

The SEMAINE API: Towards a standards-based framework for building emotion-oriented systems

Marc Schröder, DFKI GmbH, Stuhlsatzenhausweg 3, 66123 Saarbrücken, Germany,
schroed@dfki.de

Abstract

This paper presents the SEMAINE API, an open source framework for building emotion-oriented systems. By encouraging and simplifying the use of standard representation formats, the framework aims to contribute to interoperability and reuse of system components in the research community. By providing a Java and C++ wrapper around a message-oriented middleware, the API makes it easy to integrate components running on different operating systems and written in different programming languages. The SEMAINE system 1.0 is presented as an example of a full-scale system built on top of the SEMAINE API. Three small example systems are described in detail to illustrate how integration between existing and new components is realised with minimal effort.

1. Introduction

Systems with some emotional competence, so-called “affective computing” systems, are a promising and growing trend in human-machine interaction (HMI) technology. They promise to register a user's emotions and moods, for example to identify angry customers in interactive voice response (IVR) systems; to generate situationally appropriate emotional expression, such as the apologetic sound of a synthetic voice when a customer request cannot be fulfilled; in certain conditions they even aim to identify reasons for emotional reactions, using so-called “affective reasoning” technology. Ultimately, such technology may indeed lead to more natural and intuitive interactions between humans and machines of many different kinds, and thus contribute to bridging the “digital divide” that leaves non-techy users helpless in front of increasingly complex technology. This aim is certainly long-term; slightly more in reach is the use of emotion-oriented technologies in the entertainment sector, such as in computer games, where emotional competence, even in a rudimentary state, can lead to new effects and user experiences.

In fact, an increasing number of interactive systems deal with emotions and related states in one way or another. Common tasks include the analysis of the user's affective state [1] from the face [2][3], the voice [4][5], or physiological measures [6]; the evaluation of events in the world according to affective criteria [7]; and the generation of emotion-related system behaviour, such as facial expression [8][9] and voice [10][11], but also other media such as music and colour [12].

A number of elements are common to the different systems. All of them need to represent emotional states in order to process them; and many of the systems are built from components, such as recognition, reasoning, or generation components, which need to communicate with one another to provide the system's capabilities. In the past, systems used custom solutions to these challenges, usually in clearly delimited ways that were tailor-made for their respective application areas (see Section 2 for related work). However, existing emotion-oriented systems seem to be neither explicitly geared towards the use of standard representations, nor are they available as open source.

Standards enable interoperability and reuse. Nowadays, standards are taken for granted in such things as the voltage of electricity in a given country, fuel grade, or the dimensions of screw threads [13]. More recently, standards for document formats [14] have entered the public debate, under the perspective of long-term accessibility of documents. Web standards such as the Hyper-Text Markup Language HTML [15] enable access to information in the world wide web through a broad variety of software products supporting the standard format.

Proprietary formats, on the other hand, can be used to safeguard a company's competitive advantage. By patenting, or even by simply not documenting a representation format, a company can make sure not to open up the market to its competitors.

The same considerations seem to apply in the emerging area of emotion-oriented systems. Agreeing on standard formats and interfaces would enable interoperability and reuse. An initial investment of effort in defining suitably broad but sufficiently delimited standard formats can be expected to pay off in the long run by removing the need to start from scratch with every new system built. Where formats, software frameworks and components are made generally available, for example as open source, these can be used as starting points and building blocks for new systems, speeding up development and research.

This paper describes the SEMAINE API, a toolkit and middleware framework for building emotion-oriented systems in a modular way from components that communicate using standard formats where possible. It describes one full-scale system built using the framework, and illustrates the issue of reuse by showing how three simple applications can be built with very limited effort.

The paper is structured as follows. Section 2 reviews related work. Section 3 presents the SEMAINE API from the technological point of view, including the support for integrating components into a system and the supported representation formats. Section 4 describes a first larger system based on the API, the SEMAINE system 1.0, and provides some detail on the system architecture and the components used in the system. Section 5 exemplifies the use of the SEMAINE API for building new emotion-oriented systems by showing how to implement three demo systems. Section 6 presents an evaluation of the framework in terms of response times and developer friendliness.

2. Related Work

Several integrated research systems have been built in the recent past which incorporate various aspects of emotional competence. For example, the NECA project [16] generated scripted dialogues between embodied conversational agents in a web environment. Its system was based on a pipeline architecture in which a Rich Representation Language RRL [17] was successively enriched with component information. The VirtualHuman project [18] supported dialogues involving multiple humans and multiple agents. Both humans and agents were represented in the system by Conversational Dialogue Engines [19] communicating with each other using the concepts of an application-specific ontology. The FearNot! system [20], an educational application helping children to deal with bullying, uses an architecture involving reactive and deliberative layers and memory components, as well as sensors and effectors. Central to the processing of emotions in FearNot are appraisal processes realised in terms of the OCC model [21]. The project IDEAS4Games realised an emotion-aware poker game, in which two agents and a user played against each others with physical cards carrying RFID tags [22]. The emotions of characters were computed from game events using an affective reasoner [7], and realised through the synthetic voice and through body movements. Whereas all of these systems are conceptually modular, none of them is explicitly geared towards the use of standard representations, and none of the systems is available as open source.

Existing programming environments provide relevant component technologies but do not allow the user to integrate components across programming languages and operating system platforms. For example, the EMotion FX SDK [23] is a character animation engine that supports animation designers to streamline the process of designing the graphical properties of games characters and include them into a game environment. It includes facial animation such as emotional facial expressions and lip synchronisation. Luxand FaceSDK [24] is a facial feature point detection software, which can be used for face detection, the generation of 3D face models, and the automatic

creation of animated avatars. Both are relevant component technologies for an emotion-oriented system, but do not solve the issue of how to integrate heterogeneous components across platforms.

When looking beyond the immediate area of emotion-oriented systems, however, we find several toolkits for component integration.

In the area of ubiquitous computing, the project Computers in the Human Loop (CHIL) investigated a broad range of smart space technologies for smart meeting room applications. Its system integration middleware, named CHILix [25], uses XML messages for integrating the components in a smart space application. CHILix uses the freely available NIST DataFlow System II [26] as the low-level message routing middleware. The XML message format used seems to be a domain-specific, custom format; documentation does not seem to be freely available.

In the domain of interactive robots research, the project CognitiveSystems (CoSy) has developed a system integration and communication layer called CoSy Architecture Schema Toolkit (CAST) [27]. The components of a robot's architecture are structured into sub-architectures in which components work on a jointly accessible working memory. Access to data structures is through predefined *types*, similar to objects. Communication passes through the object-oriented middleware Ice [28].

The main features of the CHILix, CAST and SEMAINE API integration frameworks are summarised in Table 1. It can be seen that out of these frameworks, the SEMAINE API is the only one that is based on standard formats and can be flexibly used with closed and open source components due to its less restrictive LGPL license.

	CHILix	CAST	SEMAINE API
Application domain	Smart Spaces	Interactive Robots	Emotion-oriented systems
Integration approach	XML messages	Objects in shared working memory	XML messages
Using standard formats	no	no	yes
Operating systems	Windows, Linux, Mac	Linux, Mac	Windows, Linux, Mac
Programming languages	C++, Java	C++, Java	C++, Java
Low-level communication platform	NDFS II	Ice	ActiveMQ
Open source	no	yes, GPL	yes, LGPL

Table 1: Key properties of several component integration frameworks for real-time systems

3. The SEMAINE API

The SEMAINE API has been created in the EU-funded project “SEMAINE: Sustained Emotionally coloured Machine-human Interaction using Nonverbal Expression” [29], as a basis for the project's system integration. The project aims to build a multimodal dialogue system with an emphasis on non-verbal skills – detecting and emitting vocal and facial signs related to the interaction, such as backchannel signals, in order to register and express information such as continued presence, attention or interest, an evaluation of the content, or an emotional connotation. The system has strong real-time constraints, because it must react to the user's behaviour while the user is still speaking [30].

The project's approach is strongly oriented towards making basic technology for emotion-oriented interactive systems available to the research community, where possible as open source. While the primary goal is to build a system that can engage a human user in a conversation in a plausible way, it is also an important aim to provide high-quality audiovisual recordings of human-machine interactions, as well as software components that can be reused by the research community.

In front of this background, the SEMAINE API has the following main aims:

- to integrate the software components needed by the SEMAINE project in a robust, real-time system capable of multi-modal analysis and synthesis;
- to enable others to re-use the SEMAINE components, individually or in combination, as well as to add their own components, in order to build new emotion-oriented systems.

The present section describes how the SEMAINE API supports these goals on a technical level. First, we present the SEMAINE API's approach to system integration, including the message-oriented middleware used for communication between components, as well as the software support provided for building components that integrate neatly into the system and for producing and analysing the representation formats used in the system. After that, we discuss the representation formats used, their status with respect to standardisation, and the extent to which domain-specific representations appear to be needed.

3.1. System integration

Commercial systems often come as single, monolithic applications. In these systems, the integration of system components is as tight as possible: any system-internal components communicate via shared memory access, and any modularity is hidden from the end user.

In the research world, the situation is different. Different research teams, cooperating in research projects in different constellations, are deeply rooted in different traditions; the components they contribute to a system are often extensions of pre-existing code. In such situations, the only way to fully integrate all system components into a single binary executable would be to re-implement substantial portions of the code. In most cases, research funding will not provide the resources for that. Therefore, it is often necessary to build an overall system from components that may be running on different operating systems, and that may be written in different programming languages.

Key properties of system integration are as follows. The SEMAINE API uses a message-oriented middleware (MOM, see Section 3.1.1) for all communication in the system. As a result, all communication is asynchronous, which decouples the various parts of the system. The actual processing is done in “components”, which communicate with one another over “Topics” (see Section 3.1.2) below the named Topic hierarchy `semaine.data.*`. Each component has its own “meta messenger”, which interfaces between the component and a central system manager. When a component is started, its meta messenger registers with the system manager over a special meta communication channel, the Topic `semaine.meta`. At registration time, the meta messenger describes the component in terms of the data Topics that it sends data to and that it receives data from; if the component is an input or output component (in the sense of the user interface), that status is communicated as well. The system manager is keeping track of the components that have been registered, and checks at regular intervals whether all components are still alive by sending a “ping”. In reply to such a ping, each meta messenger confirms the respective component's status and sends debug information such as the average time spent processing incoming requests. The system manager keeps track of the information about registered components, and sends global meta messages informing all components that the overall system is ready or, if a component has an error or is stalled, that the system is not ready. Also, the system manager resets a global timer to zero when the system becomes ready. All components use this global time via their meta messenger, and

thus can meaningfully communicate about timing of user and system events even across different computers with potentially unsynchronised hardware clocks.

A centralised logging functionality uses the Topics below `semaine.log.*`. By convention, messages are sent to `semaine.log.<component>.<severity>`, e.g. the Topic `semaine.log.UtteranceInterpreter.debug` would be used for debug messages of component `UtteranceInterpreter`. The severities used are “debug”, “info”, “warn” and “error”. Through this design, it is possible for a log reader to subscribe, e.g., to all types of messages from one component, or to all messages from all components that have at least severity “info”, etc. Furthermore, a configurable message logger can optionally be used to log certain messages in order to follow and trace them. Notably, it is possible to read log messages in one central place, independently of the computer, operating system or programming language used by any given component.

Figure 1 illustrates this system architecture. Components communicate with each other via Topics in the `semaine.data` hierarchy (indicated by black arrows). Meta information is passed between each component's meta messenger and the system manager via the `semaine.meta` Topic (grey arrows). Optionally, components can write log messages, and a message logger can log the content messages being sent; a configurable log reader can receive and display a configurable subset of the log messages (dashed grey arrows).

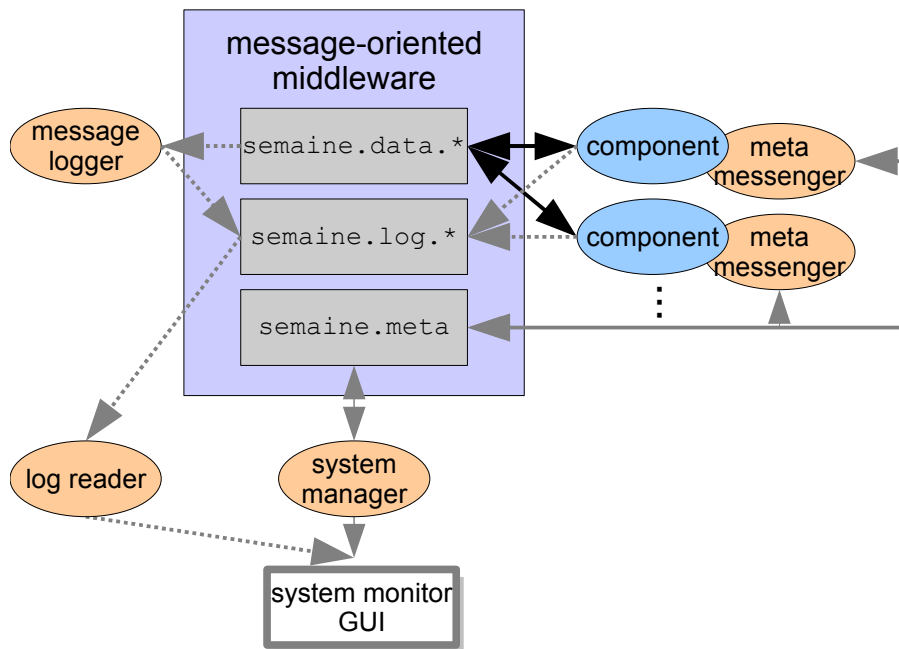


Figure 1: SEMAINE API system architecture

Optionally, a system monitor GUI visualises the information collected by the system manager as a message flow graph. Input components are placed at the bottom left, output components at the bottom right, and the other components are sorted to the extent possible based on the data input/output relationships, along a half-circle from left to right. Component B comes later in the graph than component A if A's output is an input to B or if there is a sequence of components that can process A's output into B's input. This criterion is overly simplistic for complex architectures, especially with circular message flows, but is sufficient for simple quasi-linear message flow graphs. If a new component is added, the organisation of the flow graph is recomputed. This way, it is possible to visualise message flows without having to pre-specify the layout.

Figure 2 shows the system monitor GUI for the SEMAINE system 1.0 described in Section 4. Components are represented as ovals, whereas Topics are represented as rectangles. Topics are shown in yellow when they have just transported a new message, and in grey when they have not seen any recent messages. When the user clicks on the Topic rectangle, the GUI shows a history of

the messages transported by a given Topic; debug information about a component is shown when the user clicks on the component oval. A log message reader is shown on the right-hand side. It can be configured with respect to the components and the severity of messages to show.

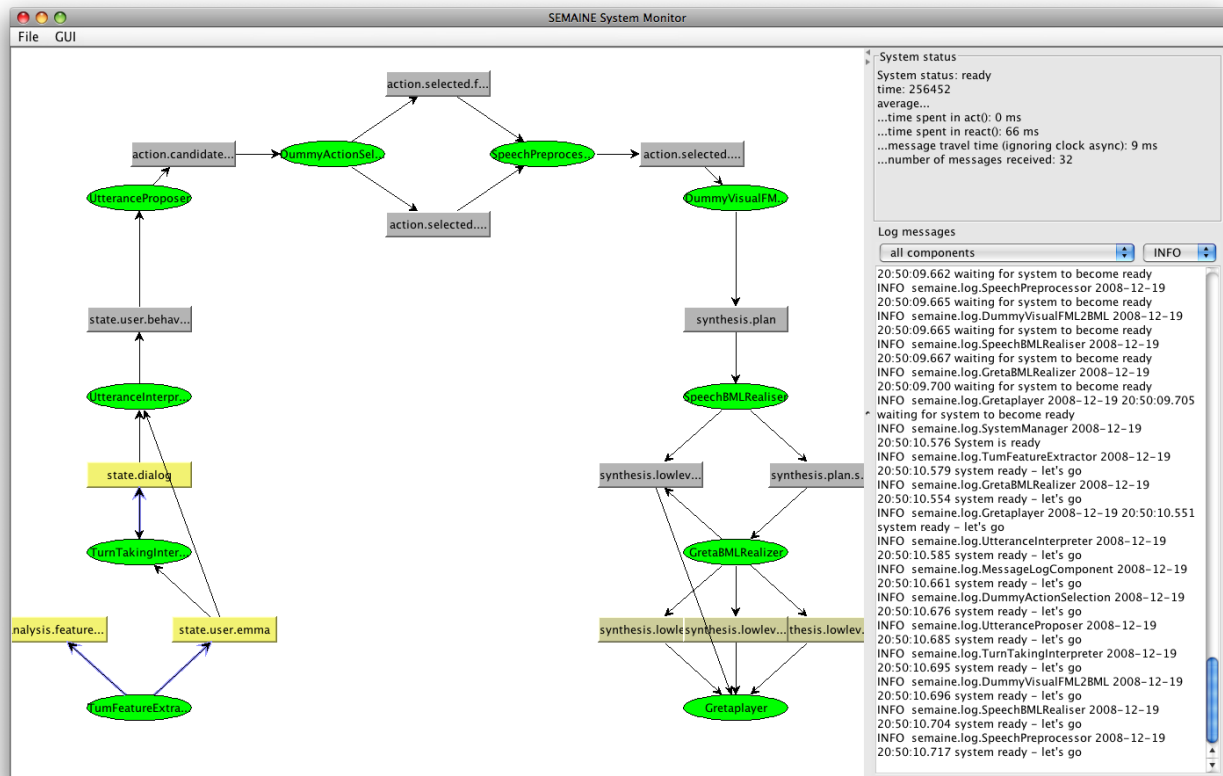


Figure 2: Screenshot of the System Monitor GUI showing the implemented SEMAINE system 1.0.

The remainder of this section describes the various aspects involved in the system in some more detail.

3.1.1. Message-oriented middleware

A message-oriented middleware (MOM) [31] is specifically designed to integrate different applications or processes through messages being routed from publishers to subscribers. The aim is to “glue together applications both within and across organizations, without having to re-engineer individual components” [31]. One method for describing how messages should be sent from sources to destinations is a message flow graph, in which the nodes represent components and the arcs represent message flows [31]. A major advantage of a generic message-oriented middleware lies in its flexibility. Through a publish-subscribe model, n-to-m connections are trivial to realise; furthermore, the system architecture can be re-arranged very easily – adding or removing a component consuming or producing a certain message type does not require any changes elsewhere in the architecture. Communication is asynchronous, so that a publisher does not need to wait for confirmation by subscribers that a message has been received. Where a response is needed, this must be sent as a separate asynchronous message, and the response's receiver needs to match it to the original request.

The SEMAINE API provides an abstraction layer over a MOM that allows the components to deal with messages in a type-specific way. The low-level serialisation and de-serialisation processes are encapsulated and hidden from the user code. As a result, it is potentially possible to exchange one MOM against another one without any changes in the user code.

The MOM currently used in the SEMAINE API is ActiveMQ from the Apache project [32]. ActiveMQ is an open-source implementation of the Java Message Service (JMS) server

specification [33]. It provides client code in Java, C++ and various other programming languages, is reasonably fast, and is actively used, supported and developed at the time of this writing.

3.1.2. Topics

In its “publish-subscribe” model, JMS routes messages via so-called *Topics* which can be identified by name. The SEMAINE API adopts this terminology. Names of Topics can be arbitrarily chosen. In order to establish a communication link from component A to component B, it is sufficient for component A to register as a “publisher” to a named Topic, and for component B to register as a “subscriber” to the same Topic. Whenever A sends a message to the Topic, B will receive the message. Topics allow for an arbitrary number of publishers and subscribers, so that it is trivial to set up n-to-m communication structures.

For a given system, it is reasonable to choose Topics such that they represent data of a specific type, i.e. with a well-defined meaning and representation format. This type of data may be produced by several system components, such as a range of modality-specific emotion classifiers. If there are no compelling reasons why their outputs need to be treated differently, it is possible to use a single Topic for their joint output, by registering all the components producing this data type as publishers to the Topic. Similarly, several components may reasonably take a given type of data as input, in which case all of them should register as subscribers to the respective Topic. Using Topics as “information hubs” in this way immensely simplifies the clarity of information flow, and consequently the message flow graph (see Figure 2 for an example).

3.1.3. Components

The creation of custom components is made simple by the base class `Component` in the SEMAINE API, which provides the basic functionality required for the component to interact with the rest of the system. The `Component` will register as a subscriber and/or as a publisher to a configurable list of Topics using suitable, possibly type-specific message receivers and message senders. Whenever a message is received, the subclass's `react()` method is called, allowing the component to process the data and perform some action, including the emission of an output message. In addition, the method `act()` is called at configurable intervals (every 100 ms by default), allowing for actions to be triggered by component-internal mechanisms such as timeouts or custom process logic.

The `Component` base implementation also instantiates the meta messenger (see Figure 1) which handles all meta communication with the system manager, without requiring customisation by user code in subclasses.

Examples of simple component classes are provided in Section 5.

3.1.4. API support for relevant representation types

The SEMAINE API aims to be as easy to use as possible, while allowing for state of the art processing of data. This principle is reflected in an extensive set of support classes and methods for parsing, interpreting and creating XML documents in general and the representations specially supported (see Section 3.2) in particular. XML processing is performed by the standards-compliant parser Xerces [34] which converts between a textual representation contained in messages and a user-friendly Document Object Model (DOM) representation [35]. Parsing is done in a namespace-aware manner in order to maintain a clear distinction between the elements used in mixed representations. Examples of mixed representations are the use of the Extensible Multimodal Annotation language EMMA to transport a recognition result expressed in EmotionML, or the use of the Speech Synthesis Markup Language SSML to encode the speech aspect of ECA behaviour in the Behavior Markup Language sBML. These combinations make perfect sense; namespaces are a

suitable method for clearly identifying the markup type of any given element when interpreting a mixed representation.

Support classes exist for the representation formats listed in Section 3.2, notably as dedicated receiver, sender and message objects. For example, when a component registers an `EmmaReceiver` to listen to a given `Topic`, it will receive messages directly as `SEMAINEEmmaMessage` objects with methods appropriate for interpreting the content of the EMMA data; a `FeatureSender` will take an array of `float` values and send it as a textual or binary feature message; `BinarySender` and `BinaryReceiver` classes can be used to transport, e.g., audio data between components.

In sum, these support classes and methods simplify the task of passing messages via the middleware, and help avoid errors in the process of message-passing by implementing standard encoding and decoding procedures. Where representations beyond those previewed by the API are required, the user always has the option to use lower-level methods such as plain-XML or even text messages and implement a custom encoding and decoding mechanism.

3.1.5. Supported platforms

The SEMAINE API is currently available in Java and as a shared library in C++, for Linux, Mac OS X and Windows. State of the art build tools (Eclipse and ant for Java, Visual Studio for C++ on Windows, GNU automake/ autoconf for C++ on Linux and Mac) are provided to make the use of the API as simple and portable as possible.

3.1.6. Current status

As of version 1.0.1, the SEMAINE API is fully functional, but the support for the individual representation formats is preliminary. Not all elements and attributes defined in the specifications mentioned in Section 3.2 are pre-defined as constants in the API support classes. This limitation is an issue of coverage rather than principle: on the one hand, it is straightforward to add the missing element and attribute names to the lists of string constants; on the other hand user code can always add custom string constants or use ad hoc strings to create or read XML elements and attributes for which no constants have been defined yet.

Other aspects are more interesting because the practical implementation has hit limits with the current version of draft specifications. For example, for the implementation of symbolic timing markers between words, the draft BML specification [36] proposes to use a `<mark>` element in the default namespace; however, we noticed that treating the speech markup as valid SSML requires the use of an `<ssml:mark>` element in the SSML namespace. Experience of this kind may be helpful in refining specifications based on implementation feedback.

3.2. Representation formats supported in the SEMAINE API

In view of future interoperability and reuse of components, the SEMAINE API aims to use standard representation formats where that seems possible and reasonable. For example, results of analysis components can be represented using EMMA (Extensible Multi-Modal Annotation), a World Wide Web Consortium (W3C) Recommendation [37]. Input to a speech synthesiser can be represented using SSML (Speech Synthesis Markup Language), also a W3C Recommendation [38].

Several other relevant representation formats are not yet standardised, but are in the process of being specified. This includes the Emotion Markup Language EmotionML [39], used for representing emotions and related states in a broad range of contexts, and the Behaviour Markup Language BML [40], which describes the behaviour to be shown by an Embodied Conversational Agent (ECA). Furthermore, a Functional Markup Language FML [41] is under discussion, in order to represent the planned actions of an ECA on the level of functions and meanings. By

implementing draft versions of these specifications, the SEMAINE API can provide hands-on input to the standardisation process, which may contribute to better standard formats.

On the other hand, it seems difficult to define a standard format for representing the concepts inherent in a given application's logic. To be generic, such an endeavour would ultimately require an ontology of the world. In the current SEMAINE system, which does not aim at any sophisticated reasoning over domain knowledge, a simple custom format named SemaineML is used to represent those pieces of information that are required in the system but which cannot be adequately represented in an existing or emerging standard format. It is conceivable that other applications built on top of the SEMAINE API may want to use a more sophisticated representation such as the Rich Description Format RDF [42] to represent domain knowledge, in which case the API could be extended accordingly.

Whereas all of the aforementioned representation formats are based on the Extensible Markup Language XML [43], there are a number of data types that are naturally represented in different formats. This is particularly the case for the representations of data close to input and output components. At the input end, low-level analyses of human behaviour are often represented as feature vectors. At the output end, the input to a player component is likely to include binary audio data or player-specific rendering directives.

Table 2 gives an overview of the representation formats currently supported in the SEMAINE API. The following sub-sections briefly describe the individual representation formats.

Type of data	Representation format	Standardisation status
Low-level input features	string or binary feature vectors	ad hoc
Analysis results	EMMA	W3C Recommendation
Emotions and related states	EmotionML	W3C Incubator Report
Domain knowledge	SemaineML	ad hoc
Speech synthesis input	SSML	W3C Recommendation
Functional action plan	FML	very preliminary
Behavioural action plan	BML	draft specification
Low-level output data	binary audio, player commands	player-dependent

Table 2: Representation formats currently supported by the SEMAINE API.

3.2.1. Feature vectors

Feature vectors can be represented in an ad hoc format. In text form (see Figure 3), the feature vectors consist of straightforward key-value pairs – one feature per line, values preceding features.

```
0.000860535 rmsEnergy
12.6699 logEnergy
-2.59005e-05 rmsEnergy-De
-0.0809427 logEnergy-De
...
```

Figure 3: Textual representation of a feature vector.

As feature vectors may be sent very frequently (e.g., every 10 ms in the SEMAINE system 1.0), compact representation is a relevant issue. For this reason, a binary representation of feature vectors is also available. In binary form, the feature names are omitted, and only feature values are being communicated. The first four bytes represent an integer containing the number of features in the vector; the remaining bytes contain the float values one after the other.

3.2.2. EMMA

The Extensible Multimodal Annotation Language EMMA, a W3C Recommendation, is “an XML markup language for containing and annotating the interpretation of user input” [37]. As such, it is a wrapper language that can carry various kinds of payload representing the interpretation of user input. The EMMA language itself provides, as its core, the `<emma:interpretation>` element, containing all information about a single interpretation of user behaviour. Several such elements can be enclosed within an `<emma:one-of>` element in cases where more than one interpretation is present. An interpretation can have an `emma:confidence` attribute, indicating how confident the source of the annotation is that the interpretation is correct; time-related information such as `emma:start`, `emma:end`, and `emma:duration`, indicating the time span for which the interpretation is provided; information about the modality upon which the interpretation is based, through the `emma:medium` and `emma:mode` attributes; and many more.

```
<emma:emma xmlns:emma="http://www.w3.org/2003/04/emma" version="1.0">
  <emma:interpretation emma:start="123456789">
    <emotion xmlns="http://www.w3.org/2005/Incubator/emotion">
      <dimensions set="valenceArousalPotency">
        <arousal value="-0.29"/>
        <valence value="-0.22"/>
      </dimensions>
    </emotion>
  </emma:interpretation>
</emma:emma>
```

Figure 4: An example EMMA document carrying EmotionML markup as interpretation payload.

Figure 4 shows an example EMMA document carrying an interpretation of user behaviour represented using EmotionML (see below). The interpretation refers to a start time. It can be seen that the EMMA wrapper elements and the EmotionML content are in different XML namespaces, so that it is unambiguously determined which element belongs to which part of the annotation.

EMMA can also be used to represent Automatic Speech Recognition (ASR) output, either as the single most probable word chain or as a word lattice, using the `<emma:lattice>` element.

3.2.3. EmotionML

The Emotion Markup Language EmotionML is partially specified, at the time of this writing, by the Final Report of the W3C Emotion Markup Language Incubator Group [39]. The report provides elements of a specification, but leaves a number of issues open. The language is now being developed towards a formal W3C Recommendation.

The SEMAINE API is one of the first pieces of software to implement EmotionML. It is our intention to provide an implementation report as input to the W3C standardisation process in due course, highlighting any problems encountered with the current draft specification in the implementation.

EmotionML aims to make concepts from major emotion theories available in a broad range of technological contexts. Being informed by the affective sciences, EmotionML recognises the fact that there is no single agreed representation of affective states, nor of vocabularies to use. Therefore, an emotional state `<emotion>` can be characterised using four types of descriptions: `<category>`, `<dimensions>`, `<appraisals>` and `<action-tendencies>`. Furthermore, the vocabulary used can be identified. The EmotionML markup in Figure 4 uses a dimensional representation of emotions, using the dimension set “valence, arousal, potency”, out of which two dimensions are annotated: arousal and valence.

EmotionML is aimed at three use cases: 1. Human annotation of emotion-related data; 2. automatic emotion recognition; and 3. generation of emotional system behaviour. In order to be suitable for all three domains, EmotionML is conceived as a “plug-in” language that can be used in different

contexts. In the SEMAINE API, this plug-in nature is applied with respect to recognition, centrally held information, and generation, where EmotionML is used in conjunction with different markups. EmotionML can be used for representing the user emotion currently estimated from user behaviour, as payload to an EMMA message. It is also suitable for representing the centrally held information about the user state, the system's "current best guess" of the user state independently of the analysis of current behaviour. Furthermore, the emotion to be expressed by the system can also be represented by EmotionML. In this case, it is necessary to combine EmotionML with the output languages FML, BML and SSML.

3.2.4. SemaineML

A number of custom representations are needed to represent the kinds of information that play a role in the SEMAINE demonstrator systems. Currently, this includes the centrally held beliefs about the user state, the agent state, and the dialogue state. Most of the information represented here is domain-specific and does not lend itself to easy generalisation or reuse. Figure 5 shows an example of a dialogue state representation, focused on the specific situation of an agent-user dialogue targeted in the SEMAINE system 1.0 (see Section 4).

```
<dialog-state xmlns="http://www.semaine-project.eu/semaineml" version="0.0.1">
  <speaker who="agent"/>
  <listener who="user"/>
</dialog-state>
```

Figure 5: An example SemaineML document representing dialogue state.

The exact list of phenomena that must be encoded in the custom SemaineML representation is evolving as the system becomes more mature. For example, it remains to be seen whether analysis results in terms of user behaviour (such as a smile) can be represented in BML or whether they need to be represented using custom markup.

3.2.5. SSML

The Speech Synthesis Markup Language SSML [38] is a well-established W3C Recommendation supported by a range of commercial text-to-speech (TTS) systems. It is the most established of the representation formats described in this section.

The main purpose of SSML is to provide information to a TTS system on how to speak a given text. This includes the possibility to add `<emphasis>` on certain words, to provide pronunciation hints via a `<say-as>` tag, to select a `<voice>` which is to be used for speaking the text, or to request a `<break>` at a certain point in the text. Furthermore, SSML provides the possibility to set markers via the SSML `<mark>` tag. Figure 6 shows an example SSML document that could be used as input to a TTS engine. It requests a female US English voice; the word "wanted" should be emphasised, and there should be a pause after "then".

```
<speak version="1.0" xmlns="http://www.w3.org/2001/10/synthesis"
  xml:lang="en-US">
  <voice gender="female">
    And then <break/> I <emphasis>wanted</emphasis> to go.
  </voice>
</speak>
```

Figure 6: An example standalone SSML document.

3.2.6. FML

The functional markup language FML is still under discussion [41]. Its functionality being needed nevertheless, a working language FML-APML was created [44] as a combination of the ideas of FML with the former Affective Presentation Markup Language APML [45].

```

<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."
      ssml:xmlns="http://www.w3.org/2001/10/synthesis">
      <ssml:mark name="s1:tm1"/>
      Hi,
      <ssml:mark name="s1:tm2"/>
      I'm
      <ssml:mark name="s1:tm3"/>
      Poppy.
      <ssml:mark name="s1:tm4"/>
    </speech>
  </bml>
  <fml xmlns="http://www.mindmakers.org/fml" id="fml1">
    <performative id="p2" type="announce" start="s1:tm1" end="s1:tm4"/>
    <world id="w1" ref_type="person" ref_id="self" start="s1:tm2" end="s1:tm4"/>
  </fml>
</fml-apml>

```

Figure 7: An example FML-APML document.

Figure 7 shows an example FML-APML document which contains the key elements. An `<fml-apml>` document contains a `<bml>` section in which the `<speech>` content contains `<ssml:mark>` markers identifying points in time in a symbolic way. An `<fml>` section then refers to those points in time to represent the fact, in this case, that an announcement is made and that the speaker herself is being referred to between marks `s1:tm2` and `s1:tm4`. This information can be used, for example, to generate relevant gestures when producing behaviour from the functional descriptions.

The representations in the `<fml>` section are provisional and are likely to change as consensus is formed in the community.

For the conversion from FML to BML, information about pitch accents and boundaries is useful for the prediction of plausible behaviour time-aligned with the macro-structure of speech. In our current implementation, a speech preprocessor computes this information using TTS technology (see Section 4.2). The information is added to the end of the `<speech>` section as shown in Figure 8. This is an ad hoc solution which should be reconsidered in the process of specifying FML.

```

<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."
      ssml:xmlns="http://www.w3.org/2001/10/synthesis">
      <ssml:mark name="s1:tm1"/>
      Hi,
      <ssml:mark name="s1:tm2"/>
      I'm
      <ssml:mark name="s1:tm3"/>
      Poppy.
      <ssml:mark name="s1:tm4"/>
      <b><pitchaccent id="xpa1" start="s1:tm1" end="s1:tm2"/>
      <b><pitchaccent id="xpa2" start="s1:tm3" end="s1:tm4"/>
      <b><boundary id="b1" time="s1:tm4"/>
    </speech>
  </bml>
  <fml xmlns="http://www.mindmakers.org/fml" id="fml1">
    <performative id="p2" type="announce" start="s1:tm1" end="s1:tm4"/>
    <world id="w1" ref_type="person" ref_id="self" start="s1:tm2" end="s1:tm4"/>
  </fml>
</fml-apml>

```

Figure 8: Pitch accent and boundary information added to the FML-APML document of Figure 7.

3.2.7. BML

The aim of the Behaviour Markup Language BML [40] is to represent the behaviour to be realised by an Embodied Conversational Agent. BML is at a relatively concrete level of specification, but is still in draft status [36].

A standalone BML document is partly similar to the <bml> section of an FML-APML document (see Figure 7); however, whereas the <bml> section of FML-APML contains only a <speech> tag, a BML document can contain elements representing expressive behaviour in the ECA at a broad range of levels, including <head>, <face>, <gaze>, <body>, <speech> and others. Figure 9 shows an example of gaze and head nod behaviour added to the example of Figure 7.

```
<bml xmlns="http://www.mindmakers.org/projects/BML" id="bml1">
  <speech id="s1" language="en_US" text="Hi, I'm Poppy."
    ssm1:xmlns="http://www.w3.org/2001/10/synthesis">
    <ssml:mark name="s1:tm1"/>
    Hi,
    <ssml:mark name="s1:tm2"/>
    I'm
    <ssml:mark name="s1:tm3"/>
    Poppy.
    <ssml:mark name="s1:tm4"/>
    <pitchaccent id="xpa1" start="s1:tm1" end="s1:tm2"/>
    <pitchaccent id="xpa2" start="s1:tm3" end="s1:tm4"/>
    <boundary id="b1" time="s1:tm4"/>
  </speech>
  <gaze id="g1" start="s1:tm1" end="s1:tm4">
    ...
  </gaze>
  <head id="h1" start="s1:tm3" end="s1:tm4" type="NOD">
    ...
  </head>
</bml>
```

Figure 9: An example BML document containing SSML and gestural markup.

While creating an audio-visual rendition of the BML document, we use TTS to produce the audio and the timing information needed for lip synchronisation. Whereas BML in principle previews a <lip> element for representing this information, we are uncertain how to represent exact timing information with it in a way that preserves the information about syllable structure and stressed syllables. For this reason, we currently use a custom representation based on the MaryXML format from the MARY TTS system [46] to represent the exact timing of speech sounds. Figure 10 shows the timing information for the word “Poppy”, which is a two-syllable word of which the first one is the stressed syllable.

```

<bml xmlns="http://www.mindmakers.org/projects/BML" id="bml1">
  <speech id="s1" language="en_US" text="Hi, I'm Poppy."
    ssm1:xmlns="http://www.w3.org/2001/10/synthesis"
    mary:xmlns="http://mary.dfki.de/2002/MaryXML">
    ...
    <ssml:mark name="s1:tm3"/>
    Poppy.
    <mary:syllable stress="1">
      <mary:ph d="0.092" end="1.011" p="p"/>
      <mary:ph d="0.112" end="1.123" p="A"/>
      <mary:ph d="0.093" end="1.216" p="p"/>
    </mary:syllable>
    <mary:syllable>
      <mary:ph d="0.141" end="1.357" p="i"/>
    </mary:syllable>
    <ssml:mark name="s1:tm4"/>
    ...
  </speech>
</bml>

```

Figure 10: An excerpt of a BML document enriched with TTS timing information for lip synchronisation.

The custom format we use for representing timing information for lip synchronisation clearly deserves to be revised towards a general BML syntax, as BML evolves.

3.2.8. Player data

Player data is currently treated as unparsed data. Audio data is binary, whereas player directives are considered to be plain text. This works well with the current MPEG-4 player we use (see Section 4) but may need to be generalised as other players are integrated into the system.

4. The SEMAINE system 1.0

The first system built with the SEMAINE API is the SEMAINE system 1.0, created by the SEMAINE project. It is an early-integration system which does not yet represent the intended application domain of SEMAINE, the Sensitive Artificial Listeners [30], but achieves a first integrated system based on existing components from the project partners.

The present section describes the system 1.0, first from the perspective of system architecture and integration, then with respect to the components which at the same time are available as building blocks for creating new systems with limited effort (see Section 5 for examples).

4.1. Conceptual system architecture

Figure 24 shows a message flow graph representing the conceptual system architecture of the intended SEMAINE system [30], which is partially instantiated by the SEMAINE system 1.0 [47]. Processing components are represented as ovals, data 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.

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

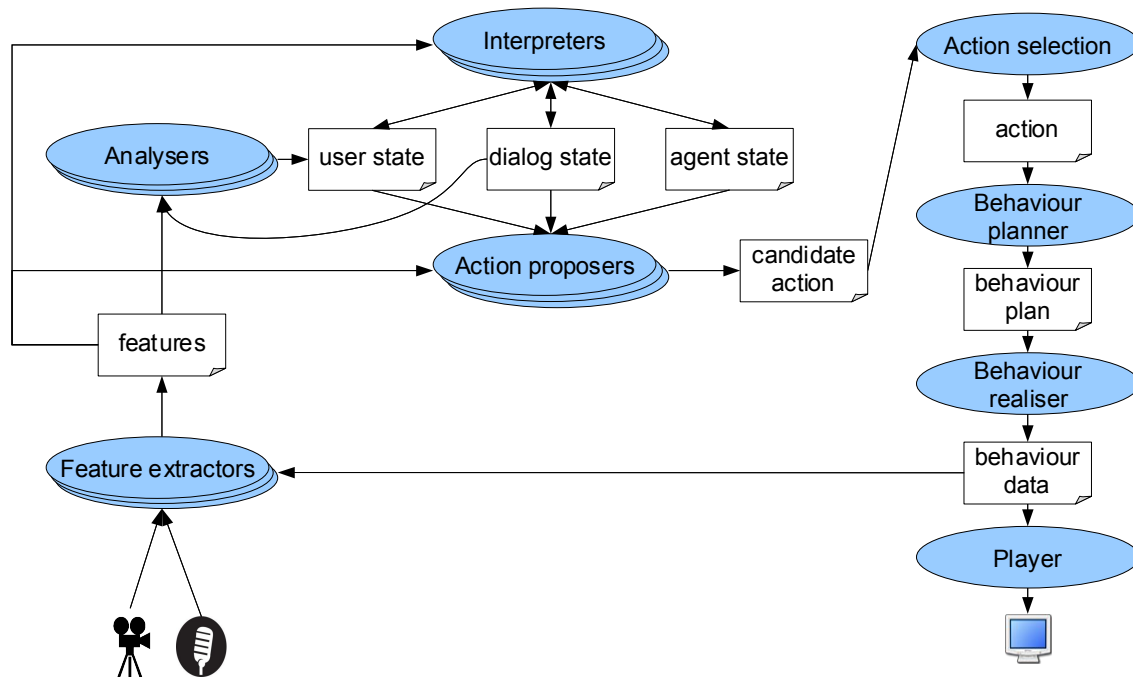


Figure 11: Conceptual message flow graph of the SEMAINE system.

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 of 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 [48], 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.

4.2. Components in the SEMAINE system 1.0

The actual implementation of this conceptual architecture is visualised in the system monitor screenshot in Figure 2. The following paragraphs briefly describe the individual system components. A more detailed description can be found in [47].

Low-level audio features are extracted using the openSMILE (Speech and Music Interpretation by Large-Space Extraction) feature extractor [49]. The SMILE automatic emotion recognition component [50] performs continuous emotion detection in terms of the emotion dimensions arousal and valence. The current models have been trained on preliminary Sensitive Artificial Listener data

from the HUMAINE project [51]. The Automatic Speech Recognition component 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/>). The speaker independent tri-phone models were trained on the Wall Street Journal corpus [52] and on the AMIDA Meeting corpus [53]. The tri-gram language model was trained on the preliminary SAL corpus [51] and therefore includes about 1900 words which typically occur in spontaneous emotionally coloured speech. The module for recognition of human interest was trained on the AVIC database [54], a database which 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 module discriminates three different levels of interest: “bored”, “neutral”, and “high interest”.

Turn taking is a complex conversational system in which the participants negotiate who will be the main speaker in the next period. The SEMAINE system 1.0 implements a simplistic mechanism: When the user is silent for more than 2 seconds, the system decides that the agent has the turn.

When the agent receives the turn, the system will analyse what the user did and said. The User utterance interpreter will look at the utterances of the user that were detected in the previous turn. The utterances are tagged with general semantic features such as the semantic polarity, the time, and the subject of the utterances, and the user state is updated accordingly.

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 Sensitive Artificial Listener script [55] and contains fitting contexts (i.e. a list of semantic features) for every possible response. When no information about the previous user utterance is available, it will pick a response from a list of generic responses which fit in almost all circumstances.

A backchannel proposer (not shown in Figure 2) can be used to generate behaviour such as the nods and “uh-huh” sounds produced by listeners during the other's turn. The current simplistic implementation triggers a non-specific backchannel after 300 ms of silence by the user.

In the case of conflicting action proposals, an action selection component is needed to decide which actions to realise. The implementation in the SEMAINE system 1.0 is a dummy implementation which simply accepts all proposed actions.

The speech preprocessing component is part of the conceptual behaviour planner component. It uses the MARY TTS system [46] to add pitch accent and phrase boundary information to FML documents in preparation of the realisation of functions in terms of visual behaviour.

Conceptually, the visual behaviour planner component identifies behaviour patterns that are appropriate for realising the functions contained in an FML document. At this stage, the component is a dummy implementation only which does nothing.

The speech synthesis component is part of the conceptual behaviour realiser component. It uses the MARY TTS system [46] to produce synthetic speech audio data as well as timing information in an extended BML document suitable as input to the visual behaviour realiser. As a proof of concept, the current version of the speech synthesiser also generates vocal backchannels upon request.

The visual behaviour realiser component generates the animation for the Greta agent [9] as MPEG-4 Facial Animation Parameters (FAP) [56] and Body Animation Parameters (BAP). 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. The list of phonemes and their respective duration, provided by the speech synthesis component, is used to compute the lips movements.

When the Behaviour Realiser 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 the player. It avoids unnatural “freezing” of the agent.

The Greta player [9] receives the animation generated by the behaviour realiser and plays it in a graphic window. The animation is defined by the Facial Animation Parameters (FAPs) and the Body Animation Parameters (BAPs). Each FAP or BAP frame received by the player carries also the time intended for its visualisation as computed by the behaviour realiser.

4.3. System properties

The combination of the system components described above enables a simple kind of dialogue interaction. While the user is speaking, audio features are extracted. When silence is detected, estimates of the user's emotion and interest during the turn are computed, and the ASR produces an estimate of the words spoken. When the silence duration exceeds a threshold, backchannels are triggered (if the system is configured to include the backchannel proposer component); after a longer silence, the agent takes the turn and proposes a verbal utterance from the SAL script [55]. Even where no meaningful analysis of the user input can be made, the script will propose a generic utterance such as “Why?” or “Go on.” which is suitable in many contexts. The utterances are realised with a generic TTS voice and rendered, either using the audiovisual Greta player or an audio-only player.

This description shows that the system is not yet capable of much meaningful interaction, since its perceptual components are limited and the dialogue model is not fully fleshed out yet. Nevertheless, the main types of components needed for an emotion-aware interactive system are present, including emotion analysis from user input, central processing, and multimodal output. This makes the system suitable for experimenting with emotion-aware systems in various configurations, as the following section will illustrate.

5. Building emotion-oriented systems with the SEMAINE API

This section presents three emotion-oriented example systems, in order to corroborate the claim that the SEMAINE API is easy to use for building new emotion-oriented systems out of new and/or existing components. Source code is provided in order to allow the reader to follow in detail the steps needed for using the SEMAINE API. The code is written in Java, and can be obtained from the SEMAINE sourceforge page [57]. The SEMAINE API parts of the code would look very similar in C++.

5.1. Hello world

The “Hello” example realises a simple text-based interactive system. The user types arbitrary text; an analyser component spots keywords, and deduces an affective state from them; and a rendering component outputs an emoticon corresponding to this text. Despite its simplicity, the example is instructive because it displays the main elements of an emotion-oriented system.

The input component (Figure 12) simply reads one line of text at a time, and sends it on. It has an input device (Figure 12, line 4) and a Sender writing TEXT data to the Topic `semaine.data.hello.text` (line 3). In its constructor, the component registers itself as an input component (l. 7), and registers its sender (l. 8). Its `act()` method, which is automatically called every 100 ms while the system is running, checks for new input (l. 12), reads it (l. 13), and sends it to the Topic (l. 14).

```

1 public class HelloInput extends Component {
2
3     private Sender textSender = new Sender("semaine.data.hello.text", "TEXT", getName());
4     private BufferedReader inputReader = new BufferedReader(new InputStreamReader(System.in));
5
6     public HelloInput() throws JMSEException {
7         super("HelloInput", true/*is input*/, false);
8         senders.add(textSender);
9     }
10
11     @Override protected void act() throws IOException, JMSEException {
12         if (inputReader.ready()) {
13             String line = inputReader.readLine();
14             textSender.sendTextMessage(line, meta.getTime());
15         }
16     }
17 }

```

Figure 12: The HelloInput component sending text messages via the SEMAINE API.

As a simplistic central processing component, the HelloAnalyser (Figure 13) makes emotional judgements about the input. It registers a Receiver (l. 7) for the Topic that HelloInput writes to, and sets up (l. 3) and registers (l. 8) an XML Sender producing data of type EmotionML. Whenever a message is received, the method `react()` is called (l. 11). It receives (l. 13) and analyses (l. 14-17) the input text, and computes values for the emotion dimensions arousal and valence from the text. Finally, it creates an EmotionML document (l. 18) and sends it (l. 19).

```

1 public class HelloAnalyser extends Component {
2
3     private XMLSender emotionSender =
4         new XMLSender("semaine.data.hello.emotion", "EmotionML", getName());
5
6     public HelloAnalyser() throws JMSEException {
7         receivers.add(new Receiver("semaine.data.hello.text"));
8         senders.add(emotionSender);
9     }
10
11     @Override protected void react(SEMAINEMessage m) throws JMSEException {
12         int arousalValue = 0, valenceValue = 0;
13         String input = m.getText();
14         if (input.contains("very")) arousalValue = 1;
15         else if (input.contains("a bit")) arousalValue = -1;
16         if (input.contains("happy")) valenceValue = 1;
17         else if (input.contains("sad")) valenceValue = -1;
18         Document emotionML = createEmotionML(arousalValue, valenceValue);
19         emotionSender.sendXML(emotionML, meta.getTime());
20     }
21
22     private Document createEmotionML(int arousalValue, int valenceValue) {
23         Document emotionML = XMLTool.newDocument(EmotionML.ROOT_ELEMENT, EmotionML.namespaceURI);
24         Element emotion = XMLTool.appendChildElement(emotionML.getDocumentElement(),
25             EmotionML.E_EMOTION);
26         Element dimensions = XMLTool.appendChildElement(emotion, EmotionML.E_DIMENSIONS);
27         dimensions.setAttribute(EmotionML.A_SET, "arousalValence");
28         Element arousal = XMLTool.appendChildElement(dimensions, EmotionML.E_AROUSAL);
29         arousal.setAttribute(EmotionML.A_VALUE, String.valueOf(arousalValue));
30         Element valence = XMLTool.appendChildElement(dimensions, EmotionML.E_VALENCE);
31         valence.setAttribute(EmotionML.A_VALUE, String.valueOf(valenceValue));
32         return emotionML;
33     }
34 }

```

Figure 13: The HelloAnalyser component. It receives and analyses the text messages from HelloInput, and generates and sends an EmotionML document containing the analysis results.

As the SEMAINE API does not yet provide built-in support for standalone EmotionML documents, the component uses a generic `XMLSender` (l. 3) and uses the `XMLTool` to build up the EmotionML document (l. 23-30).

		Valence		
		-	0	+
Arousal	+	8-(8-	8-)
	0	:-(:-	:-)
	-	*-(*-	*-)

Table 3: Ad hoc emoticons used to represent positions in the arousal-valence plane.

The output of the Hello system should be an emoticon representing an area in the arousal-valence plane as shown in Table 3. The `EmoticonOutput` component (Figure 14) registers an XML Receiver (l. 5) to the Topic that the `HelloAnalyser` sends to. Whenever a message is received, the `react()` method is called (l. 8), which analyses the XML document in terms of EmotionML markup (l. 10-12), and extracts the arousal and valence values (l. 14-15). The emotion display is rendered as a function of these values (l. 17-19).

```

1 public class EmoticonOutput extends Component {
2
3 public EmoticonOutput() throws JMSEException {
4     super("EmoticonOutput", false, true /*is output*/);
5     receivers.add(new XMLReceiver("semaine.data.hello.emotion"));
6 }
7
8 @Override protected void react(SEMAINEMessage m) throws MessageFormatException {
9     SEMAINEXMLMessage xm = (SEMAINEXMLMessage) m;
10    Element dimensions = (Element) xm.getDocument().getElementsByTagNameNS(
11        EmotionML.namespaceURI, EmotionML.E_DIMENSIONS).item(0);
12    Element arousal = XMLTool.needChildElementByTagNameNS(dimensions, EmotionML.E_AROUSAL,
13        EmotionML.namespaceURI);
14    Element valence = XMLTool.needChildElementByTagNameNS(dimensions, EmotionML.E_VALENCE,
15        EmotionML.namespaceURI);
16
17    float a = Float.parseFloat(arousal.getAttribute(EmotionML.A_VALUE));
18    float v = Float.parseFloat(valence.getAttribute(EmotionML.A_VALUE));
19
20    String eyes = a > 0.3 ? "8"/*active*/ : a < -0.3 ? "*"/*passive*/ : ":"/*neutral*/;
21    String mouth = v > 0.3 ? ")"/*positive*/ : v < -0.3 ? "("/*negative*/ : "|"/*neutral*/;
22    System.out.println(eyes+"-"+mouth);
23 }
24 }
```

Figure 14: The `EmoticonOutput` component. It receives EmotionML markup and displays an emoticon according to Table 3.

In order to build a system from the components, a configuration file is created (Figure 15). It includes the `SystemManager` component as well as the three newly created components. Furthermore, it requests a visible system manager GUI providing a message flow graph.

```

semaine.components = \
    |eu.semaine.components.meta.SystemManager| \
    |eu.semaine.examples.hello.HelloInput| \
    |eu.semaine.examples.hello.HelloAnalyser| \
    |eu.semaine.examples.hello.EmoticonOutput|

semaine.systemmanager.gui = true
```

Figure 15: The configuration file `example-hello.config` defining the Hello application.

The system is started in the same way as all Java-based SEMAINE API systems: `activemq; java eu.semaine.system.ComponentRunner example-hello.config`. Figure 16 shows a screenshot of the resulting message flow graph. As the communication passes via the middleware ActiveMQ, the system would behave in the exact same way if the four components were started as separate processes, on different machines, or if some of them were written in C++ rather than Java.

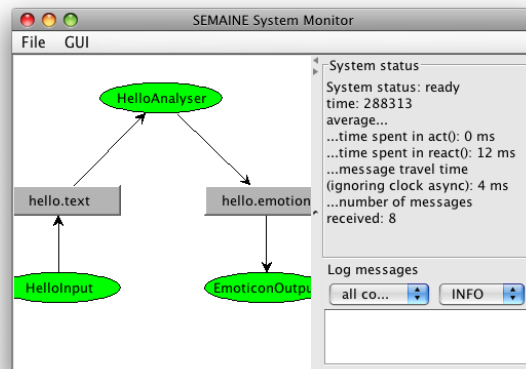


Figure 16: Message flow graph of the Hello system

5.2. Emotion mirror

The Emotion mirror is a variant of the Hello system. Instead of analysing text and deducing emotions from keywords, it uses the openSMILE speech feature extraction and emotion detection (see Section 4.2) for interpreting the user's emotion. The output is rendered using the same EmoticonOutput component from the Hello system in Section 5.1.

```

1 public class EmotionExtractor extends Component {
2     private XMLSender emotionSender =
3         new XMLSender("semaine.data.hello.emotion", "EmotionML", getName());
4
5     public EmotionExtractor() throws JMSEException {
6         super("EmotionExtractor");
7         receivers.add(new EmmaReceiver("semaine.data.state.user.emma"));
8         senders.add(emotionSender);
9     }
10
11    @Override protected void react(SEMAINEMessage m) throws JMSEException {
12        SEMAINEEmmaMessage emmaMessage = (SEMAINEEmmaMessage) m;
13        Element interpretation = emmaMessage.getTopLevelInterpretation();
14        List<Element> emotionElements = emmaMessage.getEmotionElements(interpretation);
15        if (emotionElements.size() > 0) {
16            Element emotion = emotionElements.get(0);
17            Document emotionML = XMLTool.newDocument(EmotionML.ROOT_ELEMENT, EmotionML.namespaceURI);
18            emotionML.adoptNode(emotion);
19            emotionML.getDocumentElement().appendChild(emotion);
20            emotionSender.sendXML(emotionML, meta.getTime());
21        }
22    }

```

Figure 17: The EmotionExtractor component takes EmotionML markup from an EMMA message and forwards it.

Only one new component is needed to build this system. EmotionExtractor (Figure 17) has an emotion Sender (l. 2 and l. 7) just like the HelloAnalyser had, but uses an EMMA Receiver (l. 6) to read from the topic that the Emotion detection component from the SEMAINE system (see Section 4.2) publishes to, as documented in [47]. Upon reception of an EMMA message, the method `react()` is called (l. 10). As the only receiver registered by the component is an EMMA receiver, the message can be directly cast into an EMMA message (l. 11) which allows for comfortable

access to the document structure to extract emotion markup (l. 12-13). Where emotion markup is present, it is inserted into a standalone EmotionML document (l. 16-18) and sent to the output Topic (l. 19).

The config file contains only the components SystemManager, EmotionExtractor and EmoticonOutput. As the SMILE component is written in C++, it needs to be started as a separate process as documented in the SEMAINE wiki documentation [58]. The resulting message flow graph is shown in Figure 18.

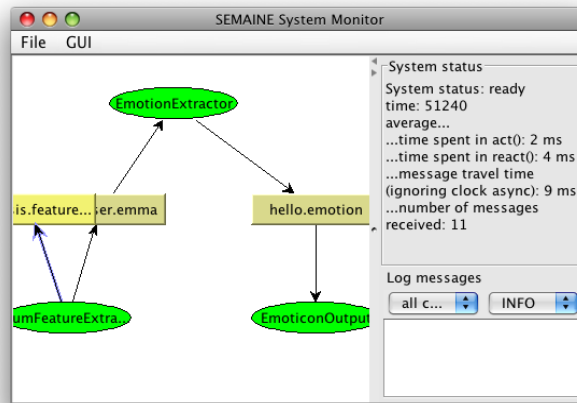


Figure 18: Message flow graph of the Emotion mirror system.

5.3. A game driven by emotional speech: The swimmer's game

The third example system is a simple game application in which the user must use emotional speech to win the game. The game scenario is as follows. A swimmer is being pulled backwards by the stream towards a waterfall (Figure 19). The user can help the swimmer to move forward towards the river bank by cheering him up through high-arousal speech. Low arousal, on the other hand, discourages the swimmer and drives him more quickly to the waterfall.

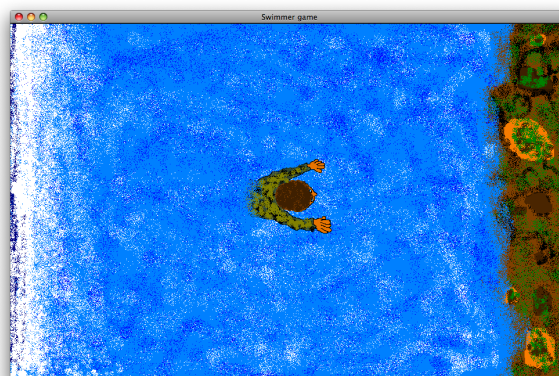


Figure 19: Swimmer's game user interface.

The system requires the openSMILE components as in the Emotion mirror system; a component computing the swimmer's position as time passes, and considering the user's input; and a rendering component for the user interface. Furthermore, we will illustrate the use of TTS output in the SEMAINE API by implementing a commentator providing input to the speech synthesis component of the SEMAINE system 1.0 (Section 4.2).

```

1 public class PositionComputer extends Component {
2     private Sender positionSender =
3         new Sender("semaine.data.swimmer.position", "TEXT", getName());
4
5     public PositionComputer() throws JMSEException {
6         super("PositionComputer");
7         receivers.add(new EmmaReceiver("semaine.data.state.user.emma"));
8         senders.add(positionSender);
9     }
10
11     @Override protected void react(SEMAINEMessage m) throws MessageFormatException {
12         SEMAINEEmmaMessage emmaMessage = (SEMAINEEmmaMessage) m;
13         Element interpretation = emmaMessage.getTopLevelInterpretation();
14         List<Element> emotionElements = emmaMessage.getEmotionElements(interpretation);
15
16         for (Element emotion : emotionElements) {
17             Element dimensions = XMLTool.getChildElementByTagNameNS(emotion, EmotionML.E_DIMENSIONS,
18                 EmotionML.namespaceURI);
19             if (dimensions != null) {
20                 Element arousal = XMLTool.getChildElementByTagNameNS(dimensions, EmotionML.E_AROUSAL,
21                     EmotionML.namespaceURI);
22                 float arousalValue = Float.parseFloat(arousal.getAttribute(EmotionML.A_VALUE));
23                 // Arousal influences the swimmer's position:
24                 position += 10*arousalValue;
25             }
26         }
27
28         @Override protected void act() throws JMSEException {
29             // The river slowly pulls back the swimmer:
30             position -= 0.1;
31             positionSender.sendMessage(String.valueOf(position), meta.getTime());
32     }

```

Figure 20: The PositionComputer component.

The PositionComputer (Figure 20) combines a `react()` and an `act()` method. Messages are received via an EMMA receiver and lead to a change in the internal parameter `position` (l. 22). The `act()` method implements the backward drift (l. 29) and sends regular position updates (l. 30) as a plain-text message.

```

1 public class SwimmerDisplay extends Component {
2
3     public SwimmerDisplay() throws JMSEException {
4         super("SwimmerDisplay", false, true/*is output*/);
5         receivers.add(new Receiver("semaine.data.swimmer.position"));
6         setupGUI();
7     }
8
9     @Override protected void react(SEMAINEMessage m) throws JMSEException {
10        float percent = Float.parseFloat(m.getText());
11        updateSwimmerPosition(percent);
12        String message = percent <= 0 ? "You lost!" : percent >= 100 ? "You won!!!" : null;
13        if (message != null) {
14            ...
15        }
16    }
17    ...
18 }

```

Figure 21: The SwimmerDisplay component (GUI code not shown).

The SwimmerDisplay (Figure 21) implements the user interface shown in Figure 19. Its messaging part consist of a simple text-based Receiver (l. 5) and an interpretation of the text messages as single float values (l. 10).

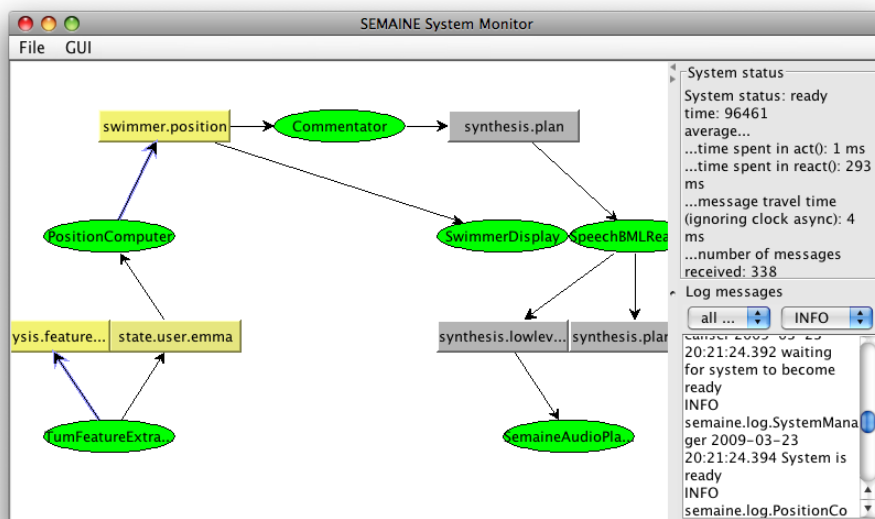
```

1 public class Commentator extends Component {
2     private BMLSender bmlSender = new BMLSender("semaine.data.synthesis.plan", getName());
3     private boolean started = false;
4
5     public Commentator() throws JMSEException {
6         super("Commentator");
7         receivers.add(new Receiver("semaine.data.swimmer.position"));
8         senders.add(bmlSender);
9     }
10
11     @Override protected void react(SEMAINEMessage m) throws JMSEException {
12         float percent = Float.valueOf(m.getText());
13         if (percent < 30 /*danger*/) say("Your swimmer needs help!");
14         else if (percent > 70 /*nearly there*/) say("Just a little more.");
15     }
16
17     @Override protected void act() throws JMSEException {
18         if (!started) {
19             started = true;
20             say("The swimmer needs your support to reach the river bank. Cheer him up!");
21         }
22     }
23
24     private void say(String text) throws JMSEException {
25         Document bml = XMLTool.newDocument(BML.ROOT_TAGNAME, BML.namespaceURI);
26         Element speech = XMLTool.appendChildElement(bml.getDocumentElement(), BML.E_SPEECH);
27         speech.setAttribute("language", "en-US");
28         speech.setTextContent(text);
29         bmlSender.sendXML(bml, meta.getTime());
30     }
31 }

```

Figure 22: The Commentator component, producing TTS requests.

Due to the separation of position computer and swimmer display, it is now very simple to add a Commentator component (Figure 22) that generates comments using synthetic speech, as a function of the current position of the swimmer. It subscribes to the same Topic as the SwimmerDisplay (l. 7), and sends BML output (l. 2) to the Topic serving as input to the speech synthesis component of the SEMAINE system 1.0 [47]. Speech output is produced when the game starts (l. 18-20) and when the position meets certain criteria (l. 13-14). Generation of speech output consists in the creation of a simple BML document with a `<speech>` tag enclosing the text to be spoken (l. 25-28), and sending that document (l. 29).



23: Message flow graph of the swimmer's game system.

The complete system consists of the Java components SystemManager, PositionComputer, SwimmerDisplay, Commentator, SpeechBMLRealiser and SemaineAudioPlayer, as well as the external C++ component openSMILE. The resulting message flow graph is shown in Figure 23.

6. Evaluation

One important aspect in a middleware framework is message routing time. We compared the MOM ActiveMQ, used in the SEMAINE API, with an alternative system, Psyclone [59], which is used in systems similar to ours (e.g., [60]). In order to compute the mere message routing time ignoring network latencies, we ran both ActiveMQ 5.1.0 and Psyclone 1.1.7 on a Windows Vista machine with a Core2Duo 2.5 GHz processor and 4 GB RAM with no other load, and connected to each using a Java client sending and receiving messages in sequence from the localhost machine. We sent text messages of different lengths to each middleware in a loop, averaging measures over 100 repetitions for each message length. We used plain string messages with lengths between 10 and 1,000,000 characters. The message routing times are shown in Figure 24. Between 10 and 1,000 characters, round trip message routing times for ActiveMQ are approximately constant at around 0.3 ms; the times rise to 0.5 ms for 10,000 characters, 2.9 ms for 100,000, and 55 ms for messages of 1,000,000 characters length. Psyclone is substantially slower, with routing times approximately constant around 16 ms for messages from 10 to 10,000 characters length, then rising to 41 ms at 100,000 characters length and 408 ms at 1,000,000 characters message length.

These results show that in this task, ActiveMQ is approximately 50 times faster than Psyclone for short messages, and around 10 times faster for long messages. While it may be possible to find even faster systems, it seems that ActiveMQ is reasonably fast for our purposes.

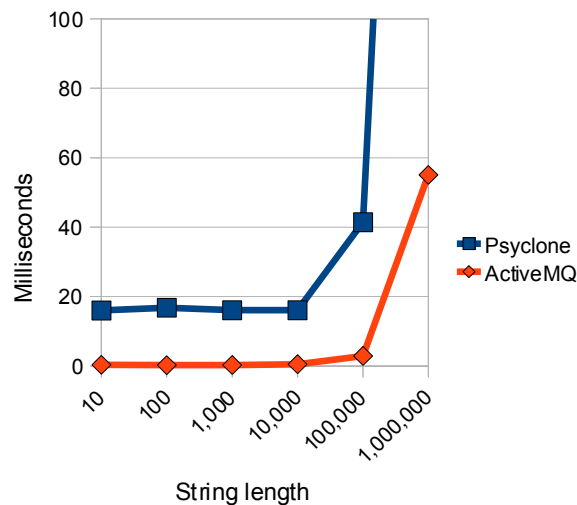


Figure 24: Round-trip message routing times as a function of message length.

Other evaluation criteria are more difficult to measure. While it is an aim of the SEMAINE API to be easy to use for developers, time will have to tell whether the system is being embraced by the community. A first piece of positive evidence is the adoption of the SEMAINE API for a real-time animation engine [61].

One aspect that should be added to the current SEMAINE API when representation formats settle is the validation of representation formats per Topic. Using XML schema, it is possible to validate that any message sent via a given Topic respects a formally defined syntax definition for that Topic. At the time of developing and debugging a system, this feature would help identify problems. At run-time, the validation could be switched off to avoid the additional processing time required for XML validation.

7. Availability

The SEMAINE API and the SEMAINE system 1.0 are available as open source [57][58]. The API is covered by the GNU Lesser General Public License LGPL [62], which can be combined with both closed-source and open source components. The components of the system 1.0 are partly released under the LGPL, partly under the more restrictive GNU General Public License GPL [63], which prohibits the proliferation together with closed-source components.

The examples in Section 5 are available from the SEMAINE sourceforge page [57] as an add-on to the SEMAINE system 1.0.

8. Conclusion

This paper has presented the SEMAINE API as a framework for enabling the creation of simple or complex emotion-oriented systems with limited effort. The framework is rooted in the understanding that the use of standard formats is beneficial for interoperability and reuse of components. The paper has illustrated how system integration and reuse of components can work in practice.

More work is needed in order to make the SEMAINE API fully suitable for a broad range of applications in the area of emotion-aware systems. Notably, the support of representation formats needs to be completed. Moreover, several crucial representation formats are not yet fully specified, including EmotionML, BML and FML. Agreement on these specifications can result from an ongoing consolidation process in the community. If several research teams were to bring their work into a common technological framework, this would be likely to speed up the consolidation process, because challenges to integration would become apparent more quickly. An open source framework such as the SEMAINE API may be suited for such an endeavour.

Acknowledgements

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement n° 211486 (SEMAINE). The work presented here has been shaped by discussions about concepts and implementation issues with many people, including Elisabetta Bevacqua, Roddy Cowie, Florian Eyben, Hatice Gunes, Dirk Heylen, Mark ter Maat, Sathish Pammi, Maja Pantic, Catherine Pelachaud, Björn Schuller, Etienne de Sevin, Michel Valstar and Martin Wöllmer. Thanks to Jonathan Gratch who pointed us to ActiveMQ in the first place. Thanks also to Oliver Wenz for designing the graphics of the swimmer's game.

References

- [1] Z. Zeng, M. Pantic, G.I. Roisman, and T.S. Huang, "A survey of affect recognition methods: audio, visual and spontaneous expressions," *Proceedings of the 9th international conference on Multimodal interfaces*, Nagoya, Aichi, Japan: ACM, 2007, pp. 126-133, <http://portal.acm.org/citation.cfm?id=1322192.1322216>.
- [2] M. Pantic and L.J.M. Rothkrantz, "Automatic Analysis of Facial Expressions: The State of the Art," *IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE*, 2000, pp. 1424-1445.
- [3] S.V. Ioannou, A.T. Raouzaïou, V.A. Tzouvaras, T.P. Mailis, K.C. Karpouzis, and S.D. Kollias, "Emotion recognition through facial expression analysis based on a neurofuzzy network," *Neural Networks*, vol. 18, 2005, pp. 423-435.

- [4] A. Batliner, S. Steidl, B. Schuller, D. Seppi, K. Laskowski, T. Vogt, L. Devillers, L. Vidrascu, N. Amir, and L. Kessous, "Combining efforts for improving automatic classification of emotional user states," *Proc. First International Language Technologies Conference, IS-LTC 2006*, Ljubljana, Slovenia: 2006.
- [1] B. Schuller, D. Seppi, A. Batliner, A. Maier, and S. Steidl, "Towards More Reality in the Recognition of Emotional Speech," *Acoustics, Speech and Signal Processing, 2007. ICASSP 2007. IEEE International Conference on*, 2007, pp. IV-941-IV-944.
- [6] C. Peter and A. Herbon, "Emotion representation and physiology assignments in digital systems," *Interact. Comput.*, vol. 18, 2006, pp. 139-170.
- [7] P. Gebhard, "ALMA — A layered model of affect," *Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-05)*, Utrecht: 2005.
- [1] N. Tsapatsoulis, A. Raouzaïou, S. Kollias, R. Cowie, and E. Douglas-Cowie, "Emotion Recognition and Synthesis Based on MPEG-4 FAPs," *MPEG-4 Facial Animation - The standard, implementations, applications*, I.S. Pandzic and R. Forchheimer, eds., Hillsdale, NJ, USA: John Wiley & Sons, 2002.
- [9] E. Bevacqua, M. Mancini, R. Niewiadomski, and C. Pelachaud, "An expressive ECA showing complex emotions," *Proceedings of the AISB Annual Convention*, Newcastle, UK: 2007, pp. 208–216.
- [10] F. Burkhardt and W.F. Sendlmeier, "Verification of Acoustical Correlates of Emotional Speech using Formant Synthesis," *Proceedings of the ISCA Workshop on Speech and Emotion*, Northern Ireland: 2000, p. 151—156.
- [11] M. Schröder, "Approaches to emotional expressivity in synthetic speech," *The Emotion in the Human Voice*, K. Izdebski, ed., San Diego, CA: Plural, 2008.
- [12] G. Castellano, R. Bresin, A. Camurri, and G. Volpe, "Expressive control of music and visual media by full-body movement," *Proceedings of the 7th international conference on New interfaces for musical expression*, New York, New York: ACM, 2007, pp. 390-391, <http://portal.acm.org/citation.cfm?id=1279740.1279829>.
- [13] ISO - International Organization for Standardization, "ISO 261: ISO general purpose metric screw threads -- General plan," 1998, http://www.iso.org/iso/iso_catalogue/catalogue_ics/catalogue_detail_ics.htm?csnumber=4165.
- [14] ISO - International Organization for Standardization, "ISO/IEC 26300:2006: Information technology -- Open Document Format for Office Applications (OpenDocument) v1.0," 2006, http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=43485.
- [15] D. Raggett, A. Le Hors, and I. Jacobs, *HTML 4.01 Specification*, 1999, <http://www.w3.org/TR/html401/>.
- [16] K.V. Deemter, B. Krenn, P. Piwek, M. Klesen, M. Schröder, and S. Baumann, "Fully generated scripted dialogue for embodied agents," *Artificial Intelligence*, vol. 172, Jun. 2008, pp. 1219-1244.
- [17] P. Piwek, B. Krenn, M. Schröder, M. Grice, S. Baumann, and H. Pirker, "RRL: A Rich Representation Language for the description of agent behaviour in NECA," *Proceedings of the AAMAS Workshop Conversational Agents*, Bologna, Italy: 2002.
- [18] B. Kempe, N. Pfleger, and M. Löckelt, "Generating Verbal and Nonverbal Utterances for Virtual Characters," *Virtual Storytelling*, 2005, pp. 73-76, http://dx.doi.org/10.1007/11590361_8.

- [19] M. Löckelt and N. Pflieger, "Multi-Party Interaction With Self-Contained Virtual Characters," *Proceedings of the 9th Workshop on the Semantics and Pragmatics of Dialogue (DIALOR)*, Nancy, France: 2005, pp. 139–142, <http://dialor05.loria.fr/Papers/21-Loeckelt.pdf>.
- [20] R. Aylett, A. Paiva, J. Dias, L. Hall, and S. Woods, "Affective Agents for Education Against Bullying," *Affective Information Processing*, J. Tao and T. Tan, eds., London: Springer, 2009, pp. 75-90, http://dx.doi.org/10.1007/978-1-84800-306-4_5.
- [21] A. Ortony, G.L. Clore, and A. Collins, *The Cognitive Structure of Emotion*, Cambridge, UK: Cambridge University Press, 1988.
- [22] P. Gebhard, M. Schröder, M. Charfuelan, C. Endres, M. Kipp, S. Pammi, M. Rumpler, and O. Türk, "IDEAS4Games: Building Expressive Virtual Characters for Computer Games," *Proc. IVA*, Tokyo, Japan: Springer, 2008, pp. 426-440, http://dx.doi.org/10.1007/978-3-540-85483-8_43.
- [23] Mystic Game Development, "EMotion FX," <http://www.mysticgd.com/site2007/>.
- [24] Luxand, Inc., "Luxand - Detect Human Faces and Recognize Facial Features with Luxand FaceSDK," <http://www.luxand.com/facesdk/>.
- [25] N. Dimakis, J.K. Soldatos, L. Polymenakos, P. Fleury, J. Curín, and J.(. Kleindienst, "Integrated Development of Context-Aware Applications in Smart Spaces," *IEEE Pervasive Computing*, vol. 7, 2008, pp. 71-79.
- [26] US National Institute of Standards and Technology (NIST), "NIST Data Flow System II," 2008, http://www.nist.gov/smartspace/sf_presentation.html.
- [27] N. Hawes, J.L. Wyatt, A. Sloman, M. Sridharan, R. Dearden, H. Jacobsson, and G. Kruijff, "Architecture and Representations," *Cognitive Systems*, H.I. Christensen, A. Sloman, G. Kruijff, and J. Wyatt, eds., published online at <http://www.cognitivesystems.org/cosybook/>, 2009, pp. 53-95.
- [28] M. Henning, *Choosing Middleware: Why Performance and Scalability do (and do not) Matter*, ZeroC, 2009, <http://www.zeroc.com/articles/IcePerformanceWhitePaper.pdf>.
- [29] "Semaine Project," <http://www.semaine-project.eu/>.
- [30] M. Schröder, R. Cowie, D. Heylen, M. Pantic, C. Pelachaud, and B. Schuller, "Towards responsive Sensitive Artificial Listeners," *Fourth International Workshop on Human-Computer Conversation*, Bellagio, Italy: 2008.
- [31] G. Banavar, T. Chandra, R. Strom, and D. Sturman, "A Case for Message Oriented Middleware," *Distributed Computing*, 1999, p. 846, http://dx.doi.org/10.1007/3-540-48169-9_1.
- [32] "Apache ActiveMQ," <http://activemq.apache.org/>.
- [33] M. Hapner, R. Burrige, R. Sharma, J. Fialli, and K. Stout, *Java Message Service (JMS) Specification Version 1.1*, Sun Microsystems, 2002, <http://java.sun.com/products/jms/docs.html>.
- [34] "The Apache Xerces Project - xerces.apache.org," <http://xerces.apache.org/>.
- [35] A. Le Hors, P. Le Hégarret, L. Wood, G. Nicol, J. Robie, M. Champion, and S. Byrne, *Document Object Model (DOM) Level 3 Core Specification*, 2004, <http://www.w3.org/TR/DOM-Level-3-Core/>.
- [36] "Behavior Markup Language (BML) Wiki," 2008, <http://wiki.mindmakers.org/projects:BML:main>.

- [37] M. Johnston, P. Baggia, D.C. Burnett, J. Carter, D.A. Dahl, G. McCobb, and D. Raggett, "EMMA: Extensible MultiModal Annotation markup language," Feb. 2009, <http://www.w3.org/TR/emma/>.
- [38] D.C. Burnett, M.R. Walker, and A. Hunt, "Speech Synthesis Markup Language (SSML) Version 1.0," 2004, <http://www.w3.org/TR/speech-synthesis/>.
- [39] M. Schröder, P. Baggia, F. Burkhardt, J. Martin, C. Pelachaud, C. Peter, B. Schuller, I. Wilson, and E. Zovato, *Elements of an EmotionML 1.0*, World Wide Web Consortium, 2008, <http://www.w3.org/2005/Incubator/emotion/XGR-emotionml-20081120/>.
- [40] S. Kopp, B. Krenn, S. Marsella, A. Marshall, C. Pelachaud, H. Pirker, K. Thórisson, and H. Vilhjálmsón, "Towards a Common Framework for Multimodal Generation: The Behavior Markup Language," *Intelligent Virtual Agents*, 2006, pp. 205-217, http://dx.doi.org/10.1007/11821830_17.
- [41] D. Heylen, S. Kopp, S. Marsella, C. Pelachaud, and H. Vilhjálmsón, eds., *Proc. Workshop on Functional Markup Language at The Seventh International Conference on Autonomous Agents and Multiagent Systems (AAMAS'08)*, Estoril, Portugal: 2008.
- [42] D. Becket and B. McBride, *RDF/XML Syntax Specification (Revised)*, 2004, <http://www.w3.org/TR/rdf-syntax-grammar/>.
- [43] T. Bray, J. Paoli, C. Sperberg-McQueen, E. Maler, and F. Yergeau, *Extensible Markup Language (XML) 1.0 (Fifth Edition)*, 2008, <http://www.w3.org/TR/xml/>.
- [44] M. Mancini and C. Pelachaud, "The FML-APML language," *Proc. Workshop on Functional Markup Language at The Seventh International Conference on Autonomous Agents and Multiagent Systems (AAMAS'08)*, Estoril, Portugal: 2008.
- [45] B. De Carolis, C. Pelachaud, I. Poggi, and M. Steedman, "APML, a Markup Language for Believable Behavior Generation," *Life-Like Characters*, H. Prendinger and M. Ishizuka, eds., New York: Springer, 2004, pp. 65-86.
- [46] M. Schröder, M. Charfuelan, S. Pammi, and O. Türk, "The MARY TTS entry in the Blizzard Challenge 2008," *Proc. Blizzard Challenge*, Brisbane, Australia: 2008.
- [47] M. Schröder, M. ter Maat, C. Pelachaud, E. Bevacqua, E. de Sevin, B. Schuller, F. Eyben, and M. Wöllmer, *SEMAINE deliverable D1b: First integrated system*, 2008, <http://semaine.sourceforge.net/SEMAINE-1.0/D1b%20First%20integrated%20system.pdf>.
- [48] D.M. Wolpert and J.R. Flanagan, "Motor prediction," *Current Biology*, vol. 11, 2001, pp. R729-R732.
- [49] F. Eyben, M. Wöllmer, and B. Schuller, "openEAR—Introducing the Munich Open-Source Emotion and Affect Recognition Toolkit," *Proc. Affective Computing and Intelligent Interaction*, Amsterdam, The Netherlands: IEEE, 2009.
- [50] M. Wöllmer, F. Eyben, S. Reiter, B. Schuller, C. Cox, E. Douglas-Cowie, and R. Cowie, "Abandoning Emotion Classes - Towards Continuous Emotion Recognition with Modelling of Long-Range Dependencies," *Proc. Interspeech*, Brisbane, Australia: 2008.
- [51] E. Douglas-Cowie, R. Cowie, I. Sneddon, C. Cox, O. Lowry, M. McRorie, J. Martin, L. Devillers, S. Abrilian, A. Batliner, N. Amir, and K. Karpouzis, "The HUMAINE Database: Addressing the Collection and Annotation of Naturalistic and Induced Emotional Data," *Affective Computing and Intelligent Interaction*, 2007, pp. 488-500, http://dx.doi.org/10.1007/978-3-540-74889-2_43.
- [52] D.B. Paul and J.M. Baker, "The design for the wall street journal-based CSR corpus," *Proceedings of the workshop on Speech and Natural Language*, Harriman, New York:

Association for Computational Linguistics, 1992, pp. 357-362,
<http://portal.acm.org/citation.cfm?id=1075614>.

- [53] J. Carletta, “Unleashing the killer corpus: experiences in creating the multi-everything AMI Meeting Corpus,” *Language Resources and Evaluation*, vol. 41, May. 2007, pp. 181-190.
- [54] B. Schuller, R. Müller, B. Höernler, A. Höethker, H. Konosu, and G. Rigoll, “Audiovisual recognition of spontaneous interest within conversations,” *Proceedings of the 9th international conference on Multimodal interfaces*, Nagoya, Aichi, Japan: ACM, 2007, pp. 30-37, <http://portal.acm.org/citation.cfm?id=1322201>.
- [55] E. Douglas-Cowie, R. Cowie, C. Cox, N. Amir, and D. Heylen, “The Sensitive Artificial Listener: an induction technique for generating emotionally coloured conversation,” Marrakech, Morocco: 2008, pp. 1-4.
- [56] J. Ostermann, “Face Animation in MPEG-4,” *MPEG-4 Facial Animation: The Standard, Implementation and Applications*, I.S. Pandzic and R. Forchheimer, eds., England: Wiley, 2002, pp. 17-55.
- [57] “SEMAINE sourceforge page,” <http://sourceforge.net/projects/semaine/>.
- [58] “SEMAINE-1.0 wiki documentation,” <http://semaine.opendfki.de/wiki/SEMAINE-1.0>.
- [59] CMLabs, “Psychlone,” 2007, <http://www.mindmakers.org/projects/Psyclone>.
- [60] R. Niewiadomski, E. Bevacqua, M. Mancini, and C. Pelachaud, “Greta: an interactive expressive ECA system,” *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems-Volume 2*, 2009, pp. 1399–1400.
- [61] A. Heloir and M. Kipp, “EMBR - A Realtime Animation Engine for Interactive Embodied Agents,” *Proceedings of the 9th International Conference on Intelligent Virtual Agents (IVA-09)*, Amsterdam, The Netherlands: Springer, 2009, pp. 393-404.
- [62] “GNU Lesser General Public License, version 3,” <http://www.gnu.org/licenses/lgpl.html>.
- [63] “GNU General Public License, version 3,” <http://www.gnu.org/licenses/gpl-3.0.html>.