*NICE project (IST-2001-35293)*

# Natural Interactive Communication for Edutainment

# NICE Deliverable D3.7 Multimodal Output Generation Module for the First Prototype

*17 October 2003*

*Authors*

*Teliasonera: Johan Boye, Joakim Gustafson and Mats Wirén*

*NISLab: Manish Mehta, Andrea Corradini, Niels Ole Bernsen*

*Liquid Media AB: Morgan Fredriksson*

*Limsi-Cnrs: Jean-Claude Martin*

| | |
|---|---|
| **Project ref. no.** | IST-2001-35293 |
| **Project acronym** | NICE |
| **Deliverable status** | Restricted |
| **Contractual date of delivery** | 1 September 2003 |
| **Actual date of delivery** | 17 October 2003 |
| **Deliverable number** | D3.7 |
| **Deliverable title** | Multimodal output generation module for the first prototype |
| **Nature** | Report |
| **Status & version** | Final |
| **Number of pages** | 24 |
| **WP contributing to the deliverable** | WP3 |
| **WP / Task responsible** | LIMSI-CNRS |
| **Editor** | Jean-Claude Martin |
| **Author(s)** | Johan Boye, Joakim Gustafson and Mats Wirén, Andrea Corradini, Morgan Fredriksson, Niels Ole Bernsen, Manish Mehta |
| **EC Project Officer** | Mats Ljungqvist |
| **Keywords** | Multimodal output, ECA specification |
| **Abstract (for dissemination)** | This report, Deliverable D3.7 from the HLT project Natural Interactive Communication for Edutainment (NICE), describes the construction of the output generation module of the system, which receives a richly parametrised semantic instruction from the dialogue manager (WP5) and splits this instruction into synchronised text instructions to the text-to-speech output module, motion instructions to the characters' lips, additional behavioural instructions to the animated character (face, gaze, body), and instructions wrt. on-screen object behaviour. |

# Table of Contents

# 1 Introduction

## 1.1 Definitions and scope

As mentioned in the NICE contract, the goal of this report is to describe the multimodal output generation module for coordinated output generation of speech, animated character behavior and illustrative graphics/on-screen objects. This includes (a) empirical research on the graphics modalities (apart from animated characters, see WP4) to be used together with spoken output; and (b) construction of the output generation module of the system, which receives a richly parameterized semantic instruction from the dialogue manager (WP5) and splits this instruction into synchronized text instructions to the text-to-speech output module, motion instructions to the characters' lips, additional behavioral instructions to the animated character (face, gaze, body), and instructions wrt. on-screen object behavior.

The experimental research on multimodal output generation conducted earlier in the NICE project was described in D3.3 "Analysis and specification of cooperation between input modalities and cooperation between output modalities".

The First Prototype (PT1) will include one multimodal output generation module and specification language for HCA's study and another for the Virtual World scenario. The differences in the specification languages for both can be explained by the facts that they address different parts of the system with different conversational abilities and interaction patterns (Virtual World vs. HCA study).

This report describes the design specification as well as the description of how verbal/non-verbal output generation actually works in the two Multimodal Output Generation Modules for the First Prototype (PT1). The complete list of non-verbal behaviors actually present in PT1 and the non-verbal specification language used for PT1 is described in D4.2 "Agent behavior and gesture controllers". A behavior subsystem will link domain-related data with the parameters representing the current emotional state of the character, producing an animation containing the gestures, emotional expressions, and lip-synchronized speech of the character.

## 1.2 Structure

This deliverable is structured as follows:

- Multimodal output generation in the architecture of the 1st NICE prototype
- Response generation for the fairy-tale world characters
- Response generation for the Hans Christian Andersen character

## 2 Multimodal output generation in the architecture of the 1st NICE prototype

Figure 3 is taken from D3.3 and recalls the NICE architecture showing main components and main information exchanges as agreed in April 2003. In this schema, the Multimodal Output Generation Module is called "Response Generation". Figure 1 and 2 provide up-to-date descriptions of the information flow and system architecture for the 1st prototype.

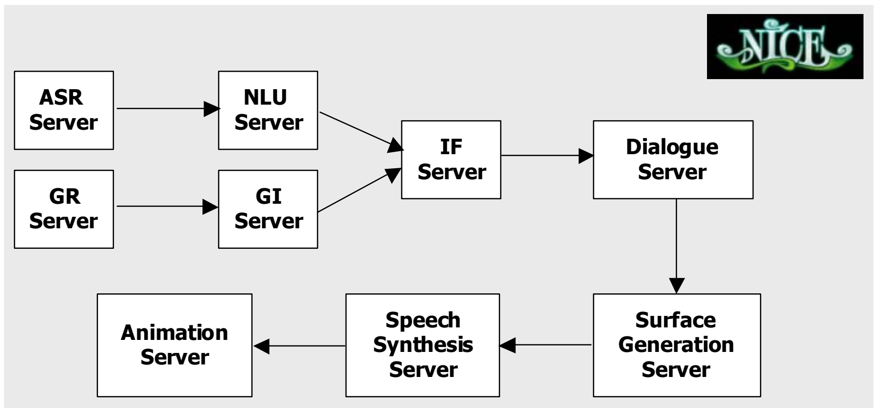


**Figure 1. The information flow in the first prototype.**

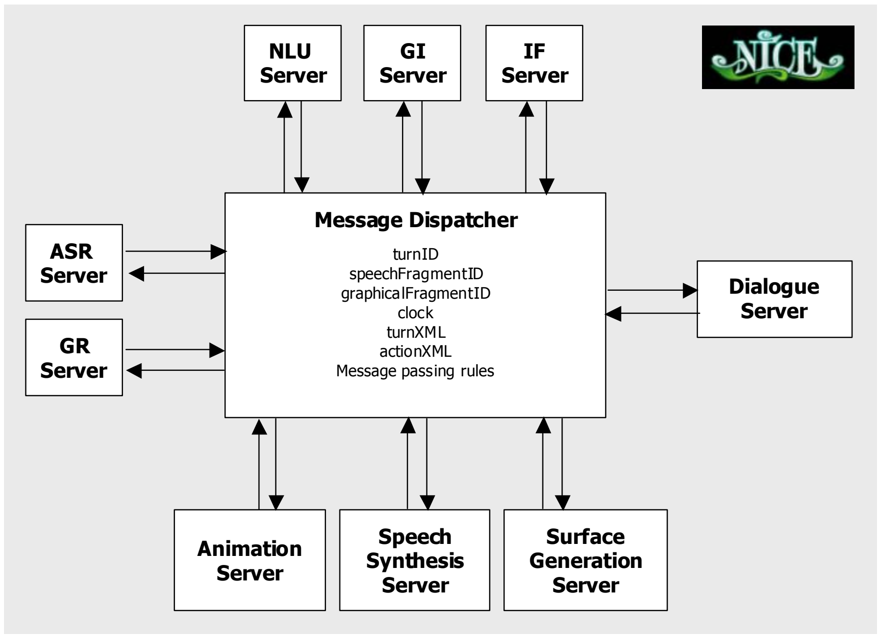


**Figure 2. The system architecture of the first prototype.**

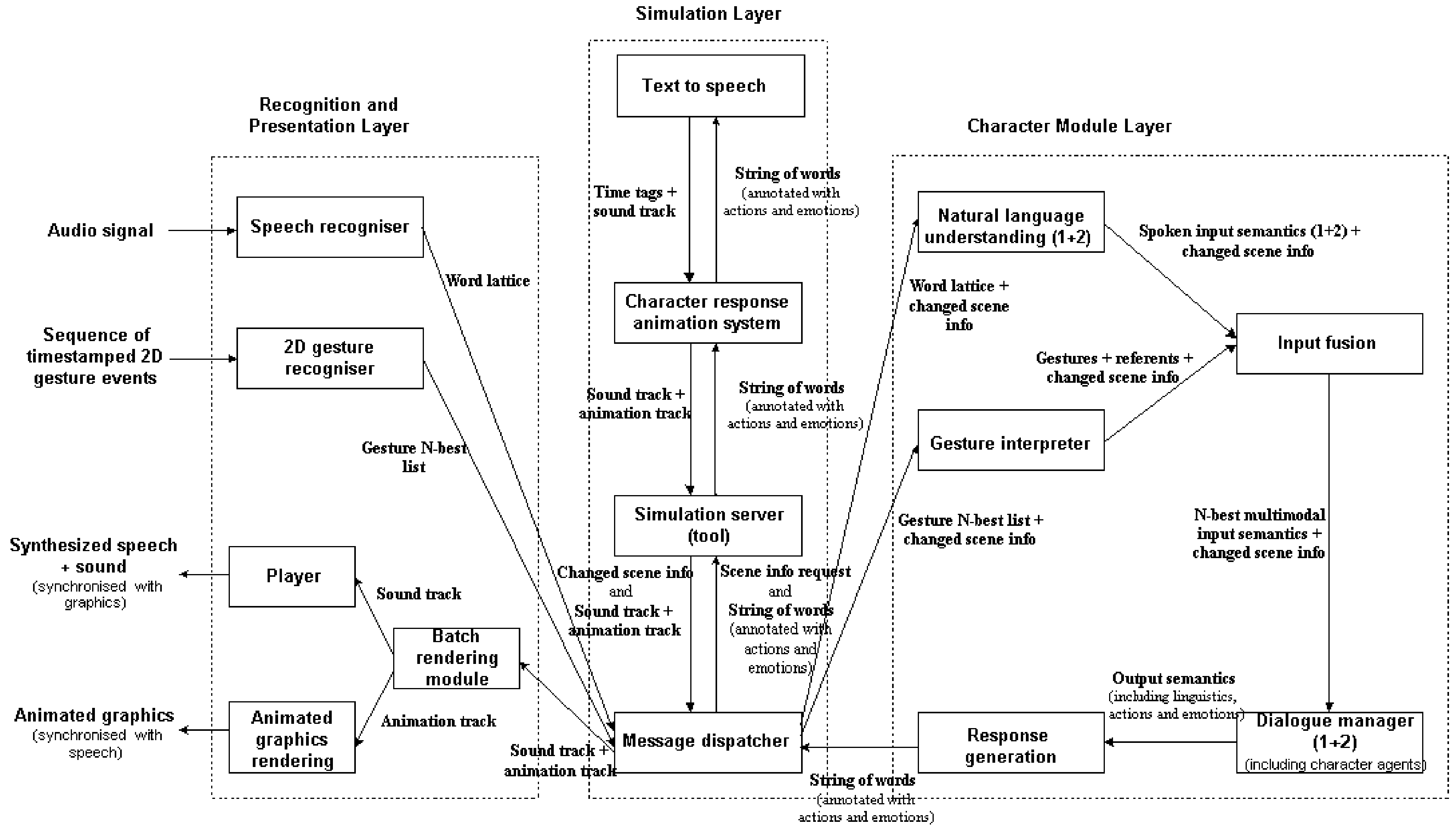

Simulation Layer
Recognition and Presentation Layer
Character Module Layer
Text to speech
Time tags + sound track
String of words (annotated with actions and emotions)
Audio signal
Speech recogniser
Word lattice
Character response animation system
Natural language understanding (1+2)
Spoken input semantics (1+2) + changed scene info
Word lattice + changed scene info
Sequence of timestamped 2D gesture events
2D gesture recogniser
Input fusion
Gestures + referents + changed scene info
Sound track + animation track
String of words (annotated with actions and emotions)
Gesture N-best list
Gesture interpreter
Simulation server (tool)
Gesture N-best list + changed scene info
N-best multimodal input semantics + changed scene info
Synthesized speech + sound (synchronised with graphics)
Player
Sound track
Changed scene info and Sound track + animation track
Scene info request and String of words (annotated with actions and emotions)
Batch rendering module
Output semantics (including linguistics, actions and emotions)
Animated graphics (synchronised with speech)
Animated graphics rendering
Animation track
Message dispatcher
Response generation
Dialogue manager (1+2) (including character agents)
Sound track + animation track
String of words (annotated with actions and emotions)

**Figure 3. NICE architecture showing main components and main information exchanges. (1+2) means that there will be an HCA version of the component and a fairy tale version of the component.**

# 3 Response generation for the fairy-tale-world characters

This section describes generation of multimodal output of the characters in the fairy-tale world. We also describe what kinds of non-verbal actions as well as how objects can be manipulated and behave in Prototype 1. We have not described the actual XML interface for animation or listed all instances of animations that we describe, since that is part of Deliverable D4.2. We refer to Section 2 in Deliverable D1.2 b for the game scenario and its implications for multimodal output.

## 3.1 Which module decides what to do?

The fairy-tale characters will be controlled at different layers. Some actions will have to be fast and reactive, while others require time to plan and produce. The fairytale character module can be divided into the three layers shown in figure 4.

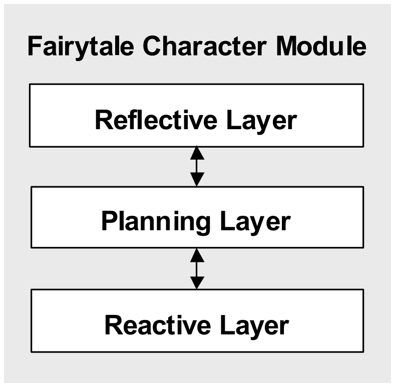


**Figure 4. The three layers in the fairytale character module.**

Allwood et al (1991) describe four basic communicative functions that correspond to the levels, identified by Clark (1994), at which problems can arise during the grounding process. These are:

**level 1** – contact, vocalization and attention

**level 2** – perception, identification

**level 3** – understanding, meaning

**level 4** – attitudinal reaction , proposal and uptake

Levels 1 and 2 would correspond to the Reactive Layer in our model and they are handled by the Message Dispatcher. It is responsible for showing that the agent has heard that the user is speaking and what the user have said. This is handled by the Message Dispatcher since the reaction have to be realized fast and since they have to be synchronized with the user's input, i.e. displaying awareness directly without thinking as soon as a user starts talking or points at a certain object. It also handles the realization of different states-of-mind, such as idle, listening and thinking. Levels 3 and 4 would correspond to the Planning Layer in our model and they are handled by the Dialogue Manager. It is responsible for showing whether the character has understood the meaning of the utterance or if the character has agreed to do what the meaning implies It is handled by the Dialogue Manager since it is necessary to have knowledge both about the previous discourse and the goals of the character to do this. The Reflecting Layer would correspond to learning and adaptation of the rules used by the Dialogue Manager to obtain the goals, something which we will not implement in the first prototype.

## 3.2 What kinds of utterances should be generated?

As mentioned above, Cloddy Hans is the only fairy-tale character present in Prototype 1. Basically, the task of generation is here limited to that of producing utterances that serve as a vehicle for an instructional dialogue. By "utterance", we here mean the verbal realization of one or more dialogue (speech) acts within a single turn, possibly accompanied by a gesture. Specifically, the following types of utterances are handled by Cloddy Hans:

- Responses to instructions from the user.
    - Confirmations (more explicit than acknowledgements).
    - Acknowledgements (like confirmations, but more like just "yes").
    - Clarification questions, for example, asking about references that are perceived as ambiguous.
- Initiatives that serve to fulfill Cloddy Hans' plan or long-term goals.
    - Requests for instructions on what goals or subgoals to pursue. For example, asking about the position of an object or where to put an object.

For Prototype 2, the following types of utterances will also be handled by Cloddy Hans:

- Social utterances, limited to responses within conventionalized social interchanges.
- Utterances for holding the floor, for example, "Wait a moment…" or "Hang on…".
- Back-channel continuers.

## 3.3 Overview of utterance generation

The basic stages of utterance generation are as follows:

1. Deciding the content of the utterance, expressed as one or several domain-dependent dialogue acts. (Compare Deliverable D1.2, Section 4.2 and Deliverable D5.1.) Note: We assume that all dialogue acts have a verbal realization.
2. Determining a) which output modalities to be used and b) the kinds of anaphoric expressions and deictic gestures to be used, if any.
3. Surface realization of the verbal utterance as text.
4. Speech synthesis of the verbal utterance.

5. Rendering of deictic gestures, if present.

Stages 1–2 and 3 can be taken to correspond to the standard distinction in text generation between text planning and linguistic realization, respectively. Since stages 1–2 require access to the dialogue context, they are carried out by the dialogue manager (see Deliverable D5.1). Stage 3 is carried out by the surface-generation module. Stages 4 and 5 are carried out by the speech-synthesis and animation modules, respectively. The latter three stages are described below.

A more fine-grained division would also include "aggregation" of utterances, that is, deciding how to fold several elements of information into one or more sentences [Reiter and Dale 1997]. This is also relevant here, but for the time being we only concatenate the realizations of dialogue acts in their respective order.

## 3.4 What is the source?

Basically, the input to stage 1 consists of actions resulting from the agenda that are realized through dialogue acts. This is further described in Deliverable D5.1.

The input to stage 3 (surface generation) is the following:

- One or more domain-dependent dialogue acts, using the same representation as that used for final analysis of utterances by the dialogue manager, except that substructures not to be realized can be hidden (see further below).
- Parameters that specify how references to each object should be realized. There are three cases, depending on the salience of the object referred to:
  - pronoun (for example, "it");
  - definite noun phrase (for example, "the axe");
  - demonstrative noun phrase with accompanying gesture (for example, "that one" accompanied by a pointing gesture).

The dialogue-act representation carries implicit information about the "verbosity" with respect to (pieces of) the utterance, that is, the degree to which linguistic material is included or left out (as in elliptic utterances). This allows us to produce different realizations from the same underlying representation depending on what is appropriate in the given context. Lack of verboseness is coded by "folding" (hiding) substructures not to be realized in the output; the corresponding substructures are denoted by a "+" sign. For example, whereas the unfolded dialogue-act representation confirm(cloddy, user, putdown(cloddy, axe, useful)) might be realized as "OK, I'll put it in 'useful'", the folded version confirm(cloddy, user, "+") would be realized simply as "OK".

## 3.5 Example

We begin by looking at an example. Consider surface realization of the dialogue act ask(cloddy, user, ?y:location(putdown(cloddy, axe, y))), where we assume that the object axe can be uniquely determined from the context. The dialogue act might therefore be paraphrased as "Where shall I put it?". For the sake of the example, we look simultaneously at two levels of verbosity, one in which the resulting utterance from Cloddy Hans is simply "Where?" and one in which he outputs the complete sentence. The former would be felicitous if the put action is salient but the destination needs to be found out (for example, if the user had just said "Put it somewhere"). The input to surface generation from the dialogue manager is:

Dialogue-act (verbosity 1): ask(cloddy, user, ?y:location("+"))

Dialogue-act (verbosity 2): ask(cloddy, user, ?y:location(putdown(cloddy, axe, y)))

Reference: axe/pronoun

The "+" indicates hidden structure corresponding to linguistic material not to be generated. We begin by decomposing the dialogue-act representation into a sequence of basic generation chunks:

Verbosity 1: ask, speaker(cloddy), addressee(user), ?y:location

Verbosity 2: ask, speaker(cloddy), addressee(user), ?y:location, putdown, agent(cloddy), object(axe)

This process is guided by a set of decomposition rules working as follows:

- Each functor is extracted in the form of an atomic expression; for example, the expression ask(...) yields ask.
- Each argument of a functor is extracted as follows:
    - If it is an atomic argument, it is extracted as an argument together with its role (formal parameter) as functor.
    - If it is a question-mark operator, it is extracted as is.
    - If it is a functor, it is (again) extracted in the form of an atomic expression.

  For example, ask(cloddy, user, ?y:location(...)) yields speaker(cloddy), addressee(user), ?y:location.

Next, we use a set of surface-generation rules to map the elements of this sequence to surface elements associated with functional and word-class information:

Verbosity 1:

ask → question/sentence_type

(ask, ?y:location) → "where"/wh

Verbosity 2:

ask → question/sentence_type

(ask, ?y:location) → "where"/wh

(speaker(cloddy), agent(cloddy)) → "I"/subject

putdown → "put"/predicate

axe/pronoun → "it"/object

Finally, we use the longest matching linearization rule with sentence_type as left-hand side to output the resulting surface text with the correct word order:

Verbosity 1:

question({X/wh}) → [X]

Verbosity 2:

question({X/wh Y/subject Z/ predicate T/object}) → [X, "should", Y, Z, T]

The argument to question is a set; hence, the order of the surface elements does no matter.

## 3.6 Outline of algorithm for surface realization

The algorithm for surface realization consists of the following steps:

1. Decomposition of the dialogue-act representation using a set of decomposition rules.
2. Mapping of generation primitives to basic lexical chunks together with functional information using a set of surface-generation rules. This means mapping semantic primitives to their lexical counterparts, as well as keeping track of the sentence type to be generated.
3. Linearization of surface elements and addition of "glue" words using linearization rules. This means reconstructing syntactic structure given lexical and functional information together with the sentence type.

## 3.7 Synthesizing the verbal utterances

The surface generation module generates an xml-structure of the multimodal utterance that contains both the text-representation of the verbal part of the utterance and animation tags that represent the animation part. This xml-structure is used as input to the Speech Synthesis Module. This module produces a sound file containing the verbal realization of the utterance along with a lip-synchronization xml-structure that the animation system uses. Lastly, it inserts time stamps into the animation tags to facilitate synchronization of speech and actions in the character output.

An important role of the actual realization of the verbal utterances in the fairytale domain is that it conveys the characters' personality. The friendly, but dunce and slow Cloddy Hans should not have the same way of expressing himself, or not even the same type of voice, as the evil, smart and selfish prince. We need to be able to tailor both their voices and their ways of expressing themselves to the personality we want them to convey. The personality of a voice is among other things conveyed by the voice quality and prosody. However, these are two research areas where a lot of research has to be done before we understand how to generate them accurately. We use unit selection synthesis to achieve a natural voice quality and prosody. In this technique the voice quality and prosody is given by a real speaker that reads a carefully design utterance database. This is a way to get one voice for one domain. It is quite time consuming and requires a lot of planning when designing the utterances that the voice professional is to read, but the result is a synthesized voice that has both the voice quality and prosody of the reader. For the first prototype in the fairytale domain we have recorded a unit selection voice for Cloddy Hans in the Fairytale machine scenario domain. Apart from utterances that are related to the machine scenario, we have designed a number of social utterances that can be used in formalized social exchanges between the user and Cloddy Hans. There are also a number of meta-oriented utterances that are used to talk about the previous discourse. A number of general utterances are also generated using a greedy algorithm to ensure that the system has coverage of all diphones. This makes it possible for Cloddy Hans to say anything. Lastly, a number of dialogue-regulating utterances have been designed. These are non-linguistic sounds (e.g. filled pauses with various prosody), words (e.g. filler words, feedback words and back-channeling words) and phrases (floor-holders and feedback phrases). During runtime, the last category will be cached in the system and generated by the Message Dispatcher to ensure a fast and responsive behavior. They will for example be generated when it is necessary to buy time while generating the next system utterance.

## 3.8 Generation of non-verbal behavior

To increase the believability of the fairytale characters, they must also be able to produce non-verbal behavior. There are a number of non-verbal behaviors in the system.

- Physical action (goTo(location), pickUp(thing), pointAt(thing))
- Emotional display (surprised, angry, happy)
- State of mind (idle, listening, thinking)
- Feedback gestures (nodding, shaking head)
- Back-channeling gestures (raising eyebrows, nodding)

There are two types of non-verbal output in the system: events and states. This division has proven useful in previous character animation system intended for multimodal dialogue systems (Beskow et al 2003). An event has a certain, often short, duration and has a predictable realization, while a state has an arbitrary, often long, duration and a semi-random realization. The realization of the events will be handled by the Animation System, (see D4.2 for details). The states are handled by the Message Dispatcher, which at random points in time asks the Animation System to generate an animation event, like blinking. There is a state-description file that describes how different states will be realized. A state has three parts where different non-verbal behaviors can be described: *enter-actions, sustain-actions* and *exit-actions*, (this is also in accordance with Beskow's model of states). In these parts animation events are listed along with a number that says how often that event will be generated. An example of how these definition might look is shown in figure 5.

```
<state name=”idle”>
   <enter-actions>
      <animation>
         <name>lookAtUser</name>
         <probability>0.5</probability>
      </animation>
   </enter-actions>
   <sustain-actions>
      <intensity>0.2</intensity>
      <animation>
         <name>blink</name>
         <probability>0.8</probability>
      </animation>
      <animation>
         <name>blinkdouble</name>
         <probability>0.5</probability>
      </animation>
      <animation>
         <name>lookup</name>
         <probability>0.1</probability>
      </animation>
   </sustain-actions>
   <exit-actions>
      </animation>
         <name>nod</name>
         <probability>0.5</probability>
      </animation>
   </exit-actions>
</state>
```

**Figure 5. An example from the state-description file.**

The intensity tag determines how often an animation will be carried out (on average), whereas the probability tag determines the relative distribution of different animations (i.e. the probability of a certain animation given the fact that an animation is to be carried out).

The Message Dispatcher can also move from an idle-state with few and small gestures to an idle state with more distinct and encouraging gestures when the users have not said anything for a long time. The advantage of letting the Message Dispatcher randomly select the animation events that make up an idle state is that it for example can refrain from telling the animation system to generate an encouraging facial gesture if the ASR just sent a StartOfSpeech event, and instead tell the animation system to generate an awareness gesture.

## 3.9 Manipulation and behavior of objects

The system can also generate actions that influence objects in the 3D world:

- create object
- move and rotate object
- delete object
- highlight object
- object-specific actions (highlight one of the slots in machine, play error-sound while opening rejection compartment)
- camera position and direction

These actions will be used to set up scenes where objects will be placed in a certain way, and where the user will have to interact with Cloddy Hans to arrange them in another way. It will also be used as visual feedback in deictic utterances by Cloddy Hans, as well as for feedback when the user has selected an object with gesture input. Lastly, the camera angle switches in cases where it is necessary to zoom in on Cloddy Hans and certain objects in order do see them clearly. The Camera position will be changed to make it possible to follow Cloddy Hans as he walks from the shelf with objects to the fairytale machine.

## 3.10 Generation of character animations

The character animation system will be able to generate all the kinds of behaviors described above. The interface will be in XML and a description of the format of that interface along with the actual animations that have been implemented so far can be found in D4.2. When the animation system has finished an animation it will inform the Message Dispatcher, which in turn will inform the Dialogue Manager that the task has been performed.

# 4 Response generation for the Hans Christian Andersen character

## 4.1 Introduction

This chapter presents a specification of the implemented response generation module for Hans Christian Andersen (HCA) for the first NICE prototype. The response generation (RG) module represents a template based approach to output generation. It has been developed using Sixtus Prolog and C++.

The RG receives from the character module (CM) the values for the current emotional state of HCA along with a list of references to template outputs and a frame that contains template value information stored in slots. The RG retrieves the actual text and graphical output corresponding to the references received, replaces template variables with values from the frame, combines the output in an XML format and sends it to the message dispatcher. Prior to that, the values of HCA's emotional state are used to modify the output graphical behaviors.

Speech synthesis for the HCA character will use Scansoft's speech synthesiser RealSpeak.

In the XML examples below, we have used Liquid's XML specification from April 2003. We expect to get a new specification in NICE Report D4.2.

## 4.2 Input from the character module to the response generator

### 4.2.1 Input Format

The RG receives from the CM the values for the current emotional state of HCA (cf. NICE Report D1.2a) along with a list of references to template outputs and a frame.

#### *4.2.1.1 Templates*

Throughout this specification we refer to template outputs (in brief: templates) as any structured output made up of text, text-only slots (to be filled with text from the frame received from the character module after further processing) and non-verbal behavior tags. For example, let the following list of prototypical templates be the first four templates stored in the Prolog file:

T1) [G1:start] Yes [G1:end] I am a [S1] [G2:start] writer [G2:end]

T2) My name is Hans Christian Andersen

T3) I like to [G1:start] [S1] [G1:end]

T4) I love [G1:start] to write [G1:end]

T5) This is the [G1:start] [S1] of [S2][G1:end]

In the list above, elements within square brackets starting with numbered G and S letters represent uninstantiated non-verbal and text parts of the template, respectively. They are uninstantiated in the sense that they need to be retrieved at run-time as described in the next section. In order to provide timing information for speech (text) and non-verbal behavior during rendering, any non-verbal element is made up of two tags to indicate both its start and end. Thus, in example T4), the tags *[G1:start]* and *[G1:end]* indicate one single movement that, whichever it is, has to be timed with the word to write. The tags *[S1]* in T1) and T3) indicate uninstantiated text.

From a notational point of view, example T2) shows what is referred to as a canned output template while the remaining examples from the above lists refer to what we will call linguistic templates or templates for short. Of course, a canned output template is nothing but a subclass of templates in which there are neither text slots nor non-verbal tags. In the list above, the templates are independent from one another, so that, for instance, the *[G1:start]* tag in T1) is not the same as the one used in T3).

#### *4.2.1.2 Frames*

The frame structure is stored as a two-column table where the first column contains the name of the field while the second one its value. The values for the second column are filled at run time after extracting information from the HCA knowledge base (KB) or from the user's input.
The frame structure is implemented as a C++ class. Without going into details of the class methods, it is worth noticing that functions like, e.g., *char* getSlotName(int n), void setSlotValue(int n, const char *slotValue)* allow to extract the name of, and to set the value for, a slot, respectively. The methods has a function *char* createPrologListForRG()* which is the one that takes care of providing the values of the slots in a format required by the RG Prolog module.

For instance, let us assume that the RG has received the template *T5) This is the [G1:start] [S1] of [S2][G1:end]* (see section 4.2.1.1) where S1="name" and S2="obj" together with the following frame F1):

```
{
        name   :        picture
        obj    :        Thumbelina
        others :        NULL
}
```

In this case, the function *createPrologListForRG()* returns a string in the form *[name="picture" obj = "Thumbelina"]* (see Figure 1). The Prolog module fills in the slots with their corresponding values and the resulting string is *"this is the [G1:start] picture of Thumbelina [G1:end]"*. When the value in the frame assumes the value *NULL* nothing is replaced and the tag is simply skipped.

#### *4.2.1.3 Examples*

Now that the formats for the templates and the frame are clear, we can discuss the full input format from the CM. The RG receives from the CM the values for the current emotional state of HCA along with a list of references to templates and a frame. The RG input will therefore look like this:

*[Emotional Values, [link1, G11, .., G1N],.., [linkM, GM1, .., GMP], frame].*

For example, assume the CM wants the RG to output: '*My name is Hans Christian Andersen. I love writing. Yes, I am a great writer*'. Using the template list shown in section 4.2.1.1, the corresponding expected multimodal output will be generated by passing on to the RG the references to templates T2), T4) and T1) and three pointers to frames in exactly this order. In that case the RG will get input that looks like this:

(Input 1): *[Emotional Values, [T2], [T4, B2], [T1, B1, B3], NULL, NULL, F1]*

(Input 2): *[Emotional Values, [T2], [T4, B2], [T1, B1, NULL], NULL, NULL, F1]*

Here *NULL, NULL, F1* is a list of pointers to frames. As templates T2) and T4) do not contain any tags there is no need for any frames, thus the value NULL is passed twice. However, template T1) requires some kind of information as it must be expanded in its component [S1]; this is retrieved from frame F1 which has to have the first field associated with the value *great* in order to produce the utterance *Yes I am a great writer*.

We will not discuss *Emotional Values* at this stage, as we have not yet carefully tested and investigated the nature and range for these parameters; *B1*, *B2* and *B3* are references to non-verbal actions that have to be stored separately from the templates. To further investigate the example, let us assume that the following elements are stored in the non-verbal template database:

B1) Nodding

B2) Writing in Air

B3) Opening Arm in Front of Body

B4) Negation

B5) Pointing

With these entries, (Input 1) says to the RG: while uttering the sentence '*My name is Hans Christian Andersen. I love writing. Yes, I am a writer*' perform a '*writing in the air*' gesture co-occurring with the word '*writing*', a '*nodding*' head movement co-occurring with the word '*yes'*, and a '*opening arm in front of body*' co-occurring with the word '*writer'*.

However, the same text sentence '*My name is Hans Christian Andersen. I love writing. Yes, I am a writer*' has a different gesture-speech relation when (Input 2) is received by the RG. In fact, (Input 2) instructs the RG to utter the sentence and still perform a '*writing in the air*' gesture co-occurring with the word '*writing*' and a '*nodding*' head movement co-occurring with the word '*yes'*, but with the difference that no gesture will co-occur with the word '*writer'*.

The described approach allows a high degree of flexibility as the binding non-verbal behavior/speech is done at run-time, allowing the same sentence to be synthesized at different times with different accompanying non-verbal behaviors. Note that the non-verbal behavior will be affected by the value(s) of the parameter *EmVals*.

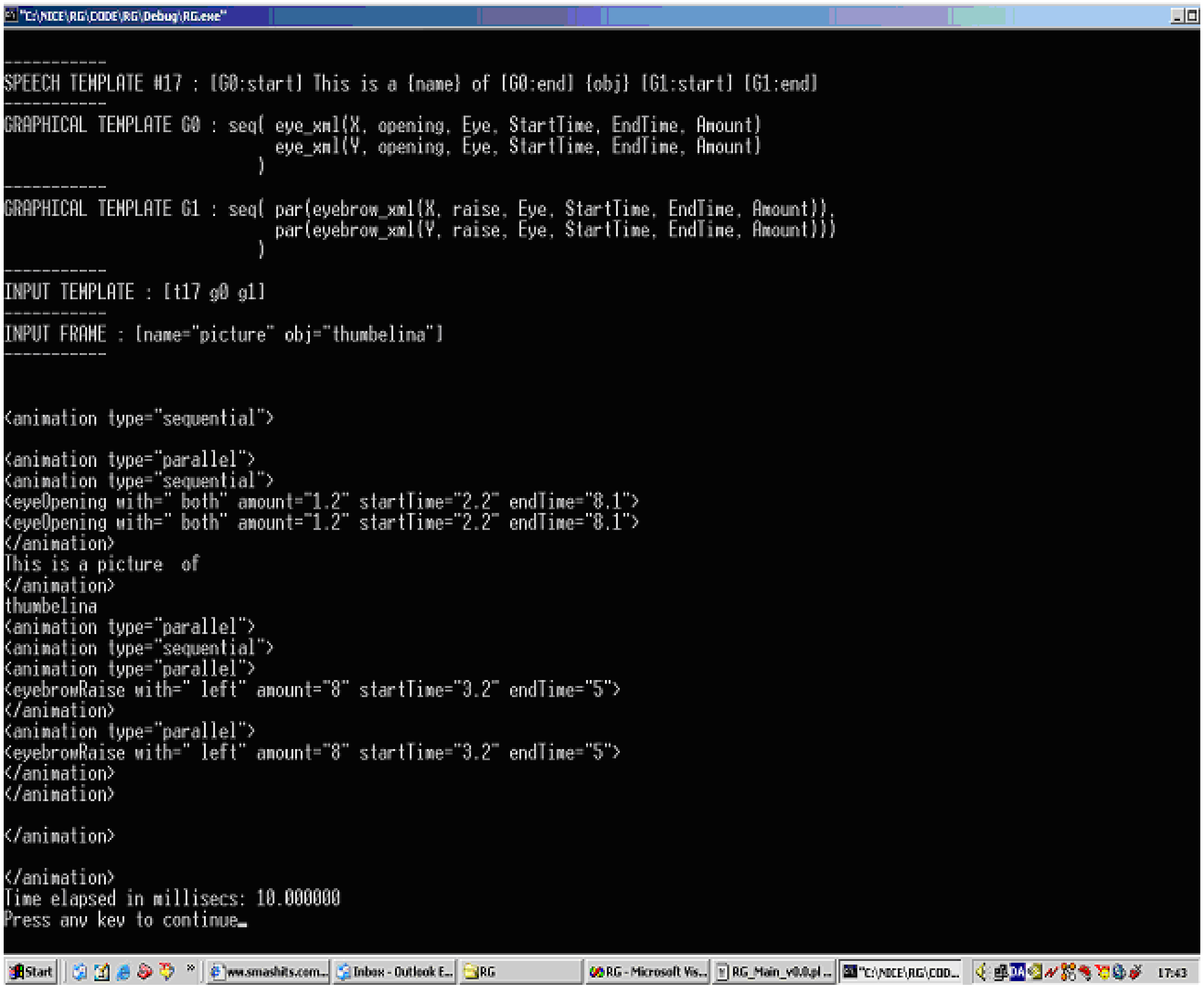
At this point it should also be clear that pointers to behavioral elements might remain uinstantiated (i.e. assume NULL value) to indicate that nothing has to be retrieved. In any case, it is important to point out that there must be a numerical correspondence between a template and the list of behavioral elements provided along with the reference to a template T and the number of behavioral units within template T. In other words, the number of non-verbal templates specified in the list provided to the RG and the one actually present in the selected template have to agree in number. Likewise, the number of frames and the templates has to agree in number.

These conditions make the following templates syntactically incorrect and would make the RG issue an error message upon their reception:

(Input 3): *[Emotional Values, [T2], [T4, B2], [**T1, B1**], F1, F2, F3]*

(Input 4): *[Emotional Values, [T2], [T4, B2], [T1, B1, B3], **F1, F2**]*

(Input 3) is syntactically wrong as the pointer to T1) is passed along with only one behavioral unit B1 while T1) contains two of these. Similarly, (Input 4) is also syntactically wrong because only two frames are passed on while three templates are provided. To fill in the slots for tag [S1] in T1) and T3) we use context information extracted from the third frame. Please also note that the behavior templates do not have entries/values referring directly to emotions like, e.g., sad or happy. They are only mere movement descriptions as emotions influence the rendering of those movements via the *Emotional Values* parameter provided by the CM.



```
-----------
SPEECH TEMPLATE #17 : [G0:start] This is a {name} of [G0:end] {obj} [G1:start] [G1:end]
-----------
GRAPHICAL TEMPLATE G0 : seq( eye_xml(X, opening, Eye, StartTime, EndTime, Amount)
                             eye_xml(Y, opening, Eye, StartTime, EndTime, Amount)
                           )
-----------
GRAPHICAL TEMPLATE G1 : seq( par(eyebrow_xml(X, raise, Eye, StartTime, EndTime, Amount)),
                             par(eyebrow_xml(Y, raise, Eye, StartTime, EndTime, Amount)))
                           )
-----------
INPUT TEMPLATE : [t17 g0 g1]
-----------
INPUT FRAME : [name="picture" obj="thumbelina"]
-----------


<animation type="sequential">

<animation type="parallel">
<animation type="sequential">
<eyeOpening with=" both" amount="1.2" startTime="2.2" endTime="8.1">
<eyeOpening with=" both" amount="1.2" startTime="2.2" endTime="8.1">
</animation>
This is a picture  of
</animation>
thumbelina
<animation type="parallel">
<animation type="sequential">
<animation type="parallel">
<eyebrowRaise with=" left" amount="8" startTime="3.2" endTime="5">
</animation>
<animation type="parallel">
<eyebrowRaise with=" left" amount="8" startTime="3.2" endTime="5">
</animation>
</animation>

</animation>

</animation>
Time elapsed in millisecs: 10.000000
Press anv kev to continue_
```

**Figure 6.** Template 17 is stored in the KB and so are the graphical templates G0 and G1. The frame is used to extract the information necessary to fill in the two tag slots present in the definition of template 17. The RG Prolog module is fed with the string defined as INPUT TEMPLATE while the results returned is the XML string starting with <animation type="sequential"> and ending with </animation>.

### 4.2.2 RG API

The RG provides the following functions to the CM.

#### *4.2.2.1 send_response(emotions, reply_list, frame)*

This function is called with the reply_list, a frame and the values for the emotional state as arguments. The CM calls the function when HCA is replying to the user. Once we have this list, we have to parse it to obtain the text output from the database. This function has been developed in C/C++.

#### *4.2.2.2 send_response(emotions, reply_list)*

This function is called with the reply_list and the values for the emotional state as arguments when HCA is listening to the user, i.e. when he is in what is referred to as the HCA communicative function state (CF). This function has been developed in C/C++.

#### *4.2.2.3 send_response(reply_list)*

This function has the reply_list as argument. It is called to start the set of behaviors corresponding to the HCA non-communicative action (NCA) state, i.e. when HCA is in an idle state. This function has been developed in C/C++.

## 4.3 Response generator internal processing

### 4.3.1 Interaction of Prolog and C++

The C++ module calls the Prolog module [1] to retrieve the XML output to be sent to the message dispatcher. The C++ module calls a Prolog predicate with an input string, emotional value and slots to be filled in the templates. The format for the frame and the input format of the slots for the Prolog predicate have been described in Section 4.2.1.2.

### 4.3.2 Prolog Module

#### *4.3.2.1 Speech Templates*

Templates are stored as Prolog facts of the type *general_template(Number, String)* where *Number* indicates the template ID as stored in the database and *String* its explanation. Here is an example:

general_template(1,'[G0:start] Hi [G0:end] welcome to [G1:start] my study [G1:end]').

Templates are used by the response generator to create text and non-verbal animation.

#### *4.3.2.2 Graphical Template Retrieval*

Once the speech template has been provided, its text representation *String* (see Section 4.2.1.1) needs to be expanded in all its components. First the graphical elements are considered. For example, in the general template of section 4.2.1.1 any occurrence of [G0:start] and [G1:start] is expanded as graphical templates. We used Prolog for this.

The main predicate has to call a graphical predicate to retrieve the correct graphical templates. It calls the graphical predicate via the *create_graphical_predicate(Number, FinalXMLTemplate)* with graphical identifiers *Number* as input parameters to retrieve the graphical XML associated with a particular graphical tag. A database of templates needs to be predefined and stored. As the code is written in Prolog, the database consists of a collection of facts. *FinalXMLTemplate* contains the final XML representation for the graphical template that, for example, for the gestural template *B4) Negation* looks like:

```
<animation type="parallel" id="25">
      <define><headTurnHorizontal to  = "left" amount = 0.5>
            <animIdentifier>animId_09<animIdentifier>
      </headTurnHorizontal></define>
      <define><headTurnHorizontal to  = "right" amount = 1.0>
            <animIdentifier>animId_09<animIdentifier>
      </headTurnHorizontal></define>
      <define><headTurnHorizontal to  = "left" amount = 1.0>
            <animIdentifier>animId_09<animIdentifier>
      </headTurnHorizontal></define>
      <define><headTurnHorizontal to  = "right" amount = 0.5>
            <animIdentifier>animId_09<animIdentifier>
      </headTurnHorizontal></define>
</animation>
```

**Figure 7.** XML representation for the *B4) Negation* gestural template.

### *4.3.2.3 Modification of Graphical Behaviors with Emotional Values*

HCA's current emotional state provided by the CM is used to modify the graphical behaviors of HCA. This is done in two methods. First, we can modify the parameterized graphical behaviors. Secondly, we can add graphical behaviors on top of the behavior list provided by the CM. The two methods are described in detail below. The first prototype does not have any parameterised animations, as these will be added later once this facility is available from the graphical system. Emotions are rendered using animations whose parameters are fixed and not tunable, i.e. there is a sad or happy state yet these emotional traits are constant and do not have any modulations.

### *4.3.2.3. Modifying Parameterized Graphical Behaviors*

The current emotional state of HCA is used to modify the parameters of the graphical XMLs. If we take the case of the *Smile* graphical behavior, the amount of lip-opening can be modified to control the facial expression of the smile. If HCA is happy, then this value will be high. Likewise, if he is in a neutral mood then the value will be in a middle range and, in general, the worse HCA's mood the lower the value. The exact number and nature of functions will have to be tuned after experimentation with the graphical setup.

```
<define>smile startTime = “” endTime = ””>
      <animation type = ”sequential”>
                    <animation type = ”parallel”>
                              <eyebrowRaise  with  =  ”both”  amount  =  “0.8”
                              startTime = “50” endTime = “100” />
                              <eyeOpening with = ”both” amount = “1” startTime
                              = “50” endTime = “100” />
                        </animation>
                        <animation type = “parallel”>
                              <lipOpening amount = ”0.8” >
                        <animation>
      </animation>
</smile></define>
```

**Figure 8.** XML representation with parameterized graphical elements.

#### *4.3.2.4 Adding Graphical Behaviors*

A set of graphical behaviors will be added on top of the list provided by the CM. For example, if HCA is very happy as indicated by the emotional value provided by the CM, we can insert a *Smile* behavior to show that he is happy.

#### *4.3.2.5 Combining Speech and Graphical XMLs*

Once the graphical elements are expanded into an XML representation they must be combined with the speech part of the template. Also this part is developed in Prolog via the function *rg_getXML(list_of_templates, list_of_frames, FinalXMLRep)* which also takes care of inserting the information from the frames into the templates. As result, the function saves an XML representation of the template into the Prolog variable *FinalXMLRep*.

Figure 6 shows an example of *FinalXMLRep* string and the intermediary steps necessary to produce it. A complete description of the XML format for each single animation can be found in NICE Report D4.2.

#### *4.3.2.6 Adding Turn-Taking Behavior*

Turn-taking behavior, which may or may not include posture changes, will have to be inserted at the beginning and end of the sentence.

#### *4.3.2.7 Adding Eye-Blink Information*

Apart from the template list received from the CM, we may have to take care of eye blink behaviors for HCA. If this has to be taken care of by the RG, then eye-blink behaviors will have to be inserted at appropriate places.

#### *4.3.2.8 Graphical Elements Associated with slots*

At this point, the variable slots to be filled in the speech template are not associated with graphical behaviors. If, in later stages of development, we find that there is a need for having graphical behaviors associated with these slots, we will have to define a methodology to do that in the Response Generator. The current implementation of the RG does not take care of that.

The C++ module receives the final XML from the Prolog module. It then sends this output to the message dispatcher listening at address ‘xx.xx.xx.xx’, port ‘yy’. The final output looks like this:

```
<animation type=”sequential”>
      <animation type=”sequential”>
            <lookAt at=”photograph_thumbelina”/>
            <pointAt with=”leftHand” at=”photograph_thumbelina”/>
            <lookAt at=”user”/>
      </animation>
      that photograph
      <emotion state = “smile” level = “0.8”>
      is of Thumbelina
</animation>
```

**Figure 9.** XML representation for combined animations.

# 5 References

Allwood, J., Nivre, J. & Ahlsén, E. (1991). On the Semantics and Pragmatics of Linguistic Feedback. Gothenburg Papers in Theoretical Linguistics No. 64. University of Gothenburg, Department of Linguistics, Sweden.

Arafa et al (2002) ” Two approaches to Scripting Character Animation”, http://www.vhml.org/workshops/AAMAS/papers/Kamyab.pdf

Beard and Reid (2002) “MetaFace and VHML: A First Implementation of the Virtual Human Markup Language” http://www.vhml.org/workshops/AAMAS/papers/beard.pdf

Beskow J, Edlund J, Nordstrand M (2003) “A model for generalised multi-modal dialogue system output applied to an animated talking head”, in Minker, W., Bühler, D. and Dybkjaer, L. (eds) Spoken Multimodal Human-Computer Dialogue in Mobile Environments, Dordrecht, The Netherlands, Kluwer Academic Publishers.

Clark, H., H (1994) “Managing problems in speaking”, Speech Communication, 15:243-250.

Dale, R., and Mellish, C. 1998. Towards evaluation in natural language generation. Pages 555–562 of: Rubio, A., Gallardo, N., Castro, R., & Tejada, A. (eds), Proceedings of the First International Conference on Language Resources and Evaluation. Also http://citeseer.nj.nec.com/article/dale98towards.html

Ehud Reiter and Robert Dale. Building applied natural language generation systems. Natural Language Engineering, 3:57--87, 1997. http://citeseer.nj.nec.com/reiter97building.html

Gustavsson, Strindkund and Wiknertz (2001) “VHML draft”, http://www.vhml.org/downloads/VHML/vhml.pdf

Huang, Eliens and Visser (2002) “XSTEP: A Markup Language for Embodied Agents”, http://www.cs.vu.nl/~eliens/projects/@archive/refs/xstep2.pdf

Kshirsagar et al (2002) “Avatar Markup Language”, http://www.miralab.unige.ch/IgnasiCFTEST/DynamicSite/3research/assdb/papers/96.pdf

Marriot and Stallo (2002) “VHML – Uncertainties and Problems. A discussion…” http://www.vhml.org/workshops/AAMAS/papers/marriott.pdf

Matthew Conway et al (2000) “Alice: Lessons Learned from Building a 3D System for Novices”, http://www.alice.org/publications/pubs/chialice.pdf

Mixing C and Prolog http://www.cs.bham.ac.uk/~pjh/Prolog_course/sicstus_manual_v3_5/sicstus_11.html