EMOTIONAL NATURAL LANGUAGE GENERATOR

Conveying Affectiveness in Leading-edge Living Adaptive Systems

CALLAS

Project IST-34800

Deliverable D1.3.1 WP1.3

Deliverable
Version 1.0 – 31 July 2007
Document. ref.: callas.D131.UOA.WP1.1.V1.0

Programme Name:	IST
Project Number:	34800
Project Title:	CALLAS
Partners:	Coordinator: ENG (IT) Contractors: VTT Electronics, BBC, Metaware, Studio Azzurro, XIM, Digital Video, Humanware, Nexture, University of Augsburg, ICCS/NTUA, University of Mons, University of Teesside, Helsinki University of Technology, Paris 8, Scuola Normale Superiore di Pisa, University of Reading, Fondazione Teatro Massimo, HITLaboratory New Zealand
Document Number:	D1.3.1
Work-Package:	WP1.3
Deliverable Type:	Specs
Contractual Date of Delivery:	31/07/2007
Actual Date of Delivery:	
Title of Document:	Emotional Natural Language Generation
Author(s):	Michael Wißner, Birgit Endraß, Elisabeth André
Approval of this report	Executive Committee
Summary of this report:	Description of CALLAS components that form an Emotional Natural Language Generator
History:	
Keyword List:	emotional natural language generation, speech processing, sentence planning
Availability	This report is public

Table of Contents

Executive Summary

This document describes two components that, taken together, form an Emotional Natural Language Generator.

The document starts with an overview of both components, their respective purpose and an introduction of the example scenario to be used throughout this document.

The second chapter describes the planning component along with its knowledge base and strategies. Furthermore, the Semantic Representation Language, a means by which the two components communicate, is introduced.

The third and last chapter describes the sentence realizer together with its subcomponents (including two different generation methods) and gives examples of how they are used. The task of corpus acquisition and annotation is also explained, along with some guidelines.

1. Introduction

The task of the emotional natural language generator is to automatically produce verbal utterances that convey an ECA's emotional state. Starting from the observation that a speaker's emotional state determines not only the content, but also the linguistic realization of verbal utterances, our contribution consists of a "What to say" and a "How to say it" component.

The "What to say" component is realized as a planner while the "How to say it" component takes the form of a corpus-based language generation system. Figure 1.1 shows how these two components and their respective subcomponents interact with each other.

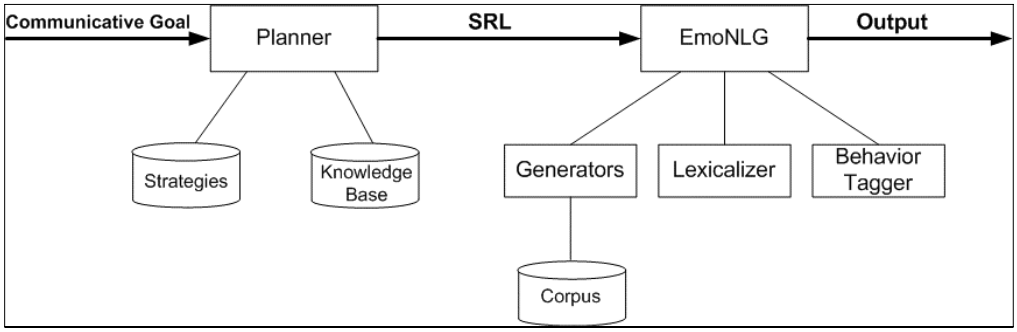

Figure 1.1: Component Overview

While each of the two components can be used on their own, they can also (as shown in the figure) be used together, employing a Semantic Representation Language for communication. The planner takes a communicative goal as input and outputs a sequence of planned utterances, which are then generated and realized by the language generation system. The output can either be plain text or text enriched with markups, e.g. with APML, the "Affective Presentation Markup Language" (de Carolis et al., 2004).

Being both implemented in the Java programming language, the components have the same hard- and software requirements: A Java-supporting operating system, an installed version of the Java Runtime Environment (v. 1.6.0 or later) and a mid-class PC with at least 512 MB of working memory.

Both components (what they are, how they work, and how they can be used) will be described in detail in the next two chapters, starting with the planner. For this end we use a comprehensive example, which we call the "Twin Sister Reunion". In this scenario, two sisters meet after not having seen each other for some time, greet each other and talk about their respective lives.

2. "What to say"

2.1 The Planner – Overall Concept

For the "What to say" component, we follow the approach described in (André, 2003). The basic idea is to make use of a hierarchical planner to decompose the speaker's communicative goal into more elementary actions. The result of this process is a dialog script that represents the elementary dialog turns to be executed by a single ECA or a team of ECAs.

Typically, dialog turns are represented in an XML-based markup language. For CALLAS, we rely on APML (Affective Presentation Markup Language) to specify dialog turns for the Greta agent and as well as SRL (Semantic Representation Language) which forms the input for a statistical sentence realizer (see next chapter).

As input, the planner expects a communicative goal, a library of plan operators that encode communicate templates and a domain knowledge base. To accomplish the communicative goal, it looks for applicable plan operators that match the communicative goal. The selection of the plan operators depends among other things on an ECA's emotional state. If a plan operator is found, all expressions in the body will be set up as new sub goals. The planning process terminates successfully if all sub goals are expanded to dialog turns that may be forwarded to the "How to say" component. Otherwise, there is no solution for a given communicative goal and the planning process fails.

To implement the approach, we rely on Kleinbauer's Nippl planner, which is a re-implementation of the PrePlan planner originally developed by (André, 2003).

In the following, we will illustrate the approach by means of a simple example.

2.2 Building a Greeting Scenario

To demonstrate the approach, we simulate the communicative behaviors of two twin sisters that meet in the street. The flow of their conversation is influenced by personality traits and their emotional state.

The two twin sisters are realized as duplicates of the Greta agent. Greta is an embodied conversational agent that incorporates conversational and emotional qualities. To determine speech-accompanying non-verbal behaviors the system relies on taxonomy of communicative functions.

Domain Knowledge Base

The domain knowledge base provides the propositional content for the agents' dialog contributions. In addition, we may specify information on the agent's personality and emotional state.

```
name: FACT

###### Personality and emotional condition of the agents #######

(name agent1 "Selma")
(extroversion agent1 "extrovert")
```

```
(attitude agent1 agent2 "positive")


(name agent2 "Patty")
(extroversion agent2 "introvert")
(attitude agent2 agent1 "positive")
```

Plan operators

Plan operators specify communicative templates. A plan operator consists of a strategy, constraints and a list of sub goals. A strategy consists of a name and a list of arguments, which may be constants or variables. Variables are marked with the prefix '?' and may be instantiated during the planning process with concrete values.

```
strategy: (GreetingScene ?agent1 ?agent2)
constraints: (FACT (extroversion ?agent1 "extrovert"))
subgoals:  (GreetEachOther ?agent1 ?agent2)
           (Smalltalk ?agent1 ?agent2)
           (FarewellEachOther ?agent1 ?agent2)


strategy: (GreetEachOther ?agent1 ?agent2)
constraints: (FACT (extroversion ?agent1 "extrovert"))
subgoals: (Greet ?agent1 ?agent2)
          (GreetBack ?agent2 ?agent1)

strategy: (Greet ?agent1 ?agent2)
constraints: (FACT (attitude ?agent1 ?agent2 ?attitude))
subgoals: (GenSRL ?agent1 ?agent2 "greeting" ?attitude)

strategy: (GreetBack ?agent1 ?agent2)
constraints: (FACT (attitude ?agent1 ?agent2 ?attitude))
subgoals: (GenSRL ?agent1 ?agent2 "greeting" ?attitude)


strategy: (Smalltalk ?agent1 ?agent2)
subgoals:
    foreach: ?topic
       with: (or (FACT (interest ?agent1 ?topic))
                 (FACT (interest ?agent2 ?topic)))
         do: (DiscussTopic ?agent1 ?agent2 ?topic)

strategy: (DiscussTopic ?agent1 ?agent2 ?topic)
constraints: (and (FACT (extroversion ?agent1 "extrovert"))
                  (FACT (interest ?agent1 ?topic)))
subgoals: (AskTopic ?agent1 ?agent2 ?topic)
          (AnswerTopic ?agent2 ?agent1 ?topic)
          (CommentAnswer ?agent1 ?agent2 ?topic)

strategy: (DiscussTopic ?agent1 ?agent2 ?topic)
constraints: (and (FACT (extroversion ?agent2 "extrovert"))
                  (FACT (interest ?agent2 ?topic)))
subgoals: (AskTopic ?agent2 ?agent1 ?topic)
          (AnswerTopic ?agent1 ?agent2 ?topic)
              (CommentAnswer ?agent2 ?agent1 ?topic)
```

```
strategy: (DiscussTopic ?agent1 ?agent2 ?topic)
subgoals: (DoNothing)

strategy: (AskTopic ?agent1 ?agent2 ?topic)
constraints: (FACT (attitude ?agent1 ?agent2 ?attitude))
subgoals: (GenSRL ?agent1 ?agent2 "question" ?topic ?attitude)

strategy:  (AnswerTopic ?agent1 ?agent2 ?topic)
constraints: (FACT (attitude ?agent1 ?agent2 ?attitude))
subgoals: (GenSRL ?agent1 ?agent2 "answer" ?topic ?attitude)

strategy: (CommentAnswer ?agent1 ?agent2 ?topic)
constraints: (FACT (attitude ?agent1 ?agent2 ?attitude))
subgoals: (GenSRL ?agent1 ?agent2 "statement_pos" ?attitude)
```

The slot constraints specifies the context in which the plan operator may be used. For instance, the operator first operator listed above may be used if there is an entry

```
(extroversion agent1 "extrovert")
```

in the domain knowledge. In this case, ?`agent1` would be instantiated with `agent1` when the strategy is called. That is the agent has to be extrovert to initialize the greeting.

The slot `subgoals` specifies the acts to be carried out when the plan operator is applied. In our case, GreetEachOther consists of two sub goals: `Greet` and `GreetBack`.

Subgoals may also contain a foreach construct. In this case the specified action might be executed several times. The strategy `Smalltalk`, for example will apply the strategy `DiscussTopic` for every `?topic` in the knowledge base that fulfils the conditions.

Communicative Goals

Communicative goals consist of a communicative act and a list of arguments. To apply a plan operator, the goal has to match its header. That is constants have to coincide and variables in the operator have to be unifiable with the arguments of the goal. For instance, the goal

(GreetingScene agent1 agent2)

matches the strategy slot of the plan operator above.

Output

The output is an SRL expression, that is exported every time the subgoal `GenSRL` is called. The SRL expression contains information about the speaker and the communicative act in. If the strategy AskTopic is instantiated with

```
(AskTopic agent1 agent2 children)
```

and the knowledge base contains information about the negative attitude of agent1, the subgoal

```
(GenSRL agent1 agent2 "question" children negative)
```

is executed and the following SRL expression is generated.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<SRL>
<Speaker>agent2</Speaker>
<MeaningElement attribute="category" value="question" />
<MeaningElement attribute="topic" value="children" />
<MeaningElement attribute="attitude" value="negative" />
 </SRL>
```

3. “How to say it” – The Emotional Natural Language Generator

This chapter describes the “Emotional Natural Language Generator” (EmoNLG), a component that generates emotional natural language sentences from semantic concepts. These semantic concepts can be represented in a specific XML-format, the “Semantic Representation Language” (SRL). For example, SRL is the means by which EmoNLG and the planner described in chapter 1 communicate.

EmoNLG provides two different ways of creating emotionally rich sentences which are employed at different levels of the generation process. First, there are the semantic concepts mentioned above. While these concepts are arbitrary, they can (and should) of course be emotion-related thus providing a high-level (e.g. happy vs. unhappy sentences) control over the emotional content of created sentences. Second, with a module called *Lexicalizer*, EmoNLG provides means of creating different shades of emotions in the generated sentences by substituting words with emotionally rich synonyms (see section 3.3 for more details) as described by (Fleischman and Hovy, 2002).

Note that EmoNLG is not an application but rather a library with which developers can build their own applications.

The remainder of this chapter will first describe the tasks of corpus acquisition and annotation, then give a detailed view about EmoNLG’s generation methods, describe the SRL, and finally take a closer look two of its modules, the *Lexicalizer* and the *BehaviorTagger*.

3.1 Corpus Acquisition and Annotation

To generate sentences, EmoNLG does not build upon rules or a grammar, but on an annotated corpus of sentences from the desired domain and language. EmoNLG generates sentences by selecting or statistically deriving them from the ones in the corpus.

On the one hand, this corpus-based approach is easy to use, extend and port, since it requires no more than an annotated corpus which is then statistically analyzed. In comparison, the task of designing grammars or rules that can generate all desired sentences usually is much more of an

On the other hand, the quality and understandability of such an approach very much depends on the underlying corpus and the statistical information derived from it. Thus, the next two sections describe in what way corpora need to be acquired, prepared, and annotated to constitute a solid base for EmoNLG.

3.1.1 Corpus Acquisition and Preparation

Corpora for EmoNLG can be acquired in any conceivable way, from user studies or spontaneous dialogs, in spoken or textual form. However, the following guidelines should be kept in mind and followed, especially when preparing the corpus for later use:

- The corpus needs to be in textual form. Spoken corpora must be transcribed.
- The corpus must consist of single sentences. Longer utterances, even from the same speaker, should be split into single sentences.
- The sentences should be rather short, containing no more than fifteen words or so. Longer sentences should be split into shorter ones, if possible.
- Sentences in the corpus should contain punctuation marks, since these add a

structure that can be learned and reproduced by the generating process.

- Words should be capitalized correctly (if applicable for the used language) to reduce ambiguity.
- The corpus should contain at least 500 sentences to yield proper statistical information.
- Exchangeable words such as names or places or words that convey certain emotional attitudes should be substituted by a proper label (see section 3.3 for more details) to yield a more general corpus.

3.1.2 Corpus Annotation

The corpus needs to be put in one or more XML-files of a specific format in order to be used with EmoNLG. Figure 3.1 shows an excerpt of the corpus file we use for our sample scenario, the “Twin Sister Reunion”.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<SentenceFrames>
	<SentenceFrame>
		<Sentence>Hey sis, how are you?</Sentence>
		<MeaningElement attribute="category" value="greeting"/>
		<MeaningElement attribute="attitude" value="positive"/>
	</SentenceFrame>
	<SentenceFrame>
		<Sentence>You still enjoy working as a $SISTERS_JOB, I hope?</Sentence>
		<MeaningElement attribute="category" value="question"/>
		<MeaningElement attribute="topic" value="career"/>
		<MeaningElement attribute="attitude" value="positive"/>
	</SentenceFrame>
	<SentenceFrame>
		<Sentence>So you're still together with that guy, what is his name again?</Sentence>
		<MeaningElement attribute="category" value="question"/>
		<MeaningElement attribute="topic" value="partner"/>
		<MeaningElement attribute="attitude" value="negative"/>
	</SentenceFrame>
	<SentenceFrame>
		<Sentence>Well, it was nice seeing you.</Sentence>
		<MeaningElement attribute="category" value="farewell"/>
		<MeaningElement attribute="attitude" value="neutral"/>
	</SentenceFrame>
</SentenceFrames>
```

Figure 3.1: Excerpt from the sample corpus

As can be seen in Figure 3.1, the top-level tag in the corpus file needs to be called *SentenceFrames*, whereas there is a child node *SentenceFrame* for each sentence from the corpus. Within this child node, the sentence is embedded in a tag called *Sentence*. The annotation itself consists of an arbitrary number of the tag *MeaningElement*. Each of these meaning elements represents a semantic concept as an attribute-value-pair. In our example we use three different meaning elements, one for the category of the sentence (“greeting”, “farewell”, “question”, “answer” and so on), one for the topic (“partner”, “children” “career” etc.), and one for the speaker’s attitude (“positive”, “neutral”, and “negative”) towards that particular topic or towards the interlocutor in case of categories that do not require a topic (e.g. “greeting”).

This way of annotating the corpus with attribute-value-pairs as semantic concepts is based on (Bhagat and Hovy, 2005).

Note that in our scenario, we make no distinction between the two speakers in regard to what sentences they can say, and hence we did not annotate speaker-dependent information in the corpus. Also note that "$SISTERS_JOB" in the second sentence is one of the above-mentioned labels that is later replaced by an appropriate expression (see section 3.3).

The number and kind of meaning elements of course strongly depends on the desired application. For example, in our scenario the three elements mentioned above suffice while Bhagat and Hovy use six. Figure 3.2 shows one annotated sentence from their corpus.

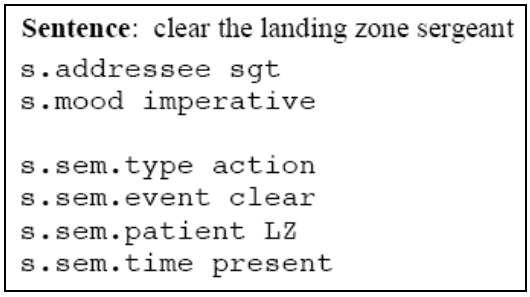

Figure 3.2: Excerpt from Bhagat and Hovy's corpus

It can be seen that Bhagat and Hovy not only use more meaning elements but also that their elements wholly describe the semantic content of the sentence. As such, the sentence and its associated meaning elements form a semantic frame as described by (Fillmore 1976). While such a detailed level of annotation is of course needed in an application like theirs (a user giving specific orders to a virtual human) it would be way too detailed for our application (two agents performing small talk).

It is needless to say that the quality and manner of annotation is the most crucial factor for the quality of the sentences generated with EmoNLG. Therefore, it needs a lot of careful consideration and training to decide the kind and number of meaning elements the corpus should be annotated with. Two example concepts of assigning meaning elements have been shown here, but of course many more are conceivable. In any case, the following guidelines should be taken into consideration:

- Meaning elements always describe the semantics of the whole sentence and are never directly associated with particular words.
- Meaning elements are one of the two ways in which the emotional content of the generated sentences can be influenced (see introduction above for more details).

3.2 The Generation Methods

EmoNLG contains two generation methods that generate sentences from the corpus, albeit using different approaches. The *SelectionGenerator*, that merely selects sentences from the corpus according to the desired meaning elements, and the *StatisticalLanguageProcessor* that employs the generation part of a bidirectional statistical procedure described by (Bhagat and Hovy, 2005). The next two sections describe these methods.

3.2.1 Method A: SelectionGenerator

The *SelectionGenerator* is a class within EmoNLG that simply selects a sentence from the collected corpus. Figure 3.3 shows some sample code describing its usage.

```
String[] filenames = new String[1];
filenames[0] = "data/corpus.xml";

//Call to constructor with corpus file
SelectionGenerator slg = new SelectionGenerator(filenames);

//Create vector with meaning elements
Vector<AttributeValuePair> mEs = new Vector<AttributeValuePair>();
mEs.add(new AttributeValuePair("category", "greeting"));
mEs.add(new AttributeValuePair("attitude", "neutral"));

//Generate a single sentence
String sentence = slg.generateRandomSelection(mEs);

//Generate all sentences
Vector<String> sentences = slg.generate(mEs);
```

Figure 3.3: Sample usage of the *SelectionGenerator*

As the figure shows, *SelectionGenerator* provides a constructor that requires a String with a reference to the corpus file(s) to be used. The corpus is then parsed and stored in an appropriate data structure.

Meaning elements are realized in the class *AtributeValuePair* that simply consists of two Strings. The constructor also takes these Strings as parameters, attribute being the first. In this example, we want to create a neutral greeting.

SelectionGenerator provides two methods for generation:

`generate` and `generateRandomSelection`.

Both methods require a Vector of *AtributeValuePair* as a parameter that represents the desired meaning elements.

A call to `generate` returns all sentences that are annotated with the desired combination of meaning elements. The application itself can then provide ways of selecting one sentence from this set as the result of the generation process.

Using `generateRandomSelection` already takes care of this by returning a randomly selected sentence from the set of all appropriate sentences.

Note that the order in which the meaning elements are provided to the methods is arbitrary. It is also possible to omit certain meaning elements. We could, for example, provide a vector that only contains the first *AtributeValuePair* from Figure 3.3. This would yield all greetings, regardless of their attitude.

If no sentence can be found that corresponds to the supplied meaning elements, `null` is returned.

Of course, the *SelectionGenerator* only provides a very simple and straightforward procedure, especially when `generateRandomSelection` is used. Nevertheless, it is sufficient for certain kinds of corpora, for example those that contain only few different meaning elements and thus enough sentences for every combination of meaning elements. In contrast, applications that need to generate sentences for combinations of meaning elements not already present in the corpus are of course better off with the *StatisticalLanguageProcessor* described in the next section.

3.2.2 Method B: StatisticalLanguageProcessor

As described above, this generation process is based upon a bidirectional statistical language processor described by (Bhagat and Hovy, 2005), which is shown in Figure 3.4 below.

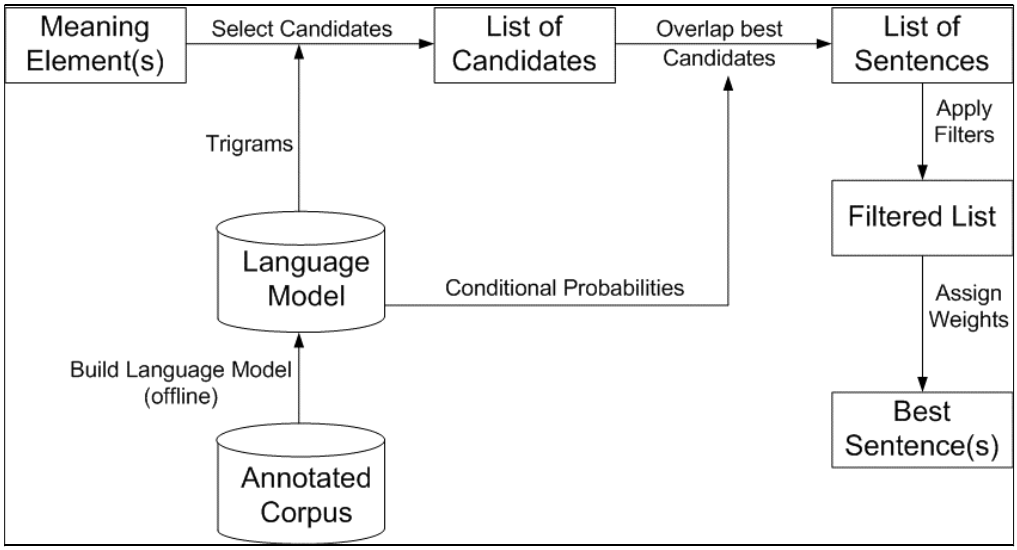

Figure 3.4: Overview of the statistical generation process

As a first and offline step, a statistical language model is built from the annotated corpus. This language model consists of conditional probabilities $P(w_{j-2}w_{j-1}w_j \mid m_i)$. This denotes the probability P of seeing the trigram $w_{j-2}w_{j-1}w_j$ as output with the meaning element m_i as an input. The language model contains such a trigram meaning dependency for each trigram meaning combination in the corpus.

The online generation process itself consists of four steps: candidate selection, candidate overlapping, sentence filtering and sentence selection. Each of these steps will now be explained in detail.

Step 1: Candidate Selection. For each meaning element in the input, all trigrams that are associated with that meaning element (i.e. their conditional probability within the language model is greater than zero) are considered as candidates. Should one trigram be "voted for" by more than one meaning element, the different probabilities are summed up and are now considered the trigram's weight. The resulting list of candidates is then sorted by that weight.

Step 2: Candidate Overlapping: The best 90 (Bhagat and Hovy use 30) candidates from the sorted list are then overlapped with each other to form sentences. Two trigrams $w_aw_bw_c$ and $w_xw_yw_z$ overlap if, and only if, $w_b=w_x$ and $w_c=w_y$. In this manner all possible (partial and full) sentences, which can be formed by continuously overlapping these 90 trigrams with each other, are built. This results in a list of sentences.

Step 3: Sentence Filtering: This step applies several filter mechanism to the generated sentences. This filtering step is not part of Bhagat and Hovy's work but was added by us. We implemented the following filters:

- "Filter Sentence Beginning": Only sentences that contain a proper sentence beginning a kept after the overlapping, others are discarded. This prevents partial sentences that occur often in the corpus from being chosen over whole but seldom sentences.
- "Filter Punctuation": Only sentences that end with a punctuation mark are kept,

others are discarded. This serves the same purpose as described above.

- "Self-Monitoring": This filter is based upon the concept of self-monitoring as described by (Levelt, 1989). According to this concept, human speakers constantly listen to what they are saying and are thus capable of correcting grammatical or content-related errors. Following this, we included a way of telling whether a generated sentence actually contains what was supposed to be "said" in the first place: We also implemented the parser from the language processing system described by Bhagat and Hovy and use it to parse all generated sentences. This yields a set of meaning elements that we then compare with those that constituted the input. This comparison is based on a list of "important attributes" (provided by the governing application): If an attribute is on that list but its generated value differs from the parsed one, that sentence will receive a negative modifier on its final weight (see next step). For example, if the parsing of a certain sentence yielded a meaning element "attitude neutral" but the input contained "attitude positive" then that sentence would receive a negative modifier if "attitude" was one of the important attributes.

Step 4: Sentence Selection: Each sentence that made it through the filtering step is now assigned a weight according to the formula shown in Figure 3.5 (*n* is the number of trigrams in the sentence).

$$\text{weightOfSentence} = \frac{\sum log(Wt(w_{j-2}w_{j-1}w_j))}{n} + \frac{n}{10}$$

Figure 3.5: Calculating a sentence's final weight

A sentence's final weight is the sum of the logarithms of the weights (as calculated in Step 1) of its trigrams, divided by its number of trigrams. The second addend is scaling factor that favors longer sentences.

All sentences are then sorted by that final weight and a certain number (set by the overlying application) of the topmost sentences is then returned as result of the generation process.

Figure 3.6 below illustrates the usage of the *StatisticalLanguageProcessor*, including the different parameters that can be set.

```
String[] filenames = new String[1];
filenames[0] = "data/corpus.xml";

//Call to constructor with corpus and model file
StatisticalLanguageProcessor slp =
    new StatisticalLanguageProcessor(filenames, "data/model.txt", true);

//Set number of candidates for step 2
slp.setNumberOfCandidates(90);

//Set filters for step 3
slp.setFilterSentenceBeginning(true);
slp.setFilterPunctuation(true);

Vector<String> importantAttributes = new Vector<String>();
importantAttributes.add("category");
importantAttributes.add("attitude");
slp.setimportantAttributes(importantAttributes);

//Set scaling for step 4
slp.setScaling(10);

//Set number of sentences to return for step 4
slp.setReturnBest(10);

//Create vector with meaning elements
Vector<AttributeValuePair> meaningElements = new Vector<AttributeValuePair>();
meaningElements.add(new AttributeValuePair("category", "question"));
meaningElements.add(new AttributeValuePair("topic", "career"));
meaningElements.add(new AttributeValuePair("attitude", "neutral"));

//Generate and get 1 from best 10 sentences
slp.generateRandomSelection(meaningElements);

//Generate and get all 10 best sentences
slp.generate(meaningElements);
```

Figure 3.6: Sample usage of the *StatisticalLanguageProcessor*

Similar to that of the *SelectionGenerator*, the constructor's first parameter is a reference to the corpus file. The second parameter is a reference to a file containing the language model. If the third parameter is set to "false" the language model is built and then written to the model file. If set to "true" the model is not built but read from the file. So normally "true" can be used (except for the first time, of course), unless the corpus changes.

The next couple of lines are responsible for setting all the variable parameters and activating the various filters mentioned above in the description of the generation process. They should all be self-explanatory, the comments indicate which step in the generation process these settings influence.

3.3 The *Lexicalizer*

In traditional natural language generation systems the task of lexicalization is a part of microplanning and deals with "choosing which words and syntactic structures should be used to express messages" (Reiter and Dale, 2000).

In EmoNLG the *Lexicalizer* applies finishing touches to the generated sentences by substituting certain general concepts (or "placeholders") with concrete words, hence the name. Based on a simplified version of the procedure described by (Fleischman and Hovy, 2002) the *Lexicalizer* takes the speaker's attitude towards these concepts into account when choosing the concrete words to put in the sentence.

These concepts and their respective lexicalizations are stored in a XML-file. Figure 3.7 shows an excerpt of the one we used in our "Twin Sister Reunion" sample.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<Speakers>
    <Speaker>
        <Name>Agent1</Name>
        <Entity>
            <Concept>SISTERS_CLOTHING</Concept>
            <Entry lexicalization="blouse" shade="0.0"/>
            <Entry lexicalization="rag" shade="-1.0"/>
        </Entity>
        <Entity>
            <Concept>SISTERS_SHAPE</Concept>
            <Entry lexicalization="slender" shade="1.0"/>
            <Entry lexicalization="thin" shade="0.0"/>
            <Entry lexicalization="skinny" shade="-1.0"/>
        </Entity>
        <Entity>
            <Concept>SISTERS_JOB</Concept>
            <Entry lexicalization="office assistant" shade="1.0"/>
            <Entry lexicalization="secretary" shade="0.0"/>
            <Entry lexicalization="typist" shade="-1.0"/>
        </Entity>
        <Entity>
            <Concept>SISTERS_CHILDREN</Concept>
            <Entry lexicalization="your kids" shade="1.0"/>
            <Entry lexicalization="your children" shade="0.0"/>
            <Entry lexicalization="your brats" shade="-1.0"/>
        </Entity>
    </Speaker>
</Speakers>
```

Figure 3.7: XML-File specifying lexicalizations

This file can contain several "Lexicons", one for each speaker (who is referenced by name or id). Each lexicon contains a number of arbitrary concepts (defined by their name, e.g. "SISTERS_JOB") and their respective possible lexicalizations.

To include a specific concept in the corpus, words describing it must be substituted with a dollar sign ("$") followed by the name of the concept (cf. "$SISTERS_JOB" in Figure 3.1).

Each of the lexicalizations also has an emotional shade, which is a floating point number. The

range of these values is arbitrary but the higher the value, the more positive the shade. It is possible to define more than one lexicalization with the same shade.

When lexicalizing a generated sentence, the *Lexicalizer* tries to substitute each concept within the sentence (recognized by the "$") with an appropriate lexicalization. For every possible lexicalization the difference between its shade and an attitude (supplied by the application) is calculated. The lexicalization with the least difference is then selected and put into the sentence. Should more than one lexicalization yield the least difference, one of them is randomly chosen.

The task of creating these Lexicons could be automated by collecting synonyms (for example from WordNet) and assigning them values based from affective Lexicons (like WordNet Affect or the Whissel emotional dictionary).

Figure 3.8 shows a code example of the *Lexicalizer*'s usage (continued from the example in Figure 3.3).

```
//Generate a sentence
String sentence = slg.generateRandomSelection(mEs);

//Creating a new Lexicalizer from lexicon file
Lexicalizer lex = new Lexicalizer("data/entries.xml");

//Lexicalize sentence for speaker "Agent1"
String lexicalizedSentence = lex.lexicalizeSentence(sentence, "Agent1", attitude);
```

Figure 3.8: Sample usage of the *Lexicalizer*

The *Lexicalizer*'s constructor must be supplied with a reference to the file containing the lexicalizations. Once initialized, a call to `lexicalizeSentence` starts the process described above. The first parameter is the sentence to be lexicalized, the second is a String containing the name of the speaker (in order to select the right lexicon), and the third is the above mentioned attitude towards the content of the sentence.

3.4 The *BehaviorTagger*

A last step in the generation of emotional natural language sentences could be the generation of appropriate accompanying nonverbal behavior such as facial expressions, gestures or eye gazes. Although these behaviors could be included in the generated sentences via markups, putting them directly in the corpus is not an option.

However, there are efforts underway in our lab to include behavioral markup in the generation process as a post-processing step, similar to the "Nonverbal Behavior Generator" described by (Lee and Marsella, 2006).

The *BehaviorTagger* is responsible for this final task of adding arbitrary markups to a generated sentence. Figure 3.9 contains a code example showing its usage.

```
//Create new BehaviorTagger with an APMLWriter
BehaviorTagger tagger = new BehaviorTagger(new APMLWriter());

//Create contexts
Hashtable<String, String> contexts = new Hashtable<String, String>();
contexts.put("category", "greeting");
contexts.put("attitude", "neutral");

//Create documents from sentence and contexts
Document doc = tagger.generateDocument(lexicalizedSentence, contexts);
```

Figure 3.9: Sample usage of the *BehaviorTagger*

The *BehaviorTagger* constructor requires one parameter, an object that implements the *MarkupWriter* interface. *APMLWriter* is such an object and developers can build their own by also implementing this interface. The *APMLWriter* makes a generated sentence AMPL-compliant so it can, for example, be animated with the Greta Agent.

A call to `generateDocument` yields the marked up sentence as an XML-Document that can be, for example, written to a file or sent over the network. This method takes two parameters: the sentence to be marked up and an arbitrary number of so-called contexts (organized in a Hashtable). These contexts can represent any information necessary for the markup-process.

For the *APMLWriter*, for example, we use the semantic concepts “category” and “attitude” to apply appropriate affect and performative types to the document.

3.5 The Semantic Representation Language

The SRL provides means to communicate with EmoNLG applications “from the outside”, e.g. from other applications or over a network. A SRL document can be seen as a “generation request”, containing semantic concepts (for the generation itself) and information about the speaker (for the lexicalization). Figure 3.10 shows an example of a SRL document.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<SRL>
      <Speaker>Agent1</Speaker>
      <MeaningElement attribute="category" value = "answer"/>
      <MeaningElement attribute="topic" value = "overall"/>
      <MeaningElement attribute="attitude" value = "positive"/>
</SRL>
```

Figure 3.10: Sample SRL document

Similar to the way in which the corpus or lexicon files are structured, an SRL document contains a speaker and semantic concepts.

Within EmoNLG, a single SRL document is represented as an instance of the *SRL* class. This class contains a String representing the speaker and a number of meaning elements, again as a Vector of *AttributeValuePair*. It also contains a convenience method `getValue` that returns the value of a designated attribute. The application code in Figure 3.12 contains a usage example this class.

3.6 Examples

This final section contains two examples, one displaying two generated dialog and a second one showing code for a complete example application built with EmoNLG.

3.6.1 Example dialog generated with EmoNLG

Figure 3.11 shows two realizations of the same dialog, with one difference: In the first, both agents have a very positive attitude and in the second they have a negative attitude towards pretty much everything.

```
Agent1: Hey sis, how are you?
Agent2: Oh, it's so good to see you.
Agent1: How's my favorite brother-in-law doing?
Agent2: He's doing great, I don't know what I'd do without him.
Agent1: Oh, I'm so happy for you, sis!
        And the kids just keep growing, I guess?
Agent2: They are just so adorable, you should see them, sis.
Agent1: I'm really glad to hear that.
Agent2: So, what's your job like these days?
Agent1: It's great, I'm quite successful as an office assitant.
Agent2: Oh, I'm so proud of you.
Agent1: Don't move, I'll be right back.
Agent2: Take care, sis.



Agent1: Oh, it's you.
Agent2: I didn't expect you to show up.
Agent1: Are you still with that guy or did you dump him?
Agent2: We're managing, you know.
Agent1: Well, go ahead, knock yourself out, it's your life.
        And are you still able to cope with your brats?
Agent2: Well, it's not easy, but I know they're worth it.
Agent1: If that's what you really want, wo am I to interfere?
Agent2: So, being a typist still doesn't bore you?
Agent1: It's not great, but I won't quit.
Agent2: That wasn't one of your better choices then, was it?
Agent1: I think that's enough chitchat for now.
Agent2: Well, that was quite some conversation.
```

Figure 3.11: Positive (top) and negative (bottom) dialog examples

These examples illustrate how both the content of the sentences and lexicalizations for the same concepts differ when realized with different attitudes.

3.6.2 An example application built with EmoNLG

Figure 3.12 below shows example code for a method that reads SRL from a file, generates a sentence using both of EmoNLG's generation methods, lexicalizes that sentence, creates APML-markups and finally writes it back into an AMPL-compliant file (that can be then, for example, be used with the Greta Agent).

```
public void generateToFile(String inputFilename, String outputFilename)
{
    //Create a SelectionGenerator
    String[] filenames = new String[1];
    filenames[0] = "data/corpus.xml";
    SelectionGenerator slg = new SelectionGenerator(filenames);

    //Create a StatisticalLanguageProcessor
    StatisticalLanguageProcessor ssp =
        new StatisticalLanguageProcessor(filenames, "data/model.txt", true);

    //Parse representation from input file
    SRL srl = XMLParser.parseSRL(inputFilename);

    //Generate a sentence
    String sentence = slg.generateRandomSelection(srl.getMeaningElements());

    //If not successful use statistical generation instead
    if(sentence==null)
        sentence = ssp.generateRandomSelection(srl.getMeaningElements());

    //Create a new Lexicalizer from lexicon file
    Lexicalizer lex = new Lexicalizer("data/entries.xml");

    //Calculate attitude
    String attitudeString = srl.getValue("attitude");
    double attitude = attitudeFromString(attitudeString);

    //Lexicalize sentence for the right speaker
    String lexicalizedSentence =
        lex.lexicalizeSentence(sentence, srl.getSpeaker(), attitude);

    //Create new BehaviorTagger with an APMLWriter
    BehaviorTagger tagger = new BehaviorTagger(new APMLWriter());

    //Create contexts
    Hashtable<String, String> contexts = new Hashtable<String, String>();
    for(AttributeValuePair avp:srl.getMeaningElements())
    {
        if(         avp.getAttribute().equals("category")
                ||  avp.getAttribute().equals("attitude"))

            contexts.put(avp.getAttribute(), avp.getValue());
    }

    //Create document from sentence and contexts
    Document doc = tagger.generateDocument(lexicalizedSentence, contexts);

    //Write to file
    writeDocumentToFile(doc, outputFilename);
}
```

Figure 3.12: Sample application using EmoNLG

The code shown above should be self-explanatory, since parts of it have already been shown in other examples.

References

E. André, *"Natural Language in Multimedia/Multimodal Systems"*, Handbook of Computational Linguistics , R. Mitkov, 650-669, Oxford University Press, 2003

R. Bhagat, E. Hovy, *"Trainable Reversible Language Analysis and Generation"*, 2005

B. de Carolis, C. Pelachaud, I. Poggi, M. Steedman, *"APML, a Mark-up Language for Believable Behavior Generation",* Life-like Characters: Tools, Affective Functions and Applications, H. Prendinger, Springer, 2004

C. Fillmore, *"Frame Semantics and the Nature of Language"*, Annals of the New York Academy of Sciences: Conference on the Origin and Development of Language and Speech, 1976

M, Fleischman, E. Hovy, *"Towards Emotional Variation in Speech-Based Natural Language Generation"*, International Natural Language Generation Conference, Arden House, 2002

J. Lee, S. Marsella, *"Nonverbal Behavior Generator for Embodied Conversational Agents"*, 6th International Conference on Intelligent Virtual Agents, 2006

W. Levelt, *"Speaking: From Intention to Articulation"*, MIT Press, 1989

E. Reiter, R. Dale, *"Building Natural-Language Generation Systems"*, Cambridge University Press, 2000