SEMAINE

THE SENSITIVE AGENT PROJECT

D1b
First integrated system

2007 - 2013

Date: 23 December 2008

Dissemination level: Public

ICT project contract no.	211486
Project title	**SEMAINE** **Sustained Emotionally coloured Machine-human Interaction using Nonverbal Expression**
Contractual date of delivery	*31 December 2008*
Actual date of delivery	*23 December 2008*
Deliverable number	D1b
Deliverable title	First integrated system
Type	Demonstrator
Number of pages	36
WP contributing to the deliverable	WP 1
Responsible for task	Marc Schröder (schroed@dfki.de)
Author(s)	Marc Schröder (DFKI) Mark ter Maat (UT) Catherine Pelachaud, Elisabetta Bevacqua, Etienne de Sevin (Paris8) Björn Schuller, Florian Eyben, Martin Wöllmer (TUM)
EC Project Officer	Philippe Gelin

Table of Contents

1 Executive Summary

The present report describes the first public system demonstrator released by the SEMAINE project after the first year of work in the project. The aim of SEMAINE is to build a Sensitive Artificial Listener (SAL) – a multimodal dialogue system with the social interaction skills needed for a sustained conversation with a human user.

As a snapshot of "work in progress", the demonstrator aims to illustrate the following:

- the SEMAINE project has proposed a system architecture for realising multimodal analysis and generation in a user-and-ECA dialogue situation;
- a SEMAINE API has been created to allow processing components to communicate with each others according to the architecture, using appropriate representation formats at interfaces between components;
- an initial set of components exist to partially realise the core functions of a SAL system.

The system is not yet intended to function as a fully operational, real-time SAL system; its main aim is to give an "early view" to technologically interested experts, indicating the direction in which the work is going. Individuals and research teams interested in cooperation around the SEMAINE platform are explicitly invited to contact the project team.

Over the next year, the system will be developed into a fully operational SAL system, including the following elements:

- the user analysis components, which are not yet reliable because they are based on very preliminary training data, will be substantially improved by the availability of suitable training data;
- visual analysis code (facial expression, gaze, head actions) will be added;
- the generation components will use specific facial models and synthetic voices for the SAL characters Poppy, Spike, Obadiah and Prudence;
- the architecture will be closer to real-time behaviour.

2 System architecture: Components and representation formats

A conceptual system architecture is the first step towards a running system, and clarity on this level makes a well-structured implementation possible. Therefore, some care was invested to identify a conceptual system architecture that lends itself to implementation and that has the properties required by the application. The resulting conceptual system architecture is depicted in Figure 1.

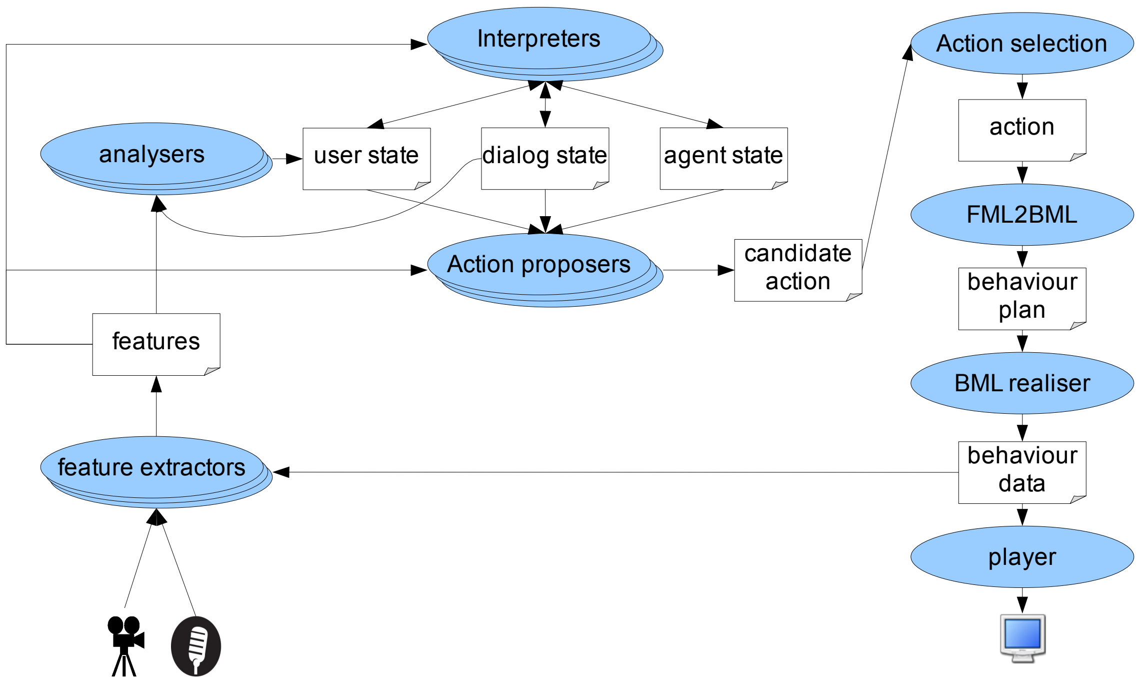


Figure 1: Architecture of the initial SEMAINE system

2.1 Overview of the conceptual architecture

Processing components are represented as ovals, data representations as rectangles. Arrows are always between components and data, and indicate which data is produced by or is accessible to which component.

It can be seen that the rough organisation follows the simple tripartition of input (left), central processing (middle), and output (right), and that arrows indicate a rough pipeline for the data flow, from input analysis via central processing to output generation. Note that this is a deliberate simplification at this stage, and is identified as a point for future improvement (Schröder et al., 2008).

The **main aspects of the architecture** are outlined as follows. Feature extractors analyse the low-level audio and video signals, and provide feature vectors periodically to the following components. A collection of analysers, such as monomodal or multimodal classifiers, produce a context-free, short-term interpretation of the current user state, in terms of behaviour (e.g., a smile) or of epistemic-affective states (emotion, interest, etc.). These analysers usually have no access to centrally held information about the state of the user, the agent, and the dialog; only the speech recognition needs to know about the dialog state, whether the user or the agent is currently speaking.

A set of interpreter components evaluate the short-term analyses of user state in the context of the current state of information regarding the user, the dialog, and the agent itself, and update these information states.

A range of action proposers produce candidate actions, independently from one another. An utterance producer will propose the agent's next verbal utterance, given the dialog history, the user's emotion, the topic under discussion, and the agent's own emotion. An automatic backchannel generator identifies suitable points in time to emit a backchannel. A mimicry component will propose to imitate, to some extent, the user's low-level behaviour. Finally, a non-verbal behaviour component needs to generate some "background" behaviour continuously, especially when the agent is listening but also when it is speaking.

The actions proposed may be contradictory, and thus must be filtered by an action selection component. A selected action is converted from a description in terms of its functions into a behaviour plan, which is then realised in terms of low-level data that can be used directly by a player.

Similar to an *efferent copy* in human motor prediction (Wolpert & Flanagan, 2001), behaviour data is also available to feature extractors as a prediction of expected perception. For example, this can be used to filter out the agent's speech from the microphone signal.

2.2 Representation formats

This conceptual architecture is rather independent of the concrete implementation components used to implement certain conceptual components in this architecture. For example, the architecture remains basically the same whether there are feature extractors for multiple modalities or just for individual modalities; action proposers may or may not include a backchannel component or a mimicry component; output may be unimodal or multimodal; analysers can recognise individual epistemic-affective states (interest, arousal, etc.) or a collection of them, from single or multiple modalities.

For all implementations, however, the **representation formats** used at the component interfaces should be the same, to allow for the type of modular design that the conceptual architecture assumes. SEMAINE has decided to follow standard representation formats where possible. The following sections characterise the data representations used at the various points in the architecture. Each section describes

- the meaning of the given type of data in the context of the architecture;
- the JMS Topics associated with these data types in the SEMAINE API (see Section 3);
- the representation format used for that data, pointing to the specifications or draft specifications where that is possible.

2.2.1 Features

Meaning	All audio and video low-level features, produced by various components in each case. All features are produced periodically, but periods may vary from one component to the other.
JMS Topics	`semaine.data.analysis.>`, including `semaine.data.analysis.audio.>` and `semaine.data.analysis.video.>`
Representation format	Feature vector in text form or in binary form. All features are float-valued. In text form, the feature vector consists of one line per feature; the line first shows

	the float value of the feature, a space character, and the feature name. In binary representation, the feature vector uses a byte array, using four bytes per feature, preceded by four bytes representing an integer which contains the number of features in the feature vector. Binary feature messages do not contain feature names. Therefore, when the system becomes ready, components sending feature vectors should send the first feature vector in text format, so that receivers can know the feature names.

2.2.2 User data: signals

Meaning	A range of descriptions of user state, including lower-level, short-term classifier output and re-interpreted abstractions over longer periods (words, sentences).
JMS Topics	`semaine.data.state.user.emma`
Representation format	Container format: EMMA (http://www.w3.org/TR/emma/). Message payload, describing the user state as determined by analysers and interpreters: ● for epistemic-affective states: EmotionML (http://www.w3.org/2005/Incubator/emotion/XGR-emotionml/); ● for behaviour: SemaineML (see Appendix I); ● for speech recognition results: SemaineML or EMMA.

2.2.3 User data: behaviours

Meaning	The current-best-guess values for behaviour (smiling, speaking, …). These are interpretations of the signals.
JMS Topics	`semaine.data.state.user.behaviour`
Representation format	SemaineML (see Appendix I)

2.2.4 User data: intentions

Meaning	The current-best-guess values for the user's intentions and mental state (intention to keep the turn, interest level, emotion, ...). These are interpretations of the signals.
JMS Topics	`semaine.data.state.user.intention`
Representation format	SemaineML (see Appendix I)

2.2.5 Dialog state

Meaning	Current best guess values for various dialog states.
JMS Topics	`semaine.data.state.dialog.>`
Representation format	SemaineML (see Appendix I)

2.2.6 Agent intentions

Meaning	Current best guess values for various agent state variables including intentions and mental state (intention to grab the turn, attitude towards a topic, the agent's emotion, interest level, …)
JMS Topics	`semaine.data.state.agent.intention`
Representation format	SemaineML (see Appendix I)

2.2.7 Agent behaviours

Meaning	Feedback about actions that have been carried out. To be filled in by action proposers after they receive confirmation that an action has been carried out.
JMS Topics	`semaine.data.state.agent.behaviour`
Representation format	SemaineML (see Appendix I)

2.2.8 Candidate action

Meaning	Proposals for action.
JMS Topics	`semaine.data.action.candidate.>`, including `semaine.data.action.candidate.function` and `semaine.data.action.candidate.behaviour`
Representation format	FML (see Appendix II) or BML (see Appendix III)

2.2.9 Action

Meaning	Selected action to be generated.
JMS Topics	`semaine.data.action.selected.>`, including `semaine.data.action.selected.function` and `semaine.data.action.selected.behaviour`
Representation format	FML (see Appendix II) or BML (see Appendix III)

2.2.10 Behaviour plan

Meaning	An atomic chunk of behaviour markup for generation.
JMS Topics	`semaine.data.synthesis.plan.>`
Representation format	BML (see Appendix III)

2.2.11 Behaviour data

Meaning	Low-level data ready to be played by the player.

JMS Topics	`semaine.data.synthesis.lowlevel.>`, including `semaine.data.synthesis.lowlevel.audio.>` and `semaine.data.synthesis.lowlevel.video.>`
Representation format	audio data and FAP+BAP parameters

3 The SEMAINE API

The SEMAINE API is the communication infrastructure for the components in the SEMAINE system. It provides the services of a message-oriented middleware (Banavar, Chandra, Strom, & Sturman, 1999), allowing communication across various programming languages and operating systems, in terms of the data representations needed in SEMAINE.

For the realisation of low-level message routing, the Java Message Service (JMS) implementation ActiveMQ (http://activemq.apache.org/) was chosen. We compared it in terms of performance with Psyclone (http://www.mindmakers.org/projects/Psyclone), and it appeared that ActiveMQ was at least one order of magnitude faster than Psyclone. In a number of simple tests, passing a simple message from one process to another took about 0.4 ms with ActiveMQ, and 15 ms with Psyclone (averaged over 1000 iterations with slightly different message content each time) – a factor of 40 in this test.[1]

ActiveMQ has the practical benefits of being an open source project actively developed at the Apache Foundation, and of providing client code in various programming languages, including Java and C++, the languages used by SEMAINE partners. ActiveMQ is available on many platforms, including Linux, Mac OS X and Windows.

ActiveMQ provides, among other things, a **publish-subscribe** communication model. The entities that client code publishes to, or subscribes from, are called **Topics**, and are identified by names. Hierarchies of topics are defined by the use of dots; subscription can use wildcards to subscribe to entire topic families. For example, a subscription to the Topic family `semaine.data.action.selected.>` will yield messages published to, e.g., `semaine.data.action.selected.function` and `semaine.data.action.selected.behaviour`.

The SEMAINE API provides abstractions from the low-level ActiveMQ handling code, in order to facilitate the communication of data in relevant representation formats between processing components. There are two versions of the SEMAINE API:

- a Java version, based on Java 6;
- a C++ version, providing build files for Microsoft Visual Studio 2005 on Windows and build for GNU automake/autoconf on Linux and Mac OS X.

The C++ version is very similar to the Java version, using C++ namespaces to reflect Java packages. This approach promotes the aim of having one consistent development environment across programming languages and operating systems. The following sections will illustrate the key properties of the SEMAINE API, using the Java syntax.

3.1 Message routing: Receiver, Sender and their subclasses

Receivers and Senders provide the SEMAINE view on subscription and publishing, respectively. A Receiver can subscribe to a certain Topic family and start receiving messages as in the following example using the simple synchronous model of message reception:

1 Further investigations, on the other hand, have provided evidence that ActiveMQ is too slow for sending large amounts of binary video data. Therefore, ActiveMQ seems appropriate for message routing in the SEMAINE architecture, but not appropriate for communicating raw input data between components. This seems to be a marginal requirement for the SEMAINE architecture – it would only concern situations where multiple feature extractors should be applied to the same input data. In those cases, alternative methods for message passing need to be used.

```
import eu.semaine.jms.receiver;
import eu.semaine.jms.message.SEMAINEMessage;
...
Receiver r = new Receiver("semaine.data.action.selected.>");
r.startConnection();
while (true) {
    SEMAINEMessage m = r.receive();
    ... // do something meaningful with the message
}
```

An alternative and more flexible way of using a Receiver is to register a SEMAINEMessageListener, which will be called in a separate thread, every time that a message is received. For example:

```
import eu.semaine.jms.receiver.Receiver;
import eu.semaine.jms.message.SEMAINEMessage;
...
Receiver r = new Receiver("semaine.data.action.selected.>");
r.startConnection();
SEMAINEMessageAvailableListener listener =
  new SEMAINEMessageAvailableListener() {
    public void messageAvailableFrom(Receiver rec) {
        SEMAINEMessage m = rec.getMessage();
        ... // do something meaningful with the message
    }
};
r.setMessageListener(listener);
// do something else in this thread
```

The default method for sending a message using a Sender is equally simple.

```
import eu.semaine.jms.sender.Sender;
...
Sender s = new Sender("semaine.data.action.selected.behaviour", "BML", "Demo
sender");
s.startConnection();
long time = System.currentTimeMillis();
String text = "<?xml version=\"1.0\" encoding=\"UTF-8\"?>" +
    "<bml xmlns=\"http://www.mindmakers.org/projects/BML\" version=\"1.0\">" +
    "<head type=\"NOD\"/>" +
    "</bml>";
s.sendTextMessage(text, time);
```

In principle, this functionality would be sufficient for enabling components to communicate. To simplify communication and to support consistent representations, however, the SEMAINE API provides a number of convenience classes representing the kinds of data that are required in the SEMAINE architecture.

In the package eu.semaine.jms.receiver, there are a number of subclasses of Receiver, producing various subclasses of SEMAINEMessage from the package eu.semaine.jms.message:

- a BytesReceiver, for binary data, produces SEMAINEBytesMessage objects;
- a FeatureReceiver, for feature vectors, produces SEMAINEFeatureMessage objects;
- an XMLReceiver, as a base class for all XML representations, produces SEMAINEXMLMessage as a base class for specific XML-based messages:
    - EmmaReceiver produces SEMAINEEmmaMessage,
    - BMLReceiver produces SEMAINEBMLMessage, etc.

User code can use these receivers to work with the data on a higher degree of abstraction. For example, receiving feature vectors using a FeatureReceiver:

```
import eu.semaine.jms.receiver.FeatureReceiver;
import eu.semaine.jms.message.SEMAINEFeatureMessage;
...
FeatureReceiver r = new FeatureReceiver("semaine.data.action.selected.>");
r.startConnection();
while (true) {
    SEMAINEFeatureMessage m = (SEMAINEFeatureMessage) r.receive();
    String[] featureNames = m.getFeatureNames();
    float[] features = m.getFeatureVector();
    ...
}
```

Similarly, subclasses of Sender can be used for sending data in different relevant formats:

- a BytesSender can be used to send binary data;
- a FeatureSender can be used to send feature vectors;
- an XMLSender is a base class for sending XML data; subclasses include
  - EmmaSender,
  - BMLSender, etc.

User code can use these senders directly, e.g. to send XML data directly from a W3C DOM representation:

```
import eu.semaine.jms.sender.BMLSender;
import org.w3c.dom.Document;
...
BMLSender s = new BMLSender("semaine.data.action.selected.behaviour", "Demo
sender");
s.startConnection();
long time = System.currentTimeMillis();
Document bmlDocument = ...; // some way of creating a BML document
s.sendXML(bmlDocument, time);
```

3.2 Component

A SEMAINE Component is a processing entity which performs a certain processing step. It corresponds to one of the ovals in Figure 1. In general, it can **receive** data from zero, one or several Topics, and it can **send** data to zero, one or several Topics. It can either **act** spontaneously due to some internal logic, or it can **react** to incoming messages.

The class `eu.semaine.components.Component` provides a base implementation for all SEMAINE components. It is designed to run as its own thread. It has a list of Receiver and Sender objects which are initialised in the constructor; for each of the Receivers, the component registers as a SEMAINEMessageAvailableListener, so that the component is notified whenever a message is available in any of its receivers.

In its main loop, implemented in `Component.run()`, the component does the following steps repeatedly:

- it calls the method `act()`;

- it checks for any incoming messages on any of its receivers, waiting for a maximum amount of time `waitingTime` (this amount can be configured as a number of milliseconds);
- if any messages arrive, it stops waiting and calls `react(SEMAINEMessage)` immediately.

As a result, the main loop calls `act()` at least every `waitingTime` milliseconds, and calls `react(SEMAINEMessage)` every time a message is received.

Subclasses of Component need to register the Receivers and Senders they need, and must implement `act()`, `react()` or both in order to do something meaningful. The following simple component reacts to every BML message by passing it on, thus implementing a trivial action selector for BML actions[2]:

```
public class PassOnBML extends Component
{
    BMLSender bmlSender;
    public PassOnBML()
    {
        super("PassOnBML");
        BMLReceiver bmlReceiver =
            new BMLReceiver("semaine.data.action.candidate.behaviour");
        receivers.add(bmlReceiver);
        bmlSender = new BMLSender("semaine.data.action.selected.behaviour",
            getName());
        senders.add(bmlSender);
    }

    protected void react(SEMAINEMessage m)
    {
        Document bml = ((SEMAINEXMLMessage) m).getDocument();
        bmlSender.sendXML(bml, m.getUserTime());
    }
}
```

The following component sends a message when a certain idle time has passed without any new messages.

```
public class IdleActionProposer extends Component
{
    BMLSender bmlSender;
    long lastMessageTime;
    long maxIdleTime = 4000; // four seconds

    public IdleActionProposer()
    {
        super("IdleActionProposer");
        BMLReceiver bmlReceiver =
            new BMLReceiver("semaine.data.action.candidate.behaviour");
        receivers.add(bmlReceiver);
        bmlSender = new BMLSender("semaine.data.action.candidate.behaviour",
            getName());
        senders.add(bmlSender);
    }

    protected void react(SEMAINEMessage m)
    {
        lastMessageTime = m.getUserTime();
    }

    protected void act()
```

2 The example code leaves out import statements and exception handling so as to maintain clarity.

```
    {
        long currentTime = meta.getTime();
        if (currentTime - lastMessageTime > maxIdleTime) {
            Document someBMLMessage = ...; // create some message
            bmlSender.sendXML(someBMLMessage, currentTime);
            lastMessageTime = currentTime;
        }
    }
}
```

3.3 ComponentRunner

The ComponentRunner is a convenience class for running several components in a single executable. In java, the list of components can be given in a config file.

```
ComponentRunner runner = new ComponentRunner("my-system.config");
runner.go();
runner.waitUntilCompleted(); // add this if you want to block
```

The file "my-system.config" should have content similar to the following:

```
# SEMAINE component runner config file
semaine.components = \
    |eu.semaine.components.meta.SystemManager| \
    |eu.semaine.components.dummy.DummyFeatureExtractor| \
    |eu.semaine.components.dummy.DummyAnalyser| \
    |eu.semaine.components.dummy.DummyInterpreter| \
    |eu.semaine.components.dummy.DummyFMLActionProposer| \
    |eu.semaine.components.dummy.DummyBMLActionProposer| \
    |eu.semaine.components.dummy.DummyActionSelection| \
    |eu.semaine.components.dummy.DummyFML2BML| \
    |eu.semaine.components.dummy.DummyBMLRealiserAndPlayer| \
    |eu.semaine.components.MessageLogComponent($semaine.messagelog.topic,
$semaine.messagelog.messageselector)|

semaine.messagelog.topic = semaine.data.>
semaine.messagelog.messageselector = event IS NOT NULL
```

In this file, components are listed as the values of the property `semaine.components`, and separated by "|" characters. Each component must be a subclass of `eu.semaine.components.Component`. A component's constructor can have zero, one or more String-valued arguments, which can be given in brackets in the config file. If the value starts with a "$" sign, then the value of the given property from the same config file is used; else, the value given is interpreted as a literal string value. In the example given, all components have a zero-argument constructor except for `MessageLogComponent`, which has two string-valued arguments; here, both are read from property entries. The constructor is effectively called as `new MessageLogComponent("semaine.data.>", "event IS NOT NULL")`.

In C++, which does not easily support reflection (i.e., instantiation of a class by its name), the components need to be given explicitly in the code.

```
std::list<semaine::components::Component *> comps;
comps.push_back(new semaine::components::dummy::DummyFeatureExtractor());
// Add any additional components here
semaine::system::ComponentRunner cr(comps);
cr.go();
cr.waitUntilCompleted(); // add this if you want to block
```

3.4 Meta messages and the System Manager

The SEMAINE architecture could function as a totally distributed system, in which a component does not need to know anything about other components – not even about their existence. The flaws of such a fully emergent system become apparent at the moment when something goes wrong: maybe a component did not start up properly, or failed due to some unexpected circumstances. In a fully distributed system, such a situation may not easily be noticed.

In order to keep track of **the state of the overall system** and of individual components, a specialised component exists: the System Manager (`eu.semaine.components.meta.SystemManager`). Each component has a meta messenger (`eu.semaine.components.meta.MetaMessenger`) communicating with the system manager. The meta messenger informs the system manager of any state changes in the component (`starting`, `ready`, `stopped`, `failure`, and `stalled`); the system manager informs the meta messenger of the state of the system as a whole (in the simplest form, a summary whether all components in the system are ready or not).

A component then has the choice how to react to the information that the system is not fully operational. In the simplest form, components can pause processing until all components are ready.

In a SEMAINE system, there must be exactly one SystemManager component. For this reason, there is only a Java implementation of SystemManager, not a C++ version. The MetaMessenger, on the other hand, obviously needs to exist for every component, so there is a Java and a C++ version like for most parts of the SEMAINE API code.

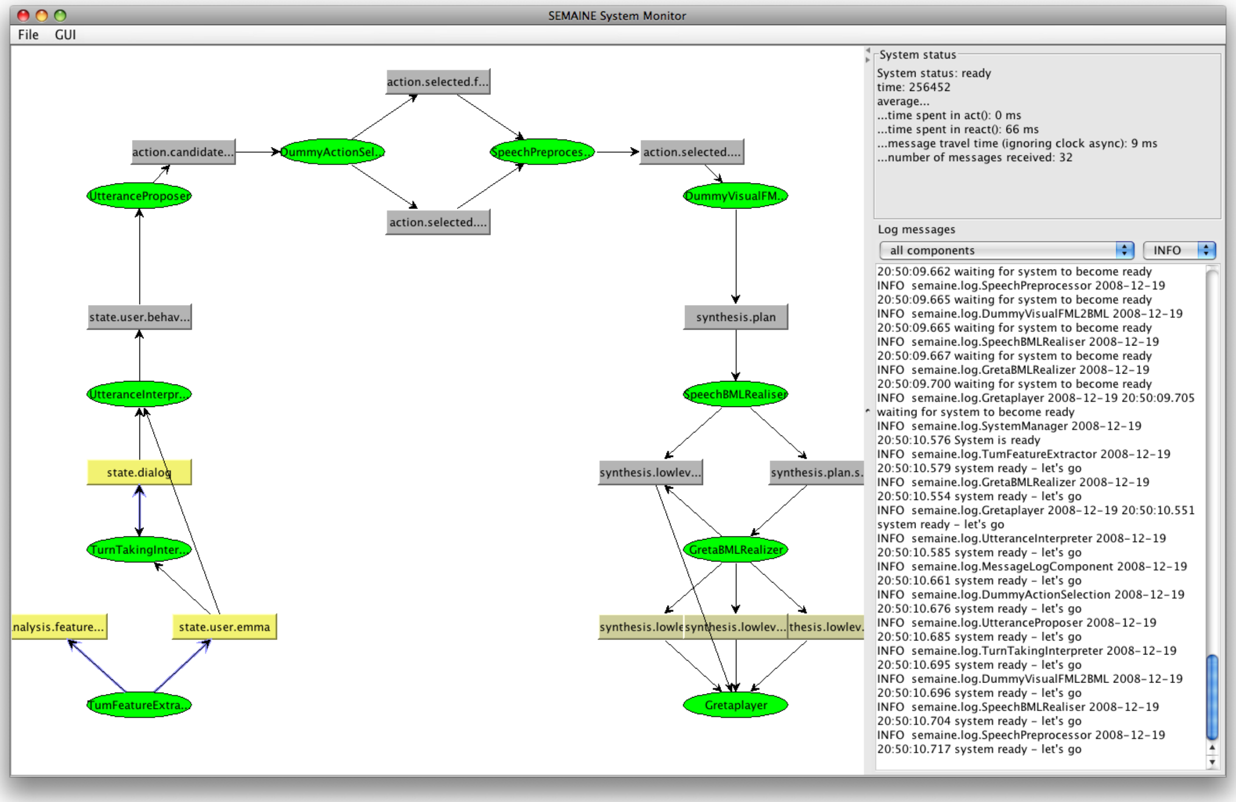
A second function of meta messages is the **synchronisation of time** across different machines whose system clocks may be set differently. The system manager informs all components of a common system time, through their respective meta messengers. Components should therefore access time exclusively by calling

```
long timeInMilliseconds = meta.getTime();
```

Communication between the System Manager and the meta messengers is happening via the Topic family `semaine.meta.>`.

Optionally, a Graphical User Interface (GUI) to the System Manager, the SEMAINE System Monitor, can be used to display information about the system. This includes the components active in the system as well as the Topics they read from and write to. The GUI also contains a summary report of the overall system state, and a configurable view of the centralised logging facility (see next section). When the user clicks on a Topic, a new window opens which displays the messages sent to that Topic. When the user clicks on a component, a window shows the detailed status of that component. Color is used to show active Topics as well as the state of individual components. The layout of components and Topics is created on the fly, using the Topics that components send to or receive from in order to sort components into an approximately meaningful order, starting with input components in the lower left and ending with output components in the lower right. Figure 2 shows an example screenshot of the GUI.

In summary, the GUI is designed to provide the developer with a live overview of the system, and help identify and diagnose problems or unexpected situations comfortably.



```
System status: ready
time: 256452
average...
...time spent in act(): 0 ms
...time spent in react(): 66 ms
...message travel time (ignoring clock async): 9 ms
...number of messages received: 32
```

```
20:50:09.662 waiting for system to become ready
INFO  semaine.log.SpeechPreprocessor 2008-12-19
20:50:09.665 waiting for system to become ready
INFO  semaine.log.DummyVisualFML2BML 2008-12-19
20:50:09.665 waiting for system to become ready
INFO  semaine.log.SpeechBMLRealiser 2008-12-19
20:50:09.667 waiting for system to become ready
INFO  semaine.log.GretaBMLRealizer 2008-12-19
20:50:09.700 waiting for system to become ready
INFO  semaine.log.Gretaplayer 2008-12-19 20:50:09.705
waiting for system to become ready
INFO  semaine.log.SystemManager 2008-12-19
20:50:10.576 System is ready
INFO  semaine.log.TumFeatureExtractor 2008-12-19
20:50:10.579 system ready - let's go
INFO  semaine.log.GretaBMLRealizer 2008-12-19
20:50:10.554 system ready - let's go
INFO  semaine.log.Gretaplayer 2008-12-19 20:50:10.551
system ready - let's go
INFO  semaine.log.UtteranceInterpreter 2008-12-19
20:50:10.585 system ready - let's go
INFO  semaine.log.MessageLogComponent 2008-12-19
20:50:10.661 system ready - let's go
INFO  semaine.log.DummyActionSelection 2008-12-19
20:50:10.676 system ready - let's go
INFO  semaine.log.UtteranceProposer 2008-12-19
20:50:10.685 system ready - let's go
INFO  semaine.log.TurnTakingInterpreter 2008-12-19
20:50:10.695 system ready - let's go
INFO  semaine.log.DummyVisualFML2BML 2008-12-19
20:50:10.696 system ready - let's go
INFO  semaine.log.SpeechBMLRealiser 2008-12-19
20:50:10.704 system ready - let's go
INFO  semaine.log.SpeechPreprocessor 2008-12-19
20:50:10.717 system ready - let's go
```

Figure 2: Screenshot of the SEMAINE System Monitor GUI for the System Manager

3.5 Centralised logging

In a distributed system, it is not trivial to keep track of things happening. A centralised log functionality alleviates the problem by creating a single central place where all components can write their log messages, no matter what programming language and what operating system they use.

This functionality is provided by the JMSLogger in Java and the CMSLogger in C++. Effectively, they are sending ActiveMQ messages to topics below `semaine.log.>` according to the scheme `semaine.log.Source.LogLevel`, where `Source` is a meaningful identifier (e.g., the name of the component logging) and `LogLevel` reflects the severity of the message (one of `error`, `warn`, `info` and `debug`). For example, the following will write a message "Hello world" to the Topic `semaine.log.MyDummy.info`:

```
JMSLogger log = JMSLogger.getLogger("MyDummy");
log.info("Hello world");
```

A simple text-based log reader can be used to display all or certain log messages: `eu.semaine.jms.JMSLogReader`.

3.6 XML handling

Many of the messages in the SEMAINE architecture are using XML as a representation format, combining and mixing various XML namespaces. As namespace-aware XML processing can sometimes be difficult to get right, the SEMAINE API provides a powerful utility class to do XML processing in the Document Object Model (DOM) framework: `eu.semaine.util.XMLTool`. The Xerces library from the Apache foundation is used because it adheres to W3C standards and it exists in a Java and a C++ version (http://xerces.apache.org/xerces-j/ and http://xerces.apache.org/xerces-c/, respectively).

For example, the following code creates an EMMA document containing a SemaineML `<behaviour>` tag:

```
Document document = XMLTool.newDocument(EMMA.E_EMMA,
      EMMA.namespaceURI, EMMA.version);
Element interpretation = XMLTool.appendChildElement(
      document.getDocumentElement(), EMMA.E_INTERPRETATION);
Element behaviour = XMLTool.appendChildElement(interpretation,
      SemaineML.E_BEHAVIOUR, SemaineML.namespaceURI);
interpretation.setAttribute(EMMA.A_START, String.valueOf(meta.getTime()));
behaviour.setAttribute(SemaineML.A_NAME, "gaze-away");
behaviour.setAttribute(SemaineML.A_INTENSITY, "0.9");
```

Again, the C++ code looks essentially the same – only, in C++, the XML code needs to be initialised and finalised properly.

```
XMLTool::startupXMLTools();
...
DOMDocument * document = XMLTool::newDocument(EMMA::E_EMMA,
      EMMA::namespaceURI, EMMA::version);
DOMElement * interpretation = XMLTool::appendChildElement(
      document->getDocumentElement(), EMMA::E_INTERPRETATION);
DOMElement * behaviour = XMLTool::appendChildElement(interpretation,
      SemaineML::E_BEHAVIOUR, SemaineML::namespaceURI);
std::stringstream buf;
buf << fm->getUsertime();
std::string usertimeString = buf.str();
XMLTool::setAttribute(interpretation, EMMA::A_START, usertimeString);
XMLTool::setAttribute(behaviour, SemaineML::A_NAME, "gaze-away");
XMLTool::setAttribute(behaviour, SemaineML::A_INTENSITY, "0.9");
...
XMLTool::shutdownXMLTools();
```

The resulting XML document is the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<emma xmlns="http://www.w3.org/2003/04/emma" version="1.0">
  <interpretation start="75838">
    <behaviour xmlns="http://www.semaine-project.eu/semaineml" intensity="0.9"
name="gaze-away"/>
  </interpretation>
</emma>
```

4 The first integrated system demonstrator

The first integrated SEMAINE system uses the SEMAINE API across operating systems and programming languages to implement the SEMAINE architecture, essentially based on existing components which were adapted to run in the SEMAINE framework. The following table gives a short overview of the components involved.

Component	Architectural role	Team	OS	Language
low-level audio features	feature extractor	TUM	Linux/Mac	C++
emotion detection	analyser	TUM	Linux/Mac	C++
ASR	analyser	TUM	Linux	C++
interest detection	analyser	TUM	Linux/Mac	C++
turn taking	interpreter	UT	any	Java
user utterance interpreter	interpreter	UT	any	Java
agent utterance proposer	action proposer	UT	any	Java
backchannel/mimicry	action proposer	Paris	Windows	C++[3]
	action selection	Paris	Windows	C++
speech preprocessing	(part of FML2BML)	DFKI	any	Java
	FML2BML	Paris	Windows	C++
speech synthesis	(part of BML realiser)	DFKI	any	Java
	BML realiser	Paris	Windows	C++
Greta	player	Paris	Windows	C++

4.1 Individual system components

The following sections briefly describe the individual system components.

4.1.1 Low-level audio features

The openSMILE (Speech and Music Interpretation by Large-Space Extraction) feature extractor was written in ANSI C. The SEMAINE architecture API provides a C++ interface. Thus, C++ wrapper classes were written for openSMILE's ANSI C components. Both, a standalone version of openSMILE and a SEMAINE component version of SMILE were provided. The SMILE feature extractor has built-in audio capture functionality relying on the PortAudio library. SMILE's architecture is highly modular while maintaining a high level of efficiency. New features and new signal processing function can be easily integrated. All audio signal processing and feature extraction is incremental, i.e. it is done frame by frame.

SMILE is capable of extracting low-level audio features (i.e. features derived directly from the signal) from overlapping frames of arbitrary size and overlap. The 38 low-level features that were implemented during the first project year are listed in Appendix IV. Basing on the low-level feature contours, delta coefficients of arbitrary order and statistical functionals in arbitrary hierarchies can be computed using SMILE (see Appendix V). In total 5,700 turn based features can be computed,

3 Components that are displayed in grey (backchannel/mimicry action proposer, action selection, and FML2BML) are ready but have not yet been integrated into the first demonstrator.

when using the standard configuration where delta and acceleration coefficients are appended to the low-level features. There is, however, no upper limit regarding the maximum number of features that can be generated, except due to computing resource limitations. Adding higher order regression coefficients and multiple hierarchies of statistical functionals can push the number of generated features beyond *1 million features*.

SMILE supports real-time, on-line stream processing, where audio data is read from a stream source, either a pipe or network stream or audio capture device. Features are output as soon as they become available. Direct interfacing with classifiers for real-time on-line classification of generated features is possible. Further, off-line chunk processing is supported, whereas features can be output in various widely used formats such as WEKA ARFF files, CSV files, LibSVM format files, and HTK feature files. SMILE has integrated voice activity detection based on energy and pitch features. This allows robust distinction of background noise and non-speech sounds from actual speech uttered by the user.

4.1.2 Emotion detection

The SMILE automatic emotion recognition (AER) component is a consequent extension of existing knowledge in the field of emotion recognition. After the extraction of large and open sets of hierarchical audio features, a feature selection is performed in order to find the most relevant features for the task of interest. Finally, a classifier (e.g. Support-Vector Machines, Multi-Layer Perceptron, Naive Bayes, K Nearest-Neighbour, etc.) or a regression method, e.g. Support-Vector Regression (SVR), is applied on the reduced feature vector.

Based on the findings in Wöllmer (2008), models for on-line emotion recognition were trained. Even though Long Short-Term Memory Recurrent Neural Nets performed better than SVR, for the on-line recognition application SVR are used due to easier integration in an on-line recognition framework. Using the openSMILE feature extraction, 5,700 features were extracted for each turn in the SAL recordings. The turns thereby were segmented by energy thresholds and the maximum turn length was limited to 10 seconds. A correlation-based feature sub-set selection was performed to find an optimal and reduced set of features for valence and activation separately. 32 features remain as relevant for activation, and 212 features for valence. Building on those reduced feature sets and the Feeltrace annotations of the SAL recordings, Support-Vector Regression models were trained, one model for each emotion dimension.

4.1.3 ASR

The Automatic Speech Recognition component for the Milestone 2 system is based on the ATK (http://htk.eng.cam.ac.uk/develop/atk.shtml) real-time speech recognition engine which is a C++ layer sitting on top of HTK (http://htk.eng.cam.ac.uk/). During the development of the ASR component, the engine was designed and parametrised as an optimal trade-off between accuracy and decoding time. The Milestone 2 ASR component processes cepstral mean normalised Mel-Frequency Cepstral Coefficients 0-12 as well as their first and second order temporal derivatives at a frame rate of 10 ms. It uses word-internal tri-phones for acoustic modelling and tri-gram language modelling. Along with the recognised string, the mean word-confidence is output.

The speaker independent tri-phone tied-state Hidden Markov Models were trained on the Wall Street Journal corpus and on the AMIDA Meeting corpus, since the preliminary SAL corpus contains only four different speakers which is not enough to train speaker independent acoustic models. Phoneme models consist of three emitting states with eight Gaussian mixtures per state. The tri-gram language model was trained on the preliminary SAL corpus and therefore included about 1900 words which typically occur in spontaneous emotionally coloured speech. Further, language models based on both, the SAL corpus and other transcriptions of spontaneous speech such as the AMIDA Meeting corpus or scripts of various TV-Series, were trained.

4.1.4 Interest detection

The module for recognition of human interest was trained on the AVIC (TUM) database. This database contains data of a real-world scenario where an experimenter leads a subject through a commercial presentation. The subject interacted with the experimenter and thereby naturally and spontaneously expressed different levels of interest.

The openSMILE feature extraction was used to extract 5,700 features serving as a basis for subsequent correlation-based feature sub-set selection. A total of 128 features remained after feature selection and were used to train Support Vector Machines. The module discriminates three different levels of interest (LOI): “bored” (LOI 0), “neutral” (LOI 1), and “high interest” (LOI 2).

4.1.5 Turn taking

Turn taking is a very complex conversational system in which the participants negotiate who will be the next main speaker in the next period. But, being too complex for the first demonstrator system a simplified version was created. The demonstrator needs a simple turn taking mechanism in order to detect that the human speaker finished speaking, so that it can give a response.

Currently it waits for user_speaking and user_silent messages from low-level audio features. When the user is silent for more than 2 seconds the system decides that SAL has the turn. It then sends this turn-information to the Dialog state.

4.1.6 User utterance interpreter

When the turn taking module decides that the agent has the turn the system will analyse what the user did and said. Eventually this will happen continuously to speed up the response time, but currently this starts when the agent takes the turn. When this happens, the User utterance interpreter will look at the utterances of the user that were detected in the previous turn. The utterances are then tagged with general semantic features such as the semantic polarity (if the user is positive or negative about something), the time (talking about the future or the past), and the subject of the utterances (if it's the user himself, the current character he's speaking to, or other people). This extended sentence is then send to the user.behaviour channel. If no sentence was detected it will send an empty string.

4.1.7 Agent utterance proposer

The function of the agent utterance proposer is to select an appropriate response when the agent has to say something. It starts working when it receives an extended user utterance from the user utterance interpreter, because in the current system this also means that the agent has the turn. Using the added features it searches its response model for responses that fit the current context. This response model is based on the transcripts of the WOz-data and contains fitting contexts (i.e. a list of semantic features) for every possible response. When an empty string is detected it will pick a response from a list of generic responses which fit it almost all circumstances. Finally, the chosen response is send as a proposed action to the action selection module and a message is send to the turn taking module to set the turn to the user again.

4.1.8 Backchannel/Mimicry action proposer

The backchannel action proposer is in charge of the computation of the agent's behaviours while being a listener when conversing with a user. This component encompasses three modules called reactive backchannel, cognitive backchannel, and mimicry. Research has shown that there is a strong correlation between backchannel signals and the verbal and nonverbal behaviours performed by the speaker (Maatman, Gratch & Marsella, 2005) (Ward, & Tsukahara, 2000). Models have been elaborated that predict when a backchannel signal can be triggered based on a statistical analysis of the speaker's behaviours (Maatman, Gratch & Marsella, 2005) (Morency, L.-P, de Kok, & Gratch, 2008) (Ward, & Tsukahara, 2000). We use a similar approach and have fixed some probabilistic rules to prompt a backchannel signal when our system recognizes certain speaker's behaviours; for example, a head nod or a variation in the pitch of the user's voice will trigger a backchannel with a certain probability. Probabilities are set based on studies from the literature (Maatman, Gratch & Marsella, 2005) (Ward, & Tsukahara, 2000). The reactive backchannel module takes care of this predictive model. On the other hand, the cognitive backchannel module computes when and which backchannel should be displayed using information about the agent's beliefs towards the speaker's speech. We use Allwood's taxonomy of communicative functions of backchannels (Allwood, Nivre & Ahlsén, 1993): contact, perception, understanding, attitudinal reactions. From a previous study (Bevacqua, Heylen, Tellier, & Pelachaud, 2007) (Heylen, Bevacqua, Tellier, & Pelachaud, 2007) we have elaborated a lexicon of backchannels. The cognitive module selects which signals to display from the lexicon depending on the agent's reaction towards the speaker's speech. The third module is the mimicry module. When fully engaged in an interaction, mimicry of behaviours between interactants may happen (Lakin, Jefferis, Cheng, & Chartrand, 2003). This module determines when and which signals would mimic the agent. So far we are considering solely speaker's head movement in the signals to mimic. A selection algorithm determines which backchannels to display among all the potential signals that are outputted by three modules.

4.1.9 Action selection

The aim of the action selection module is to choose the most appropriated action according to the perception and the mental states of the ECA at every moment in time. Each action proposer will generate priorities for their actions and send all available actions with their priorities to the action selection mechanism without making choices between conflicting ones. We are inspirited from the free flow hierarchy (Tyrrell, 1993; de Sevin, 2006). The key idea is that, during the propagation of the activity in the hierarchy, no decisions are made before the lowest level in the hierarchy, represented by the actions, is reached. The action selection mechanism of the agent has to manage the priorities of actions according to the user’s level of interest (estimated by the agent), its own level interest and its personality. All these modulations of priorities are normalized to keep control of the decision-making. Some actions are not useful in the context of the action proposer but can be important in the global context of the ECA. For instance, the priority of smiling can be higher than the one of nodding, but in the end the ECA wants to show agreement. In this case, nodding is more appropriated even so its priority is lower. In the end, cognitive backchannels will always have the priority over the reactive ones because they take into account the mental state of the agent. The action selection module modulates the action priorities to endow the ECA to have more flexibility and adaptability in its decision-making in real-time.

4.1.10 Speech preprocessing

The speech preprocessing component is part of the FML2BML conceptual component in the SEMAINE architecture. It uses the MARY TTS system (Schröder, Charfuelan, Pammi, & Türk, 2008) to add pitch accent and phrase boundary information to FML documents in preparation of the realisation of functions in terms of visual behaviour.

For FML messages containing a `<speech>` element, the component partially synthesises the textual content, and identifies the number and locations of pitch accents as well as phrase boundaries predicted by the MARY system. The locations of accents and boundaries are inserted into the FML document as the non-standard extension elements `<pitchaccent>` and `<boundary>`, respectively.

4.1.11 FML2BML

The FML2BML takes as input both the agent's communicative intentions specified by the FML-APML language and some agent's characteristics. The main task of this component is to select, for each communicative intention, the adequate set of behaviours to display. The output of FML2BML is described in the BML language. It contains the sequence of behaviours with their timing information to be displayed by our virtual agent. The agent is characterized by its general tendency to behave. These characteristics are at the level of behaviours and not at the emotional or personality level, even though both levels are intrinsically correlated. The agent's general behaviour tendency is represented by the agent's baseline. This last one contains information on the preference the agent has in using its communicative modalities (head, gaze, face, gesture and torso) and on the expressive quality of each of them. Expressivity is defined by a set of parameters that affect the qualities of the agent's behaviour (e.g. wide vs. narrow gestures). The system uses the agent's baseline to compute how a given communicative intention is shown. Our system enables us to have agents defined with different baselines to communicate accordingly. It allows us to give some coherency in the agent's behaviours through out their interaction with users. For example an agent who is always calm (baseline), in a joy state (dynamicline), could produce just a light smile, without moving the rest of his body. On the other hand, a very exuberant agent (baseline), in an angry state (dynamicline), could produce a combination of behaviours like smiling, jumping, running, stretching his arms, screaming. More precisely, the computation goes as follow. Given the agent's baseline, the system computes the dynamicline associated with each intention. The dynamicline is described by the same set of parameters of the baseline but with updated values. The dynamicline, together with the current communicative intention, is then used to select the multimodal behaviour that best conveys the given intention. This computational process is explained in more details in (Mancini, & Pelachaud, 2007) (Mancini, & Pelachaud, 2008). Baseline and dynamicline allow us to modulate agent's behaviours.

4.1.12 Speech synthesis

The speech synthesis component is part of the conceptual component “BML Realiser” in the SEMAINE architecture. It uses the MARY TTS system to produce synthetic speech audio data as well as timing information in an extended BML document suitable as input to the visual BML realiser.

As a proof of concept, the current version of the speech synthesizer also generates vocal backchannels upon request.

4.1.13 BML realiser

This module generates the animation of our agent in MPEG-4 format (Ostermann, 2002). The input of the module is specified by the BML language. It contains the text to be spoken and/or a set of nonverbal signals to be displayed. Each BML tag is instantiated as a set of key-frames that are then smoothly interpolated. Facial expressions, gaze, gestures, torso movements are described symbolically in repository files. The Behaviour Realizer solves also eventual conflicts between the signals that are scheduled to happen on the same modality at the same time. The Behaviour Realizer uses repository files of predefined facial expressions, gestures, torso movements and so on. The agent's speech, which is also part of the BML input, is synthesized by an external TTS system (EULER, Festival, MARY). The TTS system provides the list of phonemes and their respective duration. This information is used to compute the lips movements. When the Behaviour Realizer receives no input, the agent does not remain still. It generates some idle movements whenever it does not receive any input. Periodically a piece of animation is computed and is sent to Player. It avoids unnatural “freezing” of the agent. For some modalities, like the head or gaze, the Behaviour Realizer manages also signals with “infinite” duration i.e. signals with an a priori unknown ending time. In this way we can ensure the agent will keep the head turned till another BML command arising from a new communicative intention is generated by the Intent Planner module.

4.1.14 Greta player

The FAP-BAP Player receives the animation generated by the Behaviour Realizer and plays it in a graphic window. The player is MPEG-4 compliant. The animation is defined by the Facial Animation Parameters (FAPs) and the Body Animation Parameters (BAPs). The FAPs define the shape deformation or movements of a set of 68 fundamental points on a synthetic face with respect to their neutral position; the BAPs represent rotations of body parts around specific joints. Facial and body configurations are described through respectively FAP and BAP frames. Each FAP or BAP frame received by the Player carries also the time (timestamp) of its visualization computed by the Behaviour Realizer. In case the Player receives more than one frame with the same timestamp it displays the latest one it receives.

5 Availability

The demonstrator system is publicly available from http://sourceforge.net/projects/semaine/.

Documentation on how to install and run the system is available at http://semaine.opendfki.de/wiki/SEMAINE-1.0.

6 References

Allwood, J., Nivre, J., & Ahlsén, E. (1993). On the semantics and pragmatics of linguistic feedback. Semantics, 9(1).

Banavar, G., Chandra, T., Strom, R., & Sturman, D. (1999). A Case for Message Oriented Middleware. In *Distributed Computing* (p. 846). Retrieved June 17, 2008, from http://dx.doi.org/10.1007/3-540-48169-9_1.

Bevacqua, E., Heylen, D., Tellier, M., & C. Pelachaud, C. (2007). Facial feedback signals for ECAs. In AISB'07 Annual convention, workshop \Mindful Environments", pages 147{153, Newcastle, UK, April.

EULER project. http://tcts.fpms.ac.be/synthesis/euler/home.html.

Festival speech synthesis system. http://www.cstr.ed.ac.uk/projects/festival/.

Hartmann, B., M. Mancini, C. Pelachaud, Implementing Expressive Gesture Synthesis for Embodied Conversational Agents, Gesture Workshop, LNAI, Springer, May 2005.

Heylen, D., Bevacqua, E., Tellier, M., & Pelachaud, C. (2007). Searching for prototypical facial feedback signals. In Proceedings of 7th International Conference on Intelligent Virtual Agents IVA 2007, pages 147{153, Paris, France.

Lakin, J. L., Jefferis, V. A., Cheng, C. M., & Chartrand, T. L. (2003). Chameleon effect as social glue: Evidence for the evolutionary signi¯cance of nonconsious mimicry. Nonverbal Behavior, 27(3):145{162).

Maatman, R. M., Gratch, J., & Marsella, S. (2005). Natural behavior of a listening agent. In T. Panayiotopoulos, J. Gratch, R. Aylett, D. Ballin, P. Olivier, and T. Rist, editors, Proceedings of 5th International Working Conference on Intelligent Virtual Agents, volume 3661 of Lecture Notes in Computer Science, pages 25{36, Kos, Greece. Springer.

Mancini, M., & Pelachaud, C. (2007). Dynamic behavior qualifiers for conversational agents. In Catherine Pelachaud, Jean-Claude Martin, Elisabeth Andr, Grard Chollet, Kostas Karpouzis, and Danielle Pel, editors, Proceedings of 7th International Conference on Intelligent Virtual Agents, volume 4722 of Lecture Notes in Computer Science, pages 112{124, Paris, France, 2007. Springer.

Mancini, M., & Pelachaud, C. (2008). Distinctiveness in multimodal behaviors. In Lin Padgham, David C. Parkes, JÄorg MÄuller, and Simon Parsons, editors, Proceedings of Conference on Autonomous Agents and Multi-Agent Systems (AAMAS08), 2008.

MARY text-to-speech system. http://mary.dfki.de/.

Morency, L.-P., de Kok, I., Gratch, J. (2008). Predicting listener backchannels: A probabilistic multimodal approach. In H. Prendinger, J. C. Lester, and M. Ishizuka, editors, Proceedings of 8th International Conference on Intelligent Virtual Agents, volume 5208 of Lecture Notes in Computer Science, Tokyo, Japan. Springer.

Ostermann, J. (2002). Face animation in mpeg-4. In I. Pandzic and R. Forchheimer, editors, MPEG-4 Facial Animation - The Standard Implementation and Applications, pages 17{55. Wiley, England.

de Sevin E. 2006. An Action Selection Architecture for Autonomous Virtual Humans in Persistent Worlds. PhD. Thesis. VRLab EPFL.

Schröder, M., Charfuelan, M., Pammi, S., & Türk, O. (2008). The MARY TTS entry in the Blizzard Challenge 2008. In *Proc. Blizzard Challenge*. Brisbane, Australia.

Schröder, M., Cowie, R., Heylen, D., Pantic, M., Pelachaud, C., & Schuller, B. (2008). Towards responsive Sensitive Artificial Listeners. In *Fourth International Workshop on Human-Computer Conversation*. Bellagio, Italy.

Tyrrell T. 1993. Computational Mechanisms for Action Selection. In Centre for Cognitive Science. Phd. Thesis. University of Edinburgh.

Ward, N., & Tsukahara, W. (2000). Prosodic features which cue back-channel responses in English and Japanese. Journal of Pragmatics, 23:1177{1207, 2000.

Wolpert, D. M., & Flanagan, J. R. (2001). Motor prediction. *Current Biology*, *11*(18), R729-R732.

Wöllmer, M., Eyben, F., Reiter, S., Schuller, B., Cox, C., Douglas-Cowie, E., Cowie, R. (2008). Abandonning Emotion Classes – Towards Continuous Emotion Recognition with Modelling of Long-Range Dependencies. Proceedings of Interspeech 2008, pp. 597-600.

Appendix I: SemaineML Markup

A custom set of ad hoc markup tags is used to encode domain-specific information. It is neither envisaged nor does it seem appropriate to aim for a standardisation of this markup. It is rather envisaged as the “catchall” representation for information that needs to be represented somehow but does not seem to fit any of the standard formats.

The following list of examples is a starting point for SemaineML markup, as a tentative formalisation of project discussions. It will require thorough revision as the SAL domain is implemented completely.

I.1 Feature functionals and behaviour description

Here we need to use an ad hoc markup. We give it the namespace prefix “semaine:”, associated with the namespace URL “http://www.semaine-project.eu”.

```
<semaine:feature xmlns:semaine=”http://www.semaine-project.eu”
                 name=”...” value=”...”/>


<semaine:behaviour xmlns:semaine=”http://www.semaine-project.eu”
        name=”...” intensity=”...”/>
```

I.2 ASR output

Details on what needs to be represented must be spelled out. In the simplest case, we could use a simple text element:

```
<semaine:text xmlns:semaine=”http://www.semaine-project.eu”>
  this is what the user said
</semaine:text>
```

Or else we could specify a full lattice of word candidates, using the emma:lattice element (http://www.w3.org/TR/emma/#s3.4), or something custom in between.

I.3 Current best guess values for states: user, agent, dialogue

These seem to be of a different kind: they are not interpretations but rather beliefs – there is no confidence, also no time stamps – the values refer to “now”. EMMA doesn't seem right for this.

What to represent:

- “user state”: what does the agent know about the user state: he/she is in this state, has ended the turn, user's interest level, engagement in the conversation, ...
- “agent state”: agent as a character of its own: a personality, a current emotion, stance towards the user, the topic under discussion, the agent's state of interest, whether agent and user are emotionally concordant, ...
- “dialogue state”: history of the statements made, turn, current topic, ...

I.3.1 User state

epistemic/affective state information: could use EmotionML without confidence attribute, combined with semaine-specific annotations.

```
<semaine:user-state xmlns:semaine=”http://www.semaine-project.eu”>
    <emotion:category set=”emotion-quadrant” name=”active-positive”/>
    <emotion:category set=”interest” name=”interested”/>
    <emotion:dimensions set=”engagement”>
        <emotion:engagement value=”0.9”/>
    </emotion:dimensions>
    ...
    <semaine:event name=”turn-event” value=”turn-yielding-signal”
time=”123456789”/>
    <semaine:behaviour name=”gaze-at-agent” intensity=”0.9”/>
    ...
</semaine:user-state>
```

I.3.2 Agent state

can represent emotion, stance toward the user etc. using EmotionML.

```
<semaine:agent-state xmlns:semaine=”http://www.semaine-project.eu”>
    <emotion:category set=”emotion-quadrant” name=”active-positive”/>
    <emotion:category set=”interest” name=”interested”/>
    <emotion:emotion type=”stance”>
        <emotion:category set=”like-dislike” name=”like”/>
        <emotion:object name=”user”/>
    </emotion:emotion>
    <emotion:emotion type=”stance”>
        <emotion:category set=”like-dislike” name=”dislike”/>
        <emotion:object name=”topic”/>
    </emotion:emotion>
    ...
    <semaine:emotionally-concordant-with-user value=”true”/>
    ....
</semaine-user-state>
```

I.3.3 Dialogue state

the history of statements made may be a longer thing; current turn holder and topic are shorter.

```
<semaine:dialog-state xmlns:semaine=”http://www.semaine-project.eu”>
    <semaine:speaker who=”agent”/>
    <semaine:listener who=”user”/>
    <semaine:topic name=”food”/>
    <semaine:dialog-history>
        <semaine:dialog-act who=”agent” topic=”food” time=”123456789”>
            what do you like eating?
        </semaine:dialog-act>
        <semaine:dialog-act speaker=”agent” topic=”repair” time=”123453000”>
            maybe I misunderstood?
        </semaine:dialog-act>
        ...
    </semaine:dialog-history>
    ...
</semaine:dialog-state>
```

Appendix II: FML Markup

FML-APML

The FML version we are currently using is based on the APML, Affective Presentation Markup Language (DeCarolis et al 2004). FML-APML is used to model the agent's communicative intention. FML-APML is an evolution of APML and presents some similarities and differences. The FML-APML tags are an extension of the ones defined by APML, so all the communicative intentions that we can represent in APML are also present in FML-APML.

APML is based on Isabella Poggi's model of communication, where each tag corresponds to one of the communicative intentions:

- *certainty*: this is used to specify the degree of certainty the agent intends to express.
  - Possible values: *certain, uncertain, certainly not, doubt*.
- *meta-cognitive*: this is used to communicate the source of the agent's beliefs.
  - Possible values: *planning, thinking, remembering*.
- *performative*: this represents the agent's performative.
  - Possible values: *implore, order, suggest, propose, warn, approve, praise, recognize, disagree, agree, criticize, accept, advice, confirm, incite, refuse, question, ask, inform, request, announce, beg, greet*.
- *theme/rheme*: these represent the topic/comment of conversation; that is, respectively, the part of the discourse which is already known or new for the conversation's participants.
- *belief-relation*: this corresponds to the *metadiscoursive* goal, that is, the goal of stating the relationship between different parts of the discourse; it can be used to indicate contradiction between two concepts or a cause-effect link.
  - Possible values: *gen-spec, cause-effect, solutionhood, suggestion, modifier, justification, contrast*.
- *turnallocation*: this models the agent's metaconversational goals, that is, the agent's intention to take or give the conversation floor.
  - Possible values: *take, give*.
- *affect*: this represents the agent's emotional state. Emotion labels are taken from the OCC model of emotion.
  - Possible values: *anger, disgust, joy, distress, fear, sadness, surprise, embarrassment, happy-for, gloating, resentment, relief, jealousy, envy, sorry-for, hope, satisfaction, fear-confirmed, disappointment,pride, shame, reproach, liking, disliking, gratitude, gratification, remorse, love, hate*.
- *emphasis*: this is used to emphasize (that is, to convey its importance) what the agent communicates either vocally (by adding pitch accents to the synthesized agent's speech) or through body movements (by raising the eyebrows, producing beat gestures, etc.).
  - Possible values: *low, medium, high*.

FML-APML tags are used to model the agent's communicative intention. Each tag represents a communicative intention (to inform about something, to refer to a place/object/person, to express an emotional state, etc.) that lasts from a certain starting time, for a certain number of seconds. The attributes common to all the FML-APML tags are:

- *name*: the name of the tag, representing the communicative intention modelled by the tag. For example, the name *performative* represents a performative communicative intention.
- *id* : a unique identifier associated to the tag; it allows one to refer to it in an unambiguous

way.

- *type*: this attribute allows us to better specify the communicative meaning of the tag. For example, a performative tag has many possible values for the *type* attribute: *implore, order, suggest, propose, warn, approve, praise, etc.*.
- *start*: starting time of the tag, in seconds. Can be absolute or relative to another tag. It represents the point in time at which the communicative intention modelled by the tag begins.
- *end*: duration of the tag. Can be a numeric value (in seconds) relative to the beginning of the tag or a reference to the beginning or end of another tag (or a mathematical expression involving them). It represents the duration of the communicative intention modelled by the tag.
- Importance: a value between 0 and 1 which determines the probability that the communicative intention encoded by the tag is communicated through nonverbal behaviour.

The timing attributes *start* and *end* also allow us to model the synchronization of the FML-APML tags. They both can assume absolute or relative values. In the first case, the attributes are numeric non-negative values. In the second case we can specify the starting or ending time of other tags, or a mathematical operation involving them. Note that the optional *end* attribute allows us to define communicative intentions that start at a certain point in time and last until new communicative intentions are defined. Here is an example of absolute and relative timings.

```
<FML-APML>
<tag1 id="id1" start="0" end="2"/>
<tag2 id="id2" start="2" end="3"/>
</FML-APML>
```

In the above FML-APML code, *tag*1 starts at time 0 and lasts 2 seconds; *tag*2 starts at time 2, and lasts 3 seconds.

All the timings are *absolute*, that is, they are both relative only to the beginning of the actual FML-AMPL entry (equivalent to time 0).

```
<FML-APML>
<tag3 id="id3" start="0" end="2"/>
<tag4 id="id4" start="t1:end+1"
end="t1:end+3"/>
</FML-APML>
```

In this case, the first tag is the same as before. On the other hand, *tag*2 has a *relative* timing as it starts as the first tag ends and lasts for 3 seconds. FML-APML tags can be attached and synchronized to the text spoken by the agent. This is modelled by including a special tag, called *speech*, in the FML-APML syntax. Within this tag, we write the text to be spoken along with synchronization points (called *time markers*) which can be referred to by the other FML-APML tags in the same entry. For example:

```
<FML-APML>
<speech id="s1">
<tm id="tm1"/>
what are you
<tm id="tm2"/>
doing
<tm id="tm3"/>
here
<tm id="tm4"/>
</speech>
```

```
<tag3 id="id3" start="s1:tm2" end="s1:tm4"/>
</FML-APML>
```

With the above code, we specify that the communicative intention of *tag*3 starts in correspondence with the word *doing* and ends at the end of the word *here*.

Adaptation for the SEMAINE system

For the format used in the SEMAINE system, we start from the format used in the Greta system, with slight modifications to support XML namespaces. The following is an example.

A top-level element <fml-apml>, which has no namespace, encloses both a <bml> element (in the BML namespace) and an <fml> element (in the FML namespace). The FML section should contain functional information, such as information about the linguistic structure, the emotion to be expressed in the voice, etc. This format is extremely preliminary and needs careful attention to be meaningful and consistent in the SEMAINE system. For example, it can be seen in the example below that the <emotion> tag in the FML section is not yet adapted to use EmotionML.

```

<?xml version="1.0" encoding="UTF-8"?>
<fml-apml version="0.1">
   <bml xmlns="http://www.mindmakers.org/projects/BML" id="bml1">
      <speech id="s1" language="en_US" text="Hi, I'm Poppy, the eternal
optimist. What's your name? ">
         <mark xmlns="http://www.w3.org/2001/10/synthesis" name="s1:tm1"/>
         Hi,
         <mark xmlns="http://www.w3.org/2001/10/synthesis" name="s1:tm2"/>
         I'm
         <mark xmlns="http://www.w3.org/2001/10/synthesis" name="s1:tm3"/>
         Poppy,
         <mark xmlns="http://www.w3.org/2001/10/synthesis" name="s1:tm4"/>
         the
         <mark xmlns="http://www.w3.org/2001/10/synthesis" name="s1:tm5"/>
         eternal
         <mark xmlns="http://www.w3.org/2001/10/synthesis" name="s1:tm6"/>
         optimist.
         <mark xmlns="http://www.w3.org/2001/10/synthesis" name="s1:tm7"/>
         What's
         <mark xmlns="http://www.w3.org/2001/10/synthesis" name="s1:tm8"/>
         your
         <mark xmlns="http://www.w3.org/2001/10/synthesis" name="s1:tm9"/>
         name?
         <mark name="s1:tm10"/>
      </speech>
   </bml>
   <fml xmlns="http://www.mindmakers.org/fml" id="fml1”>
             <performative id="p1" type="announce" start="s1:tm1"
end="s1:tm4"/>
             <rheme id="r1" start="s1:tm1" end="s1:tm4"/>
             <performative id="p2" type="warn" start="s1:tm4" end="s1:tm6"/>
             <theme id="t1" start="s1:tm4" end="s1:tm6"/>
             <world id="w1" ref_type="person" ref_id="self" start="s1:tm7"
end="s1:tm8"/>
             <performative id="p3" type="suggest" start="s1:tm8"
end="s1:tm11"/>
             <rheme id="r2" start="s1:tm6" end="s1:tm14"/>
             <emotion id="e1" type="anger" start="s1:tm9" end="s1:tm14"/>
             <world id="w2" ref_type="person" ref_id="self"
prop_type="location" prop_value="foreign" start="s1:tm12" end="s1:tm13"/>
       </fml>
</fml-apml>
```

Appendix III: BML Markup

The BML language specifies the nonverbal signals that can be expressed through the agent communication modalities. Each BML top-level tag corresponds to a behaviour the agent is to produce on a given modality: head, torso, face, gaze, body, legs, gesture, speech, lips. In the current version for each modality one signal can be chosen from a short fixed list. Each signal has defined a start time and duration. It can be absolute (in seconds) or relative, in relation to the other verbal or nonverbal signal. The BML language version implemented for Greta contains some extensions which allow us to define labels to use a larger set of signals which can be produced by the agent. Moreover through a set of parameters, called *expressivity parameters*, it is possible to specify the quality of movement of each signal (Hartmann et al 2005): for example, Greta can perform the same gesture in different ways: quickly or slowly, smoothly or jerkily, etc.

BML as a markup language is more mature than FML. The SAIBA initiative (http://wiki.mindmakers.org/projects:BML:main) is working on a draft of a specification version 1.0. The version used in the SEMAINE project is currently a mix of the previous version of BML used in the Greta system, the latest BML draft, and attempts to be clean with respect to namespaces, notably regarding the annotation of speech content using W3C SSML (http://www.w3.org/TR/speech-synthesis/).

The following is an example BML document used in the current SEMAINE system. In addition to the BML section in the FML example of Appendix II, it features <pitchaccent> and <boundary> tags inserted by the speech preprocessor. Potentially, the BML markup would also include tags to drive visual behaviour.

```
<?xml version="1.0" encoding="UTF-8"?>
<bml xmlns="http://www.mindmakers.org/projects/BML" id="bml1">
  <speech id="s1" language="en_US" text="Tell me the details"
xmlns:ssml="http://www.w3.org/2001/10/synthesis">
         <mark xmlns="http://www.w3.org/2001/10/synthesis" name="s1:tm1"/>
        Tell
        <mark xmlns="http://www.w3.org/2001/10/synthesis" name="s1:tm2"/>
        me
        <mark xmlns="http://www.w3.org/2001/10/synthesis" name="s1:tm3"/>
        the
        <mark xmlns="http://www.w3.org/2001/10/synthesis" name="s1:tm4"/>
        details
        <mark xmlns="http://www.w3.org/2001/10/synthesis" name="s1:tm5"/>
    <pitchaccent end="s1:tm2" id="xpa2" start="s1:tm1"/>
    <pitchaccent end="s1:tm5" id="xpa14" start="s1:tm4"/>
    <boundary id="b16" time="s1:tm5"/>
  </speech>
</bml>
```

The following version of the above file includes concrete timing information, added by the speech synthesis component at the same time as producing audio data. The format is therefore suitable for realisation in the BML realiser component.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<bml xmlns="http://www.mindmakers.org/projects/BML" id="bml1">
<speech xmlns:ssml="http://www.w3.org/2001/10/synthesis" id="s1"
language="en_US" text="Tell me the details">

<ssml:mark name="tm1"/>
Tell
<mary:syllable xmlns:mary="http://mary.dfki.de/2002/MaryXML" accent="1"
stress="1">
```

```
<mary:ph d="0.091" end="0.091" p="t"/>
<mary:ph d="0.067" end="0.158" p="E"/>
<mary:ph d="0.089" end="0.247" p="l"/>
</mary:syllable>

<ssml:mark name="tm2"/>
me
<mary:syllable xmlns:mary="http://mary.dfki.de/2002/MaryXML" stress="1">
<mary:ph d="0.055" end="0.302" p="m"/>
<mary:ph d="0.085" end="0.387" p="i"/>
</mary:syllable>

<ssml:mark name="tm3"/>
the
<mary:syllable xmlns:mary="http://mary.dfki.de/2002/MaryXML">
<mary:ph d="0.074" end="0.461" p="D"/>
<mary:ph d="0.041" end="0.502" p="@"/>
</mary:syllable>

<ssml:mark name="tm4"/>
details
<mary:syllable xmlns:mary="http://mary.dfki.de/2002/MaryXML">
<mary:ph d="0.091" end="0.593" p="d"/>
<mary:ph d="0.071" end="0.664" p="I"/>
</mary:syllable>
<mary:syllable xmlns:mary="http://mary.dfki.de/2002/MaryXML">
<mary:ph d="0.124" end="0.788" p="t"/>
<mary:ph d="0.09" end="0.878" p="EI"/>
<mary:ph d="0.146" end="1.024" p="l"/>
<mary:ph d="0.217" end="1.241" p="z"/>
</mary:syllable>
<mary:boundary id="b32" type="HH" start="s1:tm5" end="0.2"
xmlns:mary="http://mary.dfki.de/2002/MaryXML"/>
<ssml:mark name="tm5"/>
<pitchaccent id="xpa2" start="s1:tm1" end="s1:tm2" time="0.1235"/>
</speech>
</bml>
```

Appendix IV: SMILE Low-Level Features

Feature Group	Features in Group	#
Signal energy (frame-wise)	Root Mean-Square (RMS-E)	1
	Logarithmic Energy (log-E)	1
Fundamental Frequency (F0) based on Autocorrelation (ACF)	F0-Frequency (Hz)	1
	F0-Strength (Peak height of ACF)	1
	F0-Quality (Zero-Crossing Rate of ACF wrt. to F0)	1
Mel-Frequency Cepstral Coefficients (MFCC)	Coefficients 0-15	16
Spectral	Centroid	1
	Roll-off (10%, 25%, 50%, 75%, 90%)	5
	Flux	1
	Frequency-band energy (0-250Hz, 0-650Hz, 250-650Hz, 1000-4000Hz)	4
	Position of Maximum	1
	Position of Minimum	1
Time signal features	Zero-Crossing Rate (ZCR)	1
	Maximum Sample	1
	Minimum Sample	1
	Mean (DC component)	1
Integration of the following features is currently in progress:		
Linear Predictive Coding (LPC)	LPC coefficients	(12)
Voice Quality	Harmonics-to-Noise-Ratio (HNR)	(1)
	Jitter	(1)
	Shimmer	(1)
Pitch by Harmonic Product Spectrum	F0-Frequency	(1)
	Probability of Voicing (pitch strength)	(1)

Appendix V: SMILE Functionals

Functionals Group	Functionals in Group	#
Extremes	Maximum/minimum value	2
	Relative position of maximum/minimum value	2
	Range	1
Mean	Arithmetic mean	1
	Arithmetic mean of absolute values	1
	Quadratic mean	1
	Maximum value - Arithmetic mean	1
Non-Zero	Arithmetic mean of all non-zero values,	1
	Percentage of all non-zero values wrt. total number of values.	1
Quartiles	25%, 50% (median), and 75% quartile	3
	Inter-quartile range (IQR): 2-1, 3-2, 3-1	3
Percentiles	95%, 98% percentile	2
Higher Moments	Skewness, Kurtosis	2
	Variance, Standard deviation	2
Centroid	Centroid: Centre of Gravity	1
Threshold Crossing Rates	Zero-Crossing Rate	1
	Mean-Crossing Rate	1
Linear Regression	2 coefficients (m,t)	2
	Linear and quadratic regression error	2
Quadratic Regression	3 coefficients (a,b,c)	3
	Linear and quadratic regression error.	2
Peaks	Number of peaks (maxima)	1
	Mean distance between peaks	1
	Mean value of all peaks	1
	Mean value of all peaks – Arithmetic mean	1
Segments	Number of segments, based on delta thresholding	1
Times	Time, within which values are above 75% of the total range	1
	Time within which values are below 25% of the total range.	1
	Rise time and fall time	2
Discrete Cosine Transformation (DCT)	Coefficients 0-5	6