

```
<UTTERANCE HEARER="USER" RETRACT="YES" SCENE="NU" SPEAKER="NUMACK">
        <CHUNK>
          <TREE-SPEC>
            <GAZE DIRECTION="TOWARDS_HEARER" FOCUS="ANY" PRIORITY="5">
            <INTONATION_TONE ENDTONE="L-L%">
            <POSTURESHIFT BODYPART="BOTH" ENERGY="LOW">after</POSTURESHIFT>
         that
        <SYNC> you will see a big building in front of you
          <GESTURE RETRACT="YES">
                  <symmetrical dominant="right_arm" symmetry="SymMS">
                    <parallel>
                      <static slot="PalmOrientation" value="DirA"/>
                      <static slot="ExtFingerOrientation" value="DirU"/>
                      <static slot="HandShape" value="BSflat"/>
                      <dynamic slot="HandLocation">
                            <dynamicElement type="linear">
                              <value type="start" name="LocUpperChest LocCenterRight LocStretched"/>
```

```
                                <value type="direction" name="DirA"/>
                                <value type="distance" name="120.0"/>
                              </dynamicElement>
                        </dynamic>
                      </parallel>
                    </symmetrical>
               </GESTURE>
             </SYNC>
             </INTONATION_TONE>
             </GAZE>
           </TREE-SPEC>
         </CHUNK>
</UTTERANCE>
```

Figure 34 Example MURML Utterance[14] from NUMACK's Brain

4.2 ECAT Compiler

As has been discussed in section 3.5.2 the Articulated Communicator Engine (ACE) has been chosen as the core component of the *Compiler*. In the following subsections the ACE system will be discussed first before the overall architecture and implementation of the *Compiler* is discussed.

4.2.1 Articulated Communicator Engine (ACE)

The ACE system is being used for several years now in multiple ECAs [36,37,39,65] and it's still under active development at the University of Bielefeld. The choice was made to use ACE as the core component of the *Compiler*. Because the system is well suited as a behavior realization system and is already being used within the Articulab and it's having support for MURML and (future) support for BML.
While not currently supporting the growing BML standard, it is however part of the ongoing development of ACE to support the language in the (near) future. Supporting BML would greatly simplify the development of *Interpreters*, since a growing number of projects utilize BML as their markup language for behavior descriptions.
This section will describe the functionality of ACE, the next section will describe the architecture of the complete *Compiler*, which is a bit more than ACE by itself, since the system needs to fit neatly within the ECAT pipeline.

Before going into the discussion of my extension of ACE, in the next subsection, this subsection will discuss the ACE system itself. This discussion is not directly reflecting any work or research carried out during this research project. It is included in this thesis because future ECAT users may or may not decide to use ECAT because of the features and inner-workings of ACE. Also an understanding of the following was necessary to get to a better understanding of the implementation of ACE. This was needed to let ACE function within ECAT as will be discussed in section 4.2.2.

The Multimodal Utterance Representation Markup Language (see section 3.3.1) is the native input language of the ACE system; therefore the processing of the input relies on the characteristics[15] underlying this language. Unlike many other behavior realization systems ACE supports animation of gestures which are specified by merely their overt form. This is in addition to the support for canned gestures from a gesticon, which can to some extend be parameterized to be suited for a specific case.

[14] The original NUMACK implementation did not utilize ACE's gesticon, but uses an internal mapping to map 'big-building'-gesture to the shown gesture-specification.
[15] Like the segmentation hypothesis and chunking, see section 3.3.1 for more details.

The processing of multimodal utterances within the ACE system uses a blackboard architecture [50]. The concept of blackboards has been introduced almost 30 years ago with the Hearsay-II-Speech-Understanding System [21]; it was designed to deal with ill-defined, complex applications [19]. A common metaphor used for blackboard systems is the scenario where a group of specialists (commonly referred to as Knowledge Sources) sit together, next to a blackboard, to solve a problem. When sufficient information is found a specialist records his contribution to the blackboard. Using this new contribution other specialist can be enabled to apply their expertise. This process is continued until the problem is solved [19]. Also in the field of ECAs blackboards have been used, for instance Gandalf, one of the first ECAs, is using the Ymir model which uses multiple blackboards [67].
ACE uses two types of blackboards for control, namely Chunk BlackBoards (ChunkBB) and Utterance BlackBoards (utteranceBB). The latter is used for control of complete utterances, which can contain multiple chunks, while the first is used for the control flow of chunks. The execution of behaviors within ACE essentially consists of executing chunks. Within a chunk numerous behaviors of different sorts can be defined like: speech acts, (arm) gestures, head gestures and facial animations. Each different sort of behavior has its own 'specialist' (or Knowledge Source), which are called BehaviorCreators. The tasks of chunk-execution could be seen as the problem to be solved on the Chunk BlackBoard by the different BehaviorCreators.

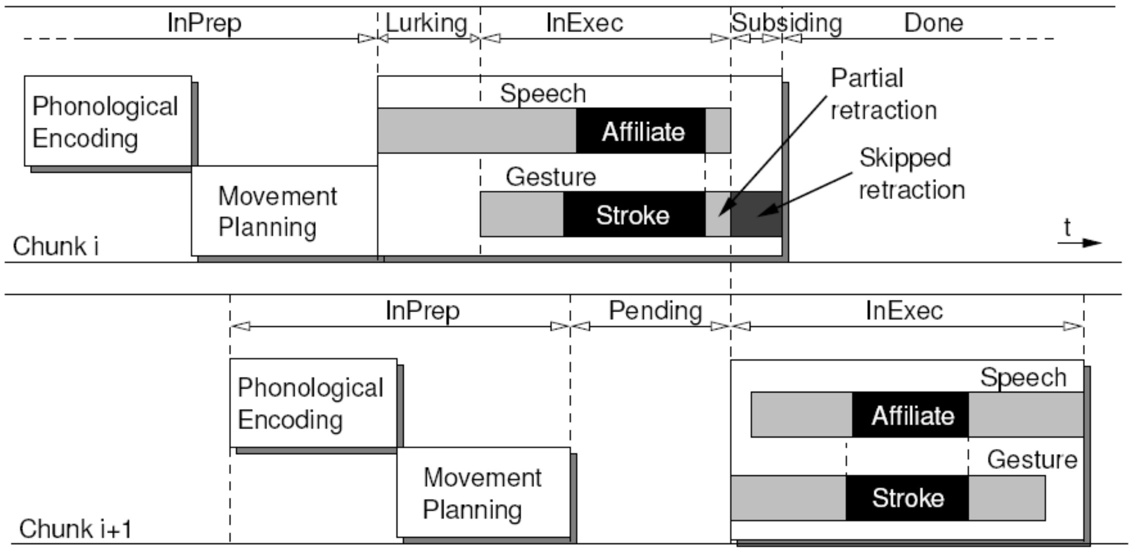


Figure 35 The states of execution of a chunk [41]

Before a chunk execution finishes it passes through different 'execution-states', which in chronological order are called: InPrep, Pending, Lurking, InExec, Subsiding and Done (see Figure 35).
During the state: InPrep (being in preparation), all the different behaviors are planned, gestures are prepared, speech synthesis is carried out, which also sets phoneme timings, which on its turn can be used to plan lip-synched animation of the face. As soon as the preparation is finished the state switches to Pending, in other words it's ready to start. When the previous chunk is finished executing (has reached the state: Subsiding or Done) the state is set to Lurking.
During Lurking intra-chunk synchrony is handled, this can for instance involve: pausing the speech, to allow the gesture to be prepared in time to synchronize its stroke with the affiliate. During Lurking the retraction of the previous gesture may not yet have been completed, which calls the need for blending the previous retraction of the previous gesture into the

preparation of the next. It could also be the other way around, meaning that the gesture does not need to be started yet, because the affiliate is located near the end of the speech. In this case the speech will be started during Lurking while the gesture 'holds'. Once all behaviors (speech, gesture, facial animations) are started the state of the chunk is set to InExec. As soon as all the behaviors are finished, the state is set to Subsiding.
During Subsiding gesture retraction will be done, once finished the state will be set to Done.
The cross-modal timing used within ACE is mainly dependent on the speech, which according to Kopp [41] is executed 'ballistically'. This means that in contrast to gesture, which is executed frame by frame, speech is started and cannot be interrupted or modified during playback. Therefore after phoneme timings are set, ACE will decide when it's time to start gesture preparation, so stroke and affiliate will be correctly synchronized. In the case that gesture preparation will take too long[16] an additional pause will be added to the speech.
As has been mentioned in section 3.3.1 MURML allows gestures to be specified using morphological features, to realize gestures of this kind ACE has a system for Motor Control which is based on Human motor control [41].
The ACE system as it is used with ECAs like Max [37] and NUMACK [65] is directly integrated with the rendering engine. In these cases this provides a complete behavior realization component. However ACE by itself is not a system which can be started and used, it is a set of C++ classes which needs to be extended for a specific implementation.
The next section will discuss the overall architecture of the complete *Compiler* and how ACE is integrated with it.

4.2.2 Compiler Architecture

The architecture of the *Compiler* is shown in Figure 36, which also shows which components of the Compiler tie into ACE and which components connect to *Interpreters* and *Translators*. Each of the different components of the *Compiler* will be discussed one by one in this section. A complete class-diagram is shown in Appendix D.

Coordinator:
The Coordinator class is the main class. It is in essence the *Compiler*. It coordinates the tasks of all the other classes. After initialization it enters its main processing loop which actually carries out two tasks: process network and process system.
Process_network instructs the listener to listen for connections as longs as there are less than two connections (which means the system is still waiting for an Interpreter and/or Translator to connect). If both Interpreter and Translator are connected the Coordinator instructs the Mediator to process input/output for both of them.
Process_system instructs the *Compiler* to process, if present, uncompiled behaviors. And after that lets the *Compiler* process updates coming from the ACE system. If this has been done, the Reporter is instructed to process the compiled behaviors.

Listener:
This is basically the socket server, which listens to incoming connections of *Interpreter* and *Translator*. If new connections are encountered these connections are stored in the Netstate.

Mediator:
The Mediator handles the input of the connections which are stored in the netstate. This initially means the handling of the identification of the connected clients as either *Translator*

[16] This could be due to a large distance between the hand location of the previous stroke and the next.

or *Interpreter*. Once both an *Interpreter* and a *Translator* have connected to the system, the Mediator will process the input received from the *Interpreter* (MURML-specs) and put them in a queue of uncompiled behaviors within the Sysstate.

Netstate:
This class is used to store the connections to the both *Interpreter* and *Translator* as well as keeping track of their roles. Since both *Interpreter* and *Translator* connect to the *Compiler* in a similar way, each connection is associated with a role, which is set during identification (see Mediator) to distinguish *Translator*-connections from *Interpreter*-connections.

Bridge:
Does nothing more than bridge between the Sysstate and the Netstate. This makes it possible for the Mediator to add uncompiled behaviors to the system and for the Reporter to issue the sending of compiled behaviors to the *Translator*.

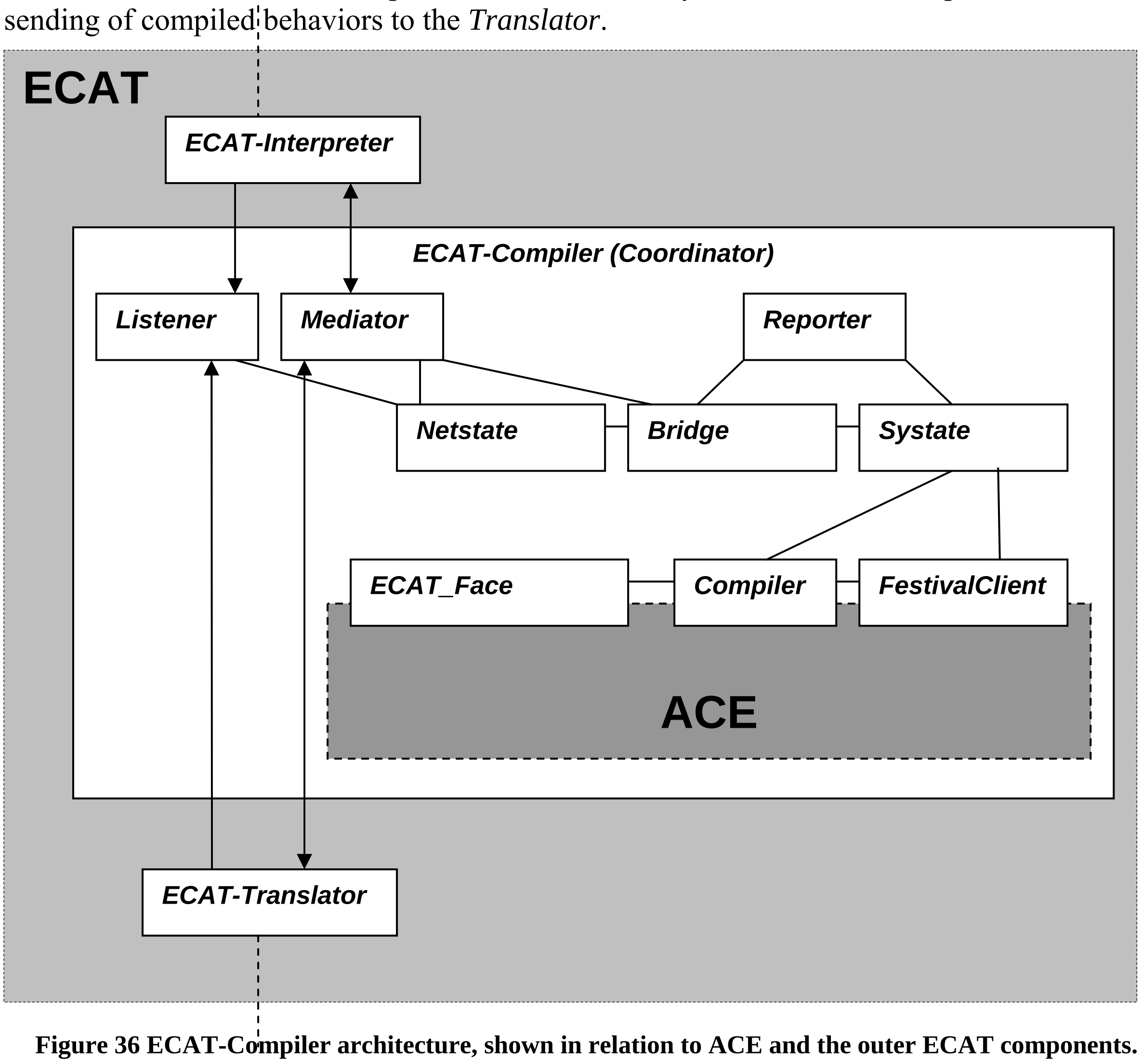


Figure 36 ECAT-Compiler architecture, shown in relation to ACE and the outer ECAT components.

Reporter:
The Reporter makes sure compiled behaviors, which are stored in a queue inside the sysstate, are sent to the connected *Translator*.

Sysstate:

This class is used to store queues of compiled and uncompiled behaviors, and includes utility functions for easy maintenance of these queues.

FestivalClient:
This class inherits from the SPU_Festival class of ACE, which is responsible for the planning, preparation and execution of speech. It uses the festival TTS [8] C++ library to utilize speech synthesis and audio storage functions of festival. After synthesis the speech is stored in a wave-file which can then be played by the *Translator*. Speech synthesis is carried out per chunk; therefore for every chunk a separate wave-file will be created.
It is also one of the specialists (see section 4.2.1) contributing to the Chunk BlackBoard, its contribution lies in providing speech duration and phoneme timing information to the blackboard. This information can be used by other specialist to time other behaviors, such as gestures, accordingly. The FestivalClient is also responsible for signaling when it's time for the *Translator* to start audio playback. It does so by creating a COL specification (referring to the created wave-file, see Figure 26) and it places this specification in the queue of compiled behaviors inside the sysstate.

ECAT_Face:
This class inherits from the HeadControl class of ACE; it is responsible for planning, preparation and execution of gaze, eyebrow raises, facial animation and also lip-synch animation.

Compiler:
This class inherits from the ACE class MultiModalFigure. This isn't the most specialistic class which ACE offers. The most specialistic class ReactiveFigure also offers behaviors like eye blinks and posture shifts. The prototype *Compiler* implementation does not inherit from this ACE class because there were unresolved dependencies. Since the reactive behaviors weren't needed for an initial prototype it was chosen to inherit from the MultiModalFigure class (see Table 8 for a description of ACE classes, with 'Compiler functionality', which can be used for inheritance).

Figure:
Base Class.
AnimatedFigure:
Base class for an articulated figure which can perform predefined animations and generates motion transitions automatically. To this end, means for inverse kinematics and keyframing are provided.
LocoFigure:
Inheriting from AnimatedFigure Adds functionality for locomotion.
MotorControlFigure:
Inherited from LocoFigure, this class contains lower levels of motor control which can be used for generating movement. Thus, the MotorContorlFigure represents the first step towards an active humanoid implemented by means of computer animation techniques.
Basically, this class contains all modules situated on the second (and final) stage of motion control hierarchy (see e.g. ECAI 2000 paper), notably "generation and execution of motor command".
Movement formation is a two-step process:
1. Overall movement specification during a planning phase. It is in this phase where local motor programs are created and sequenced. One can think of this process as anticipating the evocation of certain behaviors.
2. During runtime, the Imp's (behaviors) activate, refine, and instance themselves, yielding the ultimately executed animation. This phase resembles the "animation rendering" step."
MultimodalFigure:
Provides an articulated figure that is able to perform multimodal utterances by combining the

planning and execution of bodily or facial animations, respectively, with synthesizing prosodic speech. In addition, the figure has an explicit emotional state that affects the output behaviors produced. **behavior:ReactiveFigure:** Extends the MultimodalFigure by provisions for executing reactive behaviors. Reactive behaviors establish a stimulus-response loop that can be de-/activated and runs autonomously.

Table 8 ACE-classes shown from base class to most specialistic class

By utilizing the MultiModalFigure class the Compiler class is essentially carrying out the task of *compilation*. It is responsible for setting up initial ACE related settings, like the frameRate, which is used by ACE internally to calculate frameData.
It also initializes a FestivalClient which is a behaviorCreator (see section 4.2.1) responsible for speech production. Also it initializes ECAT_Face which is also a behaviorCreator, responsible for production of facial behaviors.

To let the 'ACE machinery' function, the Compiler class passes uncompiled behaviors (utterances), from the queue maintained in the sysstate, to ACE. It does so by calling the function **executeXMLSpec()**. This starts the processing of the utterance and its comprised chunks. The different phases of chunk execution have been discussed in the previous section. During chunk execution most of the work is carried out by the different specialists (see section 4.2.1), such as the FestivalClient, ECAT_HeadControl and GestureProductionUnit (which is an ACE class).

The functionality of the first two specialists have been discussed above, the third specialist the GestureProductionUnit is responsible for the planning and execution of gestures. The execution of gestures leads to updated 'geometry data'. 'Geometry data' is used internally by ACE and is directly used by an ECA such as Max [37]. The 'geometry data' only contains rotations for segments and links which have a corresponding visual representation in the model used by ACE (which in fact is Max [37]). This model consists of loose segments and links instead of one complete body model. The structure of the different links and segments and the corresponding visual objects are defined in Appendix C. Within ACE the term link means the same as joint, which could be: 'left_knee_joint'. Segment means the same as (sub) limb which could be for instance: 'left_lower_leg'.

In contrast to ACE its internal model NUMACK has a visual representation for each joint (see the white/grey joints in Figure 40). As can be seen in Appendix C, in ACE the knee joint has no visual representation (there is no shape-child under the knee-joint), while the ankle joint does.
The absence of a visual representation means the absence of rotation information in the 'geometry data'. As mentioned in section 3.5, ECAT supports articulated body models. And *Translators* should be able to control joints of an articulated model. For this reason the output of the *Compiler* (COL, see section 3.4) needs to contain rotational data for all joints in motion.

The original data of ACE, the 'geometry data', contains the absolute transformation of joints and segments and not the transformation relative to the parent, which is desired in the output.
Using data already present in ACE a new data structure has been created inside the Compiler class. This data structure is called 'joint data', it contains the rotation information of every joint of the model relative to its parent (see Figure 37)

With the 'joint data' the rotated joints are represented in a COL specification (see Figure 25). This COL specification is then added to the queue of compiled behaviors inside the sysstate. This queue is read by the Reporter which sends the COL specifications to a *Translator*.

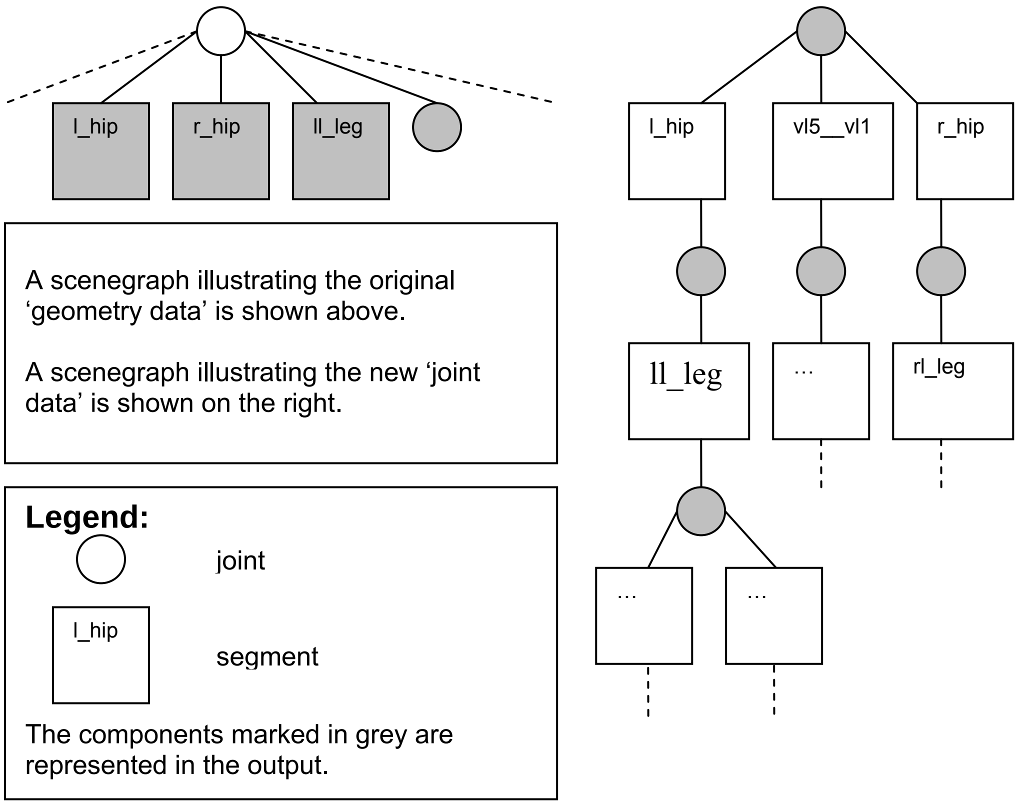

l_hip
r_hip
ll_leg
A scenegraph illustrating the original 'geometry data' is shown above.
A scenegraph illustrating the new 'joint data' is shown on the right.
Legend:
joint
l_hip
segment
The components marked in grey are represented in the output.
l_hip
vl5__vl1
r_hip
ll_leg
...
rl_leg
...
...

Figure 37 Scenegraph representation of original 'geometry data' versus new 'joint data'

4.3 Open Inventor Translator

4.3.1 The input

During compilation, the *Compiler* reads in the MURML utterances received from the Interpreter and feeds these utterances to ACE. For each frame the *Compiler* checks whether there is new information ready for realization (see section 4.2 for more details), for instance joints have been rotated or it's time to start speech playback. The new information will then be markup in the Compiler Output Language (COL) and passed on to a *Translator*.

```
<ecat>
        <audio name="utterance0_chunk0.aiff"/>
        <joints>
          <joint name="r_elbow"
                m00="0.996195" m01="0" m02="-0.0871557"
                m10="-0" m11="1" m12="0"
                m20="0.0871557" m21="-0" m22="0.996195" />
          <joint name="r_shoulder">
                m00="1" m01="0" m02="-0"
                m10="-0" m11="0.999848" m12="-0.0174524"
                m20="0" m21="0.0174524" m22="0.999848" />
           <joint name="rh_t_joint2">
                m00="1" m01="0" m02="0"
                m10="-0" m11="0.981627" m12="0.190809"
                m20="0" m21="-0.190809" m22="0.981627" />
        ...
        ...
         <joint name="rh_t_joint3">
                m00="1" m01="0" m02="0"
                m10="-0" m11="0.970296" m12="0.241922"
                m20="0" m21="-0.241922" m22="0.970296" />
        </joints>
</ecat>
```

Figure 38 COL Example

4.3.2 The translator

This *Translator* is in the first place implemented to be used with NUMACK's body. The original NUMACK system is using an older version of the Articulated Communicator Engine (ACE) which is integrated with Open Inventor [1] or Coin3D [9] in C++ for the audio/visual rendering.
To able to use the NUMACK body outside of ACE, the original skeletal data (see Appendix A) has been converted to native Open Inventor Scenegraph format (see Appendix B). This *Translator* will load the NUMACK body and also connect to the *Compiler* using a socket connection. By using input received over the socket, the scenegraph (and thus the body) can be altered, leading to an animated body of NUMACK. The *Translator* uses Open Inventor for the visual part, which relies on OpenGL [3]. For audio playback an OpenAL-player has been written, which can play wave files using the OpenAL library [4]. OpenAL which stands for Open Audio Library is to audio what OpenGL is to graphics.

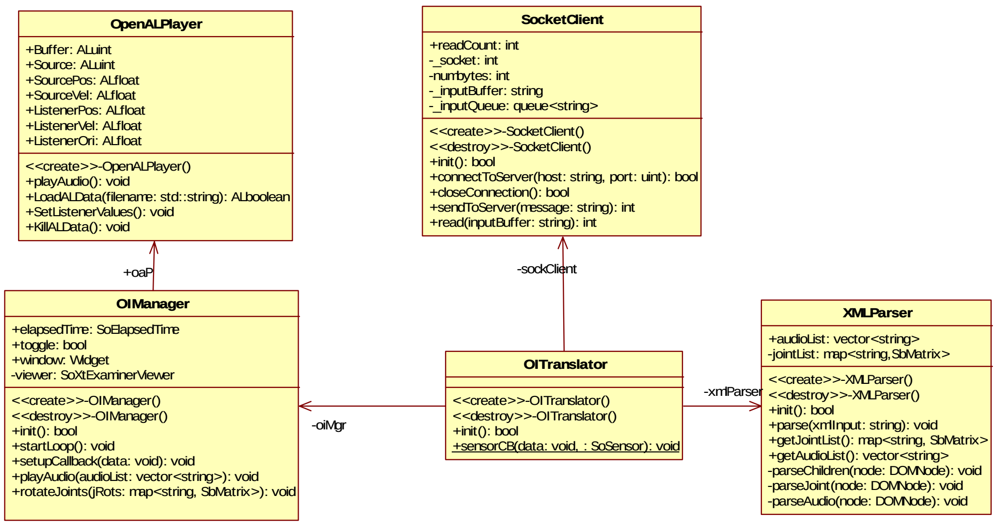

OpenALPlayer
+Buffer: ALuint
+Source: ALuint
+SourcePos: ALfloat
+SourceVel: ALfloat
+ListenerPos: ALfloat
+ListenerVel: ALfloat
+ListenerOri: ALfloat
<<create>>-OpenALPlayer()
+playAudio(): void
+LoadALData(filename: std::string): ALboolean
+SetListenerValues(): void
+KillALData(): void
SocketClient
+readCount: int
-_socket: int
-numbytes: int
-_inputBuffer: string
-_inputQueue: queue<string>
<<create>>-SocketClient()
<<destroy>>-SocketClient()
+init(): bool
+connectToServer(host: string, port: uint): bool
+closeConnection(): bool
+sendToServer(message: string): int
+read(inputBuffer: string): int
+oaP
-sockClient
OIManager
+elapsedTime: SoElapsedTime
+toggle: bool
+window: Widget
-viewer: SoXtExaminerViewer
<<create>>-OIManager()
<<destroy>>-OIManager()
+init(): bool
+startLoop(): void
+setupCallback(data: void): void
+playAudio(audioList: vector<string>): void
+rotateJoints(jRots: map<string, SbMatrix>): void
-oiMgr
OITranslator
<<create>>-OITranslator()
<<destroy>>-OITranslator()
+init(): bool
+sensorCB(data: void, : SoSensor): void
-xmlParser
XMLParser
+audioList: vector<string>
-jointList: map<string,SbMatrix>
<<create>>-XMLParser()
<<destroy>>-XMLParser()
+init(): bool
+parse(xmlInput: string): void
+getJointList(): map<string, SbMatrix>
+getAudioList(): vector<string>
-parseChildren(node: DOMNode): void
-parseJoint(node: DOMNode): void
-parseAudio(node: DOMNode): void

Figure 39 Class diagram of the Open Inventor Translator

The class diagram of the Open Inventor *Translator* shown in Figure 39 contains 5 classes. The functionality of the *Translator* will be explained class by class.

OIManager:
This is the main class, which is responsible for initialization and also includes the main processing loop. Before starting the main loop it does four things: it initializes the OpenALPlayer (for audio playback), loads the scenegraph from file (see Appendix B), opens a SoXtExaminerViewer (see Figure 40) and it sets up a callback function to be called every frame. Once this is done the viewer with the contents of the scenegraph will be shown and the callback function will be called every frame.

If the OITranslator (discussed next) thinks it's time to act the functions playAudio and/or rotateJoints will be called. The first function is responsible for audio playback (using the OpenALPlayer) and the second is responsible for applying the joint rotations to the scenegraph.

OITranslator:
The OITranslator is with no surprise the class with the core functionality. It's responsible for setting up a SocketClient and an XMLParser. The former is responsible for setting up a connection with the Compiler and the latter for parsing the incoming XML data (which is marked up in COL, see section 3.4). The OITranslator uses the SocketClient to connect and identify to the Compiler.
The callback function mentioned in the description of the OIManager resides inside the OITranslator. When called, this function checks with the SocketClient if there is any data, if so it instructs the XMLParser to parse this data and the OIManager to rotate joints and playback audio.

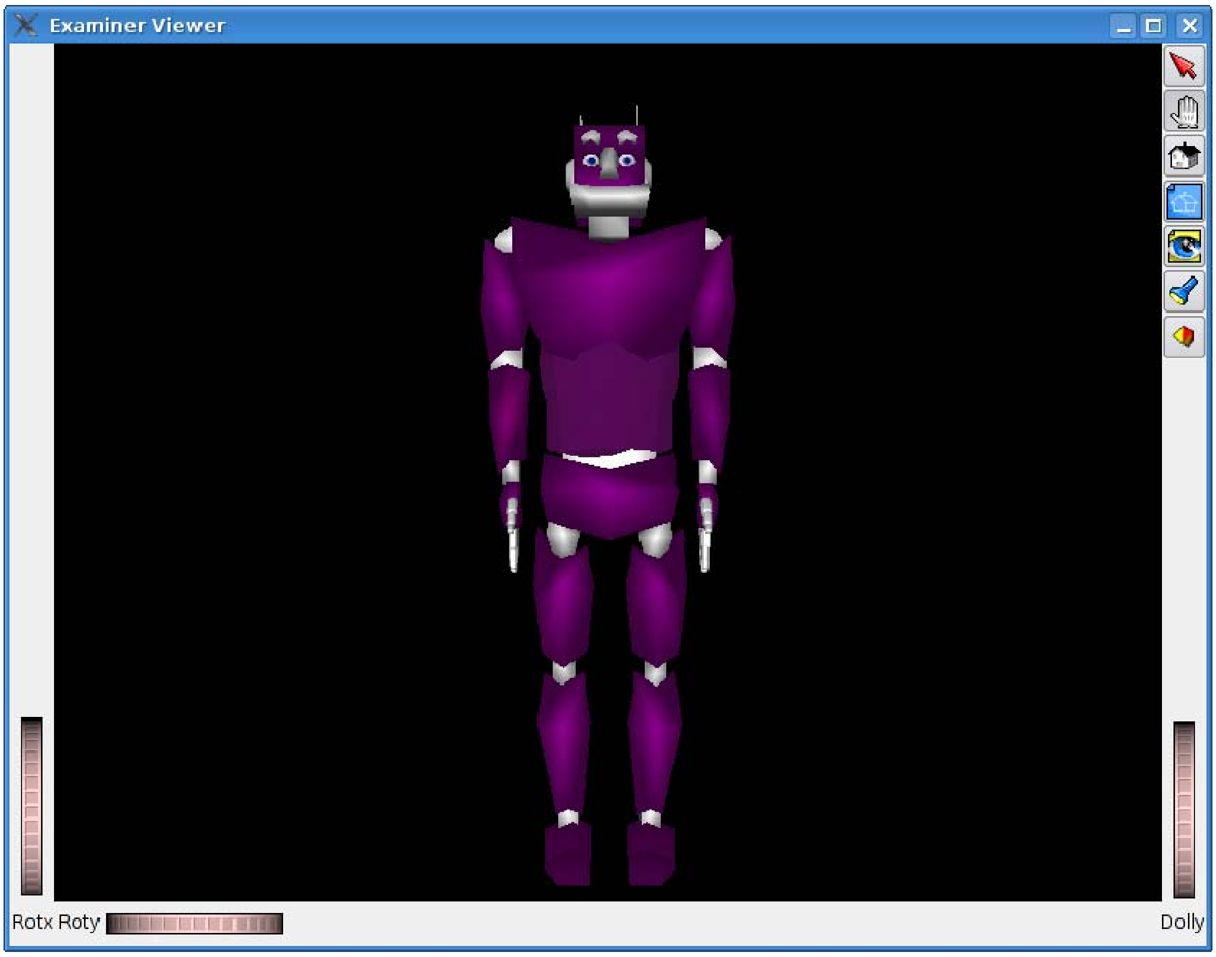


Figure 40 Examiner Viewer showing the scenegraph

XMLParser:
The XMLParser uses the XercesC library [10] to construct a parser for the incoming XML data. The XML input is expected to be marked up in COL. The current version of COL can contain joint rotations (specified in a 3x3 rotation-matrix) and audio playback directives. Parsed joint rotations are stored internally in a matrix and the filename for audio playback is also stored internally. The OITranslator uses these internally stored data to send them to the OIManager for audiovisual rendering.

The XMLParser is controlled from the OITranslator.

SocketClient:
Not much to be said about this class, it's a socket-client which can be used to connect and communicate with a socket-server, the ECAT-Compiler in this case.
It is controlled from the OITranslator.

OpenALPlayer:
Very basic audio player, which can play audio from wave files.
It is controlled from the OIManager.

4.3.2 The output

Calling the appropriate functions to play audio and to apply joint rotations, will lead to a fully realized audiovisual Embodied Conversational Agent (Figure 41).

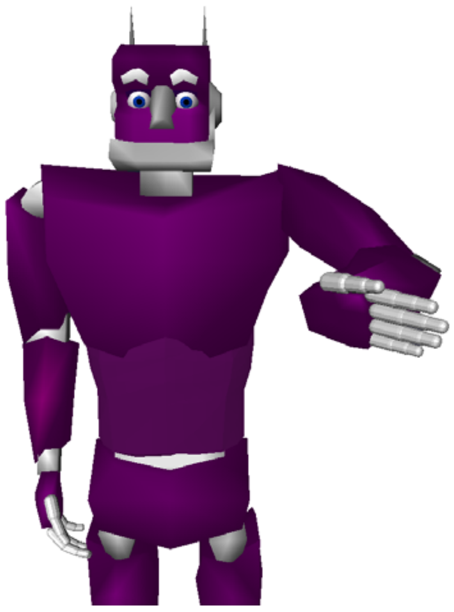
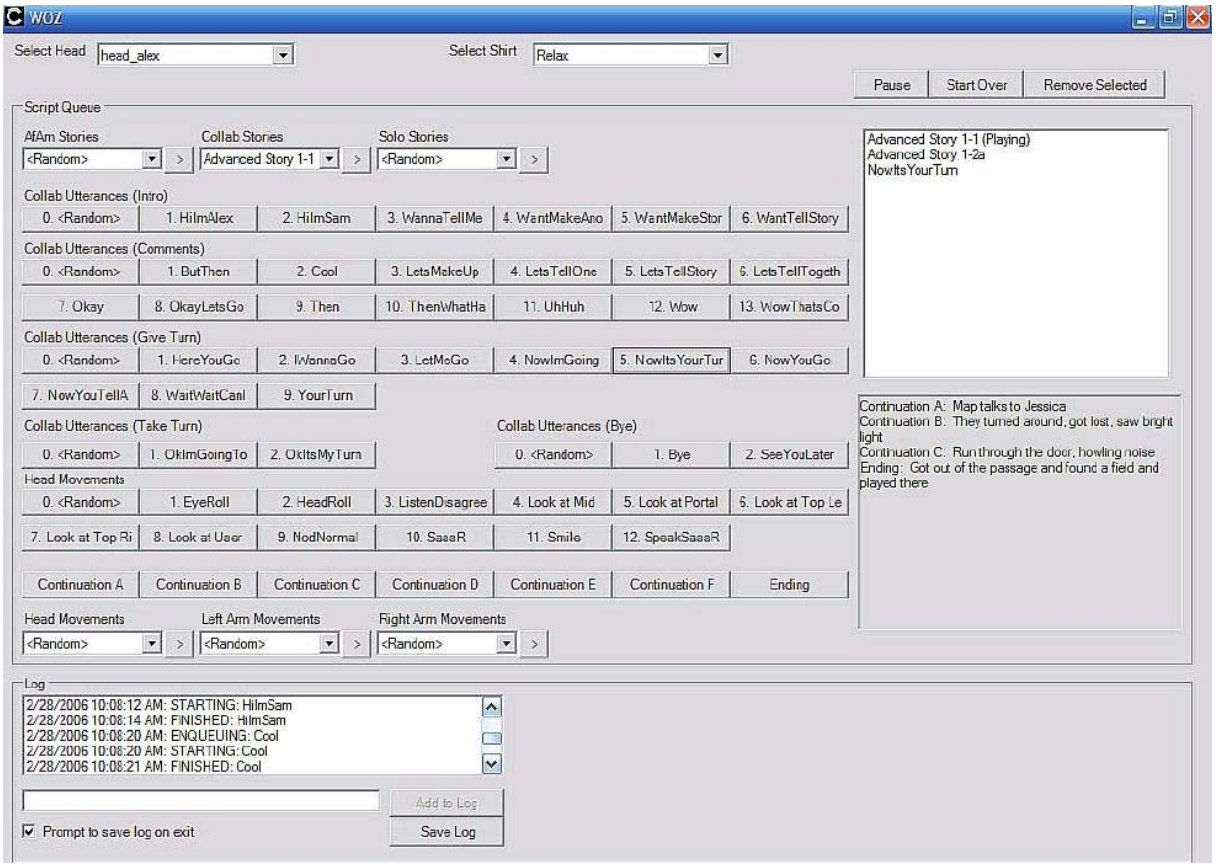


Figure 41 The result, showing a 'go left'-gesture.

5 Additional ECAT components (under development)

5.1 WOZ Interpreter

This *Interpreter* is still under development. It hooks into the existing WOZ interface (see Figure 42) used for controlling the Sam and Alex Virtual Peers. The WOZ interface normally sends its commands directly to the, currently used, Flash interface for animations. The interface is programmed in C#; a socket connection was added to the original system to connect with the ECAT-Compiler. This socket connection can now be used to send input from the WOZ to the *Compiler* instead to the Flash Interface.



Select Head: head_alex
Select Shirt: Relax

AfAm Stories: <Random>
Collab Stories: Advanced Story 1-1
Solo Stories: <Random>

Head Movements: <Random>
Left Arm Movements: <Random>
Right Arm Movements: <Random>

Advanced Story 1-1 (Playing)
Advanced Story 1-2a
NowItsYourTurn

Continuation A: Map talks to Jessica
Continuation B: They turned around, got lost, saw bright light
Continuation C: Run through the door, howling noise
Ending: Got out of the passage and found a field and played there

```
2/28/2006 10:08:12 AM: STARTING: HiImSam
2/28/2006 10:08:14 AM: FINISHED: HiImSam
2/28/2006 10:08:20 AM: ENQUEUING: Cool
2/28/2006 10:08:20 AM: STARTING: Cool
2/28/2006 10:08:21 AM: FINISHED: Cool
```

[✓] Prompt to save log on exit

Figure 42 WOZ-Panel for Sam / Alex [22]

The work which is left to do is to create (a sequence of) MURML utterance(s) for every button, so these can be send to the *Compiler* for further processing.

5.2 Ogre3D translator

This *Translator* has been far from fully tested yet, but it's using a rather simple Sam model, designed in 3D Studio Max and connects to the *Compiler* using a socket implemented in C++.

5.3 TGS Open Inventor translator

This is a possible future *Translator*, if so, it would build upon the original Sam Castlemate [60] version, using TGS Open Inventor in Java. The problem however is that TGS Open Inventor needs a license, which isn't available at the Articulab, it might be possible to use a temporary license for this purpose. Even if this is possible it's still questionable if the software is reusable at all, since it has not been used (or maintained) for years, it might be easier to find a way to load the VRML data files into the *Translator* mentioned in section 4.4.1.

5.4 Panda3D translator

This *Translator* is not fully functional yet. Since the current version does not support audio playback. The model used is a rather simple Sam model, designed in 3D Studio Max and connects to the ECAT-Compiler using a socket implemented in Python. The simplified class-diagram is shown below. The internal functionality is similar to that of the Open Inventor Translator, without a separate component for audio playback.

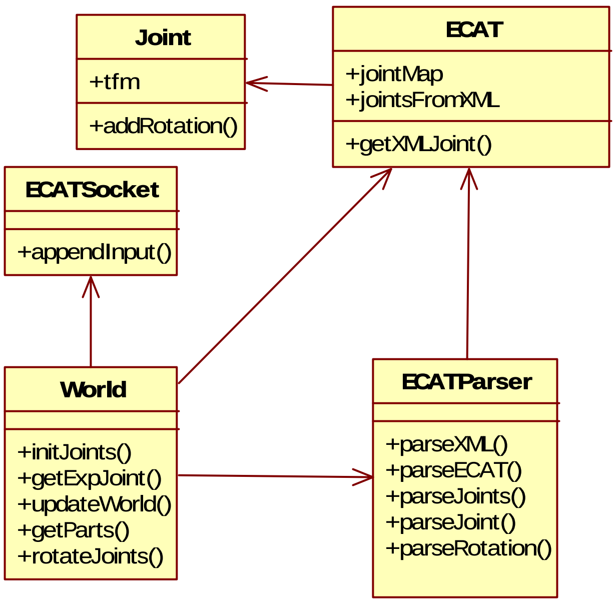


Figure 43 Class diagram of the Panda3D Translator

General functionality

The World class contains an ECATSocket for communication with the *Compiler*, an ECATParser to parse incoming XML data (specified in COL, see section 3.4) and an ECAT object in which an internal representation of the XML data is stored. The data stored in the ECAT object is read every frame and the Sam model is updated accordingly.

Mismatch between local coordinate systems of the *Translator* and the *Compiler*

The simplified class-diagram in Figure 43 does not show all the functions of the World class. This class also contains some utility classes to fix the problem shown in Figure 44. The example shows a rotation of 90 degrees of the left elbow. The ECA on the left (NUMACK) corresponds to the internal representation of the *Compiler*, while the ECA on the left

(Panda3D Sam) is the Panda3D *Translator* in action. Both should show the same pose, which isn't the case.

ECAControl

All joint rotations send by the *Compiler* are generated on the fly from high-level behaviors. At the time of testing the Panda3D *Translator* the only possible input for the *Compiler* was coming from the NUMACK's Brain *Interpreter*. It was soon clear that this input did not lead to the desired output in the *Translator*. To be able to further analyze what was causing this problem the ECAControl program was created.
ECAControl simulates the output of the *Compiler*, only supporting joint rotations for now. Figure 45 shows the Panda3D Translator in action controlled by ECAControl. ECAControl is a simple GUI with 3 sliders for every joint and with a viewer with the internal representation of the *Compiler*. By moving the sliders the different joints can be rotated along the local x, y and z-axes. The screenshot in Figure 45 is taken after the solution to the aforementioned problem was applied, that's why the caption shows 'corrected'. This solution will be discussed in the following.

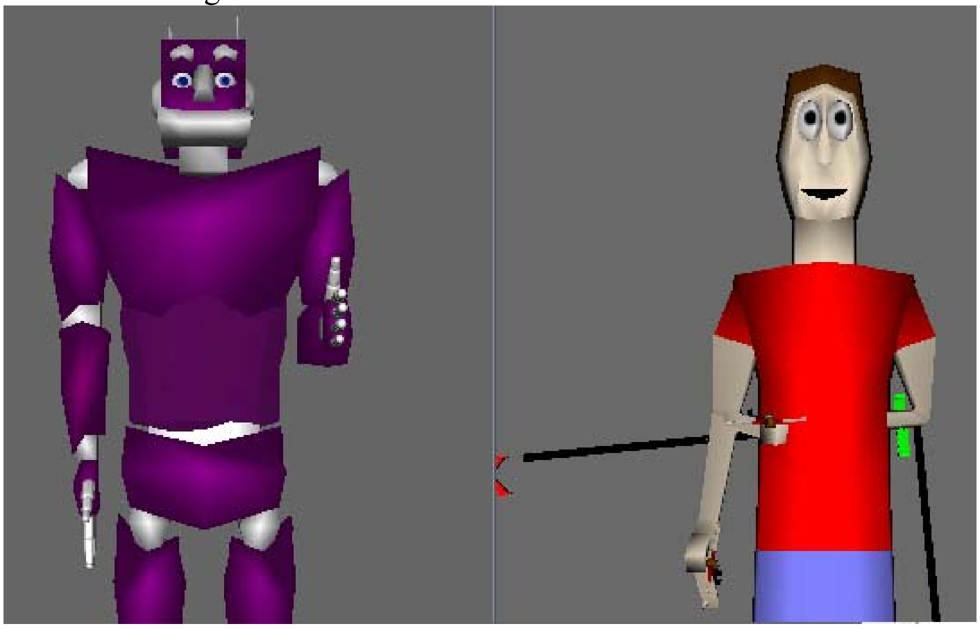


Figure 44 Panda3D model controlled by 'ECAControl'

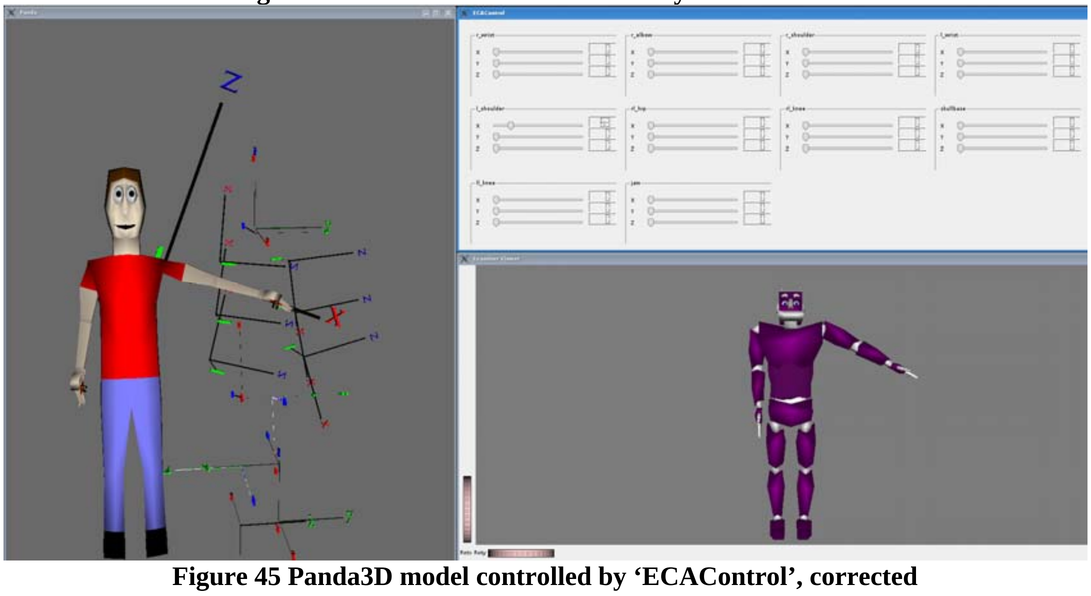


Figure 45 Panda3D model controlled by 'ECAControl', corrected

Fixing the mismatch

Having a tool to easily control the joint rotations send to the *Translator* with a viewer showing the desired pose greatly facilitated the ability to analyze and solve the problem at hand.
If we look at both screenshots above, the first shows a 90 degree rotation around the x-axis of the left-elbow and the second a 63 degree rotation around the y-axis of the left-shoulder. A close look at the second screenshot shows that the arm of the left model is rotated further away from the body as it is in the right model. This is merely a problem of default rotation, which can easily be altered. The real problem at hand is the fact that joints seem to rotate along a different axis (see Figure 44). This is because the local coordinate systems in the Panda3D model differ from the local coordinate systems used in the ECAT-Compiler.
Although it is hard to see the Panda3D model also shows a set of coordinate axes (x, y, z) for every joint and also one larger set of x, y, z coordinate axes for the previously rotated joint.

Compiler / ACE	Panda3D Sam (see Figure 45)
x-out	x-down
y-left	y-in
z-down	z-right

Table 9 Orientation[17] of coordinate axes for zero rotation of the left-elbow joint

From Table 9 we can conclude that x in 'Sam's world' corresponds to negative z in 'Compiler world', y to x and z to negative y. With this information a correction matrix can be calculated. Using this matrix, the matrix rotations received from the *Compiler* can be pre multiplied by this correction matrix and post multiplied by its inverse to get the desired result. The precise details of the calculation of this correction matrix (and how it's applied) can be found in the implementation and its documentation.

With the above-mentioned corrections this *Translator* is able to realize and render joint rotations received from the *Compiler*; audio playback is yet to be added.

[17] Note: XYZ corresponds to RGB, in other words the red-axis is the x-axis, green is y and blue is z.

6 Evaluation

The proof-of-concept prototype includes one fully functional *Interpreter* (NUMACK's Brain), one partially functional *Interpreter* (Sam/Alex WOZ), one fully functional *Translator* (Open Inventor), one partially functional *Translator* (Panda3D) and a functional *Compiler* (ACE).
For body animation and speech the complete pipeline is working from start to end. There are still some features missing, but most functionality which is in the original NUMACK system is already present in the ECAT prototype.
NUMACK reimplemented with ECAT modules can keep up with the original implementation performance wise.
The Panda3D *Translator* on the other hand is not running very fast on the one computer which it has been tested on. This computer is a laptop which is unable to run on full speed, more testing needs to be done on a computer better suited for this task.

The *Interpreter* for NUMACK's Brain was rather simple and straight-forward to implement.
It is anticipated that the *Interpreter* for the Sam/Alex WOZ can be finished without much effort. All that is left to do is to create MURML-utterances for every button on the WOZ-panel, once these are created these can be send to the *Compiler* when the corresponding button is pressed.
The most time consuming task of the development of the Open Inventor *Translator* was the conversion of the NUMACK skeleton file (see Appendix A), as was used by an older version of ACE, into an Open Inventor scenegraph (see Appendix B). The rest of this *Translator* was implemented rather quickly.
The Panda3D *Translator* took some more time to develop than the Open Inventor *Translator*. This might be due to several reasons. The first being: the fact that the authors were unfamiliar with Panda3D and Python. Another reason is the mismatch between the model used by the *Compiler* and the model used by this *Translator*. However, the idea behind the solution to solve this mismatch can be reused in the development of future *Translators*. While the *Translator* does not yet support audio playback, it should not be a too big of a problem to add this.
The *Compiler* is still missing some features which are supported by the ACE. These are features like: facial animation, lip synchronization, secondary behaviors (like: breathing and eye blinks). Both facial animation and lip synchronization are supported by ACE but they are still missing in the Compiler implementation. This is due to the fact that this functionality is tailored to needs of specific model and cannot be applied to a wide variety of rendering engines and/or models as it is. Secondary behaviors are not included in the current version of the *Compiler* because it wasn't possible to inherit from ReactiveFigure due to dependencies problems.

7 Conclusion

This thesis and the accompanied work have led to a proof-of-concept prototype of ECAT. This prototype includes at least one version of every ECAT component. As has been stated in the previous chapter the ECAT version of NUMACK can keep up with the original version. This is a very promising result, which means ECAT can be successfully used for ECA development. This wasn't trivial in the first place, because having three separate components connect via socket connections could have caused noticeable slow down. The current version is only a prototype, it remains to be seen that a full version can achieve similar results. If slow down does occur, using a compressed format like MPEG-4 might be useful for future implementations.
The current version is still lacking some important features with the most important one being facial animation. Regarding facial animation, the current ACE implementation uses 21 logical muscles which are defined as a group of vertices on the mesh of the face. This is very specific for the Max agent for which it has been written. This is why the current implementation of the *Compiler* does not include facial animation. A more generic approach needs to be added to the *Compiler*/ACE before it can be used for ECAT, since ECAT aims to support a wide variety of models and rendering engines. For this reason support for facial animation tailored to one specific model was not included in the prototype.

The time spent on *Interpreters* and *Translators* was very short compared to the development of the *Compiler*. Having a few example components should bring down development time of future *Interpreters* and *Translators* even more. As it is very important within ECAT that the outer components are easy to develop, this can also be seen as a promising result. Also having one functional *Compiler* will enable ECAT users to connect their *Interpreters* and *Translators* to it, to get a fully functional ECA.
We can also conclude on the Toolkit as a whole by looking at the different stages/modules separately.
Interpreters can help to bridge the gap between systems which are not yet SAIBA or MURML compliant, thereby increasing the size of potential ECAT, and therefore SAIBA, users. Being used by a wider public will certainly help working towards common interface standards.
Compiler(s) are situated inside ECAT. Therefore they do not directly depend on the outer systems which are connected to *Interpreters* and *Translators*. Having common interfaces on both ends of the *Compiler* does make it possible to create a new *Compiler* or use a different one. Just like it is the case for the other components, having multiple alternatives allows researchers to use the components best suited for their purpose.
Translators allow researchers to switch rendering engines, without having to reimplement a larger part of their system. With a growing number of publicly available engines this can prove to be a useful mechanism to try and compare new engines.
Having a complete repository of ECAT components allows the connection of different behavior generating applications to different rendering engines. This way researchers can share the components of their ECA which is the most prominent for their research. This can facilitate collaboration between research labs. And it will also enable researchers to focus on the part of ECAs which is most important to them.
Once the repository of ECAT components grows the SAIBA framework can grow to a wider accepted standard. Before this can happen the proof-of-concept version of ECAT has to be developed further, so it can be used to create mature ECA systems.
The proof-of-concept prototype can serve as a valuable base for this future development.

8 Recommendations

The current proof-of-concept prototype can serve as a base for future work. A logical next step would be to work on the functionality which is missing from the prototype. This includes lip synchronization, facial animation and secondary behaviors like posture shifts and eye blinks. Secondary behaviors can be added once the dependencies of the ReactiveFigure can be resolved, for a first prototype these weren't that important. Therefore the implementation was given a higher priority over resolving these dependency problems.
Facial animation and lip synchronization raise the need for animation technique for the face which can be used with a wide variety of rendering engines and with a wide variety of models. MPEG-4 which is not included in the initial prototype offers facial and body animation. The facial animation part of MPEG-4 seems like the best way to go, when the goal is to support a wide variety of models. If MPEG-4 facial animation is added to the *Compiler* it may be a very small step to also include MPEG-4 body animation.
Once facial animation support is included ECAT can be used for a lot of mature ECA systems.

Once ECAT can be used to build mature ECA systems, it would be a good time to reshift focus on modularization. Sharing and reusing modules is key to the success of ECAT and is also one of the important reasons why the project was started in the first place. An important next step towards truly modularized ECAs would be the separation of Text-to-speech. Text-to-speech (TTS) can most certainly be seen as a separate area of expertise, having a separate component for this task enables experts in the field of TTS to share their work. This enables researchers from other fields to use the TTS approach which is the most suited for their ECA. While the separation of TTS is not included in the current version of ECAT and also not discussed in this thesis, it's a rather small step. Since the current *Compiler* implementation uses a separate class for the TTS task. The communication between this class and other parts of the system mostly exists of text going to the TTS-class and phoneme-timings coming back. Just like the interfaces with *Interpreters* and *Translator*s a socket connection interface could be created between the *Compiler* and the TTS-module. Every TTS application which can provide phoneme timings can then be used as a TTS-module for ECAT.

Currently ECAT is using MURML as the input language for the *Compiler*; native support for BML should be a fact in the (near) future. Having ECAT as a SAIBA compliant behavior realization system will lower the threshold for people to start using ECAT. In addition to having a repository of ECAT components it would be very useful to also have a repository of gestures in BML. Such a repository can serve not only ECAT but the complete SAIBA community. With an ECAT repository and a gesture repository a wide variety of gestures can be tested on different models and rendering engines.

While flexibility and availability of TTS modules and gestures would be an important next step for ECAT, even more important would be to have a completely new ECA system build from scratch using ECAT components. Once the WOZ-*interpreter* is completed, facial animation included and a decent model is acquired 'Sam/Alex 2.0' can be the first ECA completely new ECA to use ECAT.

References

[1]"Open Inventor"; http://oss.sgi.com/projects/inventor/.

[2]"Unreal Tournament"; http://www.unrealtournament.com/.

[3]"OpenGL"; http://www.opengl.org.

[4]"OpenAL"; http://www.openal.org.

[5]"ArticuLab"; http://articulab.northwestern.edu.

[6]"Panda3D"; http://www.panda3d.org.

[7]"Ogre3D"; http://www.ogre3d.org.

[8]"The Festival Speech Synthesis System"; http://www.cstr.ed.ac.uk/projects/festival/.

[9]"Coin3D"; http://www.coin3d.org/.

[10]"Xerces C++ Parser"; http://xerces.apache.org/xerces-c/.

[11] Y. Arafa and A. Mamdani, "Scripting embodied agents behaviour with CML: character markup language," *Proceedings of the 8th international conference on Intelligent user interfaces*, 2003, pp. 313-316.

[12] N. Badler, R. Bindiganavale, J. Bourne, M. Palmer, J. Shi, and W. Schuler, "A Parameterized Action Representation for Virtual Human Agents," *Embodied Conversational Agents*, 2000, pp. 256-284.

[13] J. Cassell, T. Bickmore, M. Billinghurst, L. Campbell, K. Chang, H. Vilhjálmsson, and H. Yan, "Embodiment in conversational interfaces: Rea," *Proceedings of the SIGCHI conference on Human factors in computing systems: the CHI is the limit*, 1999, pp. 520-527.

[14] J. Cassell, S. Kopp, P. Tepper, K. Ferriman, and K. Striegnitz, *Trading spaces: How humans and humanoids use speech and gesture to give directions*, Engineering Approaches to Conversational Informatics. Wiley, 2007.

[15] J. Cassell, T. Stocky, T. Bickmore, Y. Gao, Y. Nakano, K. Ryokai, D. Tversky, C. Vaucelle, and H. Vilhjálmsson, "MACK: Media lab Autonomous Conversational Kiosk," *Proceedings of Imagina*, vol. 2, 2002, pp. 12-15.

[16] J. Cassell, H.H. Vilhjálmsson, and T. Bickmore, "BEAT: the Behavior Expression Animation Toolkit," *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, 2001, pp. 477-486.

[17] K. Chang, "Pantomime: Character Animation," *Motor Control System. Cambridge, MA, MIT*, 1999.

[18] R. Cole, S. van Vuuren, B. Pellom, K. Hacioglu, J. Ma, J. Movellan, S. Schwartz, D. Wade-Stein, W. Ward, and J. Yan, "Perceptive Animated Interfaces: First Steps Toward a New Paradigm for Human Computer Interaction," *Proceedings of the IEEE*, vol. 91, 2003, pp. 1391-1405.

[19] D.D. Corkill, "Blackboard systems," *AI Expert*, vol. 6, 1991, pp. 40-47.

[20] B. De Carolis, C. Pelachaud, I. Poggi, M. Steedman, and I.U.T. Paragraphe, "APML, a Markup Language for Believable Behavior Generation," *Life-Like Characters: Tools, Affective Functions, and Applications*, 2004.

[21] L.D. Erman, F. Hayes-Roth, V.R. Lesser, and D.R. Reddy, "The Hearsay-II Speech-Understanding System: Integrating Knowledge to Resolve Uncertainty," *ACM Computing Surveys*, vol. 12, 1980, pp. 213-253.

[22] J. Gratch, A. Okhmatovskaia, F. Lamothe, S. Marsella, M. Morales, R.J. van der Werf, and L.P. Morency, "Virtual Rapport," *Proceedings of the 6th International Conference on Intelligent Virtual Agents (IVA'06), Marina del Rey, CA*, 2006, pp. 14–27.

[23] J. Gratch, J. Rickel, E. Andre, J. Cassell, E. Petajan, and N. Badler, "Creating interactive virtual humans: some assembly required," *Intelligent Systems, IEEE*, vol. 17, 2002, pp. 54-63.

[24] J. Gratch, N. Wang, A. Okhmatovskaia, F. Lamothe, M. Morales, R.J. van der Werf, and L.P. Morency, "Can Virtual Humans Be More Engaging Than Real Ones?," *12th International Conference on Human-Computer Interaction, Beijing, China*, 2007.

[25] B. Hartmann, M. Mancini, and C. Pelachaud, "Formational parameters and adaptive prototype instantiation for MPEG-4 compliant gesture synthesis," *Proceedings of Computer Animation*, 2002, pp. 111-119.

[26] B. Hartmann, M. Mancini, and C. Pelachaud, "Implementing expressive gesture synthesis for embodied conversational agents," *Gesture Workshop*, vol. 2005, 2005.

[27] R.W. Hill Jr, J. Belanich, H.C. Lane, M. Core, M. Dixon, E. Forbell, J. Kim, and J. Hart, "Pedagogically Structured Game-based Training: Development of the ELECT BiLAT Simulation," *Proceedings of ASC06: The 25th Army Science Conference. Dept. of the Army*, 2006.

[28] R. Hill, J. Gratch, S. Marsella, J. Rickel, W. Swartout, and D. Traum, "Virtual Humans in the Mission Rehearsal Exercise System," *Künstliche Intelligenz*, vol. 4, 2003, pp. 5-10.

[29] Z. Huang, A. Eliens, and C. Visser, "XSTEP: A Markup Language for Embodied Agents," *Proceedings of the 16th International Conference on Computer Animation and Social Agents (CASA'2003)*, 2003.

[30] F. Iacobelli and J. Cassell, "Ethnic Identity and Engagement in Embodied Conversational Agents," *Proceedings of the 7th International Conference on Intelligent Virtual Agents (IVA'07)*, vol. 4722, 2007, pp. 57-63.

[31] M. Ishizuka and H. Prendinger, “Describing and generating multimodal contents featuring affective lifelike agents with MPML,” *New Gen. Comput.*, vol. 24, 2006, pp. 97-128.

[32] A. Kendon, “Gesticulation and speech: Two aspects of the process of utterance,” *The Relationship of Verbal and Nonverbal Communication*, 1980, pp. 207-227.

[33] P. Kenny, A. Hartholt, J. Gratch, W. Swartout, D. Traum, S. Marsella, and D. Piepol, “Building Interactive Virtual Humans for Training Environments,” *Simulation and Education Conference*, 2007.

[34] P. Kenny, T. Parsons, J. Gratch, A. Leuski, and A. Rizzo, “Virtual Patients for Clinical Therapist Skills Training,” *Proceedings of the 7th International Conference on Intelligent Virtual Agents (IVA'07)*, pp. 197-210.

[35] S. Kopp, “Surface Realization of Multimodal Output from XML representations in MURML,” *Representations for Multimodal Generation Workshop*, 2005.

[36] S. Kopp, L. Gesellensetter, N. Krämer, and I. Wachsmuth, “A conversational agent as museum guide-Design and evaluation of a real-world application,” *Proceedings of the 5th International Conference on Intelligent Virtual Agents (IVA'05)*, 2005, pp. 329-343.

[37] S. Kopp, B. Jung, N. Lessmann, and I. Wachsmuth, “Max–a multimodal assistant in virtual reality construction,” *KI-Künstliche Intelligenz*, vol. 4, 2003, pp. 11-17.

[38] S. Kopp, B. Krenn, S. Marsella, A.N. Marshall, C. Pelachaud, H. Pirker, K.R. Thórisson, and H. Vilhjálmsson, “Towards a Common Framework for Multimodal Generation: The Behavior Markup Language,” *Proceedings of the 6th International Conference on Intelligent Virtual Agents (IVA'06), Marina del Rey, CA*, 2006.

[39] S. Kopp, T. Sowa, and I. Wachsmuth, “Imitation games with an artificial agent: From mimicking to understanding shape-related iconic gestures,” *Gesture-Based Communication in Human-Computer Interaction, 5th International Gesture Workshop, Genova, Italy, April*, 2003, pp. 15-17.

[40] S. Kopp, P. Tepper, and J. Cassell, “Towards integrated microplanning of language and iconic gesture for multimodal output,” *Proceedings of the 6th international conference on Multimodal interfaces*, 2004, pp. 97-104.

[41] S. Kopp and I. Wachsmuth, “Synthesizing multimodal utterances for conversational agents,” *Computer Animation and Virtual Worlds*, vol. 15, 2004, pp. 39-52.

[42] A. Kranstedt, S. Kopp, and I. Wachsmuth, “MURML: A Multimodal Utterance Representation Markup Language for Conversational Agents,” *Proceedings of the AAMAS Workshop on “Embodied conversational agents–Let’s specify and evaluate them*, 2002.

[43] S. Kshirsagar, N. Magnenat-Thalmann, A. Guye-Vuillème, and K. Kamyab, “Avatar Markup Language,” *Proceedings of the workshop on Virtual environments 2002*, 2002, pp. 169-177.

[44] J. Lee and S. Marsella, “Nonverbal Behavior Generator for Embodied Conversational Agents,” *Proceedings of the 6th International Conference on Intelligent Virtual Agents (IVA'06), Marina del Rey, CA,* vol. 4133, 2006, pp. 243–255.

[45] A. Marriott, “VHML–Virtual Human Markup Language,” *Online), http://www. interface. domputing. edu. au/document/VHML*.

[46] S. Marsella and H. Vilhjalmsson, “Social Performance Framework.”

[47] D. Maulsby, S. Greenberg, and R. Mander, “Prototyping an intelligent agent through Wizard of Oz,” *Proceedings of the INTERACT '93 and CHI '93 conference on Human factors in computing systems*, Amsterdam, The Netherlands: ACM, 1993, pp. 277-284; http://portal.acm.org/citation.cfm?id=169059.169215.

[48] R. McDonnell, S. Dobbyn, and C. O’Sullivan, “LOD Human Representations: A Comparative Study,” *Proceedings of the First International Workshop on Crowd Simulation*, 2005, pp. 101–115.

[49] D. McNeill, *Hand and Mind: What Gestures Reveal about Thought*, University of ChicagoPress, 1996.

[50] H.P. Nii, *Blackboard Systems*, Stanford University, 1986; http://portal.acm.org/citation.cfm?id=892387.

[51] M. Nischt, H. Prendinger, E. Andre, and M. Ishizuka, “MPML3D: a reactive framework for the Multimodal Presentation Markup Language,” *IVA*, 2006, pp. 218–229.

[52] C. O'Sullivan, J. Cassell, H. Vilhjalmsson, J. Dingliana, S. Dobbyn, B. McNamee, C. Peters, and T. Giang, “Levels of detail for crowds and groups,” *Computer Graphics Forum*, vol. 21, 2002, pp. 733-741.

[53] F.C.N. Pereira and T. Ebrahimi, *The MPEG-4 Book*, Prentice Hall PTR, 2002.

[54] P. Piwek, B. Krenn, M. Schroeder, M. Grice, S. Baumann, and H. Pirker, “RRL: A Rich Representation Language for the Description of Agent Behaviour in NECA,” *Arxiv preprint cs.MM/0410022*, 2004.

[55] I. POGGI, C. PELACHAUD, F. DE ROSIS, V. CAROFIGLIO, and B. DE CAROLIS, “GRETA. A BELIEVABLE EMBODIED CONVERSATIONAL AGENT,” *Multimodal Intelligent Information Presentation*, 2005.

[56] H. Prendinger, S. Descamps, and M. Ishizuka, “MPML: a markup language for controlling the behavior of life-like characters,” *Journal of Visual Languages and Computing*, vol. 15, 2004, pp. 183-203.

[57] S. Prillwitz, *HamNoSys*, Signum-Verl., 1989.

[58] S.J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, Prentice Hall, 2003.

[59] Z.M. Ruttkay and H. Noot, “Graceful Degradation of Hand Gestures,” *Eurographics/SIGGRAPH Symposium on Computer Animation*.

[60] K. Ryokai, C. Vaucelle, and J. Cassell, *Literacy Learning by Storytelling with a Virtual Peer*, 2002; http://citeseer.ist.psu.edu/article/ryokai02literacy.html.

[61] J. Smith, “GrandChair: Conversational Collection of Grandparents' Stories,” 2000.

[62] M. Stone, C. Doran, B. Webber, T. Bleam, and M. Palmer, “Microplanning with Communicative Intentions: The SPUD System,” *Computational Intelligence*, vol. 19, 2003, pp. 311-381.

[63] K. Striegnitz, P. Tepper, A. Lovett, and J. Cassell, “Knowledge representation for generating locating gestures in route directions,” *the Proceedings of the Workshop on Spatial Language and Dialogue, Delmenhorst, Germany, October*.

[64] A. Tartaro and J. Cassell, “Authorable Virtual Peers for Autism Spectrum Disorders,” *Combined Workshop on Language-Enabled Educational Technology and Development and Evaluation fo Robust Spoken Dialogue Systems at the 17th European Conference on Artificial Intelligence (ECAI06),(Riva del Garda, Italy, 2006).*

[65] P. Tepper, S. Kopp, and J. Cassell, “Content in Context: Generating Language and Iconic Gesture without a Gestionary,” *Proceedings of the Workshop on Balanced Perception and Action in ECAs at Automous Agents and Multiagent Systems (AAMAS)*, 2004.

[66] M. Thiebaux, A. Marshall, S. Marsella, E. Fast, A. Hill, M. Kallmann, P. Kenny, and J. Lee, “SmartBody: Behavior Realization for Embodied Conversational Agents,” *Proceedings of the 7th International Conference on Intelligent Virtual Agents (IVA'07)*, 2007.

[67] K.R. Thorisson, “Layered modular action control for communicative humanoids,” *Computer Animation*, vol. 97, 1997, pp. 134–143.

[68] D. Traum, A. Roque, A. Leuski, P. Georgiou, J. Gerten, B. Martinovski, S. Narayanan, S. Robinson, and A. Vaswani, “Hassan: A Virtual Human for Tactical Questioning,” *Proceedings of the 8th SIGdial workshop on Discourse and Dialogue, S. Keizer, B. Bunt & T. Paek (eds), Antwerp, Belgium*, 2007, pp. 75–78.

[69] H. Vilhjalmsson, N. Cantelmo, J. Cassell, N.E. Chafai, M. Kipp, S. Kopp, M. Mancini, S. Marsella, A.N. Marshall, and C. Pelachaud, “The Behavior Markup Language: Recent Developments and Challenges,” *Proceedings of the 7th International Conference on Intelligent Virtual Agents (IVA'07)*, vol. 4722, pp. 90-111.

[70] M. Wooldridge, “Agent-based software engineering,” *IEE Proceedings Software Engineering*, vol. 144, 1997, p. 26—37.

Appendices

Appendix A: Original NUMACK Skeleton

```
#-----------------------------------------------------------------------------------
FigureTypeName: NUMACK
#-----------------------------------------------------------------------------------
# This file defines the articulated structure of the figure,
# ie. its kinematic skeleton. It has 4 sections defining
# the joints of the limbs, the links, the geometries attached
# to the joint, and the geometries for the links.
#-----------------------------------------------------------------------------------
# Stefan Kopp; June 2, 2004



#-----------------------------------------------------------------------------------
# section 1: definition of the limbs
#
# Each limb is defined as follows: The first line gives the
# type, name, abbreviation, number of joints, and start joint.
# Then, all individual joints are defined as follows:
#   - the type of the joint
#   - the name of the joint
#   - the number of children (joints)
#   - the rotational degrees of freedom, each given by
#     - joint angle to be set
#     - lower and upper limits
#     - two numbers for DOF-specific parameters
#
Limbs:  23

# spine
Torso_Limb Torso t_ 6 1
Multi_Joint HumanoidRoot 3  # root of the articulation
Euler_JointXYZ vl5  1  0 -10 10 1 1  0 -15 40 1 1  0 0 0 1 1 #roll/tilt in lumbar region
Euler_JointXYZ vl1  1  0 -10 10 1 1  0 -10 30 1 1  0 -10 10 1 1 #roll/tilt in lumbar region
Euler_JointXYZ vt6  1  0 -20 20 1 1  0 -10 10 1 1  0 -40 40 1 1 #roll/tilt/tors.at thoracic
base
Euler_JointXYZ vt1  1  0 -20 20 1 1  0 0 0 1 1  0 -20 20 1 1 #no tilt in thoracic
Multi_Joint vc7     3  # connection of sternoclavicular, spine, and neck

# left arm
Arm_Limb LArm l_ 4 0
Euler_JointXYZ l_clavicle 1  0 -25 35 1 1   0 -20 20 1 1   0 0 0 1 1#abd/add, rotate
Euler_JointXYZ l_shoulder 1  0 -180 180 1 1  0 -180 180 1 1  0 -180 180 1 1#add/abd, flex,
twist
Euler_JointXYZ l_elbow   1  0 0 0 1 1   0 0 159 5 1    0 0 0 1 1 #flex, twist(supin/pron)
Euler_JointXYZ l_wrist   5  0 -78 94 1 1   0 -90 90 1 1   0 -80 90 1 1

# left thumb
StdLimb LThumb lh_t_ 5 0
Euler_JointRPY lh_t_joint0 1  0 0 0 1 1  45 45 45 1 1   0 0 110 1 1
Euler_JointXYZ lh_t_joint1 1  0 0 0 1 1  0 -35 10 1 1   0 0 0 1 1
Euler_JointXYZ lh_t_joint2 1  0 -30 0 1 1  0 0 0 1 1   0 0 0 1 1
Euler_JointXYZ lh_t_joint3 1  0 -90 0 1 1  0 0 0 1 1   0 0 0 1 1
Manipulator_Joint lh_t_effector 0

# left poiting finger
StdLimb LPointing lh_p_ 4 0
Euler_JointXYZ lh_p_joint1 1  0 -90 0 1 1  0 -20 20 1 1   0 0 25 1 1
Euler_JointXYZ lh_p_joint2 1  0 -100 0 1 1  0 0 0 1 1   0 0 0 1 1
Euler_JointXYZ lh_p_joint3 1  0 -90 0 1 1  0 0 0 1 1   0 0 0 1 1
Manipulator_Joint lh_p_effector 0

# left middle finger
StdLimb LMiddle lh_m_ 4 0
Euler_JointXYZ lh_m_joint1 1  0 -90 0 1 1  0 -20 20 1 1   0 0 12 1 1
Euler_JointXYZ lh_m_joint2 1  0 -100 0 1 1  0 0 0 1 1   0 0 0 1 1
Euler_JointXYZ lh_m_joint3 1  0 -90 0 1 1  0 0 0 1 1   0 0 0 1 1
Manipulator_Joint lh_m_effector 0

# left ring finger
StdLimb LRing lh_r_ 4 0
```

```
Euler_JointXYZ lh_r_joint1 1  0 -90 0 1 1  0 -20 20 1 1   0 -12 0 1 1
Euler_JointXYZ lh_r_joint2 1  0 -100 0 1 1  0 0 0 1 1   0 0 0 1 1
Euler_JointXYZ lh_r_joint3 1  0 -90 0 1 1  0 0 0 1 1   0 0 0 1 1
Manipulator_Joint lh_r_effector 0

# left pinky
StdLimb LLittle lh_l_ 4 0
Euler_JointXYZ lh_l_joint1 1  0 -90 0 1 1  0 -20 20 1 1   0 -25 0 1 1
Euler_JointXYZ lh_l_joint2 1  0 -100 0 1 1  0 0 0 1 1   0 0 0 1 1
Euler_JointXYZ lh_l_joint3 1  0 -90 0 1 1  0 0 0 1 1   0 0 0 1 1
Manipulator_Joint lh_l_effector 0

# right arm
Arm_Limb RArm r_ 4 0
Euler_JointXYZ r_clavicle 1  0 -25 35 1 1   0 -20 20 1 1   0 0 0 1 1 #add/abd, rotate
Euler_JointXYZ r_shoulder 1  0 -180 180 1 1  0 -180 190 1 1  0 -180 180 1 1 #abd/add, flex,
twist
Euler_JointXYZ r_elbow    1  0 0 0 1 1     0 -5 159 1 1   0 0 0 1 1 #flex, twist(pron/supin)
Euler_JointXYZ  r_wrist      5   0  -94  78  1  1   0  -90  90  1  1   0  -90  80  1  1  #flex/ext,  pivot
(abd/add)

# right thumb
StdLimb RThumb rh_t_ 5 0
Euler_JointRPY rh_t_joint0 1  0 0 0 1 1  45 45 45 1 1   0 -110 0 1 1
Euler_JointXYZ rh_t_joint1 1  0 0 0 1 1  0 -35 10 1 1   0 0 0 1 1
Euler_JointXYZ rh_t_joint2 1  0 0 30 1 1  0 0 0 1 1   0 0 0 1 1
Euler_JointXYZ rh_t_joint3 1  0 0 90 1 1  0 0 0 1 1   0 0 0 1 1
Manipulator_Joint rh_t_effector 0

# right pointing finger
StdLimb RPointing rh_p_ 4 0
Euler_JointXYZ rh_p_joint1 1  0 0 90 1 1  0 -20 20 1 1   0 -25 0 1 1
Euler_JointXYZ rh_p_joint2 1  0 0 100 1 1  0 0 0 1 1   0 0 0 1 1
Euler_JointXYZ rh_p_joint3 1  0 0 90 1 1  0 0 0 1 1   0 0 0 1 1
Manipulator_Joint rh_p_effector 0

# right middle finger
StdLimb RMiddle rh_m_ 4 0
Euler_JointXYZ rh_m_joint1 1  0 0 90 1 1  0 -20 20 1 1   0 -12 0 1 1
Euler_JointXYZ rh_m_joint2 1  0 0 100 1 1  0 0 0 1 1   0 0 0 1 1
Euler_JointXYZ rh_m_joint3 1  0 0 90 1 1  0 0 0 1 1   0 0 0 1 1
Manipulator_Joint rh_m_effector 0

# right ring finger
StdLimb RRing rh_r_ 4 0
Euler_JointXYZ rh_r_joint1 1  0 0 90 1 1  0 -20 20 1 1   0 0 12 1 1
Euler_JointXYZ rh_r_joint2 1  0 0 100 1 1  0 0 0 1 1   0 0 0 1 1
Euler_JointXYZ rh_r_joint3 1  0 0 90 1 1  0 0 0 1 1   0 0 0 1 1
Manipulator_Joint rh_r_effector 0

# right pinky
StdLimb RLittle rh_l_ 4 0
Euler_JointXYZ rh_l_joint1 1  0 0 90 1 1  0 -20 20 1 1   0 0 25 1 1
Euler_JointXYZ rh_l_joint2 1  0 0 100 1 1  0 0 0 1 1   0 0 0 1 1
Euler_JointXYZ rh_l_joint3 1  0 0 90 1 1  0 0 0 1 1   0 0 0 1 1
Manipulator_Joint rh_l_effector 0

# left legg
Leg_Limb LLegg ll_ 5 0
Euler_JointXYZ ll_sacroiliac 1  0 -28 28 1 1  0 -30 30 1 1  0 0 0 1 1 #pelvic roll,tilt,no
tors
Euler_JointXYZ   ll_hip   1   -90   -141   37   1   1     0   -40   110   1   1     0   -51   56   1
1#add/abd,ext/flex,twist(lat/med)
Euler_JointXYZ ll_knee 1 0 0 0 1 1  0 -145 0 1 1  0 0 0 1 1 #knee: flexion
Euler_JointXYZ ll_ankle  1 0 0 0 1 1  0 -79 24 1 1  0 0 0 1 1 #ankle: flex(plantar)/ext(dorsi)
Manipulator_Joint ll_foot 0

# right legg
Leg_Limb RLegg rl_ 5 0
Euler_JointXYZ rl_sacroiliac 1  0 -28 28 1 1  0 -30 30 1 1  0 0 0 1 1 #pelvic roll,tilt,no
tors
Euler_JointXYZ   rl_hip   1   90   37   141   1   1     0   -40   110   1   1     0   -56   51   1   1
#abd/add,ext/flex,twist(med/lat)
Euler_JointXYZ rl_knee  1  0 0 0 1 1  0 -145 0 1 1  0 0 0 1 1 #flexion
Euler_JointXYZ rl_ankle 1  0 0 0 1 1  0 -79 24 1 1  0 0 0 1 1 #flex.(plantar)/ext.(dorsi)
Manipulator_Joint rl_foot 0
```

```
# neck/head
StdLimb Neck n_ 3 0
Euler_JointXYZ vc4 1  0 -20 20 1 1  0 -35 30 1 1  0 -20 20 1 1 # roll(lateral), ext/flex,
torsion
Euler_JointXYZ vc2 1  0 -40 40 1 1  0 -65 40 1 1  0 -77 77 1 1 # dto.
Euler_JointXYZ skullbase 7   0 -12 12 1 1  0 -10 15 1 1  0 0 0 1 1

# left eye
StdLimb LEye le_ 1 0
Euler_JointXYZ le_eye 0 0 -80 80 1 1   0 -80 80 1 1   0 0 0 1 1

# right eye
StdLimb REye re_ 1 0
Euler_JointXYZ re_eye 0 0 -180 180 1 1   0 -180 180 1 1   0 0 0 1 1

# jaw
StdLimb Jaw j_ 1 0
Euler_JointXYZ jaw  0 0 0 0 1 1   0 0 15 1 1   0 0 0 1 1

# left eyebrow
StdLimb LEyeBrow leb_ 1 0
Euler_JointXYZ l_eyebrow  0 0 0 0 1 1   0 -12 10 1 1   0 0 0 1 1

# left eyebrow
StdLimb REyeBrow reb_ 1 0
Euler_JointXYZ r_eyebrow  0 0 0 0 1 1   0 -12 10 1 1   0 0 0 1 1

# left eyelid
StdLimb LEyeLid lel_ 1 0
Euler_JointXYZ l_eyelid  0 0 0 0 1 1   0 0 85 1 1   0 0 0 1 1

# right eyelid
StdLimb REyeLid rel_ 1 0
Euler_JointXYZ r_eyelid  0 0 0 0 1 1   0 0 85 1 1   0 0 0 1 1




#----------------------------------------------------------------------------------
# section 2: definition of the links
#
# each link is defined by:
#     - the type of the link
#     - the name of the link
#     - name of the father joint
#     - name of the child joint
#     - index of the link for father joint
#     - link parameters like length and possible rotation(s)
#
# Note: AlignLink rotation = Rz*Ry*Rx
#
Links: 75

# root joint
Dummy_Link HumanoidRoot__vl5 HumanoidRoot vl5 0
Align_Link HumanoidRoot__ll_sacroiliac HumanoidRoot ll_sacroiliac 1 0 0 -90
Align_Link HumanoidRoot__rl_sacroiliac HumanoidRoot rl_sacroiliac 2 0 0 90

# spine
Euler_Link vl5__vl1 vl5 vl1 0 0 0 100
Euler_Link vl1__vt6 vl1 vt6 0 0 0 200
Euler_Link vt6__vt1 vt6 vt1 0 0 0 200
Euler_Link vt1__vc7 vt1 vc7 0 0 0 100

# vc7
Align_Link vc7__l_clavicle vc7 l_clavicle 0 0 0 -90
Align_Link vc7__r_clavicle vc7 r_clavicle 1 0 0  90
Align_Link vc7__vc4 vc7 vc4 2 0 0 0

# neck/head
Euler_Link vc4__vc2 vc4 vc2            0  0 0 60
Align_Link vc2__skullbase vc2 skullbase 0  0 0 0  0 0 40
Align_Link skullbase__le_eye skullbase le_eye       0  0 90 0  45 40 70
Align_Link skullbase__re_eye skullbase re_eye       1  0 90 0  45 -40 70
Align_Link skullbase__l_eyebrow skullbase l_eyebrow 2  0 0 0   60  40 130
Align_Link skullbase__r_eyebrow skullbase r_eyebrow 3  0 0 0   60 -40 130
Align_Link skullbase__l_eyelid skullbase l_eyelid   5  0 -45 0   50  40 65
Align_Link skullbase__r_eyelid skullbase r_eyelid   4  0 -45 0   50 -40 65
Align_Link skullbase__jaw skullbase jaw             6  0 0 0   0 0 70

# left arm
Align_Link l_clavicle__l_shoulder l_clavicle l_shoulder 0 0 0 -90 0 0 219
Euler_Link l_shoulder__l_elbow    l_shoulder l_elbow    0 0 0 315 #334
Euler_Link l_elbow__l_wrist       l_elbow    l_wrist    0 0 0 255 #288

# right arm
Align_Link r_clavicle__r_shoulder r_clavicle r_shoulder 0 0 0 90 0 0 219
Euler_Link r_shoulder__r_elbow  r_shoulder r_elbow    0 0 0 315 #334
Euler_Link r_elbow__r_wrist     r_elbow    r_wrist    0 0 0 255 #288

# left legg
Euler_Link ll_sacroiliac__ll_hip ll_sacroiliac ll_hip 0 0 0 105
Euler_Link ll_hip__ll_knee   ll_hip   ll_knee  0 0 0 434
Euler_Link ll_knee__ll_ankle ll_knee  ll_ankle 0 0 0 368
Euler_Link ll_ankle__ll_foot ll_ankle  ll_foot 0 0 0 0

# right legg
Euler_Link rl_sacroiliac__rl_hip rl_sacroiliac rl_hip 0 0 0 105
Euler_Link rl_hip__rl_knee   rl_hip   rl_knee  0 0 0 434
Euler_Link rl_knee__rl_ankle rl_knee  rl_ankle 0 0 0 368
Euler_Link rl_ankle__rl_foot rl_ankle rl_foot  0 0 0 0

# left hand
Align_Link l_wrist__lh_thumb   l_wrist lh_t_joint0 0 0 0 0 38 0 23
Align_Link lh_t_joint0__lh_t_joint1   lh_t_joint0 lh_t_joint1 0 0 0 0 0 0 5
Euler_Link l_wrist__lh_pointing l_wrist lh_p_joint1 1  36 0 91
Euler_Link l_wrist__lh_middle   l_wrist lh_m_joint1 2  12 0 91
Euler_Link l_wrist__lh_ring     l_wrist lh_r_joint1 3 -12 0 84.5
Euler_Link l_wrist__lh_little   l_wrist lh_l_joint1 4 -34 0 75
Euler_Link lh_t_joint1__lh_t_joint2   lh_t_joint1 lh_t_joint2    0 0 0 39
Euler_Link lh_t_joint2__lh_t_joint3   lh_t_joint2 lh_t_joint3    0 0 0 32.5
Euler_Link lh_t_joint3__lh_t_effector lh_t_joint3 lh_t_effector  0 0 0 23
Euler_Link lh_p_joint1__lh_p_joint2   lh_p_joint1 lh_p_joint2    0 0 0 39
Euler_Link lh_p_joint2__lh_p_joint3   lh_p_joint2 lh_p_joint3    0 0 0 26
Euler_Link lh_p_joint3__lh_p_effector lh_p_joint3 lh_p_effector  0 0 0 19.5
Euler_Link lh_m_joint1__lh_m_joint2   lh_m_joint1 lh_m_joint2    0 0 0 42
Euler_Link lh_m_joint2__lh_m_joint3   lh_m_joint2 lh_m_joint3    0 0 0 32.5
Euler_Link lh_m_joint3__lh_m_effector lh_m_joint3 lh_m_effector  0 0 0 23
Euler_Link lh_r_joint1__lh_r_joint2   lh_r_joint1 lh_r_joint2    0 0 0 42
Euler_Link lh_r_joint2__lh_r_joint3   lh_r_joint2 lh_r_joint3    0 0 0 26
Euler_Link lh_r_joint3__lh_r_effector lh_r_joint3 lh_r_effector  0 0 0 23
Euler_Link lh_l_joint1__lh_l_joint2   lh_l_joint1 lh_l_joint2    0 0 0 32.5
Euler_Link lh_l_joint2__lh_l_joint3   lh_l_joint2 lh_l_joint3    0 0 0 23
Euler_Link lh_l_joint3__lh_l_effector lh_l_joint3 lh_l_effector  0 0 0 16.5

# right hand
Align_Link r_wrist__rh_thumb   r_wrist rh_t_joint0 0 0 0 0 38 0 23
Align_Link rh_t_joint0__rh_t_joint1   rh_t_joint0 rh_t_joint1 0 0 0 0 0 0 5
Euler_Link r_wrist__rh_pointing r_wrist rh_p_joint1 1  36 0 91
Euler_Link r_wrist__rh_middle   r_wrist rh_m_joint1 2  12 0 91
Euler_Link r_wrist__rh_ring     r_wrist rh_r_joint1 3 -12 0 84.5
Euler_Link l_wrist__rh_little   r_wrist rh_l_joint1 4 -34 0 75
Euler_Link rh_t_joint1__rh_t_joint2   rh_t_joint1 rh_t_joint2    0 0 0 39
Euler_Link rh_t_joint2__rh_t_joint3   rh_t_joint2 rh_t_joint3    0 0 0 32.5
Euler_Link rh_t_joint3__rh_t_effector rh_t_joint3 rh_t_effector  0 0 0 23
Euler_Link rh_p_joint1__rh_p_joint2   rh_p_joint1 rh_p_joint2    0 0 0 39
Euler_Link rh_p_joint2__rh_p_joint3   rh_p_joint2 rh_p_joint3    0 0 0 26
Euler_Link rh_p_joint3__rh_p_effector rh_p_joint3 rh_p_effector  0 0 0 19.5
Euler_Link rh_m_joint1__rh_m_joint2   rh_m_joint1 rh_m_joint2    0 0 0 42
Euler_Link rh_m_joint2__rh_m_joint3   rh_m_joint2 rh_m_joint3    0 0 0 32.5
Euler_Link rh_m_joint3__rh_m_effector rh_m_joint3 rh_m_effector  0 0 0 23
Euler_Link rh_r_joint1__rh_r_joint2   rh_r_joint1 rh_r_joint2    0 0 0 42
Euler_Link rh_r_joint2__rh_r_joint3   rh_r_joint2 rh_r_joint3    0 0 0 26
Euler_Link rh_r_joint3__rh_r_effector rh_r_joint3 rh_r_effector  0 0 0 23
Euler_Link rh_l_joint1__rh_l_joint2   rh_l_joint1 rh_l_joint2    0 0 0 32.5
Euler_Link rh_l_joint2__rh_l_joint3   rh_l_joint2 rh_l_joint3    0 0 0 23
Euler_Link rh_l_joint3__rh_l_effector rh_l_joint3 rh_l_effector  0 0 0 16.5


#-------------------------------------------------------------------------------------
# section 4: specification of the graphical joint objects
#
# specification consists of
#   - the name of the joint to be displayed
#   - an identifier denoting the type of its visual object (e.g. PfObject)
#   - six digits giving the objects translation & rotation
#   - three optional digits giving the objects scaling along its principal axes
#
Joint_VisRep: 22

# spine:
#vl5        PfObject sphere_fix 0 0 0 0 0 0 6 6 6
#vl1        PfObject sphere_fix 0 0 0 0 0 0 7 7 7
#vt6        PfObject sphere_fix 0 0 0 0 0 0 6 6 6
#vt1        PfObject sphere_fix 0 0 0 0 0 0 5 5 5
#vc7        PfObject sphere_fix 0 0 0 0 0 0 4 4 4
#vc4        PfObject sphere_fix 0 0 0 0 0 0 4 4 4
# head:
skullbase  PfObject head 0 0 -0.05 90 0 0 800 800 800locate ivview
le_eye     PfObject AugeGanz_noscale 0 0 0 0 0 90 1.3 1.3 1.3
re_eye     PfObject AugeGanz_noscale 0 0 0 0 0 90 1.3 1.3 1.3
l_eyebrow  PfObject left_eyebrow  0.23 0 -0.05  90 0 0 200 600 700
r_eyebrow  PfObject right_eyebrow 0.23 0 -0.05  90 0 0 200 600 700
r_eyelid   PfObject right_eyelid  0 0.03 -0.005  90 0 0 450 650 450
l_eyelid   PfObject left_eyelid   0 0.03 -0.005  90 0 0 450 650 450
jaw        PfObject jaw 1000 2200 -1900 180 180 0 0.036 0.04 0.03
# right arm:
r_shoulder PfObject right_shoulderbone 0.06 -0.01 0 -90 0 180 475 374 475
r_elbow    PfObject right_elbowbone 0 -0.029 -0.07 -90 0 180 500 700 600
r_wrist    PfObject right_palm 0 0.03 0.038  -90 0 180  600 600 800
# left arm:
l_shoulder PfObject left_shoulderbone 0.06 0.02 0 -90 0 180 475 374 475
l_elbow    PfObject left_elbowbone 0 0.035 -0.07 -90 0 180 500 700 600
l_wrist    PfObject left_palm 0 -0.03 0.038  -90 0 180  600 600 800
# left leg:
ll_hip     PfObject left_hipconnect -0.03 0 0.15 -90 0 180 500 500 500
ll_knee    PfObject left_kneebone 0.1 0 -0.1 -90 0 180 500 500 500
ll_ankle   PfObject left_anklebone -0.01 0.02 -0.12 -90 0 180 500 600 800
ll_foot    PfObject left_foot 0.07 0.13 -0.04 -90 0 180 650 600 800
# right leg:
rl_hip     PfObject right_hipconnect 0 0 0.25 -90 0 180 500 500 500
rl_knee    PfObject right_kneebone 0.0 0 -0.045 -90 0 180 500 500 500
rl_ankle   PfObject right_anklebone -0.01 -0.02 -0.12 -90 0 180 500 600 800
rl_foot    PfObject right_foot 0.07 -0.13 -0.04 -90 0 180 650 600 800


#-------------------------------------------------------------------------------------
# section 5: specification of the graphical link objects
#
# same specification as for joint visual objects
Link_VisRep: 43

vl5__vl1               PfObject hips 0.2 0 0.05 -90 0 0 550 690 650
vl1__vt6               PfObject stomach_joint 0.12 0 0.05 -90 0 0 480 650 600
vt6__vt1               PfObject stomach 5 0 -40 -90 0 0 0.92 1.05 1.2
vt1__vc7               PfObject torso -0.19 0 0.0023 90 0 0 500 625 725
vc4__vc2               PfObject neck 0 0 0.18 90 0 0 500 600 500
l_shoulder__l_elbow    PfObject left_upper_arm -.02 0.078 0.17 -90 0 180 600 450 805
l_elbow__l_wrist       PfObject left_forearm_wrist 0 -0.1 0.09 -90 0 180 600 600 1000
#r_clavicle__r_shoulder PfObject r_clavicle__r_shoulder_smooth_tex 1 0 0 90 90 -90 1 1 1
r_shoulder__r_elbow    PfObject right_upper_arm -.02 .045 0.35 -90 0 180 600 450 805
r_elbow__r_wrist       PfObject right_forearm_wrist -0.01 0.1 0.09 -90 0 180 600 600 1000
#ll_sacroiliac__ll_hip  PfObject cylinder 0 0 10 0 0 90 1 1 0
ll_hip__ll_knee        PfObject left_upper_leg   -0.09 0 0.265 -90 0 180 600 600 800
ll_knee__ll_ankle      PfObject left_lower_leg   0 0 0.32 -90 0 180 600 600 900
#rl_sacroiliac__rl_hip  PfObject cylinder 0 0 10 0 0 90 1 1 0
rl_hip__rl_knee        PfObject right_upper_leg   0 0 0.46 -90 0 180 600 600 800
rl_knee__rl_ankle      PfObject right_lower_leg   0 0 0.32 -90 0 180 600 600 900
lh_t_joint1__lh_t_joint2   PfObject finger_element 0 0 22 0 0 90 1.3 1.3 1.4
lh_t_joint2__lh_t_joint3   PfObject finger_element 0 0 19 0 0 90 1.0 1.0 1.32
lh_t_joint3__lh_t_effector PfObject finger_element 0 0 18 0 0 90 0.8 0.8 0.8
lh_p_joint1__lh_p_joint2   PfObject finger_element 0 0 23 0 0 90 1.0 1.0 1.48
lh_p_joint2__lh_p_joint3   PfObject finger_element 0 0 21 0 0 90 0.9 0.9 1.1
lh_p_joint3__lh_p_effector PfObject finger_element 0 0 20 0 0 90 0.8 0.8 0.9
lh_m_joint1__lh_m_joint2   PfObject finger_element 0 0 24 0 0 90 1.0 1.0 1.6
lh_m_joint2__lh_m_joint3   PfObject finger_element 0 0 22 0 0 90 0.9 0.9 1.3
lh_m_joint3__lh_m_effector PfObject finger_element 0 0 20 0 0 90 0.8 0.8 0.9
lh_r_joint1__lh_r_joint2   PfObject finger_element 0 0 24 0 0 90 1.0 1.0 1.6
lh_r_joint2__lh_r_joint3   PfObject finger_element 0 0 22 0 0 90 0.9 0.9 1.1
lh_r_joint3__lh_r_effector PfObject finger_element 0 0 20 0 0 90 0.8 0.8 0.9
```

```
lh_l_joint1__lh_l_joint2   PfObject finger_element 0 0 22 0 0 90 0.9 0.9 1.35
lh_l_joint2__lh_l_joint3   PfObject finger_element 0 0 20 0 0 90 0.8 0.8 0.95
lh_l_joint3__lh_l_effector PfObject finger_element 0 0 18 0 0 90 0.7 0.7 0.7
rh_t_joint1__rh_t_joint2   PfObject finger_element 0 0 22 0 0 90 1.3 1.3 1.4
rh_t_joint2__rh_t_joint3   PfObject finger_element 0 0 19 0 0 90 1.0 1.0 1.32
rh_t_joint3__rh_t_effector PfObject finger_element 0 0 18 0 0 90 0.8 0.8 0.8
rh_p_joint1__rh_p_joint2   PfObject finger_element 0 0 23 0 0 90 1.0 1.0 1.48
rh_p_joint2__rh_p_joint3   PfObject finger_element 0 0 21 0 0 90 0.9 0.9 1.1
rh_p_joint3__rh_p_effector PfObject finger_element 0 0 20 0 0 90 0.8 0.8 0.9
rh_m_joint1__rh_m_joint2   PfObject finger_element 0 0 24 0 0 90 1.0 1.0 1.6
rh_m_joint2__rh_m_joint3   PfObject finger_element 0 0 22 0 0 90 0.9 0.9 1.3
rh_m_joint3__rh_m_effector PfObject finger_element 0 0 20 0 0 90 0.8 0.8 0.9
rh_r_joint1__rh_r_joint2   PfObject finger_element 0 0 24 0 0 90 1.0 1.0 1.6
rh_r_joint2__rh_r_joint3   PfObject finger_element 0 0 22 0 0 90 0.9 0.9 1.1
rh_r_joint3__rh_r_effector PfObject finger_element 0 0 20 0 0 90 0.8 0.8 0.9
rh_l_joint1__rh_l_joint2   PfObject finger_element 0 0 22 0 0 90 0.9 0.9 1.35
rh_l_joint2__rh_l_joint3   PfObject finger_element 0 0 20 0 0 90 0.8 0.8 0.95
rh_l_joint3__rh_l_effector PfObject finger_element 0 0 18 0 0 90 0.7 0.7 0.7

# end of figure file
```

Appendix B: Partial Scenegraph of the NUMACK Skeleton

```
#Inventor V2.0 ascii
DEF HumanoidRoot Separator
{#Multi_Joint
RotationXYZ { axis Y angle -1.57 }
RotationXYZ { axis X angle -1.57 }
  MatrixTransform
  {#Rotation x:0.0 y: 0.0 z: 0.0(Multi_Joint)
    matrix 1.0 0.0 0.0 0.0
    0.0 1.0 0.0 0.0
    0.0 0.0 1.0 0.0
    0.0 0.0 0.0 1.0
  }
  DEF HumanoidRoot__vl5 Separator
  {#Dummy_Link
    MatrixTransform
    {#Rotation x:0.0 y: 0.0 z: 0.0(Dummy_Link)
      matrix 1.0 0.0 0.0 0.0
      0.0 1.0 0.0 0.0
      0.0 0.0 1.0 0.0
      0.0 0.0 0.0 1.0
    }
……….
……….
  DEF HumanoidRoot__ll_sacroiliac Separator
  {#Align_Link
    MatrixTransform
    {#Rotation x:-90.0 y: 0.0 z: 0.0(Align_Link)
      matrix 1.0 0.0 0.0 0.0
      0.0 6.123233995736766E-17 -1.0 0.0
      0.0 1.0 6.123233995736766E-17 0.0
      0.0 0.0 0.0 1.0
    }
    DEF ll_sacroiliac Separator
    {#Euler_JointXYZ
      MatrixTransform
      {#Rotation x:0.0 y: 0.0 z: 0.0(Euler_JointXYZ)
        matrix 1.0 0.0 0.0 0.0
        0.0 1.0 0.0 0.0
        0.0 0.0 1.0 0.0
        0.0 0.0 0.0 1.0
      }
      DEF ll_sacroiliac__ll_hip Separator
      {#Euler_Link
        MatrixTransform
        {#Rotation x:0.0 y: 0.0 z: 0.0(Euler_Link)
          matrix 1.0 0.0 0.0 0.0
          0.0 1.0 0.0 0.0
          0.0 0.0 1.0 0.0
          0.0 0.0 0.0 1.0
        }
        DEF ll_hip Separator
        {#Euler_JointXYZ
          MatrixTransform
          {#Rotation x:-90.0 y: 0.0 z: 0.0(Euler_JointXYZ)
            matrix 1.0 0.0 0.0 0.0
            0.0 6.123233995736766E-17 -1.0 0.0
            0.0 1.0 6.123233995736766E-17 0.0
            0.0 0.0 105.0 1.0
          }
          DEF ll_hip__ll_knee Separator
          {#Euler_Link
            MatrixTransform
            {#Rotation x:0.0 y: 0.0 z: 0.0(Euler_Link)
              matrix 1.0 0.0 0.0 0.0
              0.0 1.0 0.0 0.0
              0.0 0.0 1.0 0.0
              0.0 0.0 0.0 1.0
            }
            DEF ll_knee Separator
            {#Euler_JointXYZ
              MatrixTransform
              {#Rotation x:0.0 y: 0.0 z: 0.0(Euler_JointXYZ)
                matrix 1.0 0.0 0.0 0.0
                0.0 1.0 0.0 0.0
                0.0 0.0 1.0 0.0
                0.0 0.0 434.0 1.0
              }
              DEF ll_knee__ll_ankle Separator
              {#Euler_Link
                MatrixTransform
                {#Rotation x:0.0 y: 0.0 z: 0.0(Euler_Link)
                  matrix 1.0 0.0 0.0 0.0
                  0.0 1.0 0.0 0.0
                  0.0 0.0 1.0 0.0
                  0.0 0.0 0.0 1.0
                }
                DEF ll_ankle Separator
                {#Euler_JointXYZ
                  MatrixTransform
                  {#Rotation x:0.0 y: 0.0 z: 0.0(Euler_JointXYZ)
                    matrix 1.0 0.0 0.0 0.0
                    0.0 1.0 0.0 0.0
                    0.0 0.0 1.0 0.0
                    0.0 0.0 368.0 1.0
                  }
                  DEF ll_ankle__ll_foot Separator
                  {#Euler_Link
                    MatrixTransform
                    {#Rotation x:0.0 y: 0.0 z: 0.0(Euler_Link)
                      matrix 1.0 0.0 0.0 0.0
                      0.0 1.0 0.0 0.0
                      0.0 0.0 1.0 0.0
                      0.0 0.0 0.0 1.0
                    }
                    DEF ll_foot Separator
                    {#Manipulator_Joint
                      MatrixTransform
                      {#Rotation x:0.0 y: 0.0 z: 0.0(Manipulator_Joint)
                        matrix 1.0 0.0 0.0 0.0
                        0.0 1.0 0.0 0.0
                        0.0 0.0 1.0 0.0
                        0.0 0.0 0.0 1.0
                      }
                      DEF left_foot Separator
                      {#PfObject
                        MatrixTransform
                        {#Rotation x:180.0 y: 0.0 z: -90.0(PfObject)
                          matrix 3.980102097228898E-14 -600.0 0.0 0.0
                          -650.0 -3.6739403974420595E-14 9.797174393178826E-14 0.0
                          -7.960204194457796E-14 -4.4992793479855726E-30 -800.0 0.0
                          45.500000193715096 77.99999713897705 -31.999999284744263 1.0
                        }
                        Transform
                        {#Correction
                          rotation 1 0 0 1.5707963267948966
                        }
                        File { name "left_foot.iv" }
                      }
                    }
                  }
                  DEF left_anklebone Separator
                  {#PfObject
                    MatrixTransform
                    {#Rotation x:180.0 y: 0.0 z: -90.0(PfObject)
                      matrix 3.061616997868383E-14 -600.0 0.0 0.0
                      -500.0 -3.6739403974420595E-14 9.797174393178826E-14 0.0
                      -6.123233995736766E-14 -4.4992793479855726E-30 -800.0 0.0
                      -4.999999888241291 11.999999731779099 -95.99999785423279 1.0
                    }
                    Transform
                    {#Correction
                      rotation 1 0 0 1.5707963267948966
                    }
                    File { name "left_anklebone.iv" }
                  }
                }
                DEF left_lower_leg Separator
                {#PfObject
                  MatrixTransform
```

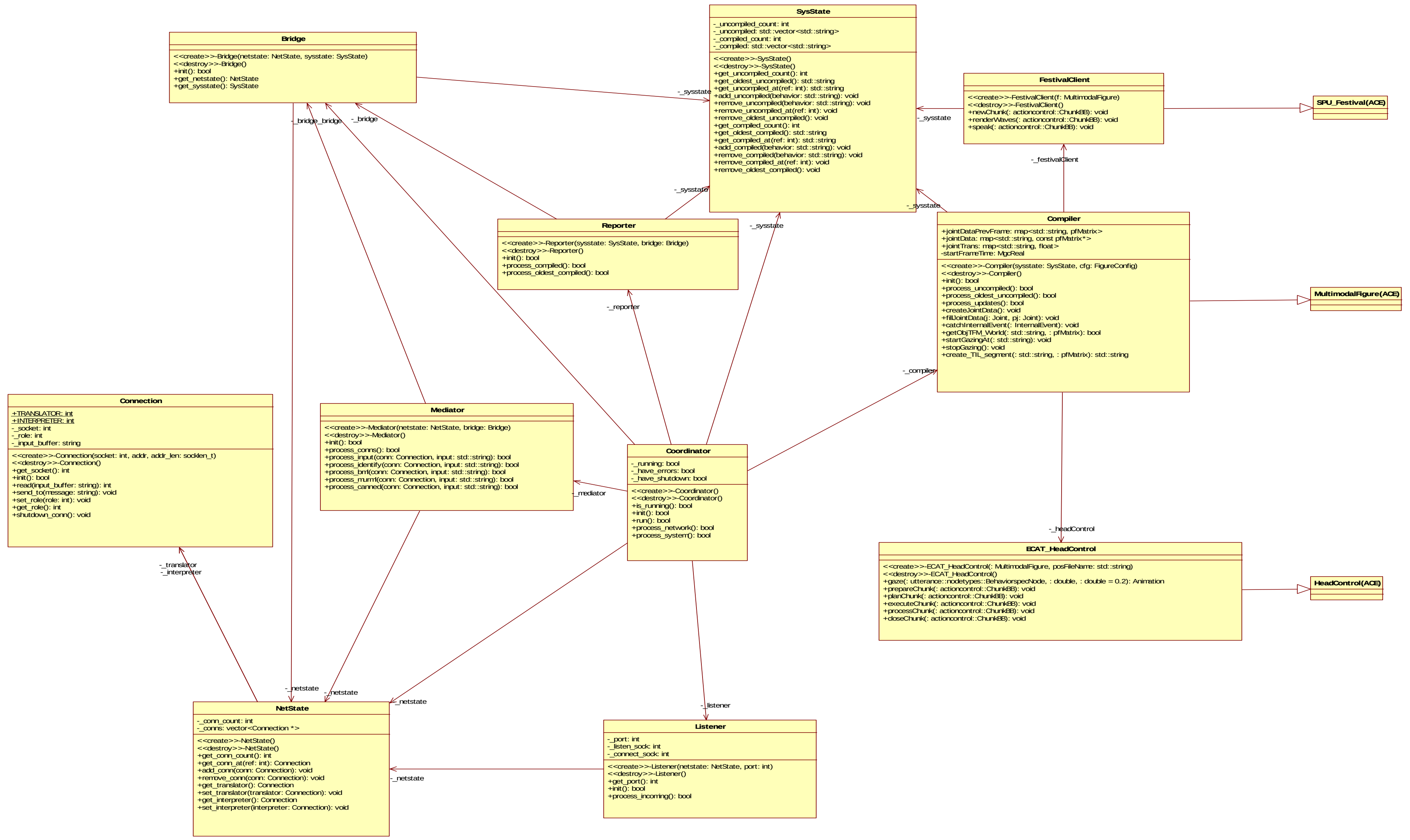
```
                  {#Rotation x:180.0 y: 0.0 z: -90.0(PfObject)
                    matrix 3.6739403974420595E-14 -600.0 0.0 0.0
                    -600.0 -3.6739403974420595E-14 1.1021821192326178E-13 0.0
                    -7.347880794884119E-14 -4.4992793479855726E-30 -900.0 0.0
                    0.0 0.0 287.99999356269836 1.0
                  }
                  Transform
                  {#Correction
                    rotation 1 0 0 1.5707963267948966
                  }
                  File { name "left_lower_leg.iv" }
                }
              }
              DEF left_kneebone Separator
              {#PfObject
                MatrixTransform
                {#Rotation x:180.0 y: 0.0 z: -90.0(PfObject)
                  matrix 3.061616997868383E-14 -500.0 0.0 0.0
                  -500.0 -3.061616997868383E-14 6.123233995736766E-14 0.0
                  -6.123233995736766E-14 -3.749399456654644E-30 -500.0 0.0
                  50.00000074505806 0.0 -50.00000074505806 1.0
                }
                Transform
                {#Correction
                  rotation 1 0 0 1.5707963267948966
                }
                File { name "left_kneebone.iv" }
              }
            }
            DEF left_upper_leg Separator
            {#PfObject
              MatrixTransform
              {#Rotation x:180.0 y: 0.0 z: -90.0(PfObject)
                matrix 3.6739403974420595E-14 -600.0 0.0 0.0
                -600.0 -3.6739403974420595E-14 9.797174393178826E-14 0.0
                -7.347880794884119E-14 -4.4992793479855726E-30 -800.0 0.0
                -54.00000214576721 0.0 211.9999885559082 1.0
              }
              Transform
              {#Correction
                rotation 1 0 0 1.5707963267948966
              }
              File { name "left_upper_leg.iv" }
            }
          }
          DEF left_hipconnect Separator
          {#PfObject
            MatrixTransform
            {#Rotation x:180.0 y: 0.0 z: -90.0(PfObject)
              matrix 3.061616997868383E-14 -500.0 0.0 0.0
              -500.0 -3.061616997868383E-14 6.123233995736766E-14 0.0
              -6.123233995736766E-14 -3.749399456654644E-30 -500.0 0.0
              -14.999999664723873 0.0 75.00000298023224 1.0
            }
            Transform
            {#Correction
              rotation 1 0 0 1.5707963267948966
            }
            File { name "left_hipconnect.iv" }
          }
        }
      }
    }
  }
  DEF HumanoidRoot__rl_sacroiliac Separator
  {#Align_Link
……
……
          DEF right_hipconnect Separator
          {#PfObject
            MatrixTransform
            {#Rotation x:180.0 y: 0.0 z: -90.0(PfObject)
              matrix 3.061616997868383E-14 -500.0 0.0 0.0
              -500.0 -3.061616997868383E-14 6.123233995736766E-14 0.0
              -6.123233995736766E-14 -3.749399456654644E-30 -500.0 0.0
              0.0 0.0 125.0 1.0
            }
            Transform
            {#Correction
              rotation 1 0 0 1.5707963267948966
            }
            File { name "right_hipconnect.iv" }
          }
        }
      }
    }
  }
}
```

Appendix C: Partial Max.xml

```
<?xml version="1.0"?>
<!DOCTYPE Humanoid SYSTEM "Humanoid.dtd">

<Humanoid name="ECAT_Figure" version="1.0">
  <Limbs>
    <Limb  name="Torso"        type="Torso_Limb"   start_joint="HumanoidRoot"   end_joint="vc7"
prefix="t_"    is_start_limb="1"/>
    <Limb name="LArm"       type="Arm_Limb"    start_joint="l_clavicle"    end_joint="l_wrist"
prefix="l_"    is_start_limb="0"/>
    <Limb   name="LThumb"              type="StdLimb"               start_joint="lh_t_joint0"
end_joint="lh_t_effector"  prefix="lh_t_" is_start_limb="0"/>
    <Limb    name="LPointing"       type="StdLimb"                  start_joint="lh_p_joint1"
end_joint="lh_p_effector"  prefix="lh_p_" is_start_limb="0"/>
    <Limb   name="LMiddle"            type="StdLimb"                start_joint="lh_m_joint1"
end_joint="lh_m_effector"  prefix="lh_m_" is_start_limb="0"/>
    <Limb   name="LRing"                type="StdLimb"              start_joint="lh_r_joint1"
end_joint="lh_r_effector"  prefix="lh_r_" is_start_limb="0"/>
    <Limb   name="LLittle"            type="StdLimb"                start_joint="lh_l_joint1"
end_joint="lh_l_effector"  prefix="lh_r_" is_start_limb="0"/>
    <Limb name="RArm"       type="Arm_Limb"    start_joint="r_clavicle"    end_joint="r_wrist"
prefix="r_"    is_start_limb="0"/>
    <Limb   name="RThumb"              type="StdLimb"               start_joint="rh_t_joint0"
end_joint="rh_t_effector"  prefix="rh_t_" is_start_limb="0"/>
    <Limb    name="RPointing"       type="StdLimb"                  start_joint="rh_p_joint1"
end_joint="rh_p_effector"  prefix="rh_p_" is_start_limb="0"/>
    <Limb   name="RMiddle"            type="StdLimb"                start_joint="rh_m_joint1"
end_joint="rh_m_effector"  prefix="rh_m_" is_start_limb="0"/>
    <Limb   name="RRing"                type="StdLimb"              start_joint="rh_r_joint1"
end_joint="rh_r_effector"  prefix="rh_r_" is_start_limb="0"/>
    <Limb   name="RLittle"            type="StdLimb"                start_joint="rh_l_joint1"
end_joint="rh_l_effector"  prefix="rh_r_" is_start_limb="0"/>
    <Limb   name="LLeg"                  type="Leg_Limb"          start_joint="ll_sacroiliac"
end_joint="ll_ankle"       prefix="ll_"   is_start_limb="0"/>
    <Limb   name="RLeg"                  type="Leg_Limb"          start_joint="rl_sacroiliac"
end_joint="rl_ankle"       prefix="rl_"   is_start_limb="0"/>
    <Limb    name="Neck"                      type="StdLimb"                start_joint="vc4"
end_joint="skullbase"      prefix="n_"    is_start_limb="0"/>
    <Limb name="LEye"       type="StdLimb"     start_joint="le_eye"        end_joint="le_eye"
prefix="le_"   is_start_limb="0"/>
    <Limb name="REye"       type="StdLimb"     start_joint="re_eye"        end_joint="re_eye"
prefix="re_"   is_start_limb="0"/>
  </Limbs>


  <Joint name="HumanoidRoot" type="Multi_Joint" num_childs="3">
    <Segment name="HumanoidRoot__vl5" type="Dummy_Link">
      <Joint name="vl5" type="Euler_JointXYZ" num_childs="1">
        <Axis1 angle="0" llimit="-10" ulimit="10"/>
        <Axis2 angle="0" llimit="-15" ulimit="40"/>
        <Axis3 angle="0" llimit="0"   ulimit="0"/>

        <Segment name="vl5__vl1" type="Euler_Link" index="0" translation="0 0 100">
         <Shape type="PfObject" src="vl5__vl1_smooth_tex" translation="0 0 35" rotation="-90 0
-90" scale="1 1.05 1"/>

         <Joint name="vl1" type="Euler_JointXYZ" num_childs="1">
           <Axis1 angle="0" llimit="-10" ulimit="10"/>
           <Axis2 angle="0" llimit="-10" ulimit="30"/>
           <Axis3 angle="0" llimit="-10" ulimit="10"/>

           <Segment name="vl1__vt6" type="Euler_Link" index="0" translation="0 0 200">
             <Shape type="PfObject" src="vl1__vt6_smooth_tex" translation="0 0 90" rotation="-
90 0 -90" scale="1 1 1"/>

             <Joint name="vt6" type="Euler_JointXYZ" num_childs="1">
………..
………..
    <Segment  name="HumanoidRoot__ll_sacroiliac"  type="Align_Link"  index="1"  rotation="0  0  -
90">

      <Joint name="ll_sacroiliac" type="Euler_JointXYZ" num_childs="1">
        <Axis1 angle="0"  llimit="-28"   ulimit="28"/>
```

```
       <Axis2 angle="0"  llimit="-30"   ulimit="30"/>
       <Axis3 angle="0"  llimit="0"     ulimit="0"/>

       <Segment  name="ll_sacroiliac__ll_hip"  type="Euler_Link"  index="0"  translation="0  0
105">

         <Joint name="ll_hip" type="Euler_JointXYZ" num_childs="1">
           <Axis1 angle="-90" llimit="-141"  ulimit="37"/>
           <Axis2 angle="0"   llimit="-40"   ulimit="110"/>
           <Axis3 angle="0"   llimit="-51"   ulimit="56"/>

           <Segment name="ll_hip__ll_knee" type="Euler_Link" index="0" translation="0 0 434">
             <Shape type="PfObject" src="ll_hip__ll_knee" translation="0 0 210" rotation="0 0
180" scale="1 1 1"/>

             <Joint name="ll_knee" type="Euler_JointXYZ" num_childs="1">
               <Axis1 angle="0"  llimit="0"    ulimit="0"/>
              <Axis2 angle="0"  llimit="-145" ulimit="0"/>
              <Axis3 angle="0"  llimit="0"    ulimit="0"/>

              <Segment  name="ll_knee__ll_ankle"  type="Euler_Link"  index="0"  translation="0  0
368">
                <Shape   type="PfObject"   src="ll_knee__ll_ankle"   translation="0   0   190"
rotation="0 0 180" scale="1 1 1"/>

                <Joint name="ll_ankle" type="Euler_JointXYZ" num_childs="0">
                  <Axis1 angle="0"  llimit="0"    ulimit="0"/>
                  <Axis2 angle="0"  llimit="-79"  ulimit="24"/>
                  <Axis3 angle="0"  llimit="0"    ulimit="0"/>
                  <Shape   type="PfObject"   src="lft_foot_lowpoly"   translation="12   0   0"
rotation="-90 0 90" scale="5 5 5"/>

                </Joint>  <!-- END OF: ll_ankle -->
              </Segment>
             </Joint>  <!-- END OF: ll_knee -->
           </Segment>
         </Joint>  <!-- END OF: ll_hip -->
       </Segment>
      </Joint>  <!-- END OF: ll_sacroiliac -->
    </Segment>

    <Segment name="HumanoidRoot__rl_sacroiliac" type="Align_Link" index="2" rotation="0 0 90">
………..
………..
                  <Axis3 angle="0"  llimit="0"    ulimit="0"/>
                  <Shape   type="PfObject"   src="rt_foot_lowpoly"   translation="12   0   0"
rotation="-90 0 90" scale="5 5 5"/>
                </Joint>  <!-- END OF: rl_ankle -->
              </Segment>
             </Joint>  <!-- END OF: rl_knee -->
           </Segment>
         </Joint>  <!-- END OF: rl_hip -->
       </Segment>
      </Joint>  <!-- END OF: rl_sacroiliac -->
    </Segment>
  </Joint>  <!-- END OF: HumanoidRoot -->

</Humanoid>
```

Appendix D: Class diagram of the Compiler


Bridge
<<create>>-Bridge(netstate: NetState, sysstate: SysState)
<<destroy>>-Bridge()
+init(): bool
+get_netstate(): NetState
+get_sysstate(): SysState
SysState
-_uncompiled_count: int
-_uncompiled: std::vector<std::string>
-_compiled_count: int
-_compiled: std::vector<std::string>
<<create>>-SysState()
<<destroy>>-SysState()
+get_uncompiled_count(): int
+get_oldest_uncompiled(): std::string
+get_uncompiled_at(ref: int): std::string
+add_uncompiled(behavior: std::string): void
+remove_uncompiled(behavior: std::string): void
+remove_uncompiled_at(ref: int): void
+remove_oldest_uncompiled(): void
+get_compiled_count(): int
+get_oldest_compiled(): std::string
+get_compiled_at(ref: int): std::string
+add_compiled(behavior: std::string): void
+remove_compiled(behavior: std::string): void
+remove_compiled_at(ref: int): void
+remove_oldest_compiled(): void
FestivalClient
<<create>>-FestivalClient(f: MultimodalFigure)
<<destroy>>-FestivalClient()
+newChunk(: actioncontrol::ChunkBB): void
+renderWaves(: actioncontrol::ChunkBB): void
+speak(: actioncontrol::ChunkBB): void
SPU_Festival(ACE)
-_sysstate
-_bridge_bridge
-_bridge
-_sysstate
-_sysstate
-_festivalClient
-_sysstate
-_sysstate
Reporter
<<create>>-Reporter(sysstate: SysState, bridge: Bridge)
<<destroy>>-Reporter()
+init(): bool
+process_compiled(): bool
+process_oldest_compiled(): bool
Compiler
+jointDataPrevFrame: map<std::string, pfMatrix>
+jointData: map<std::string, const pfMatrix*>
+jointTrans: map<std::string, float>
-startFrameTime: MgcReal
<<create>>-Compiler(sysstate: SysState, cfg: FigureConfig)
<<destroy>>-Compiler()
+init(): bool
+process_uncompiled(): bool
+process_oldest_uncompiled(): bool
+process_updates(): bool
+createJointData(): void
+fillJointData(j: Joint, pj: Joint): void
+catchInternalEvent(: InternalEvent): void
+getObjTFM_World(: std::string, : pfMatrix): bool
+startGazingAt(: std::string): void
+stopGazing(): void
+create_TIL_segment(: std::string, : pfMatrix): std::string
MultimodalFigure(ACE)
-_reporter
-_compiler
Connection
+TRANSLATOR: int
+INTERPRETER: int
-_socket: int
-_role: int
-_input_buffer: string
<<create>>-Connection(socket: int, addr, addr_len: socklen_t)
<<destroy>>-Connection()
+get_socket(): int
+init(): bool
+read(input_buffer: string): int
+send_to(message: string): void
+set_role(role: int): void
+get_role(): int
+shutdown_conn(): void
Mediator
<<create>>-Mediator(netstate: NetState, bridge: Bridge)
<<destroy>>-Mediator()
+init(): bool
+process_conns(): bool
+process_input(conn: Connection, input: std::string): bool
+process_identify(conn: Connection, input: std::string): bool
+process_bml(conn: Connection, input: std::string): bool
+process_murml(conn: Connection, input: std::string): bool
+process_canned(conn: Connection, input: std::string): bool
Coordinator
-_running: bool
-_have_errors: bool
-_have_shutdown: bool
<<create>>-Coordinator()
<<destroy>>-Coordinator()
+is_running(): bool
+init(): bool
+run(): bool
+process_network(): bool
+process_system(): bool
-_mediator
-_headControl
ECAT_HeadControl
<<create>>-ECAT_HeadControl(: MultimodalFigure, posFileName: std::string)
<<destroy>>-ECAT_HeadControl()
+gaze(: utterance::nodetypes::BehaviorspecNode, : double, : double = 0.2): Animation
+prepareChunk(: actioncontrol::ChunkBB): void
+planChunk(: actioncontrol::ChunkBB): void
+executeChunk(: actioncontrol::ChunkBB): void
+processChunk(: actioncontrol::ChunkBB): void
+closeChunk(: actioncontrol::ChunkBB): void
HeadControl(ACE)
-_translator
-_interpreter
-_netstate
-_netstate
-_netstate
-_listener
NetState
-_conn_count: int
-_conns: vector<Connection *>
<<create>>-NetState()
<<destroy>>-NetState()
+get_conn_count(): int
+get_conn_at(ref: int): Connection
+add_conn(conn: Connection): void
+remove_conn(conn: Connection): void
+get_translator(): Connection
+set_translator(translator: Connection): void
+get_interpreter(): Connection
+set_interpreter(interpreter: Connection): void
-_netstate
Listener
-_port: int
-_listen_sock: int
-_connect_sock: int
<<create>>-Listener(netstate: NetState, port: int)
<<destroy>>-Listener()
+get_port(): int
+init(): bool
+process_incoming(): bool