Deliverable D5.2

Shell for Emotional Expression

Elisabeth André, DFKI
Yasmine Arafa, ICSTM
Monika Fleischmann, Fraunhofer IMK
Patrick Gebhard, DFKI
Weidong Geng, Fraunhofer IMK

Abstract
This document constitutes Deliverable D5.2 of the IST Project SAFIRA, on the Shell for Emotional Expression. It presents the implementation details of components for affective expression to be delivered as part of the SAFIRA toolkit.

Keywords
Emotions, Affect Components, Affective Expression, Facial Expression, Bodily Expression, Affective Graphics, Affective Speech, Inter-Agent Communication.

Document Id: D-SAFIRA-WP5-D5.2
Commission Ref.: D5.2
Classification: project&commission
Distribution: project&commission
Date: October 7th, 2002
Deliverable D5.2
Shell for Emotional Expression

<table>
<thead>
<tr>
<th>Deliverable Number:</th>
<th>D-SAFIRA-WP5-D5.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version:</td>
<td>1.0</td>
</tr>
<tr>
<td>Contractual Date of Delivery to CEC:</td>
<td>August 31st, 2002</td>
</tr>
<tr>
<td>Actual Date of Delivery to CEC:</td>
<td>October 7th, 2002</td>
</tr>
<tr>
<td>Editor:</td>
<td>WP5</td>
</tr>
<tr>
<td>Deliverable Type:</td>
<td>P-Public</td>
</tr>
<tr>
<td>Deliverable Nature:</td>
<td>R-Report</td>
</tr>
<tr>
<td>Workpackages Contributing:</td>
<td>WP5</td>
</tr>
<tr>
<td>Partners Contributing:</td>
<td>INESC-ID, DFKI, FhG, ICSTM</td>
</tr>
</tbody>
</table>

This document is a Deliverable of the SAFIRA project (“Supporting Affective Interactions for Real-time Applications”), a project performed within the IST programme (project identifier IST-1999-11683).

The SAFIRA project is performed by a consortium consisting of the following partner organisations:

**INESC-ID** – Instituto de Engenharia de Sistemas e Computadores (P, Coordinating Partner)

**ADETTI** – Associação para o Desenvolvimento das Telecomunicações e Técnicas de Informática (P)

**DFKI** – Deutsches Forschungszentrum für Künstliche Intelligenz GmbH (D)

**FhG** – Fraunhofer-Gesellschaft (Previously GMD) (D)

**ICSTM** – Imperial College of Science, Technology and Medicine (UK)

**ÖFAI** – Austrian Research Institute for Artificial Intelligence (A)

**SICS** – Swedish Institute of Computer Science (SE)

Project Manager:

**Ana Paiva**

INESC-ID
Rua Alves Redol 9, 6º Esq.
P-1000-029 Lisboa
Portugal
Tel.: +351 21 3100219
Fax: +351 21 3145843
Email: ana.paiva@inesc.pt

Project Homepage:

http://gaips.inesc.pt/safira

Project Email:

safira@gaips.inesc.pt
Executive Summary

This Deliverable (D-SAFIRA-WP5-D5.2) is the second and final deliverable from WP5 (Communicating and Expressing Emotions) of the IST-sponsored SAFIRA project. It contains the implementation details of the components for affective expression being developed for the SAFIRA toolkit. The results described here are based on the work of all partners in Workpackage 5.

As defined in the Technical Annex, p. 38, there are 4 major objectives for this workpackage:

- Development of APIs and plug-ins to allow agents who are dynamically changing their behaviour and emotional state to communicate their emotions through existing facial animation packages.
- Development of tools to support the expression of emotions and personality through body movement.
- Development of tools to generate drawings in real-time which reflect an agent’s emotions and personality.
- Development of tools and techniques to use linguistic style to express emotion and personality.

The implementation details described here were based on the specifications of Deliverable 5.1, on research in each of the target application areas, and on the actual integration with the toolkit and demonstrators.

The objective of this deliverable is to provide the detailed documentation of the affective expression toolkit components that were developed and integrated in the SAFIRA Toolkit. The structure of this deliverable is as follows:

Chapter 1 describes the goal of this deliverable within the context of the entire SAFIRA project.

Then, Chapters 2 through 5 present the implementation details of the four components.

Finally, Chapter 6 draws some conclusions about each component, and the overall results of the Workpackage.
Table of Contents

1. Introduction ................................................................................................................ 9

2. Affective Facial Expression...................................................................................... 10
   2.1 Overview ................................................................................................................ 10
       2.1.1 Motivation....................................................................................................... 11
       2.1.2 Embodied Agents Markup Languages......................................................... 11
       2.1.3 Language Requirements .............................................................................. 12
       2.1.4 Dynamic Scripting of Expressive Behaviour............................................... 12
   2.2 Specification ........................................................................................................... 13
       2.2.1 Visual Behaviour Definition........................................................................... 14
       2.2.2 Classification of Motion .............................................................................. 14
       2.2.3 CML Specification....................................................................................... 16
       2.2.4 CML Representation Language .................................................................... 18
       2.2.5 CML Scripting Language ............................................................................ 20
   2.3 Architecture .......................................................................................................... 26
       2.3.1 Generating Script ......................................................................................... 26
   2.4 Use ........................................................................................................................ 27
   2.5 References ............................................................................................................ 28

3. Affective Body Expression ....................................................................................... 30
   3.1 Overview ................................................................................................................ 30
   3.2 Specification .......................................................................................................... 32
       3.2.1 Character Control ....................................................................................... 33
   3.3 Architecture ......................................................................................................... 38
       3.3.1 Geometry .................................................................................................... 39
       3.3.2 Animation Engine ..................................................................................... 46
       3.3.3 Behaviour Engine ..................................................................................... 57
   3.4 Toolkit Integration ................................................................................................. 58
   3.5 Use ........................................................................................................................ 59
       3.5.1 Modelling and Animation ........................................................................... 59
5.5.3 Dialogue Acts ................................................................. 105
5.5.4 Realisation ................................................................. 106
5.6 References ........................................................................ 112

6. Conclusions ........................................................................ 115
1. Introduction

The Workpackage 5 (Communicating and Expressing Emotions) of the IST-funded SAFIRA project defines four components of the SAFIRA Toolkit, which provide emotional expression along different modalities:

- **Affective Facial Expression:** Development of APIs and plug-ins to allow agents who are dynamically changing their behaviour and emotional state to communicate their emotions through existing facial animation packages;

- **Affective Bodily Expression:** Development of tools to support the expression of emotions and personality through body movement;

- **Affective Rendering:** Development of tools to generate drawings in real-time which reflect an agent’s emotions and personality:

- **Affective Speech:** Development of tools and techniques to use linguistic style to express emotion and personality.

The previous deliverable of WP5, Deliverable 5.1, provided the specifications for these components, and all the research background behind each of the approaches.

The present deliverable will focus on the implementation of the components taking into account the specification provided in D5.1, the integration in the SAFIRA Toolkit, and the use in the demonstrators provided by the project. It will look at the general software architecture behind each component, the functionalities provided by each interface, and the implementation techniques and algorithms that make everything work.
2. Affective Facial Expression

2.1 Overview

The delivery of facial expression as realised in the SAFIRA project is achieved through the specification of a Character Markup Language. As defined in the SAFIRA deliverable D-SAFIRA-WP2-D2.2a v2.0 the specifications of the Facial Expression component provisions for the definition of a representation and scripting language, effectively aiming at bridging the gap between the underlying emotion engines and agent animation tools. The decision not to focus on, yet another, animation tool was governed by the fact that there is much research currently being attempted at achieving believable, real-time facial expression and animation, and that a more significant contribution would be achieved by specifying a language that can serve as a glue like mechanism to tie the various visual and underlying behaviour generation tools together seamlessly, regardless of the platform that they run on and the language they are developed with.

In this regard the component here differs from the other components delivered on the SFAIRA project, specifically in the fact that it is not a proactive component issuing and receiving services dynamically. It is more like a “terminal” or “end of the pipe line” mechanism that is triggered on request. It aims at providing the language specification (both syntax and semantics) that will allow an automated, real-time generation of animation script that then is passed on to the rendering tool for the display of intended expression. This expression is not only facial but also facilitates for body expression too.

In this section we delineate the functions and specifications of a mark-up language, CML for scripting the animation of virtual characters in addition to facial expression. This multi-modal scripting language is designed to be easily understandable by human animators and easily generated by a software process such as software agents. CML is constructed based jointly on motion and multi-modal capabilities of virtual life-like figures. We further illustrate the constructs of the language and describe a real-time execution architecture (as implemented in the James the Butler demonstrator and is presented in SAFIRA-WP5-D5.2) that demonstrates the use of such a language as a 4G language to easily utilise and integrate Living Actor technology, MSAgents as well as support for MPEG-4 media objects in online interfaces and virtual environments.

To set the scene, we first set forth the motivations for the mark-up languages developed; describe the key features and capabilities they offer; and discusses the technical issues they raise based on our design and development experience in SAFIRA. The paper further sets forth the key functionality that such description and scripting languages will need to succeed in animated agent interaction applications. Finally, an implemented architecture, in which CML is integrated within the James the Butler demonstrator, is briefly described.
2.1.1 Motivation

Lifelike animated agents present ongoing challenging agenda for research in multimodal user interfaces and human-computer-interaction. Such agent metaphors will only be widely applicable to online applications when there is a standardised way to map underlying engines with the visual presentation of the agents.

The merits of embodied agents for different applications are a matter of current debate. Commercial attempts so far have been disappointing. Microsoft, for example, has been eagerly involved in efforts to drive multimodal interfaces with embodied representations, by developing the Microsoft Active Agent tool [Microsoft 1998]. Other commercial companies like Semantics have now caught on and are developing similar tools [Semantics]. However, while these tools are a relatively easy way to add some simple agent behaviour to web sites and applications and have proven to be useful for rapid prototyping, they remain very limiting to developers who require adding believability to the agents being used and reflect negatively on the merits of multimodal interfaces on the whole.

On the other hand current mechanisms for building cohesive believable characters under various theories are of even more importance. Efforts such as those by Elliott [Elliott 1993], Velázquez [Velázquez 1998], Reilly [Reilly & Bates 1993], Sloman [Sloman 1987], and others aid in the definition and planning of appropriate agent believable behaviour, but they do not aim at creating a “fourth generation language” for building them. Emotion and personality models for agents will only be widely applicable when there is an organised way to map design models to the effected representations. Current research at IIS aims at providing the tools for a 4G language to draw together the properties needed to generate embodied agents. The Character Mark-up Language (CML) is developed to bridge the gap between the available underlying engines and agent animation tools.

2.1.2 Embodied Agents Markup Languages

In recent years a number of attempts have been made to specify and develop mechanisms for the dynamic scripting of expressive behaviour in embodied characters. Early high-level control mechanisms for character animation include Improv [Perlin & Goldberg 1996], a rule based mechanism allowing animators to define how synthetic actors communicate and make decisions. Although Improv was successful in many ways, it’s use as a scripting mechanism was specific to the underlying implementation. However, Improv made the first realisation that a scripting mechanism was necessary that allowed authors of animations to define elements of a character’s expressed behaviour.

As parallel developments several animation tools and many emotion and behaviour models and their respective implementations have been produced. The appearance of
these has highlighted the need for powerful yet generic scripting languages to bridge the
gap between behaviour generation and animation tools. Notably Virtual Human Mark-
up Language (VHML) [VHML] and Human Mark-up Language (HumanML)
[HumanML] are examples of these.

As a number of such scripting languages now exist, there appears to be the need for the
research community to look at and agree upon the requirements of and expectations
from them.

2.1.3 Language Requirements

The design and formalisation process of any language needs to fulfil a set of baseline
consideration defined so as to meet the criteria for general use and implementation.

The following items are key design criteria for such languages:

- **Consistency**: Provision for predictable control of the animation output
regardless the implementation and the platforms they will be run on.

- **Generality**: Provision away from any dependency on any one prototype
character behaviour, warranting the possibility to cater for a wide range of
applications and animation tools.

- **Domain Independent**: not catering for any one domain, implementation
application or animation rendering tool.

- **Generation and Readability**: Support automated generation and human
authoring of representation and scripts. The generated output should be readable
also by humans.

- **Usable**: The language should be usable and easily implementatable with multi-
purpose applications and technologies.

2.1.4 Dynamic Scripting of Expressive Behaviour

2.1.4.1 Character Model

The character visual model is based on the MPEG-4 standard and includes two sub
models for Face and Body. MPEG-4 defines two sets of parameters to describe and
animate the character model: 1- Facial Animation Parameter Set (FAPS) and Body
Animation Parameter Set (BAPS); and 2 - Facial Definition Parameter (FDP) and Body
Definition Parameter (BDP). Base animation objects are created providing a generic
virtual face and/or body with neutral expression and a default body posture. The shape
of the virtual character is defined using the FDPs and BDPs, then facial and body
expression animation is defined by FAPS and BAPS. Real-time generation of
expression and animated behaviour is achieved by specifying the appropriate attribute values of the FAPS and/or BAPS for each frame.

The MPEG domain provides efficient coding of audio and video data to enable delivery and presentation of media episodes. MPEG-4 deals with both storing and sharing animated media objects whilst it also provisions for real-time generation and integration of animations. The actual coding of the audio and video objects are through traditional streaming of audio and video with time coding, and also the downloading of synthetic models and streaming update of their animation parameters. The specifications of spatial and temporal relationships between the objects and other streaming objects are defined by the CML decoder data, so that media composition is well-defined in the terminal as streamed objects animate jointly with their synthetic partners.

MPEG-4's model-based approach to the animation generation and transmission of facial/body information makes it suitable for real-time online applications and the implementation of multi-modal interactive character human-computer-interaction metaphors. With the small size of the expression and animation parameter set, compression ratios, that are one to two orders of magnitude better than using classical video compression, can be achieved without losing crucial content (e.g. the facial expressions and lip movements). Moreover, achieving high-level functionalities (like movement or scaling) is straightforward since the 3D models of the face and body are known. Stereoscopic visualization of a talking head or body model is also achievable without any bit-rate overhead. Body animation leverages all the same principles as facial animation for coding body models and their compressed parametric control streams during a session.

2.2 Specification

Animated lifelike character representations of agents will only be widely deployed when there are real-time mechanisms to contextually map character and affect models to effected animated personifications. To approach this the SAFIRA project utilises the design elements of an architecture for representing and scripting emotional responsiveness in an interface agent; semantic abstraction and annotation of the knowledge being manipulated; and mapping resulting semantic understanding onto appropriate behaviour; which is translated into animated expression by varying the planned response using variable emotion indicators and represented in the selected modality(ies).

Currently, there are several commercial and proprietary agent animation tools available (e.g. MS Agent, Jack, etc.), as well as, numerous emotion engines or mechanisms (Affective Reasoner, S&P, etc.) that are available to generate and govern believable behaviour of animated agents. However, there is no common mechanism or API to link the underlying engines with the animated representations until recently.

CML is developed with an aim to bridge the gap between the underlying Affect and process engines, and agent animation tools. CML provides a map between these tools.
using objects by automating the movement of information from XML Schema
definitions into appropriate relational parameters required to generate the intended
animated behaviour. This would allow developers to use CML as a glue-like mechanism
to tie the various visual and underlying behaviour generation tools together seamlessly,
regardless of the platform that they run on and the language they are developed with.

The term Character is used to denote a language that encapsulates the attributes
necessary for believable behaviour. The intention is to provision for characters that are
lifelike but are not necessarily human-like. Currently the attributes specified are mainly
concerned with visual expression, although there is a limited set of specification for
speech. These attributes include specifications for animated face and body expression
and behaviour, personality, role, emotion, and gestures.

2.2.1 Visual Behaviour Definition

Classification of behaviour is governed by the actions an agent needs to perform in a
session to achieve given tasks, and is influenced by the agent’s personality and current
mental state. A third factor that governs character behaviour is the role the agent is
given. A profile of both an agent’s personality and its role are used to represent the
ongoing influences to an agent’s behaviour. These profiles are user-specified and are
defined using XML annotation. The behaviours are defined as XML tags, which
essentially group and annotate sets of action points generally required by the intended
behavioural action. The CML processor will interpret these high-level behaviour tags,
map them to the appropriate action point parameters and generate an animation script.

The Character Mark-up Language defines the syntactic, semantic and pragmatic
character presentation attributes using structured text. CML is based on the definition
XML Schema structures. The character mark-up-based language extends the
descriptions for facial expressions used in the FACS system. FACS (Facial Action
Coding System) defines a set of all the facial movements performed by a human face
[Ekman & Rosenberg 1997]. Although FACS is not an SGML-based language in
nature, we use their notion of Action Units to manipulate expressions. Character gesture
attribute definitions are based on the research and observations by McNeill [McNeill
92] on human gestures and what they reveal.

Affective expression is achieved by varying the extent and degree values of the low-
level parameters to produce the required expression. The CML encoder will provide the
high-level script to be used in order to specify the temporal variation of these facial
expressions. This script will facilitate designing a variety of time-varying facial
expressions using the basic expressions provided by the database.

2.2.2 Classification of Motion

Along the lines defined in the Body Expression section, we assume a motor generation
module which is responsible for the basic movements along with correlated transitional
movements that may occur between them. Personified animation scripts are generated
by blending specification of different head or body poses and gestures. The base motions are further classified by generic controls that are independent of the character itself. For example a generic move motion can have different representations which are determined by the character emotional and/or personality attributes defined to represent nod, iconic gesture head, hand or body gesture, walk, etc. Additionally, the language motion categories should cater for the fact that behaviour can be expressed through and can affect different parts of the character face (or body part). To realise different parts of a character head/body skeleton which are to be affected while performing a movement CML divides the character element specifications into four units: Head, Upper, Middle and Lower parts. CML then provide specification of the constructs of each unit with varying granularity.

Action composition script is generated by a CML processor (delineated in Figure 3.1.1) which blends actions specified with an input emotion signal to select the appropriate gestures and achieve the expressive behaviour. CML also provisions for the generation of compound animation script by facilitating the definition and parameterisation of sequences of base movements.

The chosen base set of movements allows basic character control (movement and interactions) as well as assures the capability to perform unlimited character specific animations.

**Base Motions:** The initial set of the CML base motions are classified by the goal of the motion into: Movement, Pointing, Grasping, Gaze and Gesture as follows:

**Movement** defines motions that require the rotation or movement of a character from one position to another. Positions are defined by exact coordinates, an object position or a character position. Movement elements are either glance-at, nod-to turn-to or move-to.

**Pointing** defines a pointing gesture towards a coordinate, object or character. Pointing elements are point-to.

**Grasping** defines motions that require the character to hold, throw or come in contact with an object or another character. Grasping elements are grasp, throw and touch.

**Gaze** defines the movements related to the head and eyes. Gaze elements are gaze, track, blink, look-to and look-at. The gaze and track elements requires that only the eyes be moved or track an object or character. look-to and look-at require the movement of both head and eyes.

**Gesture** includes motions that represent known gestures like hand movements to convey an acknowledgement, a wave, etc. Gesture elements are gesture and gesture-at

Following is an extract of the CML base movement specifications. It shows a move-to motion. The gesture and behaviour in which the movement is made is inherited from the state of emotion, gesture and behaviours specified. Further details on the granularity and
specification of CML may be found in public SAFIRA deliverable D-SAFIRA-WP2-D2.2v2 [André et al. 2002].

```
<move-to>
  <order {0 to n/before/after} />
  <priority {0 to n} />
  <speed {0.n to n.n(unit)/default/slow/fast} />
  <target {x,y,z/object/character} />
  <begin {ss:mmm/before/after} />
  <end {ss:mmm/before/after} />
  <repeat> {0 to n/dur} />
  <transitionAction> behaviour sequence /
  <interrupt> {yes/no} />
</move-to>
```

Sample CML Base Motion Syntax

### 2.2.3 CML Specification

CML defines a script like that used for a play. It describes the actions and sequence of actions that will take place in a presentation system. The script is a collection of commands that tell the objects in the world what to do and how to perform actions. The language is used to create and manipulate objects that are held in memory and referenced by unique output-ontology objects. The structure of the language begins with a command keyword, which is usually followed by one or more arguments and tags. An argument to a command usually qualifies a command, i.e. specifies what form of action the command is to take, while a tag is used to denote the position of other necessary information. A character expression mark-up module will add emotion-based mark-up resulting from emotional behaviour generation rules to the CML descriptions.

Animated character behaviour is expressed through the interpretation of XML Schema structures. These structure definitions are stored in a Schema Document Type Definition (DTD) file using XSDL (XML Schema Definition Language). At run-time character behaviour is generated by specifying XML tag/text streams which are then interpreted by the rendering system based on the rules defined in the definition file. Its
The objective is to achieve a consistent convention for controlling character animation models using a standard scripting language that can be used in online applications.

The language contains low-level tags defining specific character gesture representations defining movements, intensities and explicit expressions. There are also high-level tags that can define commonly used combinations of these low-level tags.

```xml
<cml>
  <character name="n1" personality="p1" role="r1" gender="M" disposition="d1" transition_Dstate="t1">
    <happy intensity="i1" decay="dc1" target="o1" priority="pr1">
      <move-to order="o1" priority="pr2" speed="s" object="obj1"
               begin="s1" end="s4"/>
      <point-to order="2" priority="pr3" object="obj1" begin="s2"
                 end="s4"/>
      <utterance priority="pr2" begin="s2">
        UtteranceText
      </utterance>
    </happy>
  </character>
</cml>
```

Sample CML script

Synchronisation between the audio and visual modalities is achieved through the use of SMIL (Synchronized Multimedia Integration Language) specification [SMIL]. SMIL defines an XML-based language that allows authors to write interactive multimedia presentations. Basically, CML uses the SMIL `<par>` and `<seq>` tags to specify the temporal behaviour of the modalities being presented. The `<seq>` tag to define the order, start time and duration of execution of a sequence, whereas the `<par>` tag is used to specify the elements be played in parallel. For further flexibility, CML also provides order and time synchronisation attributes the motion and audio elements defined.
2.2.4 CML Representation Language

CML provide a set of base description/representation languages that are integrated with the face and body animation Markup languages enabling these multimodal features in a hybrid representation architecture.

2.2.4.1 Head Gesture Taxonomies

The gesture classifications here are defined as follows:

- **Symbolic**: which relate to universal symbols and commonly acknowledged gestures across cultures (e.g., OK/agree or No/disagree);
- **Iconic**: gestures that are used to demonstrate a symbolisation of a particular behaviour or action (e.g., rotating the head showing that one is dizzy);
- **Deictic**: gestures that point in the direction of objects;

2.2.4.2 Hand Gestures Taxonomy

CML captures gesture features such as postures of the hand (straight, relaxed, closed), its motion (moving, stopped), and its orientation. Over time, a stream is of gesture is then abstracted into more general gestlets (e.g., Point-at, sweep, end reference). The term and concept of 'gesture' is also used in pen computing. Although here, the recorded action takes place in the 2D plane, similar phenomena play a role as in the case of 3D hand gesturing, but with a much easier signal processing involved.

- **Posture**: Defining the position the hand is held in and a duration.
- **Motion**: Defining whether the hand is in motion or stagnant and a speed defining the transition between each state.
- **Orientation**: Defining the direction of a movement (up, down, left, right, forward, backward). Directions which are derived from normal and longitudinal vectors of the palm.
- **Gestlets**: Define a set of high level tags that are constituted from the lower-level tags described above. These make up gestures like Point, Wave…
- **Fingers**: Defines high level tags for each five fingers.

2.2.4.3 Body Gestures Taxonomy

The base of the research for the base body gestures is partly derived from the work conducted in the Body expressions section. The reader may which to refer to this section for more detail on the research and analysis there.
• **Natural**: Defines the characters default or normal posture state based on a distinct personality.

• **Relax**: Defines a relaxed posture state of the character.

• **Tense**: Defines a tensed posture state of the character.

• **Iconic**: Gestures that are used to demonstrate a symbolisation of a particular behaviour or action (e.g., rotating the head showing that one is dizzy)

• **Incline**: Defines the orientation and degree a character might lean towards or against and object.

### 2.2.4.4 Emotions

The emotions are derived from the set of specification as defined in SAFIRA-W-P2-D2.1. The table below shows the specification for the attributes for an emotion. The attributes considered for the description of an emotion, are according with the OCC theory of emotions [Ortony et al. 1988], which was set to partially supporting theory for the system implementation.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>The id of the emotion class being experienced</td>
</tr>
<tr>
<td>Valence</td>
<td>Denotes the basic types of emotional response. Neutral, positive or negative value of the reaction.</td>
</tr>
<tr>
<td>Subject</td>
<td>The id of the agent experiencing the emotion</td>
</tr>
<tr>
<td>Target</td>
<td>The id of the event/agent/object towards the emotion is directed</td>
</tr>
<tr>
<td>Intensity</td>
<td>The intensity of the emotion. A logarithmic scale between 0-10</td>
</tr>
<tr>
<td>Time-stamp</td>
<td>The moment in time when the emotion was felt</td>
</tr>
<tr>
<td>Origin</td>
<td>The component that originated the emotion.</td>
</tr>
</tbody>
</table>
The attribute Class describing an emotion refers to the type of that emotion. An emotion type represents a family of related emotions differing in terms of their intensity and manifestation, i.e., each emotion type can be realized in a variety of related forms. For example, fear with varying degrees of intensity can be seen as—concern, fright, petrified.

Initially CML defines only six emotions of the OCC model:

- Joy/Happiness
- Distress/Sadness
- Surprise (which does not exist in the classes of the OCC model)
- Gloat
- Anger
- Fear

The attribute Valence describes the value (positive or negative) for the reaction that originated the emotion. According to this theory, emotions are always a result of positive or negative reactions to events, agents or objects.

The Subject and Target attributes for emotions, define the entities related to them. The Subject defines the agent experiencing the emotion and the Target defines the event, agent or action that originated the emotion.

2.2.4.5 Personality

Is our intention to include a specification that defines personality and character roles, and relates their attributes to the animation scripting structure, but due to the lack of sufficient time this was not specified. Our intention remains that we shall provide specification and representation for these attributes and will continue to work on it even after the SAFIRA project completion.

2.2.5 CML Scripting Language

2.2.5.1 CFML Character Face Markup Language

Character face description is a set of low-level tags based on MPEG-4 BAPs, and a set of high-level tags representing facial parts which are grouped from a set of respective low-level tags. CFML base elements are as follows:

- Head Movement
  - Tilt

    `<tilt-to>`

    `<order {0 to n/before/after} />`

    `<priority 0 to n />`
<speed {0.n to n.n(unit)/default/slow/fast} />
<target {object/character} />
<degree (n\%) />
<direction (forward/backward/rightside/leftside/n angle) />
<begin {ss:mmm/before/after} />
<end {ss:mmm/before/after} />
<repeat {0 to n/dur} />
<transPos {eye,brow,head groups} />
<transSpeed default/slow/intermediate/fast />
<interrupt> {yes/no} />

</tilt-to>

• Turning

<turn-to>

<order {0 to n/before/after} />
<priority 0 to n />
<speed {0.n to n.n(unit)/default/slow/fast} />
<target {x,y,z/object/character} />
<bodyPart (head,upper,middle,lower) />
<begin {ss:mmm/before/after} />
<end {ss:mmm/before/after} />
<repeat {0 to n/dur} />
<transPos {eye,brow,head groups} />
<transSpeed default/slow/intermediate/fast />
<interrupt> {yes/no} />

</turn-to>

• Head Gesture

• Symbolic

<disagree>

<tilt-to-right>

<order {0 to n/before/after} />
<priority 0 to n />
<speed {0.n to n.n(unit)/default/slow/fast} />
<target {object/character} />
<degree (n\%) />
<direction {rightside} />
<begin {ss:mmm/before/after} />
<end {ss:mmm/before/after} />
<repeat {0 to n/dur} />
<interrupt> {yes/no} />
</tilt-to right>

<tilt-to-left>
<order {0 to n/before/after} />
<priority 0 to n />
<speed {0.n to n.n(unit)/default/slow/fast} />
<target {object/character} />
<degree (n\%) />
<direction {leftside} />
<begin {ss:mmm/before/after} />
<end {ss:mmm/before/after} />
<repeat {0 to n/dur} />
<interrupt> {yes/no} />
</tilt-to left>
</disagree>

- Deictic
- Brow Movement
  <browLeft>
  <order {0 to n/before/after} />
  <priority 0 to n />
  <speed {0.n to n.n(unit)/default/slow/fast} />
  <degree n} />
  <begin {ss:mmm/before/after} />
  <end {ss:mmm/before/after} />
  <repeat {0 to n/dur} />
</browLeft>
<transPos {eye,brow,head groups} />
<transSpeed default/slow/intermediate/fast />
<interrupt> {yes/no} 
</browLeft>

<browRight>

<order {0 to n/before/after} />
<priority 0 to n /> 
<speed {0.n to n.n(unit)/default/slow/fast} />
<degree n} />
<brbegin {ss:mmm/before/after} />
<brend {ss:mmm/before/after} />
<brrepeat {0 to n/dur} />
<brtransPos {eye,brow,head groups} />
<brtransSpeed default/slow/intermediate/fast />
<brinterrupt> {yes/no} 
</browRight>

• Mouth
• Gaze

gaze-at

<order {0 to n/before/after} />
<priority 0 to n /> 
<object objID />
<speed {0.n to n.n(unit)/default/slow/fast} />
<target {x,y,z/object/character} />
<brbegin {ss:mmm/before/after} />
<brend {ss:mmm/before/after} />
<brrepeat {0 to n/dur} />
<brtransPos {eye,brow,head groups} />
<brtransSpeed default/slow/intermediate/fast />
<brinterrupt> {yes/no} 
</gaze>
Expression

• Movement

2.2.5.2 CBML Character Body Markup Language

Character body description is a set of low-level tags based on MPEG-4 BAPs, and a set of high-level tags representing body parts which are grouped from a set of respective low-level tags. CBML base elements are as follows:

• Movement
  • Moving

  <move-to>
  <order (0 to n/before/after) />
  <priority 0 to n />
  <object objID />
  <speed (0.n to n.n(unit)/default/slow/fast) />
  <target (x,y,z/object/character) />
  <begin (ss:mmm/before/after) />
  <end (ss:mmm/before/after) />
  <repeat (0 to n/dur) />
  <transAnimat(body elements) />
  <transSpeed 0.n to n.n(unit)/default/slow/intermediate/fast>
  <interrupt> [yes/no] />
  </move-to>

  <walk-to>
  <order (0 to n/before/after) />
  <priority 0 to n />
  <object objID />
  <target (x,y,z/object/character) />
Turning

<turn-to>

<order {0 to n/before/after} />
<priority 0 to n />
<bodyPart (upper/lower/middle) />
<object objID />
<speed {0.n to n.n(unit)/default/slow/fast} />
<target {x,y,z/object/character} />
<begin {ss:mmm/before/after} />
<end {ss:mmm/before/after} />
<repeat {0 to n/dur} />
<transAnimat{body elements} />
<transSpeed 0.n to n.n(unit)default/slow/intermediate/fast>
<interrupt> {yes/no} />
</turn-to>

<turn-upperbody to>

<order {0 to n/before/after} />
<priority 0 to n />
<object objID />
<speed {0.n to n.n(unit)/default/slow/fast} />
<target {x,y,z/object/character} />
<begin {ss:mmm/before/after} />
<end {ss:mmm/before/after} />
<repeat {0 to n/dur} />
<transAnimat{body elements} />
<transSpeed 0.n to n.n(unit)default/slow/intermediate/fast>
<interrupt> {yes/no} />
</turn-to>
• Gesture
  • Iconic
  • Symbolic
  • Deictic
  • Pointing

• Grasping

• Posture (expression)

We define a set of high-level tags representing general body gestures.

  • **Natural**: Defines the characters default or normal posture state based on a distinct personality.

  • **Relax**: Defines a relaxed posture state of the character.

  • **Tense**: Defines a tensed posture state of the character.

  • **Incline**: Defines the orientation and degree a character leans towards or against and object.

2.3 Architecture

2.3.1 Generating Script

Script generation through to the effected animation process components consist of a set of MPEG-4-compliant facial and body models; high level XML-based descriptions of compound facial and body features; XML-based descriptions of user-specified personality models; behaviour definitions, a CML processor, and finally a CML decoder. The general function of this component is delineated in Figure 2-1.

The architecture of an implementation generating and using CML is divided into three conceptual components of the supporting models and database for face and body animation, CML scripting and an animation rendering tool.
The script generation component assumes state and contextual input resulting for the underlying affective processing, planning, and domain knowledge-base engines. Based on these inputs and a defined character personality, the CML processor then generates the consequent synchronised behavioural action and utterance CML script. The script is then passed onto the CML decoder which parses the CML and maps its elements onto view-specific commands for final animation rendering.

2.4 Use

The hands-on use of CML integrated with the supporting components and attempting to use different animation rendering tools in James the Butler demonstrator raised many questions regarding representation and scripting languages in general. These were listed in D-SAFIRA-WP7-D7.2 and are listed here to complete out final report and elaborating on the issue of the development and standardisation of affective mark-up languages:

- What level of granularity is required for facial and body animation points and parameter definitions?

- What taxonomy for the mark-up tag definitions should be adopted? Are current taxonomies sufficient for the requirements of believable affective behaviour animation delivery? Should a standard research the possibility of a new taxonomy?

- Are Mpeg 4 FAPs and BAPs sufficient or to granulated to provide the taxonomy required for real time animation? Given the argument that there are no Mpeg 4 players or viewers available commercially and are currently only confined to research labs and benches, is it prudent to use Mpeg 4 defined Animation
Parameters as a standard for affective mark-up languages when Mpeg 4 itself is arguably a failing standard?

- What affective and personality theories should adopted to define the tags for affective expression? Should the choice be combined to only one theory or should there be a combination of existing theories? What granularity of affective description is required?

These questions opened a significant line of discussion on the formalisation and the possibility of standardisation of such languages at the IST EU cross-project meeting on specifying representing embodied agents held in conjunction with the AMAAS 2002. This has open the path for further discussions with other researchers in this area namely AML, VHML and STEPS. This serves as positive feedback on this direction of research into such languages and manifests the significance of the area to the embodied agent community. CML will continue to be enriched and tested within various demonstrators in different domains even after the completion of the SAFIRA. Discussions with the MPML designers have been instigated to provide material on comparisons of similar languages and their specification.

2.5 References


[SMIL] SMIL – http://www.w3.org/AudioVideo/


3. Affective Body Expression

3.1 Overview

The modelling and animation of synthetic characters has only recently occupied a relevant place in the efforts of the scientific community. In fact, a few years ago, computer animation was far too expensive to be available to the common researcher. The groups who had the resources to conduct a serious work in the field were usually funded by governmental institutions with very specific interests and objectives. The research on synthetic characters was purely functional and domain specific. However, the rapid evolution of the hardware platforms allowed a sudden increase of computational performance allied with a constant decrease of costs. The research on synthetic characters became more generic and started to serve non-specific purposes.

Today, the biggest challenge in the area is how to develop a systematic, accessible and fast method for creating reusable and believable synthetic characters. There are two different approaches for achieving this goal: the traditional approach and the behavioural approach. The traditional approach follows the long-established principles of computer graphics in which the visual realism is the ultimate goal. It looks at detailed geometrical models and advanced animation techniques [Badler et al. 1993] [Fua et al. 1998] [Kalra et al. 1998] [Aubel & Thalmann 2000]. The behavioural approach relegates the visual realism to a second plane and seeks mainly the behavioural realism. The idea is that believability depends more on the character’s ability of conveying a certain inner live than on a strict visual realism. That way, it focuses on high-level architectures for real-time animation and interactive control [Perlin 1995] [Blumberg & Galyean 1995] [Perlin & Goldberg 1996] [Russell & Blumberg 1999] [Johnson et al. 1999].

At the same time, in the last years, human-like or more general articulated figures have been playing an increasing role in many areas, such as advertising, entertainment, education, and simulation. Many efforts have been spent on developing articulated body models, modelling techniques, or motion generation methods. Many theories characterize each type of motion and its purposes. However, human-like animation is still giving its first steps when compared to real human motion. In fact, when we look to the most recent works on believable humans and real-time motion generation, we quickly find some detail that doesn’t look right. Yet, it is not a computer animation limitation per se, because skilled key pose animators are able to produce excellent animations, and there are motion capture techniques that preserve all the details of real motion. What factors lead us to mark a performance as synthetic or real?

The problem is that human movements vary from person to person. They reflect not only the personality, but also inner feelings, in particular emotions. We do not walk always the same way. We walk slowly, if we are tired or sad, and we walk with more energy, if we are happy or determined. How can we identify and capture these
differences? How can we adapt a movement to reflect personality and emotions? How can we introduce some expressiveness in a movement?

Many researchers have tried to address these questions from different perspectives. Some approaches propose secondary motions as a way to add naturalness to primary motions [Perlin 95]. Others are based on signal analysis and synthesis techniques [Unuma et al. 95] [Amaya et al. 1996]. More recently, results from movement observation science have been used to define models to parameterise movements [Badler et al. 2000] [Chi et al. 2000].

The Affective Body Expression component generates expressive body movements by modifying the animations of the synthetic character in real-time. This can be done using three features supported by the proposed architecture:

- Real-time composition of stances with the active animation;
- Real-time increase or decrease of the speed of the animations;
- Real-time change of the spatial amplitude of the animations.

The use of stances comes from an empirical idea often used in the cartoons that associates each emotional state with a particular pose. Take as an example a sad movement. Sad movements are usually associated with dismaying poses, like bending the torso or the head. Combining this pose with, for instance, a walk movement generates in real-time a modified walk movement that reflects sadness. In addition, the weight of the stance can be used to denote the intensity of the emotional state. Following the previous example, the pose that represents sadness can be more or less exaggerated to reflect more or less sadness.

The variations in the speed and in the spatial amplitude are inspired in the work of Amaya [Amaya et al. 1996]. Amaya concluded that the speed and the spatial amplitude of the movement vary noticeably with different emotions. For example, sad movements are normally slow and narrow whereas happy movements are fast and wide. Using these results it is possible to parameterise the animations in real-time to characterize a particular emotion.

All this behavioural richness allows several variations of an animation in real-time that otherwise would have to be created offline as independent animations. As a result, a small database of animations can achieve results that under the classic approach would require a larger database.

As a final remark, please note that the Affective Body Expression component does not intend to establish any mapping between emotional states and movements or movement changes. The idea is to provide synthetic characters that have the ability of modifying their animations in real-time and, therefore, that are able to express emotions. How a particular emotion is expressed depends of the purpose of the application, and of the contextual aspects.
3.2 Specification

A synthetic character cannot be dissociated from its virtual environment. The applications that use synthetic characters have always an underlying graphics engine that supports this virtual environment. It manages the world inhabited by the character and renders all the geometrical information.

The graphics engine that integrates the Affective Body Expression component has a modular approach. Figure 3-1 presents a generic view of the architecture and the four core modules that are always present: the control module (or kernel), the input module, the camera module and the scene module.

![Figure 3-1: Graphics engine](image)

Each module offers a number of services that can be requested through a module interface. The **control module** is responsible for loading all the other modules, and for making their interfaces available for sharing. When loading a module, it collects the module interface and passes its own interface to the module that is being loaded. Using this simple mechanism, the control module stores a library of module interfaces, and the other modules can use the control interface to request or release any other interface they need. Additionally, the control module also manages the cycle of execution of the application, which will be depicted after a brief description of the other three core modules.

The **input module** monitors the keyboard, the mouse or any other input source. The other modules can check, for instance, if a particular key was or not pressed, and act appropriately.

The **camera module** controls the virtual camera. It offers generic services for camera control, like positioning, rotating and zooming.

The **scene module** manages the scene graph. The other modules can publish scene objects in the scene graph, and the scene module is responsible for performing the rendering of these objects.

In addition to these core modules, the engine can be easily extended with more modules to support extra features. The engine cycle starts with the input module, passes through
all the other modules, and ends with the scene module. During the cycle, each module
updates its internal information, and that changes will have impact in the rendered
scene.

Following this approach, the flexibility provided by the graphics engine allows the
inclusion of a module to manage synthetic characters, which is the implementation of
the Affective Body Expression component. The synthetic characters module loads the
available characters and, when it receives a command for creating an instance of a
specific character, it creates a scene object with a new instance of that character, and
uses the scene interface to publish the character in the scene graph. After that, when the
guide cycle passes through the synthetic characters module, the character is updated to
reflect the animations in play and other changes motivated by requests issued to the
module.

### 3.2.1 Character Control

The Affective Body Expression component has a control interface that offers three types
of commands: generic commands, animation commands, and emotional commands. The
next subsections will look more closely to all of these commands.

#### 3.2.1.1 Generic Commands

The generic commands allow the user to create and delete synthetic characters, and to
perform some generic operations like show and hide the synthetic character.

**Create**

*Prototype:*

\[
\text{create (character-name, character-class)}
\]

The *create* command issues a request to create a new synthetic character of a particular
class.

*Example:*

Create a character named *ronin* of the class *earth*:

\[
\text{create (ronin, earth)}
\]

**Delete**

*Prototype:*

\[
\text{delete (character-name)}
\]

The *delete* command issues a request to delete an existent synthetic character.

*Example:*


Delete the character named *ronin*:

```python
delete (ronin)
```

**Set-Position**

*Prototype*:

```python
set-position (character-name, position)
```

The **set-position** command issues a request to place an existent synthetic character in a specific position of the virtual world.

**Example:**

Place the character named *ronin* at the point (10, 10, 10):

```python
set-position (ronin, point(10, 10, 10))
```

**Get-Position**

*Prototype*:

```python
position get-position (character-name)
```

The **get-position** command issues a request to get the position in the virtual world of an existent synthetic character.

**Example:**

Get the position of the character named *ronin*:

```python
get-position (ronin)
```

**Set-Orientation**

*Prototype*:

```python
set-orientation (character-name, orientation-angles)
```

The **set-orientation** command issues a request to orientate an existent synthetic character in a specific direction.

**Example:**

Orientate the character named *ronin* with the orientation-angles (0, 45, 0):

```python
set-position (ronin, orientation-angles(0, 45, 0))
```

**Get-Orientation**

*Prototype*:
orientation-angles get-orientation (character-name)

The \textit{get-orientation} command issues a request to get the orientation of an existent synthetic character.

\textit{Example}:

Get the orientation of the character named \textit{ronin}:

\begin{verbatim}
get-orientation (ronin)
\end{verbatim}

\textbf{Hide}  
\textit{Prototype}:

\begin{verbatim}
hide (character-name)
\end{verbatim}

The \textit{hide} command issues a request to hide an existent synthetic character.

\textit{Example}:

Hides the character named \textit{ronin}:

\begin{verbatim}
hide (ronin)
\end{verbatim}

\textbf{Show}  
\textit{Prototype}:

\begin{verbatim}
show (character-name)
\end{verbatim}

The \textit{show} command issues a request to show an existent synthetic (if it is hidden).

\textit{Example}:

Shows the character named \textit{ronin}:

\begin{verbatim}
show (ronin)
\end{verbatim}

3.2.1.2 Animation Requests

The animation requests allow the user to play and stop animations of the synthetic character, and to query the module about the current state of a particular animation.

\textbf{Play-Animation}  
\textit{Prototype}:

\begin{verbatim}
animation-tag play-animation (character-name, animation-name)
\end{verbatim}

The \textit{play-animation} command issues a request to play an animation of the synthetic character. The available animations may vary from character class to character class. It
returns an animation tag that is necessary to stop an animation, or to query the module about current state of the animation.

Example:

Plays the animation walk of the character named ronin:

```scheme
play-animation (ronin, walk)
```

Stop-Animation

Prototype:

```scheme
stop-animation (character-name, animation-tag)
```

The stop-animation command issues a request to stop an active animation of the synthetic character. This command is only applicable to loops, because the actions or the stances cannot be interrupted.

Example:

Stops the animation with tag 20 of the character named ronin:

```scheme
stop-animation (ronin, 20)
```

Animation-Is-In-Play

Prototype:

```scheme
boolean animation-is-in-play (character-name, animation-tag)
```

The animation-is-in-play command issues a query to verify if an animation of the synthetic character is in play or not.

Example:

Verifies if the animation with tag 20 of the character named ronin is in play:

```scheme
animation-is-in-play (ronin, 20)
```

Animation-Is-In-Queue

Prototype:

```scheme
boolean animation-is-in-queue (character-name, animation-tag)
```

The animation-is-in-queue command issues a query to verify if an animation of the synthetic character is in the playing queue or not.

Example:

Verifies if the animation with tag 20 of the character named ronin is in queue:
animation-is-in-queue (ronin, 20)

Animation-Has-Ended

Prototype:

boolean animation-has-ended (character-name, animation-tag)

The **animation-has-ended** command issues a query to verify if an animation of the synthetic character has ended or not (in play or in the playing queue).

Example:

Verifies if the animation with tag 20 of the character named *ronin* has ended:

animation-has-ended (ronin, 20)

3.2.1.3 Emotional Commands

The emotional commands allow the user to set the emotional state of the synthetic character, or to query the module about the current emotional state of the character.

Set-Emotional-State

Prototype:

set-emotional-state (character-name, emotion-name, intensity)

The **set-emotional-state** command issues a request to set the emotional state of the synthetic character.

Examples:

Sets the emotional state of the character named *ronin* to *sad* with intensity 1.0:

set-emotional-state (ronin, sad, 1.0)

Sets the emotional state of the character named *ronin* to *sad* with intensity 0.1:

set-emotional-state (ronin, sad, 0.1)

Sets the emotional state of the character named *ronin* to *neutral*:

set-emotional-state (ronin, neutral, 1.0)

Get-Emotional-State

Prototype:

emotion-name get-emotional-state (character-name)
The **get-emotional-state** command issues a request to get the current emotional state of the synthetic character.

*Example:*

Queries about the current emotional state of the character named *ronin*:

```
get-emotional-state (ronin)
```

**Get-Emotional-State-Intensity**

*Prototype:*

```
intensity get-emotional-state-intensity (character-name)
```

The **get-emotional-state-intensity** command issues a request to get the intensity of the current emotional state of the synthetic character.

*Example:*

Queries about the intensity of the current emotional state of the character named *ronin*:

```
get-emotional-state-intensity (ronin)
```

### 3.3 Architecture

![Architecture Diagram](image)

Figure 3-2: Architecture

The architecture that supports the synthetic characters module defined in the previous section is inspired in the works of Blumberg and Perlin. Figure 3-2 shows the three layers of the architecture: the geometry, the animation engine, and the behaviour engine.

The **geometry** is responsible for maintaining the body of the synthetic character, and for controlling the related rendering process.
The animation engine controls the character’s motion. It manages the active animations, and it is responsible for updating the body over the time.

The behaviour engine is responsible for the higher level control over the character’s body, and it sends combined requests to the animation engine in order to create complex behaviours.

This approach allows a clear conceptual separation that distinguishes:

- The body, as the entity managed by the geometry layer;
- The animations, as the entities managed by the animation engine;
- The behaviours, as the entities managed by the behavioural engine (parameterised combinations and sequences of animations).

### 3.3.1 Geometry

The geometry layer implements the internal structure of the synthetic character. It offers direct control over the body of the synthetic character, as well as an easy manipulation of both the skeleton and the skin surface.

The internal structure is a simple joint-link model, as it is usually described in the literature. In a joint-link model, the body is characterized by a hierarchical skeleton and by a deformable skin surface that is linked to the bones of the skeleton.

The hierarchical skeleton is collection of bones connected by a tree structure. There is a root bone, which is the entry point to the tree, and each bone can have an undetermined number of children. The movement of a particular bone affects not only the bone itself, but also all of its children and grandchildren until the end of the branch. This is very similar to what happens in a real skeleton. For instance, when we move the elbow, the wrist and all the bones in the hand are also moved.

The skin surface is a triangular mesh, and each vertex of the mesh is connected to one or more bones. The movement of the bones modifies the skin surface that is adjusted to reflect the underlying skeleton. Each vertex is calculated looking at the weighted contribution of the bones that are linked to it.

Figure 3-3 shows the UML diagram that represents an overview of the all the classes involved.

---

1 Sometimes the bones are referred as joints, thus the designation joint-link model.
3.3.1.1 Body

The body is characterized by a skeleton, a skin surface and a set of materials. The skeleton is a set of bones, and the skin is a set of meshes, as they were defined in the internal structure. The materials store the information related to the materials (usually textures) that cover the skin (see Figure 3-4).

All these sets are stored as dynamic arrays, because dynamic arrays offer the advantages of a static array and the advantages of a list at the same time:

- The elements of a dynamic array are contiguous like in a static array, which leads to significant improvements in performance because it reduces the iteration time; moreover, the arrays associate an index to each element, which reduces the search time because the retrieval of an element is immediate if the search is by index;
• A dynamic array can grow and shrink in size like a list, which allows an easier manipulation of the elements and dynamic allocation of the array in runtime.

To take benefit from these properties, all the elements of the collections are internally identified by the index occupied in the associated dynamic array. This decision has an important impact in the lightness of the structures, and also reduces the complexity of most algorithms.

3.3.1.2 Skeleton

The implementation of the skeleton takes into account its hierarchical structure. The skeleton is organized in a tree structure with a root bone, and an undetermined number of children per bone. The root bone is the first element of the dynamic array that stores the skeleton. The information about the topology of the skeleton is distributed for all the bones. Figure 3-5 shows the UML diagram that represents a bone of the skeleton.

![Figure 3-5: Bone [UML]](image)

Each bone has links to the parent bone and to the children bones. These links are represented by indexes of the dynamic array that stores the skeleton. With this approach, each bone is responsible for its own topologic information and, together, all the bones keep the complete topology of the skeleton. Figure 3-6 shows an example of a skeleton.

Internally, a bone is characterized by the translation and the rotation in relation to the parent bone (boneTranslation and boneRotation). In addition, it also keeps the translation and the rotation in relation to the world (worldTranslation and worldRotation), which simplifies the algorithm that updates the skin surface. These transforms in the world space are updated over the time to maintain the consistency with the transforms in the bone space. The equations that update the skeleton are the following:

\[
wt_m = wt_n + wr_n \cdot bt_m \quad m,n=0..bones \quad n=parent(m)
\]

Equation 3-1: Transform positions from bone to world space
Equation 3-2: Transform orientations from bone to world space

\[
wr_m = wr_n \cdot br_m \quad m,n=0..\text{bones} \quad n=\text{parent}(m)
\]

The position of each bone in the world space \((wt_m)\) is the position in the bone space \((bt_m)\) after being transformed by the orientation of the parent bone in the world space \((wr_n)\) and added to the parent bone position in the world space \((wt_n)\).

The orientation of each bone in the world space \((wr_m)\) is the orientation in the bone space \((br_m)\) after being transformed by the orientation of the parent bone in the world space \((wr_n)\).

3.3.1.3 Skin

The skin surface plays a very important role because it is the visible part of the synthetic character. Typically modellers use a single mesh to preserve the continuity of the surface during the animations, but it is also very common to use a separate mesh for the head in order to have a more detailed model of the face. Figure 3-7 shows the UML diagram that represents a mesh of the skin surface.

Figure 3-6: Skeleton
A mesh is characterized by a set of vertices and faces. A vertex stores the coordinates (coord) of a skin point. The faces organise the vertices in triangles. However, there is an indirection between the faces and the vertices, something that can be defined a face point. Therefore, although vertices can be shared between faces, the indirection through the face points guarantees an individual structure where the properties of the relation face-vertex can be kept. Figure 3-8 shows an example of a skin surface.
A final aspect concerning the skin surface is the support of skin deformations. As it was mentioned before, the skin surface is automatically updated to reflect the poses of the underlying skeleton. However, it does not take into account the usual deformations provoked by the tightening or the growth of the muscles, because the internal structure does not have an intermediate layer to simulate the muscles. To overcome this limitation, the geometry layer allows the direct manipulation of the skin points to achieve similar results if needed.

At this point, let us look more closely how the skeleton is linked to the skin. Figure 3-9 shows the UML diagram that represents a skin link.

![Figure 3-9: Skin Link [UML]](image)

A skin link is a relation between a bone and a vertex of a specific mesh. It defines the offset between the bone position and the vertex position, and also the influence that bone has over the vertex. The offset works like a rigid link between the bone and the skin point. When the skin is updated, if the position or the orientation of the bone has changed, then all the attached skin points are also moved to maintain the offset. If a skin point is connected to more than one bone, then the influence is used to weight the contribution of each bone to the skin point movement. The equation that updates each vertex of the skin surface is the following:

\[
\nu_m = \sum_{n=0}^{bones} wt_n + (wr_n \times offset_{nm} \times influence_{nm} )_{m=0..\text{vertices}}
\]

Equation 3-3: Update a vertex of the skin surface

The position of each vertex (\(\nu_m\)) is equal to the sum of the contributions of all the bones. The contribution of a single bone is the sum of the position of the bone in the world space (\(wt_n\)) with the offset between the bone and the vertex (\(offset_{nm}\)) after being transformed by the orientation of the bone in the world space (\(wr_n\)) and multiplied by the influence of the bone over the vertex (\(influence_{nm}\)).
3.3.1.4 Materials

The skin surface can be covered by one or more materials. Figure 3-10 shows the UML diagram that represents a material, and its connections to the skin mesh.

A material has a name defined by the modeller of the synthetic character, and a series of properties that usually define a material: the ambient, diffuse, specular and emissive colors (RGB values), the shininess and the alpha value. Most of the times, a material is associated with a texture, which is a bitmap file with an image used to “dress” the character.

Internally, each mesh of the skin surface can have a single material that covers all the mesh, or different materials for each face. If the material is a texture, each face point must have the mapping coordinates (texCoord) defined (see Figure 3-7). These points define how the texture is attached to the skin surface. When the skin surface moves, as a consequence of the skeleton movement, the texture is adjusted (stretched or shrunk), thus maintaining the attachment points.

3.3.1.5 Interface

The body interface offers the following methods for update:

- The method **setPose** that receives an entire skeleton pose, and updates the skeleton accordingly; this is the main method for updating the body, and is
extensively used by the animation engine; therefore, the next section will explain in detail what is a pose, and how it can change the skeleton;

- The method `updateSkeleton` that updates the skeleton as it was described in the Section 3.3.1.2; basically, it is used to update the internal skeleton structure after changing the position or the orientation of any bone; for example, it should be used after using the `setPose` method to guarantee the internal coherence of the skeleton;

- The method `updateSkin` that updates the skin as it was described in the Section 3.3.1.3; it is used for updating the skin surface in order to illustrate the changes in the underlying skeleton; for example, it should be used after the `updateSkeleton` to guarantee that the skin reflects the last pose of the skeleton.

As it should be clear, the normal use of the body leads to a sequence of calls `setPose`, `updateSkeleton`, and `updateSkin`. The reason why these methods are separated is to allow changes in the skeleton without making them immediately visible. It may be necessary, in some circumstances, to use a `setPose` and an `updateSkeleton`, but delay the `updateSkin` for a little while. Since the rendering cycle only looks at the skin surface, because it is the visible part of the character, it will remain unchanged until the `updateSkin` method is called.

### 3.3.2 Animation Engine

![Animation Engine Overview UML](image)

The animation engine is implemented by the character class. Figure 3-12 shows the UML diagram that represents an overview of the all the classes involved. It is responsible for the animation cycle of the synthetic character.
3.3.2.1 Character

A **character** is characterized by a **body**, and by a set of **animations**. The body was described previously (see Section 3.3.1.1). The animations are stored in a hash table that facilitates a quick access when needed. Each character has also a timer that controls the animation cycle as will be explained later.

![Figure 3-13: Character [UML]](image)

The character class is the heart of the animation engine. It keeps the available animations, and manages the requests. At the end of each animation cycle, it updates the body of the synthetic character so it can reflect the active animations. It also offers the support for the affective body expression, because it manages the real-time composition of stances, the increase and decrease of the animation speed, and the changes in the spatial amplitude of the animations. But let us start by the beginning, and look at the implementation of the animations.

3.3.2.2 Animations

The animations are the principal ingredient of the animation engine. With the dynamics conveyed by the body, it is possible to represent animations with very few information because there is no need to store the variation of each skin point, only the variation of the underlying bones. An **animation** is simply defined as a series of skeleton poses over the time. The skin surface is calculated in real-time taking into account the position of the bones at each instant. Figure 3-14 shows the UML diagram that represents an animation.

An animation can have three **types**:

- **Stance** – a single key-frame, i.e. a single skeleton pose; an example of a stance is the head and the torso bended;
- **Action** – a sequence of key-frames that runs once and stops, i.e. a sequence of skeleton poses; an example of an action is scratching the head;
- **Loop** – a sequence of key-frames that runs in an endless cycle, i.e. a loop of skeleton poses; an example of a cycle is walking.
However, the differences are purely functional, because the structures that support these three types of animation are the same. An animation is characterized by a set of frames and an internal clock time duration in milliseconds.

Each frame represents a key-frame of the animation, and contains the “static image” of the synthetic character in a particular point of the animation. Together, the sequence of all the “static images” over the time produces the related motion. For the purpose of the present work, this “static image” only contains a skeleton pose. The skeleton pose is sufficient to describe the synthetic character in a particular moment, because the movement of the skeleton affects the rest of the body as it was described in previous sections (Figure 3-15 and Figure 3-16). However, a frame can also contain one or more morph targets that can change the skin of the synthetic character, which can be very useful for simulating deformations.
Each pose is associated with a particular body, meaning a particular skeleton. A pose has a certain number of bone nodes that should match the number of bones in the skeleton. Otherwise, it leads to unpredictable results. A bone node keeps the position and the orientation of a bone in the bone space. When a pose is applied to the body (skeleton) using the method `setPose` (see Section 3.3.1.5), each bone node is used to modify the position and the orientation of the associated bone.

An important aspect, that should be mentioned, is the internal conversion of the animations to make them compatible with the overall model. In fact, the body of the synthetic character has already a neutral stance (or pose), and each bone node must keep the position and the orientation of the bones in relation to that neutral stance, and not an absolute position or orientation. This means that the poses of the animations (usually created in the world space) must be converted to the bone space, and then become relative to the neutral stance of the body. The equations that support this conversion are the following:

\[
bt_m = \left( wr_n^{-1} \cdot (wt_m - wt_n) \right) - nbt_b \\
\]

Equation 3-4: Transform positions from world to bone space (with neutral stance)

\[
br_m = nbr_b^{-1} \cdot \left( wr_n^{-1} \cdot wr_m \right) \\
\]

Equation 3-5: Transform orientations from world to bone space (with neutral stance)

The position of each bone node in the bone space \(bt_m\) is the difference between its position in the world space \(wt_m\) and the position of the parent in the world space \(wt\) transformed by the inverse of the orientation of the parent bone in the world space \(wr_n^{-1}\); the difference for the neutral stance is done by subtracting the position in the bone space of the neutral bone associated with the bone node \(nbt_b\).
The orientation of each bone node in the bone space ($w_{rm}$) is the orientation in the world space ($wr_{m}$) after being transformed by the inverse of the orientation of the parent in the world space ($wr_{n}^{-1}$), and calculated the difference for the neutral stance by the inverse transform of the orientation in the bone space of the neutral bone associated with the bone node ($nbr_{p}^{-1}$).

Usually the above described conversion is performed once when the animations are being loaded.

At this point, let us look closely into the `setPose` method, and see how each bone node is capable of modifying the position and the orientation of the related bone. The equations involved are:

\[
bt_{b} = nbt_{b} + bt_{m} \quad b=0..bones \quad m=0..boneNodes \quad m=boneNode(b)
\]

Equation 3-6: Update of the bone position

\[
br_{b} = nbr_{b} \times br_{m} \quad b=0..bones \quad m=0..boneNodes \quad m=boneNode(b)
\]

Equation 3-7: Update of the bone orientation

The position of each bone in the bone space ($bt_{b}$) is the sum of its neutral position in the bone space ($nbt_{b}$) with the position in the bone space of the associated bone node ($bt_{m}$).

The orientation of each bone in the bone space ($br_{b}$) is the orientation in the bone space of the associated bone node ($br_{m}$) transformed by the neutral orientation of the bone in the bone space ($nbr_{b}$).

As a result, the `setPose` method changes all the bones in the bone space to a specific pose. After, the `updateSkeleton` and `updateSkin` methods do the rest by calculating the skeleton in the world space, and by changing the skin accordingly.

Following this approach, playing an animation is very straightforward, because it can be done by simply providing to the body the successive poses in the appropriate time. The next step is the animation cycle.

### 3.3.2.3 Animation Cycle

The animation cycle implements the necessary functionalities to control the animation play, and all the rules that regulate the coexistence of multiple animations:

1. The animation engine cannot play more than one loop or action at the same time, but the stances can coexist with both of them;

2. The loops are played continuously until the animation engine receives a request for another loop or action, or an explicit command to end the loop;
3. The actions are always played until the end, and cannot be interrupted by any request;

4. The stances are combined in real-time with the active loop or action, but the animation engine cannot handle more than one stance at the same time.

Although very simple, the previous set of rules guarantees that all the animations run properly without ambiguous visual results. These rules implement a minimal resource mechanism that prevents two loops or actions from being played at the same time, but offer the necessary flexibility to interrupt a loop or to introduce stances in the middle of the animation in play (either loop or action).

Figure 3-17 shows the UML diagram that represents the classes involved in the animation cycle.

![UML Diagram](image)

The character keeps a **playlist** of animations. The first animation in the playlist is the animation in play; the following animations are the animations in queue. The playlist is only used for loops or actions, because the stances are kept in a separate variable named **stance**, which is a pose. With this internal organisation the animation engine cannot play more than one loop or action at the same time, because it has space for only one active loop or action in the playlist. At the same time, only a single stance can coexist with the animation in play.

When the animation engine receives a request for an animation, several things can happen:

- If the animation request is invalid, because the animation doesn’t exist, then nothing happens;

- If the animation request is a stance, then the animation engine substitutes the current stance by the new one, and in the next update the active animation will be combined with the new stance;

- If the animation request is a loop or an action, then the animation engine starts by checking which animation is currently in play: if the animation in play is a loop, or an animation resultant from the automatic idle behaviours (either loop
or action), then it is removed from the playlist to guarantee that the new request is played immediately; if the animation in play is an action, then it remains in the playlist because it cannot be interrupted by any request. After this step, the animation engine creates an animation node in the playlist for the new request, which contains a tag for identifying the request, and a frame counter that keeps the current frame (currentFrameID). Two things can happen: this new animation node is the first in the playlist, and the timer is restarted (start) to control the new animation; the animation node is in queue, and it will be processed later.

As a result, the requests to the animation engine are always played in sequence, but if a particular request cannot be completed it will be queued. A typical situation is when an action is in play and the animation engine receives a request for another action; the resource mechanism places the new request in queue until the first action ends. When there are no more requests in queue, the animation engine plays an idle behaviour.

Actually, the automatic activation of the idle behaviours is an important task of the animation engine. The idle behaviours are essential to attain a certain degree of believability, because they fill the gaps between the requests for specific animations, and prevent the synthetic character from being rigid and immovable, which looks very unnatural. The animation engine keeps a list of idle behaviours that are played randomly when the character is idle. Usually, these idle behaviours are loops that will be active until the animation engine receives a new request.

However, the management of the request is only a small part of the animation cycle. In fact, most of the time is spent updating the internal structures to reflect the active animation. The current frame of the active animation is given by the getFrameID method with the animation time got from the cycle timer. Basically, the current frame is calculated by:

\[
\text{currentFrame} = \frac{\text{currentTime}}{\text{duration}} \times \text{numberFrames}
\]

Equation 3-8: Current frame calculation

If the current frame is equal to the currentFrameID stored in the animation node, then the elapsed time since the last update was not sufficient to the animation advance to the next frame, and the update is not needed. Otherwise, both the currentFrameID and the body are updated. This simple trick eliminates many unnecessary body updates because the number of updates per second is usually greater than the number of animation frames per second.

When a body update is needed, the animation engine calculates an update pose combining the pose associated with current frame (calculated in the previous step) with the current stance. The stance has an associated weight that represents how much it influences the current animation. The equations that describe the composition of each bone node are:
\[ bt_m = fbt_m + \left( sbt_m \times \text{weight} \right) \]

Equation 3-9: Weight composition of the bone position

\[ br_m = fbr_m \times \text{slerp}(I, sbr_m, \text{weight}) \]

Equation 3-10: Weight composition of the bone rotation

The combined position of each bone node \((bt_m)\) is the sum of the position of that bone node in the current frame \((fbt_m)\) with the weighted position in the current stance \((sbt_m)\).

The combined orientation of each bone node \((br_m)\) is the weighted orientation of that bone node in the current stance \((sbr_m)\) transformed by the orientation in the current frame \((fbr_m)\). In this case, the weight influence is calculated using a spherical linear interpolation \((\text{slerp})\) between the identity rotation \((I)\) and the orientation of the bone node in the current stance.

Figure 3-18 and Figure 3-19 show an animation before and after being composed with a non-neutral stance:
After calculating the update pose, the animation engine uses the services provided by
the geometry layer (see Section 3.3.1.5), and updates the body of the character.

Next, the animation engine checks if the current animation has ended, and if it has, two
things can happen: if the animation is a loop, then it is simply restarted by resetting the
timer and the frame counter; otherwise, if the animation is an action, then the animation
node is removed from the playlist, and the next animation in the queue becomes active
(if the queue is empty, an idle behaviour is used).

At this point, in the transition between the two animations, the animation engine blends
the end of the old animation with the beginning of the new one to achieve a more
smooth and realistic transition. It uses a fade out / fade in technique that gradually
decreases the influence of the old animation and increases the influence of the new
animation. The process is very straightforward, and it simply creates some intermediate
poses between the last frame of the old animation, and the first frame of the new
animation. These intermediate poses are created using weighted functions similar to the
ones applied to the stance when computing the update pose:

\[ b_{tm} = b_{tm} \times \text{weight} \quad m=0..\text{boneNodes} \]

Equation 3-11: Weight function of the bone position

\[ b_{rm} = \text{slerp}(I,br_m,\text{weight}) \quad m=0..\text{boneNodes} \]

Equation 3-12: Weight function of the bone rotation

The weighted position of each bone node \((bt_m)\) is equal to the current position \((bt_m)\)
multiplied by the \text{weight}.

The weighted orientation of each bone node \((br_m)\) is the result of the spherical linear
interpolation \((\text{slerp})\) between the identity rotation \((I)\) and the current orientation \((br_m)\)
using the \text{weight} as balance factor.

When the weight decreases, the pose is gradually changed towards the neutral stance,
and vice-versa. This way, decreasing the weight of the old pose, and then increasing the
weight of the new pose causes a very soft transition between the two animations.

A final aspect related with the animation cycle, is how it can be influenced by changes
in the speed and in the spatial amplitude of the animations:

- The changes in the speed of the animations are easy to implement, and have a
direct relation with the timer that controls the animation cycle. Increasing or
decreasing the animation speed is simply achieved by multiplying the animation
time by a factor. For example, using a time factor of \(2.0\) duplicates the speed of
the animations, whereas a time factor of \(0.5\) decreases to half the same speed.
Although very simple, this approach reduces or increases the time between two
consecutive frames, thus increasing or reducing the frame rate per second. However, if the frame rate falls below a reasonable value, the animation may appear very rough. The solution is to use the same technique used in the transitions, and to generate intermediate frames within the animation to maintain the frame rate constant.

- The changes in the spatial amplitude are achieved with one of the weight functions used for the generation of the intermediate poses (see Equation 3-12). In fact, decreasing or increasing the weight of a pose, and more particularly of a bone, has the exact same effect than the variation of the spatial amplitude. Increasing the spatial amplitude increases the angles between the bones, like when the weight is increased. Decreasing the spatial amplitude of the animation reduces the angles between the bones, like when the weight is decreased. Therefore, the spatial amplitude is easily modified by applying the referred function to the update pose, before updating the body. Figure 3-20 shows an example of the use of the spatial amplitude.

![Figure 3-20: Spatial Amplitude = 0.5 (regarding Figure 3-16)](image)

In conclusion, the animation cycle implemented by the animation engine is very straightforward. It receives requests for animations (loops, actions or stances), and plays them sequentially. If the resource mechanism that manages the coexistence of animations blocks a particular request, then this request is queued until it can be played. When there are no more requests and the character is idle, the animation engine automatically plays an idle behaviour. The stances are combined in real-time with the active animation; and the speed and spatial amplitude of the overall movement can be modified in real-time. The animation engine manages all this information, and updates the geometry of the synthetic character to reflect the desired behaviour.
3.3.2.4 Interface

The character interface offers the following methods related with the animation engine:

- The method **playAnimation** that receives an animation name (loop, action or stance), and executes the procedures related with the handling of a new request for an animation, as it was described in the previous section; it returns the animation tag associated with the request for future queries and requests;

- The method **stopAnimation** that receives an animation tag, and stops the related animation (this is only valid for loops, because the actions and the stances cannot be stopped);

- The method **setStanceWeight** that changes the weight of the current stance, and the method **getStanceWeight** that returns the weight of the current stance; this has impact in the influence of the current stance over the animation in play, as it was described in the previous section;

- The method **setSpeed** that changes the speed factor, and the method **getSpeed** that returns the speed factor; this factor affects the speed of the animations, as it was described in the previous section;

- The method **setSpatialAmplitude** that changes the spatial amplitude factor, and the method **getSpatialAmplitude** that returns the spatial amplitude factor; this factor influences the amplitude of the animations, as it was described in the previous section;

- The method **update** that is the main method of the animation engine, and controls the iterations of the animation cycle; it does the necessary updates to reflect the changes introduced by the methods described above;
• The `animationIsInPlay`, `animationIsInQueue` and `animationHasEnded` methods that are querying methods which check if a particular animation request is in play, is in queue, or has ended, respectively;

The methods `setEmotionalState`, `getEmotionalState` and `getEmotionalStateIntensity` are related with the behaviour engine that will be depicted in the next section.

### 3.3.3 Behaviour Engine

The behaviour engine is usually seen as the lowest level of the character’s mind. It is able to map complex behaviours into basic animations of the animation engine. A typical behaviour is the expression of an emotional state. For example, to express sadness the behaviour engine can activate a specific stance that reflects sadness and decrease the speed of the animations. The combination of these requests to the animation engine with the animation in play will create the desired result.

Obviously, it is possible to define more complex behaviours. The behaviours can be any sequence of animations, or any other requests supported by the animation engine. The behaviour engine can be as complex as needed, and that depends mostly on the mind’s model and if the upper levels need more accurate control over the animations or not. A richer behaviour engine offers a complex set of behaviours but less control over the animations, whereas a simpler behaviour engine offers a simpler set of behaviours but more control over the animations.

The Affective Body Expression component uses a very simple behaviour engine that, like the animation engine, is implemented by the character class. It is a very simple version of a behaviour engine that only deals with emotional states, and knows how to map them into requests for the animation engine, thus the decision of implementing them together. Basically, the behaviour engine receives requests for a specific emotion with a particular intensity and, using a set of mapping rules stored internally, issues a series of requests to the animation engine.

#### 3.3.3.1 Interface

The character interface (see Figure 3-21) offers the following methods related with the behaviour engine:

- The method `setEmotionalState` that receives an emotion and its intensity, verifies if there is a rule for that emotion, and executes the necessary requests to the animation engine;
- The method `getEmotionalState` that returns the current emotional state;
- The method `getEmotionalStateIntensity` that returns the intensity of the current emotional state.
3.4 Toolkit Integration

The Affective Body Expression component needs to be associated to a small graphics engine in order to be integrated in the SAFIRA Toolkit. The implementation of this basic graphics engine was done in C++ and OpenGL. The four core modules: control, input, camera, and scene were implemented together using the OpenGL library GLUT, which already offers services for window management and input control.

Basically, the input module is implemented by the glutKeyboardFunc using a handler that maps keyboard keys into services provided by the other modules, either camera, scene or external modules like the synthetic characters module.

The camera module implements a simple virtual camera based on two positions: the eye position (i.e. the position of the camera in the virtual world), and the focus position (i.e. the target of the camera). The services offered allow the following operations:

- Panning operations that position the camera in the virtual space: panUp, panDown, panLeft, panRight, panForward, and panBackward;
- Arc rotating operations that rotate the camera around the focus point: arcUp, arcDown, arcLeft, and arcRight;
- Rotating operations that rotate the camera (pitch, yaw and roll): pitchUp, pitchDown, yawLeft, yawRight, rollLeft, and rollRight;
- Zooming operations that approach more or less the focus position: zoomIn, and zoomOut.

The scene module implements the scene graph as a plain list of scene objects. It offers the following services:

- Add, get, or remove scene objects: addObject, getObject, and removeObject;
- Change the position and the orientation of a scene object: setObjectPosition, getObjectPosition, setObjectOrientation, and getObjectOrientation;
- Change the visibility of a scene object: setObjectVisibility, and getObjectVisibility.

The engine cycle is implemented by the glutDisplayFunc, which updates the camera and draws the entire scene.

All these services offered by the graphics engine together with the character control interface define previously (see Section 3.2.1) were wrapped inside a JAVA interface class that was integrated into the Toolkit. Each method is associated with a request encoded using the SAFIRA XML message protocol.
When the Affective Body Expression component is in use, it registers itself into the SAFIRA CRS, and waits for requests that are processed through method calls using the JAVA interface class. Actually, this is a very simple process because the Affective Body Expression component is a terminal component in the application that acts like a server that simply waits for, and processes requests.

3.5 Use

3.5.1 Modelling and Animation

The use of the Affective Body Expression component has a preliminary phase that consists in modelling and animating the synthetic characters (the Toolkit provides by default the characters used in FantasyA – one of the SAFIRA demonstrators).

There are several commercial tools available for modelling synthetic characters, like 3D Studio MAX, Maya, or Lightwave, just to mention the most well known. However, the bodies created with these tools must be compatible with the internal structure presented before. To ensure that compatibility the modeller must follow some guidelines:

- The body must have a skeleton and a skin surface;
- The skeleton must have a root, and the child bones must be organized hierarchically;
- Each point of the skin surface must be linked at least to one bone.

These guidelines do not imply a pre-determined skeleton with a certain number of bones, and this is a significant point. There are several approaches that are bound to a specific skeleton with well defined bones, and that is a major restriction from the point of view of the modeller. The present work deals with any kind of skeleton, and the modeller is free to use the number of bones s/he needs to achieve the desired results. Likewise, there are no restrictions to the skin surface imposed by the model. The modeller can use a single or multiple meshes, and s/he only as to ensure that all the skin points are connected to one or more bones.

There are also several commercial tools to create animations for the synthetic characters, but once more the animator has to follow some guidelines to ensure the compatibility with the internal structure:

- The skeleton configurations used in the animations must have the same number of bones of the body, and the number of bones must not change along the key-frames of the animation;
- The time interval between the key-frames must be fixed.
Note that an exact match between the bones of the body and the bones in each skeleton configuration is of critical importance, for it is the only way to maintain the links to skin surface along the animation, and thus to guarantee the proper update of the internal structure.

In addition to the guidelines stated above, there are also a few practices that ensure the creation of more consistent animations:

- The actions should begin and end with a neutral skeleton configuration to minimize the transitions;
- The last key-frame of a loop should have a smooth and natural transition to the first key-frame to guarantee a true endless loop where one cannot notice where it starts and where it ends.

Following this practices and guidelines, it was developed an exporting plug-in for the 3D Studio MAX that is able to convert the character models and animations into the structures/objects defined in the previous sections, which can then be saved to the hard disk using the serialization methods. After that, importing the information to an application that uses the character is simply achieved by using the de-serialization methods to rebuild all the structures/objects that represent the synthetic character.

The synthetic character’s library is composed by the character model and all the animations, plus a configuration file that keeps all the information together. Next is an example of such a file:

```xml
<Class id='EARTH'>
  <Body>ronin.y</Body>
  <Animation id='NEUTRAL' type='STANCE'>ronin_idle.y</Animation>
  <Animation id='ANGRY' type='STANCE'>ronin_angry.y</Animation>
  <Animation id='SAD' type='STANCE'>ronin_sad.y</Animation>
  <Animation id='HAPPY' type='STANCE'>ronin_happy.y</Animation>
  <Animation id='FEAR' type='STANCE'>ronin_fear.y</Animation>
  <Animation id='SURPRISED' type='STANCE'>ronin_surprise.y</Animation>
  <Animation id='GLOAT' type='STANCE'>ronin_gloat.y</Animation>
  <Animation id='GREETING' type='ACTION'>ronin_greeting.y</Animation>
  <Animation id='OFFENSIVE' type='ACTION'>ronin_cast_offensive.y</Animation>
  <Animation id='DEFENSIVE' type='ACTION'>ronin_cast_defensive.y</Animation>
  <Animation id='HIT' type='ACTION'>ronin_hit.y</Animation>
  <Animation id='DIE' type='ACTION'>ronin_die.y</Animation>
  <Animation id='LOOSE' type='CYCLE'>ronin_sad.y</Animation>
  <Animation id='WIN' type='CYCLE'>ronin_gloat.y</Animation>
  <Animation id='WALK' type='CYCLE'>ronin_walk.y</Animation>
  <Animation id='IDLE' type='CYCLE'>ronin_idle.y</Animation>
</Class>
```
The file contains the definition of a character class with the names of the files that store the body of the character, and all the existent animations.

### 3.5.2 Behaviour Definition

The next step is the definition of the behaviours that map emotional states into animation engine commands (by using stances, and speed and spatial amplitude variations, as defined previously). This can be illustrated by the next figures featuring characters from FantasyA:

**“sad” pose, spatial amplitude = 0.5, speed = 0.5**

Figure 3-22: Ronin "sad"

**“fear” pose, spatial amplitude = 1.0, speed = 1.0**

Figure 3-23: Alvegha with “fear”
3.5.3 Application

After creating the characters and defining the emotional behaviours, the Affective Body Expression component can be integrated into the application that will use it. As an example, look at FantasyA. In a few words, FantasyA is a magic duel where two characters fight each other using emotions. The human player uses a sympathetic interface, the SenToy, that is able to recognize six emotions: sad, happy, gloat, angry, fear, and surprise. With these emotions, it can influence the emotional state of the human controlled synthetic character. At each turn, the combination of the emotional states of the two characters leads to actions, either offensive or defensive, that can damage the opponent or protect the character from future attacks. Figure 3-26 shows the magic arena, and an ongoing duel between two characters of FantasyA: Alvegha, and Ronin.
As it is easy to understand, the emotional display is extremely important for game mechanics, because the player must be able to recognize all the emotional states in order to discover the combinations that produce each action.

3.6 References


4. Affective Rendering

4.1 Overview

The goal of the Affective Rendering component is to give software systems the ability to draw pictures in real-time which express emotion and personality, both in the content of the drawing and in the dynamics of the drawing process. We aim to communicate a human, emotional, subjective perspective on a situation. Children's drawings, while requiring a very low level of expertise, communicate a great deal of charm and personality. The goal of the affective rendering component is to provide an artificial system with a similar ability to make child-like drawings which communicate its emotions and personality. While such drawings are not likely to become a universal tool in autonomous agents’ skill sets, the principles that drive and structure the system are drawn and developed from other forms of agent expression and are likely to be usable for other such applications.

Affective Rendering, as defined for the SAFIRA project, focuses on two aspects of graphics which are pertinent to SAFIRA: emotional expression and real-time dynamics within the rendering process. The algorithms implemented in this component provide the ability to express emotional information through the very process by which pictures are rendered. In addition, these pictures are drawn in real-time, as the user watches. The dynamics of this drawing process convey information about the emotional state of the agent.

Our most direct inspiration is Ed Burton’s Rose (Burton 1997a) (Burton 1997b), a program which generates childlike drawings of 3D models. The Rose program takes as input a CAD/CAM model, analyses it in a way metaphorically similar to children’s perception, and then produces a childlike drawing of it.

Figure 4-1: Screenshot of Rose, showing input 3D model, Rose’s internal representation, and the resulting picture Rose draws.
Ed Burton argues strongly that this work should not be seen as an attempt to directly model children’s drawings within the computer. Instead, he explores the kinds of systems one could build when drawing on particular, metaphorical understandings of children’s drawings. We follow this view by being inspired in the construction of the system by stories of children’s drawings. We draw these largely from the depth psychology literature, which unlike the cognitive science literature discusses not the general properties of drawing, but the individual emotional and subjective meaning of these drawings, in a narrative form.

According to depth-psychological theories, children's drawings can be 'read' in order to determine the child's emotional and developmental situation. For example, a closed circle means the child has learned to distinguish between inside and outside, or himself and the world. Depth-psychological theories propose a psychological reading for the elements that are possible in children's drawings, suggest an order in which those elements are likely to appear, and give reasons for the changes from one element to another. These statements can be combined to generate a developmental graph, which lists the pictorial elements that are created in each developmental state, and the emotional reasons for moving from one state to the next. In this way, these theories provide an excellent basis for an algorithm for emotional rendering. Examples of these developmental graphs are presented below.

![Developmental Graph Example](image-url)

**Figure 4-2: Development of early, abstract forms from ages 0 to 3.**
Figure 4-3: Development of the human figure from ages 3 to 5.

Figure 4-4: The development of house forms, and their relationship to emotional states.
There is an important conceptual difficulty with these theories, however: they are no longer accepted by art pedagogians. They are considered to be based far too strongly on psychologist's interpretations. Rhoda Kellogg, for example, simply classifies children's drawings by form, but states that no one really knows why a particular child draws a particular thing at a particular moment. Nevertheless, this need not stop us from using the now discredited theories to generate drawings; we simply accept that while the theories are good for the algorithm, they are not the absolute truth about children's drawings. In this sense, again, the Influencing Machine is focused on child-like drawing, rather than a representation of real children's drawings.

Our extension to previous work in this area is focused on the following points:

1. The ability to explicitly model and communicate emotion and personality through the drawings.

2. The generation of dynamic drawings, i.e. where not only the output, but also the real-time process of doing the drawing is child-like.

3. The elimination of an input model, which is not realistic for children’s art before the age of 4 or 5. Instead, the ‘content’ of the drawing is driven by the agent’s emotional state.

4. The modelling of development, so that the program is capable of generating different stages of child-like drawings.

### 4.2 Specification

The basic principle of SAFIRA’s affective rendering is a two-step pipeline. A low-level stroke-drawing model creates nonphotorealistically rendered pen strokes in real-time with humanlike dynamics. This model’s input parameters are provided by higher-level drawing behaviours, which map the input emotional and developmental states to control parameters for the stroke drawing. The pipeline architecture is as follows.

![Figure 4-5: The pipeline for emotion-driven rendering](image-url)
This pipeline architecture provides the opportunity to develop interfaces at two different levels to the affective rendering system:

1. A low-level interface (Stroke System) allows other software components to call the stroke-drawing component directly. This consists of a direct interface to the nonphotorealistic rendering system, which takes as input pen-control commands and draws these in a way resembling human motion to the screen. This interface is at a low level, and is intended to give subsequent researchers the ability to replicate our work on affective rendering using other frameworks and algorithms.

2. The high-level interface (Behavioural Drawing) consists of an interface to the behavioural drawing system, which implements the complete two-phase affective rendering pipeline. It takes as input emotions and developmental states, and draws forms which correspond to those states. This interface works at the same level as the other components developed for this workpackage.

4.3 Architecture

4.3.1 Stroke System

The overall goal of the affective rendering component is to provide a toolkit of creating and painting childlike drawing based on emotions. The emotion-guided rendering process could be considered as a visualization of emotions and behaviours. Its assumptions are that the drawing behaviours of children are influenced by their emotions, and vice versa, their development of emotions could be observed by what they are drawing on the paper. The stroke-rendering system supports the expression of these emotions and development by providing a system that can create human-like strokes dynamically in real-time.

The behaviour architecture (described below) selects commands for the agent’s virtual pen, including the following: setting the colour; setting the pressure; resetting the position of the pen; and moving the pen towards a point. The resulting strokes create shapes on the screen. The stroke-rendering component includes a set of APIs that simulate childlike drawing in a specified canvas. Its simulation model (shown in Figure 5-4) is composed of NPR (non-photorealistic rendering) strokes, moving arms and canvas (paper) with given size. The NPR stroke consists of the basic brush type, ink color and maximum width & offset. The track of the moving arm determines the path drawn in the canvas. A lot of dynamic properties, including current position, next position, heading noise and target noise, could be set to mimic the arm movement during the painting process, and control the shape style formed in canvas. The canvas will have a system default background color and size. It could be resized before the drawing task is launched.
There are two major aspects to the stroke system: (1) the determination of the path of the stroke; (2) the calculation of the features of the stroke itself. Here, we will describe each in turn.

4.3.1.1 Stroke Path

The path the stroke travels along determines the final shape of the object drawn: round, straight, etc. While the behavioural drawing system can suggest that the strokes form, for example, a circle, the output will not seem human-like if the strokes move directly, cleanly, and exactly along the line which the behavioural drawing system suggests. Instead, the line must be, like those in children’s drawings, wavery, meandering across the page with error and noise, and with the agent’s ability to control the resulting line decreasing with the speed at which the line is drawn. It is the responsibility of the stroke path subcomponent to determine a suitably errorful path for the stroke, to create the impression of alive- and humanness in the resulting line.

Conceptually, the determination of the path of the stroke is done as follows. At any point, the system considers that it is at a particular point with a particular velocity. It implements two simple algorithms to create a lifelike feel to the line.

1. Noise in the direction it goes. When a stroke command comes in, the system attempts to go towards the target point of the stroke command. Call the target point “TargetExact.” Instead of going directly to this point, the system goes to a “TargetInexact”, which is a point randomly chosen out of a square around the TargetExact, of diameter HeadingNoise (Figure 4-7). The system goes halfway to TargetInexact, then chooses a new target in the square around TargetInexact.
of diameter $\frac{1}{2}$ HeadingNoise. This repeats every time the drawn line comes halfway closer to approaching TargetInexact, so that the system slowly hones in on TargetInexact. In this way the system meanders more or less uncertainly towards the target, creating the effect of lower stages of motor control as one might expect in a small child.

![Diagram](image1)

**Figure 4-7:** The system heads towards an inexact approximation of the target.

2. **Momentum of the arm in drawing.** When the system has calculated the direction it wants to go in, it must work against its previous momentum to go in that direction, by using a variable Force, specified by the calling system. This is implemented in the following way: the system calculates the vector it is currently going to, and the ideal direction. It then adds a vector proportional to the Force to the current direction to bend it towards the ideal direction. So, it starts to move in the direction of the ideal direction, but still maintains some momentum from before. By making Force low, the resulting strokes curve more, whereas with a high Force, the resulting strokes can have sharper angles.

![Arm Simulation](image2)

**Figure 4-8:** Arm simulation adds a sense of dynamics to the movement of the drawings.

### 4.3.1.1 Stroke Attributes

The stroke itself must also appear to be something drawn by an instrument a human being would typically use, varying, for example, in color and pressure. Stroke has
different definitions in different illustration scenarios. In classical pen-and-ink illustration, a ‘stroke’ is produced by placing the point, or ‘nib’, a pen in contact with the paper and allowing the nib to trace out a path. The thickness of the stroke can be varied by varying the pressure of the nib. In brush-painting, a stroke is a set of parameters (e.g. position and pressure) which evolve as a function of an independent variable. We use the first definition of stroke.

4.3.1.1.2 Overall stroke specification

The component user can set the attributes of strokes such as brush-types and ink-colors, defining the track of arm-moving and its perturbation, feedback what have been drawn actually in canvas, and resizing the paper/canvas and saving the painting result into JPEG file.

The pipeline for real-time simulation of drawing in the stroke system is divided into five sub-components:

1. Initialization of drawing and paper size;
2. Setting the drawing parameters and launching the drawing task;
3. Drawing graphical segments according to the received commands;
4. Feedback the painted graphic data;
5. Stop the current drawing task.

At first, the system will create a NPR stroke object using the default drawing parameters and paper size, and then the Arm object will be created to mimic the moving of arm. The component user can also resize the canvas during the initialization phase. Then application program sets the brush type of the stroke, moves the arm to the current specified position, and specifies the perturbation during the drawing. The real pixel-painting will be fired by the AddStroke command, and generate the path on the canvas. The 2D geometric data of the path will be transferred back to the user automatically after the current AddStroke command has been executed successfully. The overall drawing task can be terminated by the Stop command.

4.3.2 Behavioural Rendering

Within the SAFIRA framework, the task of the Affective Rendering component is to take as input emotional and developmental states, and to create as output animated images rendered to the screen, which in their form, dynamics, and pacing reflect the current emotional and developmental state. The drawings should not be a simple, one-to-one mapping of internal emotion to an observable graphical token (e.g. a 3-inch red triangle that appears whenever the agent is angry). Instead, graphics should be dynamic, be newly generated in real time, and evoke complex and open-ended interpretations of multiple simultaneous emotional states.
The basis of the graphics engine is a behaviour-based, reactive control architecture similar to those used to control animated computer characters. This architecture selects drawing forms and styles based on the current emotional state, and executes them by sending commands to control the movement of a virtual pen over the screen. These commands are then implemented by the underlying stroke renderer, which implements a simple model of the natural variation of human drawing. This section will describe the behaviour architecture developed for Affective Rendering, explain how the architecture worked in practice and the lessons learned in its use, and present possibilities for future work.

4.3.2.1 Requirements

The graphics engine must generate drawings in real time that reflect in content, form, and dynamics the current state of the emotional model, which may include up to 11 different emotional states to express. Choosing what to draw and how to draw it is a variation of one of the fundamental problems in agent research: the appropriate, dynamic selection and execution of activity (see, for example, [Maes 1990]). Hence, we solve the drawing problem by developing a specialised agent architecture.

This architecture’s decision mechanism must allow the agent to make complex and interdependent decisions about what to draw, while remaining reactive to both the drawing process and to new emotional input from the user.

The agent architecture must:

- Be interruptable / reactive to
  - Changing emotions
  - Visual feedback from drawing process
- Steer the drawing process over time (a control problem, rather than a standard discrete-action planning problem).
- Make (complex) decisions about planning the drawing process. The decisions are complex in the following ways:
  - Multiple attributes must be decided upon
  - There are potentially complex reasons for a particular action
Decisions are interrelated (Structure affects form affects color affects...)

A changing developmental process. I.e., the decision structure changes over time, as the agent develops new abilities.

A simplifying attribute of this problem is that correctness is not strongly pertinent. I.e., it does not matter if the agent chooses the "wrong" (i.e., nonoptimal) drawing from time to time. It is more important that something is being drawn, and that it is responsive to the agent's emotions.

### 4.3.2.1.1 Characteristics of the Drawing Problem

The drawing problem proceeds at multiple, hierarchical levels. Those levels are as follows:

1. Decide structure (if any)
2. Decide "theme" (if any)
3. Decide
   a. Combinations of forms (if any)
   b. Variations of forms (if any)
4. Decide
   a. Form
   b. Color
   c. Pressure
5. Draw decided-upon form (with pertinent structure, variation, color, pressure, etc.).

Each level of the problem must be reactive, both to outside events and to decisions elsewhere in the hierarchy. For example, the form level must be reactive to problems in actually drawing the form, while the structure level could decide to change the structure of the picture owing to mistakes in or decisions about the form being drawn.

Also, the levels must be reactive to new emotional input.

### 4.3.2.2 General Approach

This analysis suggests that an appropriate approach to the drawing problem may be as a multi-level reactive control problem, such as addressed by Brooks's subsumption architecture [Brooks 1986] [Brooks 1991]. A particularly suitable approach may be that
proposed by the Airsoar architecture, which is a multi-level reactive hierarchy, with a fixed set of levels but multiple goals at each level, and which integrates control and planning [Pearson 1993].

The Airsoar approach, which is used to drive a simulated airplane, works as follows:

- The agent has multiple, hierarchical levels ("problem spaces") at which it works simultaneously.
- Each problem space works at a different level of abstraction.
- Each problem space may contain multiple goals.
- Each problem space continually monitors environmental conditions and changes its actions based on changes to the environment.
- When higher-level problem spaces change, the lower-level problem spaces are retracted and redecided with respect to higher-level decisions.

A system like Airsoar may be modified to be appropriate for the rendering problem as follows:

- Problem spaces contain behaviours, rather than goals. These behaviours must run "in parallel".
- Higher-level problem spaces decide which behaviours run at lower-level ones, and set their parameters.
- Bits of behaviours must be run in a round-robin fashion, so that they can be continually interruptable.
- Some kind of communication system between behaviours must be set up, for example a blackboard architecture.

4.3.2.3 Behaviour Architecture

Conceptually speaking, the behaviour architecture organizes behaviours in multiple levels which run simultaneously, as shown in the following figure. These levels range in complexity from low to high, from simple, direct attributes of drawing to complex decisions about the nature of what to draw. Along with the complexity, there is also a gradual increase in time unit. Decisions at the lowest level are made every tick, while decisions at the high level (e.g., the theme of the drawing) are made and revised rarely.
In practice, two levels are implemented: (1) a low level, which sets the drawing attributes, and (2) a high level, which selects and executes the drawing forms. Both levels incorporate multiple emotions in every aspect of decision-making and execution. In order to avoid watering down the emotional effects by averaging between all possible emotions, the behaviour architecture considers only the top 4 strongest emotions in its decisions.

### 4.3.2.3.1 Low Level: Drawing Attributes

The low level resets the drawing attributes – pressure, speed, wobbliness, and colour – once per animation frame (15 times a second) to reflect the current emotional state. The drawing attributes are the most low-level decisions the behavior system must make. For pressure, speed, and wobbliness, this is done by assigning a preferred level for each emotion which can be expressed through that degree of freedom. For example, extreme anger will greatly increase speed, while extreme sadness will greatly reduce it. At any point in time, the current value of the attribute is calculated by multiplying the preferred level for each emotion by the current strength of that emotion (measured from 0.0 to 1.0) and averaged over the strengths of the total number of emotions considered. Thus, the strongest emotions have the most influence on the resulting level of the attribute.
Colour is calculated in a different manner, since colour cannot meaningfully be simply averaged. Instead, each emotion that is expressible through colour is assigned a particular colour range. This colour range consists of 3 subrange of hue, saturation, and brightness. The ranges are marked with how much they express the corresponding emotion; so, for example, a very saturated red will express more anger than a less saturated red. On the other hand, a less saturated blue will express more coldness than a more saturated blue, so that its range will be marked as an inverse variance. In this way, the colour can be varied depending on the level of the emotion being expressed, so that subtleties of emotional strength can be conveyed through shades of colour.

When the agent begins to draw a new form (i.e., has lifted its pen from the paper), it selects the colour that corresponds to the strongest emotion expressible through colour.\(^2\) A specific color in the color range corresponding to that emotion is chosen, with an appropriate hue, and with its levels of saturation and value selected to correspond to the strength of the emotion. While drawing a form, the agent varies the saturation and brightness (but not the hue) of that colour to express the current emotional strength. In addition, a small amount of noise is added to the saturation and brightness of the colour so that the stroke can still be seen when the agent is scribbling over previously drawn lines. This also adds to the colour interest and attractiveness of the drawn scribblings.

### 4.3.2.3.2 High-Level Architecture: Behaviours

The high-level architecture must continually redécide the form to draw and execute it appropriately for the currently active emotions. We implement this in a behaviour-based framework, as described e.g. in [Maes 1990]). In this framework, each form the agent can draw is implemented as a ‘behaviour,’ a self-contained piece of code which is reactive to changes in the environment – in this case, the emotional model – and responds to them by executing actions – in this case, by sending drawing commands to the pen. The agent stays up to date by using an action-selection algorithm; once per frame, it redécides whether the current form is appropriate, and then engages in a small amount of computation for that form. This allows it to continuously respond to changes in input from the emotional model.

\(^2\) If there are no strong emotions, it uses beige.
The forms that have been implemented are those that have been identified in the literature on the early stages of children’s drawings (see especially [Kellogg 1970]). It is important to note that a single behaviour does not correspond to a single graphically visible form. Rather, a behaviour describes procedurally how a shape is drawn, and can result in many graphical variations of that form. This is essential for the interpretational complexity of output (i.e., the forms are not simply ever-repeating tokens). It is also essential for expressing emotion; each behaviour encodes within it the variations necessary to express many different emotions by varying the appearance of the form.

![Some available forms: circular scribbles, line scribbles, crosses, centred circles, ladders.](image)

Figure 4-13: Some of the available forms: circular scribbles, line scribbles, crosses, centred circles, ladders.

![A single behaviour can draw many variations of a form. These are all line scribbles.](image)

Figure 4-14: A single behaviour can draw many variations of a form. These are all line scribbles.

The fundamental unit of computation for the behaviour architecture is a behaviour. Behaviours are defined as Java classes, where each behaviour corresponds to and implements a development stage of which the agent is capable. This general class has the following components:

- **An activation level.** A behaviour is activated when its developmental state has been triggered. This allows it to begin being considered for execution.

- **An appropriateness calculator.** The appropriateness calculator determines how appropriate the behaviour is for expressing the current emotional state. The degree to which any particular form expresses an emotion is derived in part from...
the depth-psychological literature on children’s drawings, which states the emotional interpretation of different shapes. To this are added considerations about the concrete implementation of the behaviour and how it can vary with different emotional states. The appropriateness calculator is used by the behaviour architecture to select the behaviour that can best express the current emotional state. If a behaviour is not activated, its appropriateness is zero.

Once a behaviour class has been chosen for execution, it is instantiated to create a concrete, running behaviour. These behaviour instances have the following components:

- A set of **parameters**, which specify the detailed functioning of this instance of the behaviour class. Parameters vary depending on the form, and include such aspects as size, radius, or number of iterations. The parameters can be modified externally to alter the functioning of the behaviour - also while the behaviour is running. Communication between behaviours occurs when they modify each other’s parameters. Some of the behaviour’s parameters are particularly useful for expressing emotions. These are defined as **emotional variables**. Emotional variables recalculate their values every time they are used in order to reflect the current emotional state. This is done by first setting a "home value," which represents the variable’s value in the absence of emotional influences. Coefficients are set up that express how much this home value should be varied based on emotional state. For example, when introverted, the radius of a circle should be drawn smaller; this is represented by choosing a coefficient less than 1 for the introverted emotional state. When the value of an emotional variable is retrieved, it calculates the combined emotional influence of the current emotional state on the variable and returns the "home value" of the variable multiplied by that coefficient. This allows the parameters of the behaviour to adapt continuously to the changing emotional state.

- An **initialization function** that sets the behaviour’s parameters to reasonable values.

- A method **expressEmotions()** that tweaks the behaviour’s parameters to express the current emotional state. This allows the behaviour to automatically modify itself to reflect emotions appropriately when it is created.

- A method **runOneStep()** that, when executed, implements one step of the behaviour. A “step” should be brief, no more than 100 msecs. This is to allow real-time response, and rethinking the choice of what to do frequently between executions. Typically, this method engages in some computation to determine the next stage of the stroke, and then implements this stage by calling rendering methods of the stroke system. Frequently, behaviours also create subbehaviours, modify their parameters, and run them in order to implement their own functionality. For example, the centred circle spawns first a circle behaviour, then a cross behaviour, where the parameters of the cross are altered to be at the
centre of the previously drawn circle. This method is passed the current drawing attributes, which it uses to determine the path and strength of the stroke.

- A drawing sensor. This is a data structure which keeps tracks of the points which this behaviour has drawn so far. These points are returned by the stroke system, and represent the actual points drawn (rather than the points where the behaviour attempted to draw; because of the noise in the drawing system, which simulates human-like drawing, there is usually a divergence between these two!). These sensed points can then be used to modify later behaviour. For example, the “ladder” drawing behaviour draws two lines, and then draws crossbars between the sensed points from the lines.

4.3.2.3.3 Levels

In our initial conception of the architecture, not only the architecture as a whole, but also the behavioural substrate was organized into multiple levels. A single level consists of the following pieces:

- A set of behaviours, as defined above.

- A currently running behaviour instance. This is an instance of the Java class for that behaviour, which is currently in the middle of execution.

- A selection method. This method allows the level to choose among the behaviours that are at that level. The selection method works by checking the appropriateness of each behaviour for the current emotional state, then choosing probabilistically among them, weighted by the appropriateness. The selection method includes the following attributes:
  
  o The appropriateness of the currently running behaviour is doubled, in order to reduce the problem of dithering [Blumberg 1994] [Blumberg 1996].
  
  o Behaviours inhibit themselves [Blumberg 1994] [Blumberg 1996]. That is, when a behaviour has run, it triggers an increase in its inhibition level, which in turn decreases its appropriateness. This inhibition is cumulative; once a behaviour has run a few times it is unlikely to be chosen again. This increases the variance of observed behaviour for users.

- A runOneStep method. This method causes the currently selected behaviour to execute for one step.

In practice, multiple levels within the behavioural architecture were not necessary. It proved just as usable and more flexible to implement all behaviours as a single level (though behaviours can call other behaviours directly, so that more than one behaviour may be executed for a particular form).
4.3.2.3.4 Total Architecture

The behaviour architecture consists of the high and low levels as defined previously. At each tick, the behaviour architecture follows a typical action-selection algorithm [Maes 1989], continually interleaving of selecting and running one step of the currently (probabilistically) optimal behaviour. Specifically, the architecture engages in the following steps:

1. The emotional model used by the drawing component is updated to reflect new emotional input.
2. For each newly activated developmental state, the corresponding behaviour is activated.
3. For each newly deactivated developmental state, the corresponding behaviour is deactivated.
4. The drawing attributes for all behaviours are calculated to reflect the current emotional state.
5. The selection method is run to determine the currently active behaviour. This involves calculating each behaviour’s appropriateness, then probabilistically choosing the best behaviour, as described above.
6. One step of the level’s best behaviour is run.

In practice, 21 behaviours were implemented. As described in D7.3, writing a behaviour involved the most work in two areas: (1) fine-tuning the geometry and pacing of the behaviour; (2) varying the behaviour properly for the 22 different emotions, including setting the appropriateness of the behaviour and varying the behaviour’s parameters. The first behaviours written generally dealt with pure forms --- drawing lines, drawing circles; then came behaviours which varied forms --- for example, varying a circular scribble to create a spiral; finally came behaviours which combined already existing behaviours --- for example, a circle with a cross at the centre. These combination behaviours were trivial to write, implying that further extension of the behaviour to deal with complex combinations of forms should be straightforward.

4.4 Toolkit Integration

The Affective Rendering component is part of the SAFIRA Toolkit. Further details can be found in the component documentation provided with the toolkit.

4.5 Use

The Affective Rendering component was used in the Influencing Machine demonstrator.
4.6 References


5. Affective Speech

5.1 Overview

The ASM is a software component, which belongs to the Expression Components of the SAFIRA toolkit. They are responsible for the communication and expression of affective traits – like emotion and personality. The task of the ASM is to generate affective speech, vary in length and intonation of words according to several aesthetic and domain dependant modalities.

The use of the ASM in an emotionally expressive agent application will enhance the global believability of such systems. It has also the potential to spirit the human-machine interaction, which would result in a more enjoyable and engaging experience on the users’ side. Newer studies [C. Nass, K. Lee 01] have shown that users prefer applications which mirror the users’ personality character. If in addition the application will react in an “emotional” manner, this may raise the acceptance even more. A priori speech is well suited to convey emotions.

A first step into this direction is the PhD thesis by Hovy which describes one of the first natural language generators that also considers pragmatic goals, such as conveying the social relationship between speaker and listener, during the generation process [Hovy 87]. His generation system PAULINE is able to produce a number of linguistic variants in dependency of parameters, such as the tone of interaction, the speakers’ opinion, and the available time.

While Hovy focuses on the generation of text, Walker and colleagues examine how social factors influence the semantic content, the syntactic form and the acoustic realisation of conversations [Walker et al. 97]. The generation of their dialogues is essentially influenced by the power the listener has on the speaker and the social distance between them. For speech synthesis, Walker and colleagues rely on the pioneering work by [Cahn 90] which represents one of the first approaches to the production of affective speech.

Rosis and Grasso [Rosis & Grasso] present an approach to affective text generation for medical applications. They start from the observation that the behaviour of a patient is highly influenced by his or her affective involvement and investigate how emotions and personality can be considered during the design of a natural-language generator. In particular, they argue that the implementation of an affective text generator requires a relaxation of the Gricean maxims. For instance, an affective text generator does not need to be sincere under all circumstances. Instead, Rosis and Grasso recommend to avoid insincerity where not needed.
5.2 Specification

The overall generation process is divided into two steps. The first step is to create an utterance structure – an internal data representation, which is annotated by several speech parameters, like pitch, volume, etc. We call them prosodic mark-ups. The utterance structure is the output of the text generator. This output is generated by some kind of rules, which are applied on the input. This input defines what to generate and how to generate. We call it input structure. In the second step utterance structure is passed to an integrated speech synthesiser, which then creates affective speech.

In detail, the generation process is divided into the following tasks:

- Mapping of high-level input on the text generator’s input format. See subsection 5.3.4.
- Generation of utterance structures. Basically the utterance structure is a document, which holds beneath the utterance text additional mark-up sequences, which control the speech synthesiser. Subsection 5.3.5 describes the supported mark-up languages. The sub-tasks are:
  - Selection of prosodic mark-ups according to the actual prevailing settings of emotions and personality.
  - Enrichment of utterance structure with selected prosodic mark-ups to create annotated utterance structures for controlling a given speech synthesiser.
- Speech synthesis of the annotated utterance structure. This step is optional.

5.3 Architecture

Figure 5-1 gives a schematic overview on the ASM architecture. The design of the ASM includes three interfaces to external modules.

ASM Interfaces are bound to several requirements, which must be met by the individual external software module:

- The Knowledge Interface, which provides methods for the manipulation of the text generator’s knowledge, i.e. the utterance generation rule set. The ASM provides methods for loading different rule sets during run time. Requirement: Knowledge bases for the text generator must be stored in files.
- The Text generator Interface, which controls an external text generator module via a socket communication. Requirement: The text generator must be controllable by a socket interface.
The **Speech Synthesiser Interface**, which controls an external speech synthesiser via a JAVA™ – API. For integration of an external speech synthesiser it must provide a JAVA™ Speech API (JSAPI) interface.

Requirement: The speech synthesiser must be controllable by the JSAPI.

The Input/Output (I/O) – Interface handles the communication with other modules of the SAFIRA toolkit or other applications. If using the SAFIRA toolkit, the ASM can use the Central Register Service (CRS) for obtaining and exchanging relevant data. This is not used in the current version of the ASM, because the text generator’s input structure is flexible enough for storing all information, which is needed for the generation process. In general, this means that the ASM have no internal domain knowledge.

The I/O – Interface implements a socket and a common JAVA™ – API for controlling each sub-module (Knowledge, Text Generator and Speech Synthesiser) and for the invocation of the speech generation process. The socket interface provides a XML message interface, which maps message commands on ASM methods.

### 5.3.1 External Software

First of all, a text generator and a speech synthesiser are need. They must meet the particular – see above – interface requirements, so that they can be integrated. The subsections below describe the features of the text generator and the speech synthesiser we currently use.

#### 5.3.1.1 Template-based Generator TG/2

For the generation of language, we use the DFKI generator TG/2 [Busemann et al. 00]. TG/2 stands for a new generation of template-based generators, which integrates canned text, templates and context-free rules into a single formalism, which we call *text*...
generation rule set. A major advantage of TG/2 with respect to the SAFIR A project is the fact that it can be parameterised to produce the preferred formulation first (regarding style, grammar, fine-grained rhetoric’s, affective state etc.). TG/2 also contains methods for pre- and post processing data. This is relevant for the mapping of an affective state to specific prosodic mark-ups. The effort to accommodate TG/2 to new tasks is approximately three person-months, taking the modelling task, which is described in section 4 into consideration. The full version is available in Lisp, TG/2 Light as JAVA™ version. Subsection 5.3.4 gives detailed information about the TG/2 modelling capabilities.

5.3.1.2 Speech Synthesiser

For the synthesis of spoken language, we use the freely available FESTIVAL speech synthesis system. A major advantage of FESTIVAL with respect to the SAFIRA project is the fact that it contains a basic implementation of SABLE [R. Sproat et al. 98]. For a more flexible integration, we have implemented a full JSAPI [JSAPI] interface.

In addition, we use the speech synthesisers which come with the Microsoft Agent Technology [MS Agent].

5.3.2 Hardware Requirements

Tests have shown that the generation of text and speech needs specific hardware requirements to meet the real-time characteristics of the SAFIRA toolkit. The hardware requirements in detail are:

- An IBM-compatible PC, with 256MB main memory. This is basically used for holding the voice data of the speech synthesiser and the generation rules, respectively the text templates of the text generator. The CPU speed has a direct impact on the overall generation speed. A minimal setting for the CPU is an Intel Pentium III 800 MHz or an AMD Athlon 900 MHz. If the PC system is intended for executing other programs in parallel to the ASM, it should have more main memory and a faster processor.

- To install the system the minimal space of 300MB on the computer’s hard-drive is needed.

- If it is intended to produce spoken speech, the computer system should have a sound card. To avoid technical problems, a sound card of the CREATIVE LABS sound card manufacturer should be used. Relevant sound cards are the Sound Blaster 128 PCI or one of the Sound Blaster Live! series.

- The ASM basically runs under nearly every operating system, which provides a JAVA™ – Edition (J2SE™) Software Development Kit (JSDK). Due to the requirements of the external speech synthesiser Festival and the text generator
TG/2, the ASM currently runs within the Windows NT™ 4.x, Solaris 2.x, and Linux (Kernel version 2.4.x) operating system (OS).

In the case the ASM is for the generation of affective utterances structures and not for generation lower hardware requirements are needed.

5.3.3 Software Environment

Due to the fact, that the ASM is developed with the JAVA™ programming language, it primarily needs the JAVA™ runtime environment for execution. The following list shows in detail, which (other) modules are needed:

- JAVA™ 2 Standard Edition (J2SE™) Software Development Kit (JSDK) version 1.3.1 or higher [JSDK]
- JAVA™ Speech API (JSAPI) [JSAPI]
- IBM XML Xerces parser version 3.1.1 or higher [Xerces]
- Allegro Common LISP runtime image of the TG/2 text generator [Busemann et al. 00]
- Festival Speech Synthesiser version 1.4.2 or higher with at least one voice packet (i.e. the English male voice) [Festival]

5.3.4 TG/2 interface

The following subsection will describe the TG/2 sub-component modelling languages.

5.3.4.1 TG/2 Modelling Languages

This subsection presents the different modelling languages of the TG/2 text generator. The following is a brief description of some relevant properties TG/2 ’s Template Generation Language (TGL).

Here is a BNF\(^3\) definition for TGL rules:

\[
<\text{rule}> ::= (\text{PUT-TEMPLATE} \ <\text{symbol}> \ <\text{string}> \ <\text{tgl-rule}>)
\]

\[
<\text{symbol}> ::= '\text{Any symbol different from NIL Ex: PRONOUN}'
\]

\[
<\text{string}> ::= 'A sequence of characters embraced by double quotes'
\]

\(^3\)“<” and “>” delimit BNF variables. ”|” denotes alternative definitions. ”*” after a BNF variable denotes zero or more occurrences. Postponed ”+” denotes one or more occurrence. Single quotes delimit informal, abbreviating explanations. All other right-hand side symbols are terminal symbols of the language defined.
<tgl-rule> ::= (:PRECOND (:CAT <symbol> :TEST (<test>+))

:AIM (<template-part>+) :CONSTR {<constraints>} :SIDE-EFFECTS <code>

<test> ::= 'A call to a lisp function evaluating to T or NIL'
<template-part> ::= (<label> :RULE <category> <accfun>) |

(<label> :OPTRULE <category> <accfun>) |

(<label> :FUN <funcall>) |

<string>

<label> ::= X1 | X2 | X3 | ...

<accfun> ::= 'A predefined access function'

<funcall> ::= 'A call to a lisp function evaluating to a string'

<constraints> ::= <symbol> (<label0>+) :EQ |

<symbol> (<label0>+) :VAL <symbol>

<label0> ::= X0 | <label>

<code> ::= 'arbitrary Lisp code to be evaluated'

5.3.4.2 Input Format

The input format corresponds to tree structures represented as sets of pairs consisting of a feature name and a feature value. Names are symbols. Values are symbols or tree structures.

Here is a BNF definition for the input format:

<structure> ::= { <structures> } | <simple-str>
<structures> ::= <structure> | <structure>, <structures>
<simple-str> ::= <ascii>+ | [ <label-value-pair>* ]
<label-value-pair> ::= (<label> <value>)
<label> ::= <ascii>+
<value> ::= <ascii>+ | *NULL* | <structure>
<ascii> ::= a | ... | z | A | ... | Z | 0 | ... | 9 | # | $ | & | * | - | + | _
The curly brackets denote disjunctive structures. The special value *NULL* terminates list representations. Lists of feature structures should be represented according to the following recursive definition:

\[
\text{<list>} ::= \text{*NULL*} | \[(\text{FIRST <structure>}) (\text{REST <list>})\]
\]

Usage note: The input to TG/2 is usually a complex feature structure. It may, however, be of interest to have a grammar enumerate its derivations without using any input. Since a call to TG/2 with an empty feature structure ("["]") leads to an immediate stop of computation - there is nothing to generate from, after all! - it is recommended to use a dummy atom such as “no-input” as an argument.

5.3.4.3 Template Rules

The following is a brief description of some relevant properties TG/2's template generation language (TGL). There is also information about how to create, modify, or extend a TGL rule set. Here is a BNF definition for TGL rules:

\[
\text{<rule>} ::= (\text{PUT-TEMPLATE <symbol> <string> <tgl-rule>})
\]

\[
\text{<symbol>} ::= \text{'Any symbol different from NIL Ex: PRONOUN'}
\]

\[
\text{<string>} ::= \text{'A sequence of characters embraced by double quotes'}
\]

\[
\text{<tgl-rule>} ::= (:\text{PRECOND} (:\text{CAT <symbol>} :\text{TEST} (<test>+)) :\text{ACTIONS} (:\text{TEMPLATE} <template-part>+ { :\text{CONSTRAINTS} (<constraints>+)} { :\text{SIDE-EFFECTS} <code> })})\]
\]

\[
\text{<test>} ::= \text{'A call to a lisp function evaluating to T or NIL'}
\]

\[
\text{<template-part>} ::= (\text{<label> :RULE <category> <accfun>}) |
\quad (\text{<label> :OPTRULE <category> <accfun>}) |
\quad (\text{<label> :FUN <funcall>}) |
\quad <\text{string}>
\]

\[
\text{<label>} ::= X1 | X2 | X3 | ... \]

\[
\text{<accfun>} ::= \text{'A predefined access function'}
\]

\[
\text{<funcall>} ::= \text{'A call to a lisp function evaluating to a string'}
\]

\[
\text{<constraints>} ::= <\text{symbol} (\text{<label0>}+:):\text{EQ} |
\quad <\text{symbol} (\text{<label0>}+:):\text{VAL} <\text{symbol}>}
\]

\[
\text{<label0>} ::= X0 | \text{<label>}
\]
For further explanations see [Busemann 1996] and [Busemann et al. 2000]. A manual, which holds the basic explanation of the TG/2 system and modelling possibilities, comes with the ASM package.

5.3.5 Speech Synthesis Mark-up Language Specification

This section describes the used mark-up languages for controlling the output of speech synthesisers, which are used together with the ASM.

5.3.5.1 SABLE

SABLE [R. Sproat et al. 98] is an XML (Extensible Markup Language)/SGML (Standard Generalized Markup Language)-based [Cover et al. 91] mark-up scheme for text-to-speech synthesis, developed to address the need for a common TTS control paradigm. SABLE is based in part on two previous proposals by a subset of the present authors: the Spoken Text Markup Language (STML [R. Sproat et al. 97]; and see also [Taylor P., Isard A 97] for an even earlier proposal SSML) and the Java Speech Markup Language (JSML) [JSML].

The SABLE specification is part of a larger set of mark-up specifications for voice browsers developed through the open processes of the W3C. It is designed to provide a rich, XML-based mark-up language for assisting the generation of synthetic speech in web and other applications. The essential role of this mark-up language is to give authors of “synthesiseable” content a standard way to control aspects of speech output such as pronunciation, volume, pitch, rate and etc. across different synthesis-capable platforms.

Here is a definition of the element and attributes for prosody and style:

5.3.5.2 Voice Element

The voice element is a production element that requests a change in speaking voice.

The attributes are:

- gender: optional attribute indicating the preferred gender of the voice to speak the contained text. Enumerated values are: "male", "female", "neutral".

- age: optional attribute indicating the preferred age of the voice to speak the contained text. Acceptable values are of type (integer)

- category: optional attribute indicating the preferred age category of the voice to speak the contained text. Enumerated values are: "child", "teenager", "adult", "elder".
• **variant**: optional attribute indicating a preferred variant of the other voice characteristics to speak the contained text. (e.g. the second or next male child voice). Acceptable values are of type (integer).

• **name**: optional attribute indicating a platform-specific voice name to speak the contained text. The value may be a space-separated list of names ordered from top preference down. Acceptable values are of the form (voice-name-list).

Example:

```xml
<voice gender="female" category="child">Mary had a little lamb,</voice>
<!-- now request a different female child's voice -->
<voice gender="female" category="child" variant="2">It's fleece was white as snow.</voice>
<!-- platform-specific voice selection -->
<voice name="Mike">I want to be like Mike.</voice>
```

### 5.3.5.3 Emphasis Element

The **emphasis** element requests that the contained text be spoken with emphasis (also referred to as prominence or stress). The synthesiser determines how to render emphasis since the nature of emphasis differs between languages, dialects or even voices.

The attributes are:

- **level**: the "level" attribute indicates the strength of emphasis to be applied. Defined values are "strong", "moderate", "none" and "reduced". The default level is "moderate". The meaning of "strong" and "moderate" emphasis is interpreted according to the language being spoken (languages indicate emphasis using a possible combination of pitch change, timing changes, loudness and other acoustic differences). The "reduced" level is effectively the opposite of emphasising a word. For example, when the phrase "going to" is reduced it may be spoken as "gonna". The "none" level is used to prevent the speech synthesiser from emphasising words that it might typically emphasise.

Example:

```xml
That is a <emphasis> big </emphasis> car!

That is a <emphasis level="strong"> huge </emphasis> bank account!
```

### 5.3.5.4 Break Element

The **break** element is an empty element that controls the pausing or other prosodic boundaries between words. The use of the break element between any pair of words is optional. If the element is not defined, the speech synthesiser is expected to
automatically determine a break based on the linguistic context. In practice, the "break" element is most often used to override the typical automatic behaviour of a speech synthesiser.

The attributes are:

- **size**: the "size" attribute is an optional attribute having one of the following relative values: "none", "small", "medium" (default value), or "large". The value "none" indicates that a normal break boundary should be used. The other three values indicate increasingly large break boundaries between words. The larger boundaries are typically accompanied by pauses.

- **time**: the "time" attribute is an optional attribute indicating the duration of a pause in seconds or milliseconds. It follows the "Times" attribute format from the Cascading Style Sheet Specification, e.g. "250ms", "3s".

Examples:

```
Take a deep breath <break/>then continue.

Press 1 or wait for the tone. <break time="3s"/>I didn't hear you!
```

5.3.5.5 Prosody Element

The prosody element permits control of the pitch, speaking rate and volume of the speech output.

The attributes are:

- **pitch**: the baseline pitch for the contained text in Hertz, a relative change or values "high", "medium", "low", "default".

- **contour**: sets the actual pitch contour for the contained text. The format is outlined below.

- **range**: the pitch range (variability) for the contained text in Hertz, a relative change or values "high", "medium", "low", "default".

- **rate**: the speaking rate for the contained text, a relative change or values "fast", "medium", "low", "default".

- **duration**: a value in seconds or milliseconds for the desired time to take to read the element contents. Follows the "Times" attribute format from the Cascading Style Sheet Specification. e.g. "250ms", "3s".

- **volume**: the volume for the contained text in the range 0.0 to 100.0, a relative change or values "silent", "soft", "medium", "loud" or "default".
Relative changes for any of the attributes above are specified as floating-point values: "+10", "-5.5", "+15.2\%", "-8.0\%". For the pitch and range attributes, relative changes in semitones are permitted: "+5st", "-2st". Since speech synthesizers are not able to apply arbitrary prosodic values, conforming speech synthesis processors may set platform-specific limits on the values. This is the second of only two exceptions allowed in the conformance criteria for an SSML processor.

Example:

```
The price of XYZ is <prosody rate="-10%">
  <say-as type="currency">$45</say-as></prosody>
```

### 5.3.5.6 Pitch contour

The pitch contour is defined as a set of targets at specified intervals in the speech output. The algorithm for interpolating between the targets is platform-specific. In each pair of the form (interval, target), the first value is a percentage of the period of the contained text and the second value is the value of the "pitch" attribute (absolute, relative, relative semitone, or descriptive values are all permitted). Interval values outside 0\% to 100\% are ignored. If a value is not defined for 0\% or 100\% then the nearest pitch target is copied. Example:

```
<prosody contour="(0\%,+20)(10\%,+30\%)(40\%,+10)">
  good morning</prosody>
```

A detailed description of the general document structure, all usable elements and attributes are defined on the related W3C web site http://www.w3.org/TR/speech-synthesis.

### 5.3.6 Microsoft Agent Speech Output Tags

Microsoft Agent [MS Agent] does not support all the tags specified in the Microsoft Speech API. In addition, support for some parameters may depend on the text-to-speech engine installed. For further information, see [MS Agent Speech Output Tags].

The following tags for modifying speech parameters are supported:

```
Chr, Emp, Map, Pau, Pit, Spd, Vol
```

The tags are primarily designed for adjusting text-to-speech (TTS)-generated output. Only the Mrk and Map tags can be used with sound file-based spoken output.

Speech output tags use the following rules of syntax:

- All tags begin and end with a backslash character (\).
• The single backslash character is not enabled within a tag. To include a backslash character in a text parameter of a tag, use a double backslash (\)

• Tags are case-insensitive. For example, \pit\ is the same as \PIT\.

• Tags are whitespace-dependent. For example, \Rst\ is not the same as \ Rst \.

For C, C++, and Java™ programming, precede backslashes and double quotes with a backslash. For example:
\SPD=120\\VOL=220\\EMP\\What has happened in the \SPD=80\past three hours?

The following paragraphs describe the supported tags:

5.3.6.1 Chr Tag

Sets the character of the voice.

Syntax: \Chr=string\


Remarks: This tag is supported only for TTS-generated output. The range of values for the parameter may vary depending on the installed TTS engine.

5.3.6.2 Emp Tag

Emphasizes the next word spoken. This tag must immediately precede the word.

Syntax: \Emp\

Remarks: This tag is supported only for TTS-generated output. The range of values for the parameter may vary depending on the installed TTS engine.

5.3.6.3 Map Tag

Maps spoken text to text displayed in the word balloon.

Syntax: \Map="spokentext"="balloontext"\

Remarks: spokentext, specifies the text for spoken output. balloontext specifies the text for word balloon output. This tag enables you to use different spoken text than that displayed in the word balloon.

5.3.6.4 Pau Tag

Pauses speech for the specified number of milliseconds.
Syntax: \Pau=number\ 

Remarks: number specifies the number of milliseconds to pause. This tag is supported only for TTS-generated output. The range of values for the parameter may vary depending on the installed TTS engine.

5.3.6.5 Pit Tag
Sets the baseline pitch of the output to the specified value in hertz.

Syntax: \Pit=number\ 

Remarks: number specifies the pitch in hertz. This tag is supported only for TTS-generated output. The range of values for the parameter may vary depending on the installed TTS engine.

5.3.6.6 Spd Tag
Sets the baseline average talking speed of the speech output.

Syntax: \Spd=number\ 

Remarks: number specifies the baseline average talking speed, in words per minute. This tag is supported only for TTS-generated output. The range of values for the parameter may vary depending on the installed TTS engine.

5.3.6.7 Vol Tag
Sets the baseline speaking volume of the speech output.

Syntax: \Vol=number\ 

Remarks: number specifies the baseline speaking volume: 0 is silence and 65535 is maximum volume. The volume setting affects both left and right channels. You cannot set the volume of each channel separately. This tag is supported only for TTS-generated output.

5.4 Toolkit Integration
The SAFIRA Toolkit is a platform for creating affective real time applications. It has a generic module integration interface, which allows any programmer to integrate a module easily. During the project period several modules are developed, which can be used by the Toolkit. The Affective Speech Module is one of them. In association with the SAFIRA Toolkit, the ASM can be used straightforward by any other modules which are registered in the Toolkit.
The ASM generates written affective utterances or affective speech output depending on affective parameters. In general two ways exist how the ASM can be parameterised: by direct definition in the generation command (i.e. `generate say-hello-happy`) or by global parameter definitions of needed parameters. The Toolkit supports these two ways how the ASM can work together with other modules.

The following sections give information about what can be handled by the ASM and which methods are available for Toolkit application designers.

### 5.4.1 Integration Design

The ASM is designed to operate in distributed real time applications. It can communicate via a socket interface. A JAVA™ – API is also provided. In general the ASM can be used in combination with animated agents (lifelike characters), but not restricted to these. The following list gives an overview, in which application scenarios the ASM could be used:

1. **Standalone.** In this scenario the ASM is used for the generation of affective speech, which will be used later. It is also possible to generate written utterances (utterance structures), which, if wanted, have prosodic annotation. In addition, audio files can be produced. This may be useful, if presentations with affective agents are planned before they are shown to users.

2. **Dialogue System:** Here the ASM is used to generate the utterance structures or speech during runtime by request. In a dialogue system that uses the MS Agent technology the ASM can generate annotated utterance structures with prosodic mark-ups in the MS Agent speech synthesiser format.

3. **Human Emulation Environment:** This is the most complex scenario. Here the ASM produces affective speech for a virtual human-like animated talking head or figure. The hardest technical part in this scenario is the synchronisation of the gestures and mimic. Synchronisation on the level of facial animations parameters (FAP) like defined in the mpeg-4 [MPEG 4] format or defined by the FACS format [P. Ekman, W. Friesen, 78] are not created by the ASM. However, the speech synthesiser Festival is able to produce a utterance time dates, which can be used for synchronisation. This has to be done by superior modules, which wrap the output of the ASM in an appropriate format.

As the above list shows the operation of the ASM depends always on component, which triggers it. It modern dialogue system this part is mostly taken by a discourse planner.

### 5.4.2 Interfaces

For accessing globally defined parameters, which are needed for the generation process, the ASM inherits methods from the SAFIRA CRS (see Section 2 – Overall Specification).
In addition, all methods from JSAPI are inherited, for controlling JSAPI – conform speech synthesiser applications.

The methods for creating a speech synthesiser or for communicating with the CRS are not described in this document, see SAFIRA Deliverable 6.1. The JSAPI methods description can be found here [JSAPI].

5.4.3 AffectiveSpeechModule Constructor

```java
javax.speech.Central
   |
   +--de.dfki.safira.speech.AffectiveSpeechModule
```

public class AffectiveSpeechModule extends Central

The AffectiveSpeechModule class (super-class to Central) is the initial access point to all speech input and output capabilities. The Central class provides the ability to locate, select and create speech recognisers and speech synthesisers.

The createSynthesizer method is used to create speech engines. The method accepts a single parameter that defines the required properties for the engine to be created. The parameter is an EngineModeDesc, and may be of the sub-class: SynthesizerModeDesc, see JSAPI documentation.

A mode descriptor defines a set of required properties for an engine. For example, a SynthesizerModeDesc can describe a synthesiser for Swiss German that has a male voice.

An application is responsible for determining its own functional requirements for speech input/output and providing an appropriate mode descriptor. There are three cases for mode descriptors:

- null
- Created by the application
- Obtained from the availableRecognizers or availableSynthesizers methods of AffectiveSpeechModule.

The mode descriptor is passed to the createSynthesizer method of AffectiveSpeechModule to create a synthesiser. The created engine matches all the engine properties in the mode descriptor passed to the create method. If no suitable speech engine is available, the create methods return null.

The create engine methods operate differently for the three cases. That is, engine selection depends upon the type of the mode descriptor:
null mode descriptor: the AffectiveSpeechModule class selects a suitable engine for the default Locale.

Application-created mode descriptor: the AffectiveSpeechModule class attempts to locate an engine with all application-specified properties.

Mode descriptor from availableSynthesizers: descriptors returned by these two methods identify a specific engine with a specific operating mode. AffectiveSpeechModule creates an instance of that engine.

Other required parameters are:

- templateDir: Path to the set of the accepted input structures from the text generator.
- inputStrDir: Path to the set of generation templates (grammar) for the text generator.
- prosodyMapDir: Path to the set of the accepted input structures from the text generator.
- prosodyMapFile: The set of the affective state prosody element mapping

5.4.4 Methods

In the SAFIRA Toolkit environment, the ASM provides methods for controlling the speech and the text generator engines. Amongst other things, these methods allow to manipulate the knowledge base of the text generator during runtime (i.e. changing the prosodic mapping). There are several controlling mechanisms for the speech synthesiser engine, like pausing, cancelling and resuming of an active synthesis task. For the generation purpose, the ASM provides different methods for creating text and/or speech output. It is also possible to generate non-affective speech output. The Java Language Specification [JavaDoc 1.3] is the formal specification for public available methods of the ASM.

Method overview:

```java
public static final boolean createAll(String gammaDir, 
String gammaName, 
String inputStrDir, 
String prosodyMapFile, 
EngineModeDesc EMD)
```

Creates a text generator and the speech synthesiser object.
public static final boolean destroyAll()
Destroys the text generator and the speech synthesiser objects.

public static final boolean createTextGenerator
(String grammarDir,
String grammarName,
String inputStrDir,
String prosodyMapFile)
Creates a text generator object.

public static final boolean destroyTextGenerator()
Destroys the text generator object.

public static final boolean createSpeechSynthesiser
(EngineModeDesc EMD)
Create the speech synthesiser object.

public static final boolean destroySpeechSynthesiser()
Destroys the speech synthesiser object.

public static final File readCurrentInputStructure()
Read the current set of the accepted input structures from the text generator.

public static final boolean setInputStructure
(String inputStrDir)
Set the set of the accepted input structures of the text generator.

public static final File readCurrentGrammar()
Read the current set of generation templates (grammar) from the text generator.

public static final boolean setGrammar
(String grammarDir,
String grammarName)
Set the set of generation templates (grammar) for the text generator.

public static final File readProsodyMap()
Read the set of the affective state prosody element mapping.

```java
public static final boolean setProsodyMap
        (String prosodyMapFile)

Set the set of the affective state prosody element mapping.

```java
public static final boolean generateAffectiveText
        (String inputID)

Generates affective written speech output, the parameter inputID must be member of the set of the accepted input structures of the text generator (see setInputStructureFile method definition).

```java
public static final boolean generateAffectiveSpeechbyText
        (String text)

Generates affective spoken speech output of the text.

Note: In this case the affective state prosody element mapping will be not applied. The content of the text parameter can contain SSML prosody parameters.

```java
public static final boolean generateAffectiveSpeech
        (String inputID)

Generates affective spoken speech output, the parameter inputID must be member of the set of the accepted input structures of the text generator (see setInputStructureFile method definition).

```java
public static final boolean pauseSpeechSynthesiser()

Immediately stops the speech output.

```java
public static final boolean resumeSpeechSynthesiser()

Immediately stops the speech output.

```java
public static final boolean cancelSpeechSynthesiser()

Immediately cancels the speech output.

```java
public static final String getState()

Gives various information about the current state of the ASM.
For detailed information of the public ASM methods, check the JAVA™-Doc reference manual, which is part of the ASM package. For the Toolkit an ASM module definition XML file has been created, which enables the above functionality for any other Toolkit module.

5.5 Use

As an example scenario, we have chosen a wine sales scenario, within a virtual animated (lifelike) character plays the role of wine salesmen. A similar scenario provides one of the SAFIRA demonstrators (cf. James the Butler, see SAFIRA Deliverable 6.1), where the ASM will be integrated, too. However, our modelling of wine sales dialogues does not correlate with the ones in the demonstrator.

First of all, we consider that it is important to define a background for the wine sales character. This helps in the case of identifying a set of possible dialogue actions (dialogue acts) and their verbal affective characteristics. In our case, we defined that the virtual character Herbert – the wine seller – owns a shop. He is forced to sell wine; otherwise he will loose the shop. In this situation, the presence of varied emotions (Herbert’s) is by all means conceivable. Based on this, it is exciting how utterances vary according to the affective state vector of Herbert. This state is derived of the reactions and actions of a customer against the above-described background. Normally this is done by an affective reasoning process [R. Picard 97], which is not part of the ASM. However, if using the ASM, it is relevant to know which affective dimensions are used in the reasoning process. Affective dimensions define the used set of personality and emotions for processing an affective state vector. Technically the affective state vector is an ordered list of elements of each affective dimension.

The modelling of the text generation rule set for creating utterances depends on them. This implies, that an ASM rule set is bound to the affective dimensions and had to be remodelled, if they change. The following sub section gives an overview on the underlying models of personality and emotions.

5.5.1 Preconditions

There a several preconditions, which must be meet, before the ASM can be reasonable used. The basic assumption is, that there is a scenario, in which inherently “emotions” can occur. This means it makes no sense to use the ASM in a database environment to verbalize database query results in an affective way, because there are needed in most cases as true facts. Taking into account that such an affective scenario exists (see subsection above), one should agree on a definition affective parameters and their modelling. If the ASM is used in an dialogue system environment (see subsection Realisation below), a dialogue scheme (dialogue phases and dialogue acts) had to be defined. According to this scheme the discourse management module triggers the ASM.
<table>
<thead>
<tr>
<th>Group</th>
<th>Local Variables</th>
<th>Specification</th>
<th>Emotion Type and Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Well-being Desirability</td>
<td>Appraisal of a situation as an event.</td>
<td>Joy: an event is desirable for self. Distress: an event is undesirable for self.</td>
<td></td>
</tr>
<tr>
<td>Prospect-based Desirability Likelihood</td>
<td>Appraisal of a situation as a prospective event.</td>
<td>Hope: a prospective event is desirable. Fear: a prospective event is undesirable.</td>
<td></td>
</tr>
<tr>
<td>Confirmation Desirability Likelihood</td>
<td>Appraisal of a situation as confirming or disconfirming an expectation.</td>
<td>Satisfaction: a prospective desirable event is confirmed. Relief: a prospective undesirable event is disconfirmed. Fears-confirmed: a prospective undesirable event is confirmed. Disappointment: a prospective desirable event is disconfirmed.</td>
<td></td>
</tr>
<tr>
<td>Attraction Appealingness Familiarity</td>
<td>Appraisal of a situation as containing an attractive or un-attractive object.</td>
<td>Liking: finding an object appealing. Disliking: finding an object unappealing.</td>
<td></td>
</tr>
<tr>
<td>Well-being/Attribution Praiseworthiness</td>
<td>Compound emotions</td>
<td>Gratitude: admiration + joy Angr: reproach + distress Gratification: pride + joy Remorse: shame + distress</td>
<td></td>
</tr>
<tr>
<td>Attraction/Attribution Praiseworthiness Appealingness Familiarity</td>
<td>Compound emotions (extensions*)</td>
<td>Love: admiration + liking Hate: reproach + disliking</td>
<td></td>
</tr>
</tbody>
</table>

Table 5-1: Emotion specification of 24 emotion types (*extended by Elliot).

The following paragraphs describe the work, which had to be done to realize the use of the ASM in the proposed scenario in the above subsection.

### 5.5.1.1 Modelling of Affect

The *Five Factor Model (FFM)* [R. McCrae, O. John 92] is a descriptive model, with the five dimensions (extraversion, agreeableness, conscientiousness, neuroticism, and openness) being derived from a factor analysis of a large number of self- and peer reports on personality-relevant adjectives. In our system we concentrate on the trait *extraversion*.

On the one hand, the present rule set for generating utterance structures relay on the extraversion-introversion dimension. Experiments have shown, that this dimension has a clear impact on verbal expression, like verbal cues, word/phrase selection, and number of words [K. Isbister, C. Nass 00] On the other hand, we use the *politeness* dimension, which is in fact not a basic dimension in the sense of the FFM, but can be considered as a compound dimension of personality dimensions. Politeness has a clear relevance in sales scenarios and is relevant in human-human communication [P. Brown, S. Levinson 87].

With regard to emotions, we relay on the theory of Ortony, Clore, and Collins [Ortony et al. 88], which defines emotions as *valenced reactions* to events of concern to us, actions of those we consider responsible for such actions, and objects.
The OCC model works at the level of emotional clusters (emotion types), where the emotions in each cluster share a similar cause, e.g. the “distress type” describes all emotions caused by displeasing events (sad, distraught, lovesick, etc.). Emotions are defined as the result of three types of subjective appraisal:

1. Desirability of events with respect to the agent’s goals (event-based emotions)
2. Praiseworthiness of actions with respect to a set of standards (agent-based emotions)
3. Appealingness of objects with respect to a set of attitudes (object-based emotions)

Clark Elliot has augmented this model from twenty-two to twenty-four emotion types (see Table 5-1) and used these as the basis of a system for synthesising and recognising emotions based on cognitive reasoning [C. Elliot 92].

As mentioned above, an affective state vector contains elements of each affective dimension. In our example scenario, the affective state vector contains three elements, which are the current dominant emotion, the extraversion setting and the politeness setting. The last two elements can be viewed as fix during the dialogue, whereas the emotion element is dynamic.

All elements consist of a label and a numerical value ([0, 1]), which represents the intensity, e.g. (extraversion = extravert, 1; politeness = polite, 0.5; dominant emotion = hope, 0.8). The labels for extraversion are: extravert, neutral and introvert. The labels for politeness are polite, neutral, and impolite. The labels for the dominant emotion are defined in the above table.

5.5.2 Dialogue Phases

Our wine sales dialogue consists of three main dialogue phases, the init phase, the sales phase, and the end phase. The sales phase is divided into the presentation sub phase and the discussion sub phase.

Figure 5-2 shows an example dialogue between the salesman (S) and a customer (C), which is grouped by the mentioned dialogue phases.
As a next step all dialogue acts, which are needed in a wine sales scenario, are identified. Within the dialogue act it is defined, what utterance the ASM should create.
According to the example dialogue shown in figure 2, figure 3 shows the corresponding dialogue acts. For generating Herbert’s utterances, it only relevant to identify the dialogue acts in the System column. Identifying the users utterance may helps speech recognisers and text analysers for better recognition rates. However, the users dialogue acts may be used for generating fix answers (by the ASM), which the user could use. E.g. by clicking on them if displayed as links on a web page. This was not our focus, and left for further experiences with the ASM.

Whereas the dialogue act roughly points the direction, what subset of the text generation rule set should be used, so called **utterance units** define more precisely, how the utterance fits into the whole dialogue. We use the theory of Traum and Hinkelmann on utterance units for a fine-grained structuring of dialogue acts [D. Traum, E. Hinkelmann 92]. Currently we use the following utterance units for refining the dialogue acts:

- **Initiate.** Initial utterance of a dialogue act
- **Continue.** A continuation of a previous dialogue act, performed by the same speaker.
- **Acknowledge.** Shows understanding of a previous utterance (back channel response)
- **Repair.** Changes the content of the current dialogue act, and defines a new one.
- **Cancel.** Closes the current dialogue as ungrounded. E.g. an unfinished utterance, like: “Can you show me …”.

### 5.5.4 Realisation

For understanding the work of the ASM, it is necessary to know little more about the application, which uses it. As mentioned above, we use an animated character (Herbert) as a virtual wine seller in a virtual shop on a web page. A visitor can engage Herbert in a sales talk by entering his web page. The underlying system consists among other components of a discourse management component, which is responsible for the actions respectively the reactions of Herbert. In fact the discourse management component reasons during runtime, how to present wine at best. This includes which gestures, utterances (in an abstract format) Herbert should uses at a specific dialogue state (we call it **dialogue phase**). In other words: a discourse management component, usually a *discourse planer*, triggers the ASM with a detailed input specification in which the content of the next utterance is defined. The input specification always depends on the current **dialogue act**, which is performed. Common dialogue acts are: inform and request.

Creating a rule set for generating utterances with the ASM requires a strict view of the dialogue. Dialogue phases and dialogue acts help to structure a dialogue. This abstract view defines the interface between the discourse planer and the ASM. The following
two sub-sections give an overview about the identified dialogue phases and dialogue acts, which are needed for this wine sales dialogue example.

5.5.4.1 TG/2 Input Structure

```plaintext
(INPUT [(LANGUAGE %language)
  (OUTPUTFORMAT %outputformat)
  (AP [(EMOTION [(NAME %emotionname) (VALUE %emotionvalue)])
       (EXTRAVERSION [(BIAS %extraversionbias) (VALUE %extraversionvalue)])
       (POLITENESS [(BIAS %politenessbias) (VALUE %politenessvalue)])])
  (CA [(DU %dialogueact)
       (UU [(NAME %utteranceunit) (NEWDU %newdialogueact)])
       (EVENT [(NAME %eventname) (EECCUE %eventeeccue)])
       (OBJECT [(CLASS %objectclass)
                 (TYPE %objecttype)
                 (NAME %objectname)
                 (USEANAPHOR %useanaphor)
                 (EECCUE %objecteeccue)
                 (WINEDETAILS [(YEAR %wineyear)
                               (PRODUCER %wineproducer)
                               (REGION %wineregion)
                               (GRAPE [(NAME %grape)
                                        (POS %grapeposfeature)
                                        (NEG %grapenegfeature)])
                               (COLOR [(NAME %color)
                                        (POS %colorposfeature)
                                        (NEG %colornegfeature)])
                               (FLAVOUR [(NAME %flavour)
                                          (POS %flavourposfeature)
                                          (NEG %flavournegfeature)])
                               (ACIDITY [(NAME %acidity)
                                         (POS %acidityposfeature)
                                         (NEG %aciditynegfeature)])
                               (FRUITY [(NAME %fruity)
                                         (POS %fruityposfeature)
                                         (NEG %fruitynegfeature)])])
                 (ADDRESSEE [(NAME [(PRENAME %h_prename) (SURNAME %h.surname) (SEX %h.sex)])
                             (FLAGS [(KNOWN %known) (KNOWNBUYER %knownbuyer)])])
                 (SPEAKER [(PRENAME %s_prename) (SURNAME %s.surname)])])
```

Figure 5-4: TG/2 input structure of a text generation rule set for utterance structures in a wine sales scenario.
Flat generation of utterances requires a rich specification of input. Figure 5-4 shows the current TG/2 input structure.

A discourse management component can send a full or partial completed input structure to the ASM. Therefore the ASM provides direct JAVA-API calls and a XML messaging interface, which is part of the SAFIRA toolkit (cf. Deliverable 2.2b).

Essential for the generation of utterance structures are the highlighted slots AP (Affective Parameters) and CA (Communication Act). According to what should be generated, different (sub) slots of the above input structure had to be filled with content.

An example for a valid input structure is shown in Figure 5-5.

```plaintext
;;; requestorder msagent
(defun requestorder-e-p-msagents*
  "((LANGUAGE english)
   (OUTPUTFORMAT msagents)
   (AP ([EMOTION [(NAME hope) (VALUE 1)]])
        (EXTRAVERSION [(BIAS extravert) (VALUE 1)])
        (POLITENESS [(BIAS polite) (VALUE 1)]))
   (CA [(DU dialoguecontrol_requestorder)
      (UU [(NAME initiate)])
      (EVENT [(NAME new_buyer) (ECCUE potential_income)])]))")
```

Figure 5-5: Example of a valid input structure.

5.5.4.2 Prosodic mark-up mapping

One important task of the ASM (in fact of the text generator) is the mapping of the affective state vector elements (see paragraph Modelling of Affect) on prosody mark-ups. They are used to manifest the actual affective parameter settings in speech. The set of prosody mark-ups depends on the deployed speech synthesiser and the synthesiser control language. Currently, we use the prosody mark-ups, which are defined by the SABLE language. They allow a manipulation of the following voice parameters: loudness (L), mean fundamental frequency (F0), frequency range (F0R), speech rate (SR). All voice parameters, which are quoted below, are defined for the British/American English voices.

In the case of the extraversion personality settings, we use the below mapping on L, F0, F0R, SR, which are associated with judgments of extraversion [K. Scherer 78, 79][Apple et al. 79] [J. Pittam 94]:

- Loudness is positively associated with extraversion [J. Pittam 94][K. Tusing, J. Dillard 00].
  Formula: \[ L := L + base_L * (extraversion\text{\ intensity} - 0.5) * scaling\text{\ factor} \]
• Fundamental frequency is raised according to the intensity of extraversion [J. Pittam 94][K. Tusing, J. Dillard 00].  
  Formula:  \( F_0 = F_0 + \text{base } F_0 \times (\text{extraversion intensity} - 0.5) \times \text{scaling factor} \)

• The frequency range is raised according to the intensity of extraversion [C. Aronovitch 76][K. Scherer, H. London, J. Wolf 73].  
  Formula:  \( F_{0R} = F_{0R} + \text{base } F_{0R} \times (\text{extraversion intensity} - 0.5) \times \text{scaling factor} \)

• Faster speech rate is associated with extraversion [C. Aronovitch 76][K. Tusing, J. Dillard 00].  
  Formula:  \( SR = SR + \text{base } SR \times (\text{extraversion value} - 0.5) \times \text{scaling factor} \).

The above settings on speech parameters are tested successfully in terms of experimental tests of recognition in a study from Clifford Nass and Kwan Min Lee [C. Nass, K. Lee 01]. In the case of the politeness settings, we use the following mapping on L, F0, and SR, which are associated with judgements of politeness [P. Brown, S. Levinson 87]:

• Loudness is negatively associated with politeness.  
  Formula:  \( L := L + \text{base } L \times (-\text{politeness intensity} + 0.5) \times \text{scaling factor} \)

• A higher mean frequency is negatively associated with politeness.  
  Formula:  \( F_0 := F_0 + \text{base } L \times (- \text{politeness intensity} + 0.5) \times \text{scaling factor} \)

• A higher speech rate is negatively associated with politeness.  
  Formula:  \( SR := SR + \text{base } L \times (- \text{politeness intensity} + 0.5) \times \text{scaling factor} \)

The scaling factor usually depends on the deployed speech synthesiser. For Festival we use the scaling factor 10, for Lernout & Hauspie, we use the scaling factor 30. The final values of L, F0, F0R, and SR for extraversion and politeness will be add together, but weighted. The dominant personality trait has a higher impact.

The mapping of emotions is different. Due to the fact, we use the augmented OCC-Model by Elliot (see subsection Modelling of Affect above), with 24 types of emotion; we need a reduction on basic (Joy, Sadness, Anger, Fear, Surprise, and Boredom) set emotion type before mapping them on voice parameters. The reduction mapping is shown in Figure 5-6. These match in general with the identified basic emotion set from Ekman and Johnson-Laird with Oatley [P. Ekman 92] [P. Johnson-Laird, K. Oatley 89].
Figure 5-6: Reduction mapping of the OCC emotion types on a set of emotions, which can be expressed by speech synthesers.

However, we do not use Disgust as a basic emotion, because this will require a different approach in speech synthesis software. Table 5-2 gives an overview of the voice parameters. These parameters are identified as successful prosody rules for emotion expression in synthetic speech [M. Schröder 01].

<table>
<thead>
<tr>
<th>Emotion</th>
<th>Parameter settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Joy</td>
<td>F0 mean: +50% F0 range: +100% Tempo: +30% Voice Quality: modal or tense; &quot;lip-spreading feature&quot;: F1 / F2 +10% Other: &quot;wave pitch contour model&quot;: main stressed syllables are raised (+100%), syllables in between are lowered (-20%)</td>
</tr>
<tr>
<td>Sadness</td>
<td>F0 mean: &quot;0&quot;, reference line &quot;-1&quot;, less final lowering &quot;-5&quot; F0 range: &quot;-5&quot;, steeper accent shape &quot;+6&quot; Tempo: &quot;-10&quot;, more fluent pauses &quot;+5&quot;, hesitation pauses &quot;+10&quot; Loudness: &quot;-5&quot; Voice Quality: breathiness &quot;+10&quot;, brilliance &quot;-9&quot; Other: stress frequency &quot;+1&quot;, precision of articulation &quot;-5&quot;</td>
</tr>
<tr>
<td>Anger</td>
<td>F0 mean: +10 Hz F0 range: +9 s.t. Tempo: +30 wpm Loudness: +6 dB Voice Quality: laryngealisation +78%; frequency -175 Hz Other: increase pitch of stressed (2ary: +10% of pitch range; 1ary: emphatic: +40%)</td>
</tr>
<tr>
<td>Fear</td>
<td>F0 mean: &quot;+150%&quot; F0 range: &quot;+20%&quot; Tempo: &quot;+30%&quot; Voice Quality: falsetto</td>
</tr>
<tr>
<td>Surprise</td>
<td>F0 mean: &quot;0&quot;, reference line &quot;-8&quot; F0 range: &quot;+8&quot;, steeply rising contour slope &quot;+10&quot;, steeper accent shape &quot;+5&quot; Tempo: &quot;+4&quot;, less fluent pauses &quot;-5&quot;, hesitation pauses &quot;-10&quot; Loudness: &quot;+5&quot; Voice Quality: brilliance &quot;-3&quot;</td>
</tr>
<tr>
<td>Boredom</td>
<td>F0 mean: end frequency 65 Hz (male speech) F0 range: excursion size 4 s.t. Tempo: duration rel. to neutrality: 150% Other: final intonation pattern 3C, avoid final patterns 5&amp;A and 12</td>
</tr>
</tbody>
</table>

Table 5-2: Prosody rules for emotions in synthetic speech.
5.5.4.3 Personality and Emotion in Text

Within the text generation rules the statement of the studies made by Isbister and Nass [K. Isbister, C. Nass 00] finding cues of the extraversion trait in written text is realised. The extrovert utterances were relatively lengthy and used strong and descriptive language expressed in the form of confident assertions. Extravert characters tend to hang on control of the situation. Introvert utterances were relative short and used weaker language expressed in the form of suggestion. Introvert characters tends to dispense control.

For conveying politeness with textual cues, we try to adapt the work done by P. Brown and S. Levinson [P. Brown, S. Levinson 87] in the text generation rules.

We believe that the influence of emotions on phrases and word selection strongly depends on the extraversion and politeness traits. Emotions are used to comment events, actions, and objects, which are the cause for the current emotion according to the OCC model. These emotional comments will be inserted (by extra text generator rules) in the current generated utterance. As mentioned above the settings of the personality have an impact which emotion will appear as an emotional comment. Comments of negative emotions (like Anger, Sadness, …) did not appear, if setting the politeness trait to polite. Similar positive emotions are treated.

If setting the extraversion trait to extravert, emotional comments are relatively long and use strong descriptive language. If setting the trait to introvert, emotional comments are short (or even not used) and use weaker language expressed in the form of suggestions. Remembering the dialogue example shown in subsection Dialogue Phases above, the following utterances for the dialogue act dialoguecontrol_requestorder will be created:

- Affective parameters (hope, extravert, polite):
  Let us see, what I can do for you. I am sure I can help you.

- Affective parameters (hope, introvert, polite):
  What I can do for you. Hopefully I can help you.

- Affective parameters (hope, extravert, impolite):
  What do you want?

- Affective parameters (hope, introvert, polite):
  What do you need?

The ASM will be part of the Wine-Sales-Demonstrator James the Butler, to give the wine seller – an animated character – an affective speech output, which can enhance the believability of the character’s utterances.
5.6 References


[Festival 1.4.1] *Festival Speech Synthesiser*. A freely available speech synthesiser application, which provides a JSAPI interface, http://www.cstr.ed.ac.uk/projects/festival.html


[JSML] Java Speech Markup Language


6. Conclusions

The present document describes the components of the SAFIRA toolkit responsible for the affective expression: Affective Facial Expression, Affective Body Expression, Affective Rendering, and Affective Speech.

The Facial Expression component defines a Character Markup Language the provisions for the specification of a representation and scripting language, effectively aiming at bridging the gap between the underlying emotion engines and agent animation tools. The decision not to focus on, yet another, animation tool was governed by the fact that there is much research currently being attempted at achieving believable, real-time facial expression and animation, and that a more significant contribution would be achieved by specifying a language that can serve as a glue like mechanism to tie the various visual and underlying behaviour generation tools together seamlessly, regardless of the platform that they run on and the language they are developed with. The goal was achieved.

The Affective Body Expression component provides a way of generating expressive body movements using simple techniques that change the animations of a synthetic character in real-time. It combines neutral animations with emotional stances, and/or changes in the speed and in the spatial amplitude of the animations in order to create expressive bodily behaviours. The component provides a control interface that allows the manipulation of the synthetic characters, and also of the small graphics engine that is associated with the component. The characters produced by the component were used in FantasyA, a computer game in which the emotional display of the characters is essential for understanding and playing the game. The results were very positive, and the overall goals of the Affective Body Expression component were fulfilled.

The goal of the Affective Rendering component is to give software systems the ability to draw pictures in real-time which express emotion and personality, both in the content of the drawing and in the dynamics of the drawing process. It features a low-level stroke-drawing model that creates nonphotorealistically rendered pen strokes in real-time with humanlike dynamics. The input parameters of this model are provided by higher-level drawing behaviours, which map the input emotional and developmental states into control parameters for the stroke drawing. The component was used as part of the Influencing Machine demonstrator, and the objectives were all realised.

The Affective Speech component generates affective speech by varying the length and the intonation of words according to several affective and domain dependant modalities. The generation process depends on a representation on what to generate, the input structure, and on an affective state vector, which contains values for personality and emotions. This vector influences the format of utterance structures, like the content, syntactical word and phrase placement, and speech parameters. These utterance structures used in conjunction with a speech synthesiser provide a way of synthesising affective speech. The component was used in a wine selling scenario in a virtual shop.
on a web page, as part of the James the Butler demonstrator. The results were the expected, and the main goals were achieved.

As a final conclusion, the overall objective of WP5 that was to create reusable components capable of addressing affective expression using four different modalities was met. The components integrate the SAFIRA Toolkit, and they are part of the SAFIRA demonstrators.