The Applicability of Common Generative Techniques for Textual Non-Code Artifact Generation

Johannes Müller Ulrich W. Eisenecker

University of Leipzig
Information Systems Institute
Marschnerstraße 31, 04109 Leipzig, Germany
http://www.iwi.uni-leipzig.de
{eisenecker, jmueller}@wifa.uni-leipzig.de

Abstract

Configuration or generation of software artifacts is a widely adopted approach to implement software system families e.g. by applying the generative software development paradigm. The generation of artifacts not directly related to software but rather related to a delivered software product is not widely examined. This paper discusses the applicability of three well-known software artifact generation techniques to natural language textual non-code artifacts. Therefore an overview of these techniques and adequate tools is given. The frame technology with the adaption and the abstraction concept and and the template approach of the model driven software development are examined. The tools *XFramer*, *XVCL* and *openArchitectureWare* are used to evaluate these techniques by implementing an exemplary toy use case. The experience gained by implementing the use case is presented. The three selected tools are compared with respect to the task to generate natural language texts as non-code artifacts.

Categories and Subject Descriptors D [2]: m

General Terms GSE, Non-code artifact

Keywords frame, MDSD, XVCL, XFramer, oAW

1. Introduction

Generative software development (GSD) [6] is one widely accepted approach to develop software intensive systems within a software system family amongst others such as model driven software development (MDSD) and aspect oriented software development (AOSD). There are mainly two development processes for software system families, namely *domain engineering* (development for reuse), and *application engineering* (development with reuse).

A software intensive system comprises more artifacts than only source files or executable program files, e.g. user-manual or man-pages but also graphics, sounds, animations, test data and so on. Such artifacts are subsumed under the term *non-code artifacts*. A special kind of such non-code artifacts are natural language textual non-code artifacts. These kinds of non-code artifacts will be subject of this paper.

When the generative paradigm is applied to develop a system family, the elementary reusable components of the solution space should be maximally reusable and minimally redundant, as the authors point out in [6]. The features of a system family member can be described by a *domain specific language* (DSL) which belongs to the problem space. The *configuration knowledge* is required for mapping the specification of a system family member to the solution space by means of a (configuration) generator. These components form the generative domain model (GDM).

The term *generator* covers not only facilities to generate code artifacts, but also to configure and parameterize available components which have well defined, asserted and possibly tested qualities. An exemplary realization of such a configuration generator in C++ could apply template meta-programming. Obviously this technique is not really suitable for generating non-code artifacts. There are other techniques which are more appropriate to generate non-code artifacts. One approach which explicitly aims to synthesize, beside code-artifacts, non-code-artifacts is the AHEAD tool suite [4]. The ancestor of it is the GenVoca approach [5], which is a methodology for creating system families.

Other approaches, which also use GenVoca as architectural style, are described as *technology projections* of the GSD. A *technology projection* is a mapping of the GDM to a specific technique, platform or programming language (see [6] for details). Some of them are also able to generate non-code artifacts, namely the projection to the adaption and the abstraction concept of the frame technology as described in [10] and the projection to the generator-framework openArchitectureWare [14]. The benefit of using one of these approaches to generate non-code artifacts is, that, if the stated approaches are used to realize a system family, one can use the existing environment to also generate non-code artifacts: No further tools are required. The paper will examine the ability of these three projections to generate non-code artifacts by using the theoretical background given by the specification of the projections.

A member of a system family may require a considerable number of related documents, e.g. program-documentation, instructions for manual system tests, customized licensing agreement or a contract between software supplier and customer to account a created software product. The listed documents raise distinct demands for a text generation system. Documentation, for example, could be created by simply assembling the specific documentation of the miscellaneous components, the system is built from. A contract on the other hand requires the creation of parts of a sentence, which are differently composed depending on the distribution model for the software system and the assembled components, to get a well-defined content. An example of a contract generation system can be found on [19].

The preceding example, the contract generation, could be achieved by *natural language generation* (NLG), a field of research in the area of artificial intelligence and computer linguistics [20, p. 1]. Its aim is to create natural language texts out of non-linguistical representations—e.g. out of a database. Another approach is to create texts by assembling text components. In [20, p. 4] the thesis is stated that every functionality which is realized by a NLG-system can also be realized by assembling text components. For this reason this paper focuses exclusively on the text-assembly approach, while NLG-techniques will be not pursued.

A human readable document must be well presented to make the contained information easily accessible to human readers. To format the output of a generation run, amongst others, there are the following techniques available:

- Tag the document by LaTeX commands. A TeX processor translates the tagged document into a human readable form in natural language.
- Tag the document by XHTML[1] commands and format it with CSS[2].
- Create a XML[3]-document and format it with XSL-FO[4].

If the approach with the LaTeX tags is used, the document creation process is a multi level generation process. At a first stage, the document with LaTeX tags is created. At a second stage, a TeX processor transforms the representation created before into a document ready for press. Because of that, LaTeX sometimes is referred to as a DSL for the domain of document creation [16].

2. Techniques for Text Generation

Not all technology projections to the generative software development paradigm which are useful to generate software artifacts are useful for generating non-code artifacts. Applying template meta programming in C++ for example is an effective way to create configuration generators to configure software components in C++. In fact this technique uses the features of the C++ language and requires adequate components for assembly. Therefore it can not be reasonably used to generate natural language texts. Furthermore a technique for non-code artifact generation must provide a possibility to modularize text blocks to handle potentially complex non-code artifacts within a system family. These prerequisites are provided by the adaption and the abstraction concept of the *frame* technology [2] and the template approach of *model driven software development* used to transform a model to code [8, 14].

2.1 Frames

The idea of *frames* was developed in the research area of artificial intelligence. In the seminal work "*A Framework for Knowledge Representation*" [13] Marvin Minsky explains a system to describe the mental processing of concepts appearing in the real world.

A frame defines constant values for a concept. These constant values are part of all instances of a frame. Beside this a frame contains also variable parts which are organized in so called *slots*. A slot of a frame can be either an instance of an other frame or finally a terminal value. Because of this relationship between frame instances a complex frame hierarchy can be composed which is useful to represent or analyse concepts of the real world [9, p. 120].

[1] XML Hypertext Markup Language

[2] Cascading Style Sheets

[3] Extensible Markup Language

[4] Extensible Stylesheet Language-Formating Objects

In 1987 Bassett describes in his seminal work [2] at the first time the usage of frames to foster reuse of software components. Independently of this work the company *Delta Software Technology* developed a technology to generate software components based on the idea of Minsky.

In [11, p. 55] the approach of Bassett is called the *adaption concept* and the solution of *Delta Software Technology* is called *abstraction concept*.

2.1.1 Adaption Concept

In the adaption concept frames are stepwise specialized. From general frames more special frames will be assembled. Higher level frames *adapt* lower level frames. The more general frames are customized to the circumstances and requirements of the more special frames [3, p. 88]. A frame will be changed on its slots, which are the variation points of the frame hierachy. A frame has default values for variation points, so an adapting frame must only change the slots if there are special requirements which differ from the defaults [3, p. 89]. Furthermore a frame at the adaption concept did not have any mutable state. So its processing compares to the processing of variables at the functional programming paradigm.

2.1.2 Abstraction Concept

Within the abstraction concept frames are not part of an other frame, rather they are instantiated. The created instances will be referenced from other frame instances. In this way a hierarchy of frame instances is constructed. It is possible to instantiate a frame more than once. Every instance gets its own set of slot-values and references to other frames. This feature of the abstraction concept resembles the class/instance-scheme of object-oriented programming. Thereby the abstraction concept is similar to object-oriented programming.

2.2 Templates

The MDSD aims at automatic generation of executable software out of a formal model [24, p. 11]. A formal model represents rules which make a statement about the models meaning. In this context a model could be a class diagram in the *Unified Modeling Language* (UML). Moreover also other types of models e.g. textual models can be used. Automatic generation means that the source code is generated without manual intervention. No modifications may be applied to the generated source code because the model adopts the role of code.

Underlying every concrete model is a meta-model. A meta-model is a formal description of a domain in which a system is generated. It defines the abstract syntax of the models. The abstract syntax consists of the meta-model elements and their relation amongst each other [24, p. 29]. A transformation is described by means of a template language. A template consists of static text which contains tags on specific spots. During the generation process these spots will be assigned text according to a input model [24, p. 146]. The tags can contain additional instructions which are executed while the model is processed, e.g. to modify an input string. While templates are defined on the basis of a meta-model, the generated text will be created according to a concrete model. In this approach the templates are the reusable components of the solution space.

MDSD aims at generating code artifacts. Because code is generated mainly as text, it is also possible to apply MDSD for generating textual non-code artifacts.

2.3 Techniques not Examined

The techniques mentioned before are not the only options to generate text. Subsequently some other techniques are listed. Every technique has its own weakness which renders it not suitable for the specific purpose of text generation. In [24, p. 149ff] it is remarked that common programming languages, such as Java or C# can be used to create generators. However this idea is rejected, because of language related problems such as processing of strings which requires to use escaped sequences for special characters. Moreover, language structures which are not designed for text generation have to be used for the generator. Thereby it is hardly recognizable, how the result will be structured. In [24, 147ff] another alternative is discussed, namely the application of XSLT[5] to build a generator. The availability of *XPath* as a powerful navigation language is pointed out as a special advantage. *XPath* supports the navigation of basically every object structure. But an obvious disadvantage is the hardly readable syntax of XML [6], which has to be used to describe the transformation in XSL files. Preprocessors like those used in C/C++, can be used independently from the compiler as well [24, p. 143f]. Therefore it is basically possible to realize a text generator. A preprocessor is often used for small replacements of texts, for instance the value of a constant. A frequent consequence is that preprocessors are mostly not Turing-complete, they lack especially control structures for iterations. These are required in order to implement more complex generators. For this reason it is not practical to realize the generation of textual non-code artifacts with a preprocessor.

3. A Use Case for Text Generation

As use case to gain experience with the tools a family of documents is implemented. A document of these family accompanies a pizza and contains the following elements:

- The address and the title of the customer
- The price of the ordered pizza
- A list describing the ingredients. The list also contains a warning if an ingredient could be allergenic.
- A complimentary close customized to the order. Friends of chili get the spanish wish *¡buen provecho!*, garlic fans get the italian wish *Buon appetito* and if no or both extras are chosen, the close will be the english phrase *enjoy your meal*.

The document is generated to be suitable to the corresponding pizza. The example covers a wide range of imaginable non-code artifact generation scenarios. So one can gain experience with the utilized tools. First the price contains no static contents: the generator is able to compute all values. Second the list of ingredients consists of distinct text blocks which will be assembled with respect to a certain order. The third part, the complementary close, is a mixture of the both preceding variants. The closings will not be changed but a generator must choose the right text block with respect to the chosen extras.

4. Tools for Text Generation

In the last section, techniques for textual non-code artifact generation were examined. This section focuses on tools for implementing these techniques. There is more than on implementation for each of the preceding introduced techniques. Tools to realize one of the two frame concepts are presented in [11, p. 56]. The template approach is realized by MDSD tools. An overview of available tools is provided in [7, p. 622]. The evaluation of these techniques will be based on open source or freely available tools so that the following descriptions can be easily reproduced[7]. However it is important that the used tools adequately support the examined technique. This requirements are fulfilled by the following tools: For the abstraction concept *XFramer* is chosen and for the adaption concept *XVCL* is selected. To examine the template approach *openArchitectureWare* will be applied.

In addition to the aforementioned requirements, there are also the following properties of the tools expected, to realize a family of non-code artifacts:

- It must be possible to define a domain specific language.
- It must be possible to preserve a certain locality of the parts of the GDM to achieve a sufficient maintainability
- Control structures for processing text modules must be available.
- Debugging facilities must be available.
- It should be possible to externalize huge text blocks in order to improve the readability of the generator modules.
- It must be possible to exactly control the output of white spaces to get the expected output by the TEX-processor.
- Tools, i.e. editor, must be available for creating the text modules.

4.1 XFramer

XFramer is available from [23] as freeware for Linux and Windows for non-commercial purpose. It is used to extend the programming languages C++, C# and Java with the capability to process frames [11, p. 55] Nevertheless the compiler is still required to use it as a full frame processor. *XFramer* works as a preprocessor, which translates the frame definitions to valid source code of the language of the used compiler. Even if one of the languages C++, C# or Java is used to process the frames, it does not imply that only one of this languages can be used exclusively. In fact all textual representation—any programming language, HTML, XML or natural language are imaginable—of information can be processed by the tool [11]. The tool extends the supported languages with new elements.

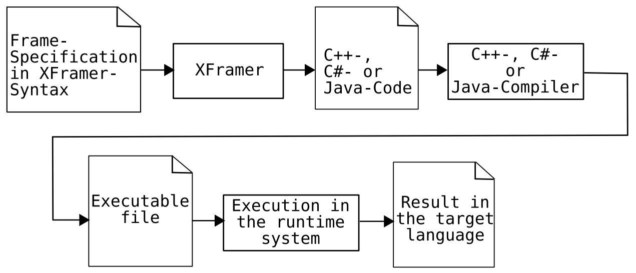
Frame-Specification in XFramer-Syntax
XFramer
C++-, C#- or Java-Code
C++-, C#- or Java-Compiler
Executable file
Execution in the runtime system
Result in the target language

Figure 1. Workflow when applying XFramer [11, p. 57]

Figure 1 depicts the flow of a XFramer run. The XFramer preprocessor reads frame specifications and transforms them into source code modules—classes in the case of the chosen host programming language. The output of the preprocessor is processed by the compiler of the used host language. In this step the real generator is created. This generator is executed in the runtime system and creates the result in the target language. XFramer creates the generator in the selected programming language, so it is fully amenable to the debugger of this language [10, p. 80]. In addition all libraries available in the host language can be used to build the generator. So it is e.g. imaginable to make the generator configurable by XML using a XML processing library like Xerces [10, p 80].

[5] Extensible Stylesheet Language Transformations

[6] Extensible Markup Language

[7] The complete source-files of all three implementations can be downloaded as a zip archive from [15]

4.2 XVCL

XVCL is an acronym for *XML-based Variant Configuration Language* and names a language to define frames according to Bassetts adaption concept. Frame definitions in XVCL are processed by the tool *XVCL processor* [12, p. 1]. Language, tool as well as a methodology for domain analysis are developed by the *National University of Singapore*. XVCL processor can be freely downloaded from [1] under the permission of the *GNU Lesser General Public License* (LGPL) version 2.1.

XVCL is a XML-based language and is shipped with a *Document Type Definition* (DTD). It is written in Java so it executes on many host systems. The tool is invoked with the specification frame and starts the generation process in which all frames directly or indirectly referenced by the specification frame are stepwise processed and assembled to one or more output documents. There lies one key difference to *XFramer*, which instantiates the frames, so the content can be changed during the generation process.

If the parameter `-B` is passed to the execution of *XVCL* superfluous white spaces are removed. Otherwise all blank lines of the frames are written into the output documents. Lines with XVCL commands become blank lines. An exemplary invocation of the tool is `java -jar xvcl.jar -B pizza-document.xvcl`. A detailed description of the XVCL-tool can be found in [18].

4.3 OpenArchitectureWare

OpenArchitectureWare (oAW) is a generator framework from the area of model driven software development. It covers a language family to check and transform models. The language *Xpand* is used to describe model to text transformations. The framework offers editors and plug-ins for the eclipse platform but it is also possible to use it independently [17]. It is available under the terms of the *Eclipse Public License* (EPL).

Within the subproject *Xtext* a tool is developed which allows to define the syntax of a DSL with a sort of *Extended Backus Naur Form* (EBNF). With this definition a model is generated which represents the abstract syntax tree (AST) of the language. In addition an editor for the eclipse platform is generated which assists the user with error checking, syntax highlighting and so on.

It is not possible to describe all features of the oAW project here. For details the reader is referred to the manual of oAW [8].

4.4 Comparison of the Tools

Essential properties of techniques and of tools for generating non-code artifacts in natural language are enumerated in section 2 and 4 respectively. These properties will be studied more closely with respect to the selected tools now.

Definition of the DSL *XFramer* uses a host language. Hence the only restriction implementing a DSL is given by the selected host language. As it is the case in the described example the DSL is embedded into the host language. If this approach is not powerful enough, it is imaginable to externalize the DSL and include a parser which reads the specification. *XVCL* on the other hand only allows to define a rudimentary DSL in the specification frame. In the implementation of the use case with *XVCL* multi-valued and single-valued variables are used to realize the DSL. Another approach could be to define a DSL as an XML lanugage and use XSLT[8] to transform a specification into a valid XVCL specification frame. The *oAW* solution in this paper is showing the realization of a textual DSL with the Xtext tool. It has the ability to create expressive DSLs. A specification in the DSL can be written in its own editor which has the capability to check the specification and highlight keywords.

[8] Extensible Stylesheet Language Transformations

Locality of the parts of the GDM With *XFramer* it is possible to define one configuration frame which contains all the knowledge to configure the elements of the solution space given a specification from the problem space. Another possible approach with *XFramer* is to define intelligent frames with each containing some specific part of the configuration knowledge and elements of the solution space. Thus some elements of the configuration knowledge can be reused but the locality of the components of the configuration knowledge degrades. Because of the stepwise adoption of the frames in *XVCL* only the second approach is feasible. The framework *oAW* is able to use model to model transformations. So the configuration knowledge is localized in the transformation rules. If the direct model to text generation approach is used the maintainability degrades.

Available control structures *XFramer* can use all the facilities of the host language, so very expressive solutions are possible. In contrast *XVCL* provides only a few basic commands which permit the definition of the logic, but they are much harder to use—also because of the XML syntax—than the control structures of a general purpose language. The Xpand template language provides also just rudimentary support for control structures. But it is possible to define the logic in a functional sub-language of the oAW project called Xtend which has the ability to call statically defined Java-methods. Thereby all the power of the Java language is available.

Debugging facilities If errors are detected during the generation process it would be helpful to use a debugger to understand the generation process. In fact *XFramer* uses a host language, hence the debugging facilities of this language are available. If there is any error at the generation process with the *XVCL* tool it produces error messages. Another approach to support debugging is to produce messages which the generator displays during the generation process. At present, other debugging facilities do not seem to be available. The *oAW* project contains debugging facilities to debug templates, workflows and transformations.

Externalize and modularize text blocks If huge text blocks are used, it would be useful to define them externally and reference them by a generator module. This is directly supported only by *XFramer*. Using *XVCL* there is no possibility to do so. Basically the same restriction applies to *oAW* but this functionality can be realized with the capability to use Java-methods. Related to this problem is the ability to modularize text blocks. All tools allow to distribute the modules over several files.

Control white spaces Even if the final document is typeset by $\LaTeX$ it is nevertheless important that the resulting document does not contain superfluous white spaces because they could have a meaning. At $\LaTeX$ e.g. an empty line results in a new paragraph in the target document. So the techniques must provide a facility to control the output of white spaces. Using *XFramer* all blanks are outputted as defined in the frame. So the output of the blanks can not be well controlled. The *XVCL* tool has the ability to remove superfluous white spaces but if there are intended blank lines they are also removed. The lines in the templates of *Xpand* which contains escaped commands will be removed if a minus is noted at the end of the escaped sequence.

Tool support The creation of text modules for one of the tool could be made more convenient by having any tool support, e.g. a text editor with syntax highlighting. At present, it seems that there is no editor available which assists the creation of frames for *XFramer*. But perhaps it is more adequate to use a common text editor which supports the selected host language. *XVCL* is a XML dialect so any XML editor can be used. Because a DTD is given, some editors can even perform syntax highlighting and code completion. *oAW* is well integrated into the eclipse platform. There are editors, wizards and other plug-ins available, which make the usage of the different languages convenient.

Table 1. summary of the comparison of the tools

Criteria	XFramer	XVCL	oAW
Defining DSLs	−	−−	++
Locality of parts of the GDM	++	−−	++/−[a]
Available Control structures	++	−−	++/−−[b]
Debugging facilities	++	+	+
Externalize text blocks	++	−−	+[c]
Modularize text blocks	++	++	++
Control white spaces	−−	+	+
Tool support	−−	++	++
Overall effort to implement the use case	−	++	+

(−−) bad to (++) good

[a] Good, if model to model transformation are used, otherwise bad.

[b] By using Xtend all possibilities of Java are available.

[c] By using Xtend and Java.

5. Conclusions

This work has analyzed three technology projections for their applicability to generate textual non-code artifacts in natural language. The techniques and tools which realize them were presented. The three tools are used to implement a use case. The three tools were compared with respect to the aforementioned criteria. As table 1 suggests, the toolset of oAW is well suited to realize the text generator. But it needs some effort to set up the environment to get the generator run (define a grammar, install the plugin and so on). The fastest way to implement the text generator of the use case provides XVCL. If a simple DSL suffices or there is an other way to configure the specification frame XVCL is a lightweight alternative to oAW. XFramer was somewhat harder to use then the other two approaches. But if complex decision logic is to be implemented, it can be an alternative because of the integration in a host language. To decide which tool is best suited for a given environment one must check the ease of integration in a present tool chain to generate system family members. Therewith it is possible to use one and the same DSL to specify the software system and the adequate textual non-code artifacts in natural language.

This paper only examines textual non-code artifacts in natural language. Another survey should reveal other relevant types of non-code artifacts. The result of this survey could be organized in a taxonomy of non-code artifacts. With such a taxonomy it would be possible to examine generation tools for all other types of non-code artifacts.

Real life use cases probably contain much harder requirements to textual non-code artifacts in natural language than the presented toy use case. To implement such requirements some ideas can be found in work related to NLG ([20], [21] and [22]).

As the implementation of the use case demonstrates, the conceptual framework of the generative software development [6] is also well-suited to generate textual non-code artifacts in natural language. With this aspect in mind, further technology projections specialized for generating non-code artifacts could be developed. As the usage of a multistage generation process in the use case demonstrates (e.g. XVCL and TEX processor), the usage of more than one tool to realize the generation is a promising approach. A further example would be to generate graphics by producing svg-files (which are text-based) and then use one of the *svg*-tools to convert it to a required file type.

This paper has shown that existing techniques and tools are applicable to generate (textual) non-code artifacts. To get a deeper insight the aforementioned next steps should be pursued.

References

[1] Xvcl download: `http://fxvcl.sourceforge.net`.

[2] P. G. Bassett. Frame-based software engineering. *IEEE Software*, 4(4):9–16, 1987.

[3] P. G. Bassett. *Framing Software Reuse: Lessons from the Real World*, volume 1 of *Yourdon Press Computing Series*. Yourdon Press, Prentice Hall, 1997.

[4] D. Batory. The road to utopia: A future for generative programming.

[5] D. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, 1(4):355–398, 1992.

[6] K. Czarnecki and U. W. Eisenecker. *Generative Programming Methods, Tools, and Applications*. Addison-Wesley, Boston, 2000.

[7] K. Czarnecki and S. Helsen. Feature-Based Survey of Model Transformation Approaches. *IBM Syst. J.*, 45(3):621–645, 2006.

[8] S. Efftinge, P. Friese, A. Haase, C. Kadura, B. Kolb, D. Moroff, K. Thoms, and M. Völter. *openArchitectureWare User Guide*. n.p., 1 edition, September 2007.

[9] U. W. Eisenecker and R. Schilling. Zum Entwickeln entwickelt. *iX*, 10:114 – 121, 2002.

[10] M. Emrich. Generative Programming Using Frame Technology. Diploma thesis, University of Applied Sciences Kaiserslautern, 2003.

[11] M. Emrich and M. Schlee. Codegenerierung mit XFramer und Programmiertechniken für Frames. *Objektspektrum*, 5:55 – 61, 2003.

[12] S. Jarzabek, P. Bassett, H. Zhang, and W. Zhang. Xvcl: Xml-based Variant Configuration Language.

[13] M. Minsky. A Framework for Representing Knowledge. Technical report, MIT, Juni 1974.

[14] J. Müller. Technikprojektion zur generativen Programmierung mit openArchitectureWare. Diploma thesis, University of Leipzig, Marschner Straße 31, 04109 Leipzig, Germany, May 2008.

[15] J. Müller and U. W. Eisenecker. Zip-archive with the sample code. `http://w3l.wifa.uni-leipzig.de/sw/public/ncag.zip`.

[16] N.A. Domain Specific Language. online, Oktober 2007. `http://c2.com/cgi/wiki?DomainSpecificLanguage`.

[17] N.A. openArchitectureWare. homepage, nov 2007. `http://www.openarchitectureware.org/`.

[18] N.A. Xml-based Variant Configuration Language (xvcl), 2007. `http://sourceforge.net/project/showfiles.php?group_id=58966&package_id=54953&release_id=305328`.

[19] N.A. Creative Commons. online, 2008. `http://creativecommons.org/license/`.

[20] E. Reiter and R. Dale. Building Applied Natural Language Generation Systems. *Journal of Natural Language Engineering*, 3(1):57–87, 1997.

[21] E. Reiter and R. Dale. *Building Natural Language Generation Systems*. Cambridge University Press, 2000.

[22] E. Reiter, S. Sripada, and R. Robertson. Acquiring correct knowledge for natural language generation, 2003.

[23] M. Schlee. Xframer 1.45 download: `http://www.geocities.com/mslerm/downloads.html`.

[24] T. Stahl, M. Völter, S. Efftinge, and A. Haase. *Modellgetriebene Softwareentwicklung. Techniken, Engineering, Management*. Dpunkt Verlag, Heidelberg, 2 edition, 2007.