

# XtraGen — A Natural Language Generation System Using XML- and Java-Technologies

**Holger Stenzhorn**

XtraMind Technologies GmbH  
Stuhlsatzenhausweg 3  
66123 Saarbrücken, Germany  
holger@xtramind.com

## Abstract

In this paper we present XtraGen, a XML- and Java-based software system for the flexible, real-time generation of natural language that is easily integrated and used in real-world applications. We describe its grammar formalism and implementation in detail, depict the context of how the system was evaluated and finally provide an outlook on future work with the system.

## 1 Introduction

In this paper we present XtraGen, a recently developed software system for the flexible, real-time generation of natural language that can be easily integrated into real-world, industrial application environments through its open XML- and Java-based implementation and interfaces.

Our motivation for developing a completely new generation system started when we made the same observation as stated in the following quote:

There are thousands, if not millions, of application programs in everyday use that automatically generate texts; but probably fewer than ten of these programs use the linguistic and knowledge-based techniques that have been studied by the natural language generation community. (Reiter, 1995)

The goal of our company is to develop state-of-the-art software and hence we wanted to change the portrayed situation at least in our environment for the applications we create.

Therefore we started to experiment with XSL (World Wide Web Consortium, 2001) to generate natural language as suggested in (Cawsey, 2000) and (Wilcock, 2001). But we figured out fairly soon that XSL did not satisfy our needs and desires because with this mechanism

- we were not able to appropriately handle the issue of morphology,
- we could not parameterize the generation process at the desired level and
- we had no possibility to generate alternatives or recover from dead ends during generation since XSL is lacking a backtracking mechanism.

Therefore we decided to develop our own natural language generation system that incorporates on the one hand many ideas found in XSL but on the other hand tries to give a solution the above described problems.

## 2 Generation Grammars

### 2.1 Formalism

The grammar formalism conceived for the XtraGen system has been developed from an application-oriented point of view. This means that from our standpoint real-world applications hardly ever require a full and complete linguistic coverage which is striven for by linguistically motivated generation systems. Therefore our formalism is based on extended templates that allow the inclusion of predefined and dynamically retrieved text, constraint-based inflection and a context-free selection mechanism. The development of this formalism was strongly influenced by the ideas found in the (Lisp-based) formalism of the TG/2 system (Busemann, 1996; Wein, 1996) and the YAG system (Chanarukul, 1999).

A template has the overall form as depicted in the Backus-Naur Form in figure 2.1. Each part of the template will be elaborated below.

### 2.2 Conditions

Conditions describe the exact circumstances under which a certain template can be applied and its actions executed. There are two distinct basic types

```

<template id="String"
          category="String">
  <conditions>
    Condition*
  </conditions>
  <parameters>
    Parameter*
  </parameters>
  <actions>
    Action+
  </actions>
  <constraints>
    Constraint*
  </constraints>
</template>

```

Figure 1: Overview of a XtraGen template in Backus-Naur Form

of conditions: Simple-Conditions and Complex-Conditions. They in turn are the supertypes for more specific conditions:

**Simple-Condition** They form the actual tests that are applied to the input structure. A set of commonly used conditions is already predefined such as ones that test for equality or that test whether certain information is existent in the input structure. If there is a need for some very specific conditional testing that cannot be realized with the existing means a developer is free to implement and add its own conditional types.

**Complex-Condition** This type of condition makes it possible to combine several conditions into a more complex one. Three predefined Complex-Conditions exist: the And-Condition, the Or-Condition and the Not-Condition. Additional Complex-Conditions can also be added by providing an implementation for them.

### 2.3 Parameterization

Parameterization is an easy and flexible means to guide and control the generation process with regard to different linguistic preferences such as matters of style or rhetorical structures. Parameterization works by introducing a preference mechanism that provides the possibility of dynamically sorting the application of templates according to a given set of parameters.

```

<conditions>
  <or>
    <and>
      <condition type="equal">
        <get path="/recall"/>
        <value>95</value>
      </condition>
      <condition type="less">
        <get path="/accuracy"/>
        <value>90</value>
      </condition>
    </and>
    <not>
      <condition type="exist">
        <get path="/exception"/>
      </condition>
    </not>
  </or>
</conditions>

```

Figure 2: Example of some complex interleaved conditions

The way parametrization works in our system is a two-step process:

**Adding of parameters to templates** During the design of a generation grammar the writer adds one or more parameters to some templates as in the example in figure 3.

Here the upper template is intended to be used during the generation of text targeted at experts and the lower one in case text is to be produced for novices (*level* is *expert* in one template and *novice* in the other). Both of the templates are preferably used when a low verbosity level is desired (*verbosity* is *low* in both cases).

**Setting of the parameters at runtime** At runtime the parameters corresponding to the ones defined in the grammar are set to the desired values. To continue our example, we now set the value of the parameter *level* to *expert* (see figure 4) and hence the template in the upper box would be selected.

The particularity of our system is that parameters can be assigned a weight and thus a priority. In our example we might want to give a higher priority to the parameter *level* than to the parameter *verbosity* as shown in figure 5 This now sorts the application of templates in a way that they are first sorted according to their level of verbosity and

the result is further sorted according to the level of expertise.

```
<template id="explainExpert"
  category="explain">
  <parameters>
    <parameter name="level"
      value="expert">
    <parameter name="verbosity"
      value="low">
  </parameters>
  ...
</template>
```

```
<template id="explainNovice"
  category="explain">
  <parameters>
    <parameter name="level"
      value="novice">
    <parameter name="verbosity"
      value="low">
  </parameters>
  ...
</template>
```

Figure 3: Example of using parameters on the level of generation grammars

```
generator.addParameter("level",
  "expert");
```

Figure 4: Example of using parameters on the level of programming code

```
generator.addParameter("level",
  "novice",
  0.75);
generator.addParameter("verbosity",
  "low",
  0.5);
```

Figure 5: Example of using parameters with a weight specified on the level of programming code

## 2.4 Actions

In the case that all conditions of a given template have been tested successfully (see section 2.2) the actions contained in the actions-part of the template are executed.

There are four different types of actions that can appear: String-Action, Getter-Action, Inflection-

Action and Selection-Action. The actual purpose of each of them is quite different but all of them return a result string when executed successfully.

**String-Action** This type of action simply returns a statically specified string as a result — a so-called canned text.

**Getter-Action** With a Getter-Action it is possible to directly access and retrieve data from the entered input structure. The syntax used for specifying the path to the data conforms to the syntax of XPath (World Wide Web Consortium, 1999). There is no additional processing done on the returned data.

```
<get path="/values/startTime"/>
```

**Inflection-Action** This action inflects a stem according to the defined morphological constraints and returns the result.

The stem can be stated statically in the grammar as in case (a) or can be dynamically retrieved from the input structure as in case (b).

The needed morphological constraints are furnished by the constraints-part of the template to which the given label provides a link (confer to section 2.5 below for details).

```
(a) <inflect stem="bring"
  label="X0"/>
```

```
(b) <inflect path="/action"
  label="X0"/>
```

**Selection-Action** The Selection-Action can actually be seen as the most important of the actions since it accounts for the context-free backbone of the system.

It allows to select another template directly via a specified identifier as in case (a) or via a given category as in case (b). In the second case several templates might have the given category and hence backtracking might be invoked at his point. (see section 5.1)

```
(a) <select id="top"/>
```

```
(b) <select category="top"
  optional="true"/>
```

Selections can also be declared optional as in (b) which means that in case the selection of the template fails no backtracking is invoked and simply an empty string is returned.

## 2.5 Constraints and Morphology

The treatment of morphology is naturally one of the major issues in the context of a complete natural language generation especially when working with morphologically rich languages such as German, Russian or Finnish. Therefore we took great care to design and develop a morphological subsystem that is powerful and flexible yet easy to understand and use. The actual realization of the component is based on a constraint-based inheritance algorithm that follows the example of PATR-II (Shieber et al., 1989).

In the (overly simplified) example in figure 6 one can get a glimpse on how the morphology works: There are two Selection-Actions, the first one labelled X0, the second one labelled X1. The given constraint now tells that the attribute `number` of X0 is the same as the attribute `number` of X1 and sets it dynamically to the value retrieved by the Getter-Action.

```
<template ...>
  <actions>
    <select category="determiner"
            label="X0"/>
    <select category="noun"
            label="X1"/>
  </actions>
  <constraints>
    <constraint>
      <place label="X0"
            attribute="number"/>
      <place label="X1"
            attribute="number"/>
      <get path="/categoryNumber"/>
    </constraint>
  </constraints>
</template>
```

Figure 6: Example of using constraints

## 2.6 Compilation

In order to be able to work with a generation grammar the generation engine requires the grammar (and its templates) to exist in the form of a Java object. But since the original format of the grammar is plain XML this format must be transformed through a compilation process into the internally needed representation. Our system is capable to perform such a compilation in two different ways:

**Just-in-time Compilation** With this technique the required templates are compiled from their

XML source into their corresponding Java objects at runtime of the generation engine, i.e. during the actual generation process. This type of compilation is advised only for smaller grammars or during the development and testing of a new grammar since the constant interleaving of compilation on the one hand and the actual generation process on the other leads to some quite noticeable overhead. This overhead is naturally not acceptable when XtraGen is used in real-time applications.

**Pre-Compilation** This type of compilation allows to compile the whole grammar before its actual deployment during the generation process. The pre-compilation of grammars can improve the performance of the generator-engine tremendously and is therefore to be preferred in most situations. (The pre-compilation of generation grammars is very similar to the Translets approach in XSL (Apache XML Project, 2002) where XSL stylesheets are compiled in advance into Java objects.)

## 3 Input

In contrast to other generation systems that require their input to adhere to some particular (and most of the time proprietary) encoding format the core engine of our system only demands its input to be a valid XML structure.

The actual restrictions on the input are imposed only at the level of the generation grammars in terms of their access to the input (see section 2.4 on Getter-Actions and Inflection-Actions). This can obviously lead to a severe drawback: In case that either the generation grammar or the input structure changes heavily there might emerge a complete mismatch between the XPath specified in the actions and the actual structure of the input.

Under circumstances when it is not feasible to change either the input structure or the grammar, we propose to introduce an additional mapping layer between input and generator that is based on a XSL stylesheet and that dynamically maps the input in the way that is needed by the grammar.

## 4 Editor

We have stressed in the sections before that we believe our formalism to be powerful yet very straightforward to implement and use. But when developing larger grammars for real-world applications it becomes quite a demanding, non-trivial task to keep

track of all the templates and especially of the relations between them (e.g. relations on the level of morphological constraints) Common XML editors are of no help at this point since they cannot show such relations at all.

Therefore the development of egram, a Java-based graphical editor for generation grammars is on its way at the site of our cooperation partner DFKI (German Research Center for Artificial Intelligence). After the completion of its development this tool will allow to comfortably edit all aspects of generation grammars and templates. Among many other things the editor will be able to depict the whole generation grammar and process in a graphical tree format in which dependencies between templates are shown in an intuitive way.

## 5 Implementation

The realization and implementation of the XtraGen system is based entirely on the two cornerstones Java and XML.

XML was chosen because it has become the de-facto standard language in many (if not most) scenarios where information transfer takes place. This in turn is caused by its unique capabilities to encode information in a way that is easy to read, process, and generate (even for human beings as in the case of our formalism).

Java was chosen because it provides many mechanism to bolster the productivity of a programmer during the development of new software with such things as an extensive programming interface or automatic memory management for example. An additional advantage of Java is the availability of many free and readily usable open-source packages that provide a host of diverse functionalities. The most important ones in our project were the different XML packages and in particular the XML parser Xerces or the XSLT engine Xalan (Apache XML Project, 2002).

### 5.1 Backtracking

During the generation process it is possible that two different templates are applicable at the same time (i.e. they have the same category and all of their conditions are satisfied). Now if one of the templates is selected this leads to one of two different results:

- The application of the template was successful which means that all of its actions could

be successfully executed and a result was returned.

- The application of the template failed because the execution of one or more of its actions was not successful.

But the failure of a template described above does not mean that there exist no solution at all. Therefore we backtrack to the point where the unsuccessful template was selected and apply another template. This procedure is repeated until there are no more templates at this backtrack point.

The underlying implementation of the backtracking mechanism is quite elaborated since it has to take several important issues into account, the most important ones are:

**Performance issues** We implemented several different mechanisms that help to tremendously enhance the performance during the backtracking phase such as the memorization of partial solution.

**Constraint issues** We had to take great care of the constraint inheritance mechanism during the backtracking implementation so that an invocation of backtracking does not lead to a misguided percolation of constraints and hence a corrupted morphology.

### 5.2 Programming Interface

So far we have talked about the deployment of XtraGen only on the level of generation grammars and their XML-based formalism. Now we turn to the description of the tasks that have to be undertaken on the level of programming code to make the system run.

The following shows the individual steps that are be taken to generate some output with XtraGen:

**Creating a new generator-engine** The very first thing to do in order to get the whole system running is to create a new `Generator` object which represents the core generation engine:

```
Generator generator =  
    new Generator();
```

By doing so one implicitly creates objects for the internal subcomponents such as the already mentioned morphological component and puts them under the control of the generation engine.

**Setting the start-category or -id** The generation engine needs to know which template it should start from. This is done by specifying either a start-category as in case (a) or a start-identifier as in case (b).

```
(a) generator.setStartCategory(  
    String category);
```

```
(b) generator.setStartId(String id);
```

**Setting the grammar** Without a generation grammar it would naturally be impossible to generate any output at all. There are two different possibilities to pass a grammar to the generation engine:

```
(a) generator.setGrammarDocument(  
    Document grammar);
```

```
(b) generator.setGrammar(  
    Grammar grammar);
```

In the first case a `Document` object (World Wide Web Consortium, 2000) that contains the grammar in parsed XML-format is passed, in the second case a pre-compiled `Grammar` object is passed. (see section 2.6)

**Setting the input** In addition to the grammar the generation engine needs an input structure to generate from. This can be set as follows:

```
generator.setInputDocument(  
    Document input);
```

Again, the parameter passed is a `Document` object that contains the input in parsed XML format. The input can be reassigned between two calls to the generation engine.

**Setting parameters** In subsection 2.3 we talked about the use of parameters to control and guide the generation process. The way parameterization works is explained in detail there. To set parameters at runtime one has to add the following methods:

```
(a) generator.addParameter(  
    String name, String value);
```

```
(b) generator.addParameter(  
    String name, String value,  
    double weight);
```

This step is only needed if parameterization is desired. Otherwise these methods can be omitted and parameterization is turned automatically off.

**Run the generation process** To now actually start the generation process and get some output, one of the following calls can be used:

```
(a) String result =  
    generator.generate();
```

```
(b) Document result =  
    generator.generateDocument();
```

The difference between the two calls is that in case (a) a simple `String` containing the result is returned whereas in the case (b) a `Document` object is passed back.

## 6 Evaluation

At the end of a software development phase any newly created system must proof in an evaluation phase whether it reaches its predefined goals. (Melish and Dale, 1998) This is especially true in an industrial context with commercial applications as in our case.

The context for the evaluation of XtraGen was provided by X-Booster (Beauregard et al., 2002) which is another project at our site that was concurrently developed with our system. This system is an optimized implementation of Slipper (Cohen and Singer, 1999), a binary classifier that induces rules via boosting and combines a set of those classifiers to solve multi-class problems. It was the goal to successfully integrate XtraGen into this system.

The motivation behind this is based on the fact that common classification systems are quite non-transparent in regard to their inner workings. Therefore it becomes rather difficult to understand how and why certain classification decisions are made by those systems.

We departed at exactly this point: XtraGen was to be used to automatically generate texts that explain the learning phase of the X-Booster and hence make the classification more transparent. As an additional “gadget” we wanted to create the explanations in all the languages that are spoken at our company site: English, German, French, Italian, Russian, Bulgarian and Turkish.

### 6.1 Integration Tasks

As the very first task of the integration procedure we needed to answer the question what to actually output to the user and in which exact format this output should be. We decided on producing a description of the complete learning phase with two kinds of output texts: One targeted at experts and one for

novice users. The format chosen for the final output was HTML.

Now we needed to adapt the code of X-Booster slightly to make the meta data about the learning phase accessible from the outside. To do so we wrote some small methods that simply returned in XML format the meta data which were only stored in internal variables up to this point.

The next step was to add to X-Booster's own code the code for calling the generation engine and for transforming the result into the final output-format. This was done as described in section 5.2.

Finally (and most importantly) the generation grammars for the different languages were developed. This happened in the way that we first set-up a prototypical grammar for English which we tested extensively. Then in a second step the grammars for the other languages were modelled according to this exemplary one. For this we worked together with different native-speaking colleagues that translated the original English grammar into their language.

## 6.2 Sample Template and Output

Figure 7 below shows a small portion of the output returned after running X-Booster together with the integrated XtraGen on a given training set. This result was produced by using the English generation grammar and the parameters set for producing texts targeted at novice users.

*The number of documents is 37, divided into 2 different categories. The results have been produced using 3 fold-cross-validation which means that the data-set is divided into 1/3 test-set and 2/3 training-set. The learner is trained on the training-data only and evaluated on the test-set which has not been presented before. We repeat this process 3 times on non-intersecting test-data. The overall result is then computed finally as the average of all performed tests. The average accuracy reaches 100.0% which is achieved by applying 5 rules.*

Figure 7: Sample output from X-Booster

Figure 8 shows parts of the template that produced the second paragraph of this sample output. The template is complete in the sense that all different aspects of a template are exposed and it is

only simplified in the respect that similar parts of the template are left out.

```
<template id="foldNumberNovice"
  category="foldNumber">
  <conditions>
    <and>
      <condition type="exists"/>
      <get path="/foldNumber"/>
    </condition>
    ...
  </and>
</conditions>
<parameters>
  <parameter name="level"
    value="novice"/>
  ...
</parameters>
<actions>
  ...
  We repeat this process
  <get path="/foldNumber"/>
  <inflect stem="time"
    label="X0"/>
  ...
</actions>
<constraints>
  <constraint>
    <place label="X0"
      attribute="number"/>
    <get path="/foldNumber"/>
  </constraint>
  ...
</constraints>
</template>
```

Figure 8: Sample template from X-Booster

## 7 Outlook and Future Work

Because the above described evaluation proofed to be quite successful it was decided to further deploy XtraGen at our site and to integrate it into new products and projects. One of the first of these projects is Mietta-II (Multilingual Information Environment for Travel and Tourism Applications), an European Commission funded industrial project with the goal of developing a comprehensive web search and information portal that specializes on the tourism sector. (Additional application scenarios are already envisaged for a possible later stage of the project.) In this environment we will apply natural language generation to produce texts and messages for various types of media such as dynamically generated web pages, paper leaflets, hand-held devices and

in particular cellular phones. For the last of those media we are exploring the possibility of producing voice-enabled output with a dedicated voice server that is based on VoiceXML (World Wide Web Consortium, 2002) or JSML (Sun Microsystems, 1999). We are experimenting at the moment with the different possible outputs and on how these outputs can be encoded in a generation grammar enriched with VoiceXML or JSML tags.

## 8 Conclusion

In this contribution we presented XtraGen, a real-time, template-based natural language generation system designed for real-world applications and based on standard XML- and Java-technologies. We described in detail the different aspects of its generation grammars with an emphasis on their formalism. Then we covered architectural and implementational issues of the system. After depicting the evaluation done for XtraGen we concluded with presenting a new project where the system is used and where new ideas are experimented with.

## Acknowledgements

We are grateful to Sven Schmeier, Huiyan Huang and Oliver Plaehn for their collaboration on X-Booster and the Mietta-II project and especially to Stephan Busemann for many fruitful discussions on TG/2. This work was carried out in the Mietta-II project funded by the European Commission under the Fifth Framework Programme IST-2000-30161.

## References

- Apache XML Project. 2002. Apache XML Project Website. <http://xml.apache.org/>, June.
- Stéphane Beauregard, Huiyan Huang, Karsten Konrad, and Sven Schmeier. 2002. Industrial-Strength Text Classifiers. In *Proceedings of the Fifth International Conference on Business Information Systems (BIS2002)*, Poznań, Poland.
- Stephan Busemann. 1996. Best-first surface realization. In *Proceedings of the 8th. International Workshop on Natural Language Generation (INLG '96)*, pages 101–110, Herstmonceux, England, June.
- Alison Cawsey. 2000. Presenting tailored resource descriptions: will XSLT do the job? In *Proceedings of the Ninth World-Wide Web conference (WWW9)*.
- Songsak Channarukul. 1999. YAG: A Natural Language Generator for Real-Time Systems. Master's thesis, Department of Electrical Engineering and Computer Science, University of Wisconsin-Milwaukee, December.
- William W. Cohen and Yoram Singer. 1999. A Simple, Fast, and Effective Rule Learner. In *Proceedings of the Annual Conference of the American Association for Artificial Intelligence (AAAI-99)*, pages 335–342.
- Chris Mellish and Robert Dale. 1998. Evaluation in the Context of Natural Language Generation. *Computer Speech and Language*, 12(1):349–373.
- Ehud Reiter. 1995. NLG vs. Templates. In *Proceedings of the Fifth European Workshop on Natural Language Generation*, pages 95–105, Leiden, The Netherlands, May. Faculty of Social and Behavioural Sciences, University of Leiden.
- Stuart M. Shieber, Gertjan van Noord, Robert C. Moore, and Fernando C. N. Pereira. 1989. A Semantic-Head-Driven Generation Algorithm for Unification-Based Formalisms. In *Meeting of the Association for Computational Linguistics*, pages 7–17.
- Sun Microsystems. 1999. Java Speech API Markup Language Specification . <http://java.sun.com/products/java-media/speech/forDevelopers/JSML/>, October.
- Michael Wein. 1996. Eine parametrisierbare Generierungskomponente mit generischem Backtracking. Diploma thesis, Department of Computer Science, University of the Saarland, Saarbrücken.
- Graham Wilcock. 2001. Pipelines, Templates and Transformations: XML for Natural Language Generation. In *Proceedings of the first NLP and XML Workshop; Workshop session of the 6th Natural Language Processing Pacific Rim Symposium*, Tokyo, November.
- World Wide Web Consortium. 1999. XML Path Language (XPath) Version 1.0. <http://www.w3.org/TR/xpath>, November.
- World Wide Web Consortium. 2000. Document Object Model (DOM) Level 2 Core Specification Version 1.0. <http://www.w3.org/TR/DOM-Level-2-Core/>, November.
- World Wide Web Consortium. 2001. Extensible Stylesheet Language (XSL) Version 1.0. <http://www.w3.org/TR/xsl>, October.
- World Wide Web Consortium. 2002. Voice Extensible Markup Language (VoiceXML) Version 2.0. <http://www.w3.org/TR/voicexml20>, April.