UNIVERSITY OF TWENTE.

Faculty of Electrical Engineering,
Mathematics & Computer Science

SimpleNLG-NL: Natural Language Generation for Dutch

Ruud de Jong

M.Sc. Thesis
August 2018

Supervisors:
Dr. Mariët Theune
Prof. dr. D.K.J. Heylen

Human Media Interaction
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Summary

This thesis presents SimpleNLG-NL, a Dutch adaptation of SimpleNLG. SimpleNLG is a Java-based surface realiser, which performs the last step in Natural Language Generation. Natural Language Generation is the process of transforming non-linguistic information into understandable texts. With SimpleNLG-NL, developers and researchers can generate Dutch sentences based on dynamically generated input. SimpleNLG-NL was developed using an iterative process recreating sentences from a Wikipedia corpus. After four rounds, out of 86 sentences for which the input was manually written, a total of 75 (87.2%) could be generated in an acceptable manner. 69 of those were exact matches (80.2%). A proof-of-concept demonstrates that dependency trees can be automatically converted into input code for SimpleNLG-NL. In the current state, the proof-of-concept can only handle basic sentences, but the coverage can be increased by continuing the same development method. A more advanced version can be useful to quickly generate multiple variants of a sentence, for instance.

SimpleNLG-NL will be released as open source software. Several suggestions for further development are made.

Contents

List of Figures

List of Listings

List of Tables

Chapter 1

Introduction

Natural Language Generation (NLG) is the process of generating understandable texts from non-linguistic information using computational knowledge of the desired output language (Reiter & Dale, 2000). For example, an NLG system can take weather predictions in the form of numbers and transform that data into text that is understandable by humans. Creating text computationally has the advantages of being quicker than humans writing it manually, allowing for quick production of large amounts of text. It can also make sure that every requirement for the text is fulfilled, without human errors, although new problems may occur if the system is not accurate or complete. Because NLG systems use a non-linguistic representation of information as their starting point, output can be generated in multiple languages.

An NLG system can be made up of multiple steps, from planning the content to be described, all the way to applying the grammar rules of the output language (see Section 2.2). The first step, as described by Reiter and Dale (2000), is to plan the document: what information should be communicated and in what order? Next, the sentences have to be planned. The words to use have to be chosen, referring expressions, such as names and pronouns, have to be planned and in the case of multiple sentences being merged ('aggregation'), words may be left out or replaced. Lastly, the words are inflected and ordered correctly, together with punctuation being added. This final step, called 'surface realisation', results in an understandable sentence. It is this surface realisation that this thesis will focus on.

There are three main approaches for NLG: template-based, corpus-based and rule-based. Template-based NLG uses templates similar to the mail merge function in Microsoft Word: preprogrammed sentences have gaps that are filled dynamically to form complete sentences. The level of complexity of templates can differ. To a large extent, the document and sentence planning is performed by writing the template. Corpus-based NLG analyses a corpus of text and derives statistics from it. These statistics can be used for, for example, lexicalisation (choosing words) and surface realisation. Thirdly, the rule-based approach uses predetermined rules in all three steps. A rule can prescribe to leave the chance of snow out of a weather prediction if it is summer; the sentence planner may choose words based on readability; and the surface realiser uses programmed grammar rules to realise a sentence. This thesis focuses on surface realisation using a rule-based approach to NLG. A more detailed comparison between the three approaches can be found in Chapter 2.

There are a few NLG systems available, only some of which are open source. A few open source examples are NaturalOWL[1] (Androutsopoulos et al., 2013), OpenCCG[2] and KPML[3] (Bateman, 1997). All these systems try to accomplish multiple steps in the sequential process of NLG.

In contrast, SimpleNLG (Gatt & Reiter, 2009) is a so called surface realiser, taking the rule-based approach. It only performs surface realisation: the last step in the sequential process of NLG. When generating texts from non-linguistic information, the information about a sentence that has been generated by the sentence planner has to be realised, that is, the chosen words have to be put in the desired order and inflected according to the grammatical rules built into the surface realiser.

SimpleNLG is a surface realiser that was built to be simple to use. It is written in Java, which makes it usable on most operating systems. The input for SimpleNLG consists of a clause in the form of a variable of the class `PhraseElement`, to which the subject, verb and other elements can be added using Java methods like `sentence.setSubject("he")` and `sentence.setVerb("run")`. In this minimal example, the system would return "*He runs.*". A more detailed description of using SimpleNLG can be found in Section 3.1. Since its development in 2009, SimpleNLG has been

[1] `http://nlp.cs.aueb.gr/software.html`
[2] `https://github.com/OpenCCG/openccg`
[3] `http://www.fb10.uni-bremen.de/anglistik/langpro/kpml/README.html`

adapted for multiple other languages besides the original English, however, it has not been adapted for Dutch. Having a Dutch version would allow researchers and companies alike to realise sentences in Dutch created by their own sentence planner. Such a Dutch adaptation was developed during this research and will be called ‘SimpleNLG-NL’.

SimpleNLG-NL will probably be used in the POSTHCARD[4] project. The POSTHCARD project is developing a simulation of Alzheimer patients. Such simulations can be used as a training for caregivers and help them in their interaction with patients. While simulations for English and French will be built using a bilingual version of SimpleNLG, SimpleNLG-EnFr (Vaudry & Lapalme, 2013), the Dutch simulation will be using SimpleNLG-NL. Another project for which it may be useful is Data2Game[5], a project that researches the use of a serious game in a training for crisis teams. Such teams are formed by the regional emergency services and the municipality, a.o. The project group Data2Game intends to build a simulation of a crisis scenario, such as a derailed train releasing toxic gasses, which will train the personnel to handle the situation correctly. SimpleNLG-NL can be used to realise dynamically created content, such as fake news articles or tweets, which inform or purposefully distract the player.

The research described in this thesis tried to answer the following research questions:

R1: How can SimpleNLG be adapted for Dutch? This is the main question that will be answered using the answers to its subquestions.

R1.1: How can the subset of Dutch grammar to be implemented be determined? Dutch grammar is a large collection of rules and exceptions. A subset has to be determined to simplify the grammar to a complexity that is feasible to implement in SimpleNLG-NL. In this context, Dutch grammar’ covers phonology, morphology and syntax.

[4]`http://posthcard.eu/`

[5]`https://www.nwo.nl/en/research-and-results/research-projects/i/02/28502.html`

SimpleNLG-NL was developed using a process that iteratively increased the coverage of Dutch grammar in SimpleNLG-NL. Using the bilingual SimpleNLG-EnFr as a basis, sentences from a Wikipedia corpus were used as realisation targets. For each sentence, the input code for SimpleNLG-NL was written and the system was run to realise the input. The resulting sentence was compared to the target sentence and any differences were analysed. This analysis was then used to determine the Dutch grammar rules that needed to be implemented in SimpleNLG-NL. As the system was based on the French part of SimpleNLG-EnFr, the first sentences would use French grammar. As the process went along, the Dutch grammar became more and more prevalent. The development process is described in Chapter 4.

R1.2 "What parts of SimpleNLG have to be changed to implement Dutch grammar?" SimpleNLG is divided into modules, with each multiple classes inside. This research question explores the internal structure of SimpleNLG and what parts of it have to be changed to adapt it for Dutch. The structure of SimpleNLG is described in Chapter 3, while the specific files that were changed are noted in Section 5.2.

R1.3: How can SimpleNLG-NL be evaluated? Evaluating Natural Language Generation systems can be done in multiple ways. In the case of SimpleNLG-NL, it was evaluated during the iterations of development. After each iteration, the resulting generated sentence was manually checked by comparing it with the target sentence. A sentence was accepted as being correct if it met one of the correctness criteria described in Section 4.2. If the result did not match the target sentence exactly, but differed only in a way that kept the meaning intact (e.g. extra, unnecessary punctuation or word order), the sentence was accepted as correct.

R2: How can SimpleNLG-NL be used to generate sentence variations? The second main question is aimed at a potential use case of SimpleNLG-NL. It explores how scenario writers for serious games can use SimpleNLG-NL to writes sentence variants in a semi-automated fashion. The answer to its subquestion will be used to answer this.

After SimpleNLG-NL was built, a prototype was created to show a potential use case. This proof-of-concept consisted of a Java program that converts dependency trees of sentences into input for SimpleNLG-NL, allowing the user to generate multiple variations of the same input sentence.

These variations allow, for example, game copywriters to quickly pick a different variant of their sentence to increase variety. It can also be used to rewrite articles when the described future events have turned into past events.

R2.1: "How can parse trees be used with SimpleNLG-NL?" The answer to this question will provide a method of using parse trees with SimpleNLG-NL, which can then be part of the system answering question R2.

The proof-of-concept built after SimpleNLG-NL uses dependency trees to gain information about an input sentence. The user can feed it a sentence, which will be parsed by the Dutch Alpino parser (Bosch et al., 2007). The resulting tree, containing information about the words and their roles in the sentence, is read by a converter, which traverses the tree and writes the corresponding input for SimpleNLG-NL. SimpleNLG-NL then realises the input in multiple tenses, voices (active and passive) and forms (perfect).

This research consisted of two phases in accordance with the two main research questions. The first phase was used to build SimpleNLG-NL. The second phase was aimed at answering research question R2 by building a proof-of-concept system.

This document starts with describing Natural Language Generation, its uses and multiple approaches to it (Chapter 2). It then focuses on SimpleNLG in Chapter 3 to explain how it is used and its technical architecture. Also, adaptations for other languages, such as Spanish and German, are described. Chapter 4 explains the method used to build SimpleNLG-NL, while Chapter 5 describes the subset of Dutch grammar covered by SimpleNLG-NL and how it was implemented. The proof-of-concept that generates sentence variants is described in Chapter 6. The results of both phases are discussed in Chapter 7. Finally, the thesis is concluded in Chapter 8 and future work is suggested in Chapter 9.

Chapter 2

What is NLG?

This chapter describes what Natural Language Generation is and how it can be used. It also makes the distinction between three approaches to NLG. This chapter is based on work performed by the author in preparation of this graduation project.

2.1 Uses of NLG

NLG has been researched since the 1950s, and even more since the 1980s (Reiter & Dale, 2000, pp 19-20), but has only started to become commercially viable in the last decade. With the large amount of data collected by sensors and online user analysis, effective reporting of insights gathered from the data has become a greater and more difficult task for human workers. The time consuming task of writing understandable reports can now be (partially) automated by the use of NLG. NLG systems can take the data and turn it into information within minutes, if not seconds.

The simplest example, and one of the first examples, of a real-world use of NLG is generating weather reports with systems like FOG (Goldberg et al., 1994). The computer models used to forecast weather output large amounts of numbers. The NLG system then takes those numbers, extracts useful data from them and generates a text describing the most important points. With this technology, weather forecasts can be generated live and need not be (re)written manually by human forecasters. This saves time, but also makes sure the user gets the most up-to-date information, whenever he requests it.

Similar to weather reports are reports on business intelligence. Businesses are constantly tracking large amounts of metrics, ranging from sales numbers and client on-boarding to employee performance and website usage. To make use of this data, the companies write reports. Understandably, this process takes a lot of time and effort. This is where NLG comes in. Several companies (like Arria NLG[1], Automated Insights[2] and Yseop[3]) allow businesses to input all their data, from which a report is generated automatically. As with the weather reports, this has the advantages of saving time and effort, as well as having the very latest insights almost instantly.

Besides reports, NLG can also be used in direct user interaction. This can include chatbots[4], but also (video) games[5]. Implementing NLG in such systems allows the sentences to be different each time, making the user feel more like he is interacting with a human, instead of a robot with preprogrammed dialog. One step further would be to adapt the output based on user preferences and/or behaviour. This would require an extra step in choosing what to say and using which words.

Like many technologies, NLG, too, can be found in marketing. Phrasee[6] uses an unspecified NLG method to generate copy with the optimal amount of directness, friendliness and other marketing-specific ratios. The system uses machine learning to optimise email subject headings, email bodies and other texts. The company claims that their generated language outperforms human language more than 95% of the time.

2.2 Steps of NLG

There are three main approaches for NLG: rule-based, template-based and corpus-based. These approaches are described in Sections 2.3, 2.4 and 2.5, respectively. An NLG system has to perform multiple sequential tasks. Reiter and Dale (2000) divide the tasks into three modules: docu-

[1]https://www.arria.com/
[2]https://automatedinsights.com
[3]https://yseop.com/
[4]https://dialogflow.com/docs/dialogs
[5]http://botcolony.com/
[6]https://phrasee.co/

ment planning, microplanning and surface realisation.

1. Document planning
    - Content determination: Determining the information to be communicated in the text.
    - Document structuring: Structuring the story (by grouping and ordering information).
2. Microplanning/sentence planning
    - Lexicalisation: Choosing the words to use.
    - Referring expression generation: Planning the use of referring expressions (pronouns, names, descriptions, etc.).
    - Aggregation: Merging sentences to increase information density and readability. For example, two sentences with the same subject can probably be merged and the second mention of the subject can be left out.
3. Surface realisation
    - Linguistic realisation: Applying grammar, morphology and punctuation to the sentence structures and their words. This converts the abstract information into text.
    - Structure realisation: Adding mark-up (like HTML or TeX) for document presentation.

2.3 Rule-based NLG

As mentioned before, there are three main approaches to NLG: rule-based, template-based and corpus-based. This section describes rule-based NLG. During each step in this approach to the NLG process, rules determine the outcome. For instance, the document planner may decide to leave out information based on a rule that determines the relevancy of that content. Similarly, the sentence planner uses lexicalisation rules to choose the best fitting words. In the last step of the NLG process, the surface realisation engine will also use rules. Here, the set of rules describes the grammar and semantics of the target language. These rules are then applied to the words, based on the grammatical function they have each been given by the sentence planner.

This method has the advantage of the rules being domain independent: unlike grammar rules extracted from a context-specific corpus using statistics, the rules in the rule-based approach are universal and not tailored to just the systems context. The only limits are the systems knowledge of vocabulary (in the form of a lexicon) and the completeness of the grammar implemented in its rules. Another advantage is that, given great completeness of the grammar, the generated text will be grammatically correct. Although the use of proper grammar may result in texts that differ in writing style from spoken words, the result may be suitable for use in business reports and other documents requiring proper style.

One disadvantage is that applications may require specific rules or domain knowledge. More generally, each language requires its own set of rules to be implemented. Some individual grammar rules may be usable in other languages, but adding a new language requires a lot of work upfront. This thesis describes an Dutch adaptation of the SimpleNLG surface realiser, which required adding Dutch grammar rules.

2.4 Template-based NLG

A second approach is template-based: not unlike mail merge in text processors like Microsoft Word, template-based NLG uses a template that is populated with the required data, such as names. Some variables in the template are replaced by single words or short sentences, often with only simple inflection like plural rules applied. In the scientific research field, this is often performed on individual sentences, creating more advanced sentences by populating them with phrases, modifiers or other linguistic elements (Reiter, 1995).

The advantage of this approach is that a template is created easily, the author has great control over the final result and it is easy to instantiate variables and add new ones. Since templates are often written on the sentence level, the author takes the roles of both the document planner and the sentence planner. Computationally, populating a template is very cheap and quick, as there are often only a few variables to be changed. However, the downside of templates is the manual work that the author has to put into the creation and maintenance of the template. Another disadvantage is that, depending on the complexity of the templates and the number of templates, the results often use a similar structure and/or wording, therefore increasing the chances of a reader noticing that the text is generated automatically. Both of these disadvantages can be reduced by implementing more variables and more randomly chosen text elements (Deemter et al., 2005). However, this would increase the amount of manual labour during both creation and maintenance of the template. Research is being done on the automated extraction of template from corpora, which would reduce the manual work required, e.g. by Ell and Harth (2014).

2.5 Corpus-based NLG

Then there is corpus-based generation. Corpus-based NLG uses a body of text to generate statistical models that can be used to choose words (Gatt & Krahmer, 2018). This machine learning approach mimics the average writing style of the corpus. One simple approach would be to gather statistics on n-grams (a sequence of n words) in the corpus. With those statistics, one word can be given as input and the most common next word can be calculated. This has the downside that it only has a ‘memory’ of the last n words to base the choice of the next word on, which can result in ungrammatical sentences. Another approach is to use a neural network or a similar model that can also be used to plan the sentences. One advantage of using a corpus is the flexibility of the system: when (informal) language evolves, the text domain is changed or another language is chosen, the system can be updated by simply using a different corpus to train the model. The structure and word choice of the generated sentences will then be based on the new corpus. A disadvantage is that there is no guarantee for the grammatical correctness of the text, depending on the corpus and the machine learning method. The corpus would need to be checked for correctness and completeness of the used grammar and vocabulary (Reiter et al., 2003).

2.6 Conclusion

The NLG process can be divided into three steps: document planning, sentence planning and surface realisation. Each step requires input from the step above. Surface realisation is the last step in the process and it is the step on which this thesis focuses.

The surface realiser that will be adapted for Dutch is SimpleNLG, which uses a rule-based approach. This has the advantage of being domain independent. It requires only a lexicon that can be very general, or easily extended with domain-specific jargon.

When combined with (a document planner and) a sentence planner, SimpleNLG-NL may be useful for generating reports, dialog or other texts. This will be explored to answer research question R2.

Chapter 3

SimpleNLG

SimpleNLG is an open source project started in 2006 by Ehud Reiter (Reiter, 2016). It is a surface realiser library for Java, which allows it to be used in applications on many platforms. It takes the rule-based approach to NLG and uses a lexicon to get its information about words and their grammatical roles and inflections. The lexicon is a word list written in *Extensible Markup Language* (XML). As SimpleNLGs first use was describing medical baby data, the lexicon was partially based on the NIH Specialist Lexicon[1], which includes medical terms and phrases (Gatt & Reiter, 2009). The other part included general English entries. The resulting lexicon contains over 300,000 entries. This lexicon is still available for use in SimpleNLG, but by default, a much smaller lexicon with only 6314 entries is used. This much smaller file size (349 MB and 670 kB, respectively) decreases the memory load of SimpleNLG, making it more efficient.

Currently at version 4.4.8, SimpleNLG has seen several big changes. The biggest change came with version 4.0: by removing the dependency on a commercial library, SimpleNLG could be released under the Mozilla Public License[2], granting commercial use, too. With this new license, companies are free to adapt and use SimpleNLG and sell it or its results. On a code level, version 4 has brought modularization, described in Section 3.2, which makes it easier for outside developers to adapt the system to their needs.

[1]https://lsg3.nlm.nih.gov/LexSysGroup/Projects/lexicon/current/web/index.html

[2]https://www.mozilla.org/en-US/MPL/

The efficiency of SimpleNLG was evaluated by measuring the time it took to generate 26 summaries (Gatt & Reiter, 2009). Each summary was generated 100 times, to be able to measure timing accurately. This is only one way of assessing its performance, as will be shown in the sections describing adaptations of SimpleNLG, under Section 3.3. This chapter is largely based on work performed by the author in preparation of this document. The next section will describe how to use SimpleNLG.

3.1 Using SimpleNLG

As SimpleNLG is just a surface realiser, all previous steps of NLG have to be performed by another system. SimpleNLG requires input in the form of words and their role in the sentence that it is supposed to build. It is the task of the sentence planner to generate the input for SimpleNLG. SimpleNLG was designed to allow (a combination of) both canned and 'non-canned' text as input (Gatt & Reiter, 2009). Canned text is text that should not be inflected and should be printed as is, e.g. names. The 'non-canned' text will be subjected to the grammar rules programmed into SimpleNLG. The developer using SimpleNLG will have to make sure that the sentence planner output is written in or mapped to SimpleNLG input.

At the very least, every sentence in SimpleNLG needs a subject and a verb. A sentence is started using the `createClause()` method and is stored in a variable. This variable has methods to set its subject and verb. A demonstration of this minimal code can be found in Listing 3.1. The first three lines of code describe the initialisation of the lexicon, factory and realiser. The lexicon is used to get information on word categories, irregular inflections, usage and more. The factory is used to create the sentence elements, e.g. the `SPhraseSpec` that is used as a variable class for clauses. The realiser is used to actually perform the realisation of the sentence. Note that it defaults to using the present simple tense.

```
// These first lines are required regardless of the sentence
final static Lexicon lexicon = new XMLLexicon();
final static NLGFactory factory = new NLGFactory(lexicon);
final static Realiser realiser = new Realiser();

// Start the creation of a new clause
SPhraseSpec sentence = factory.createClause();

// Add the subject and verb to the sentence
sentence.setSubject("Marie);
sentence.setVerb("run");

// Realise the sentence and print the results
String output = realiser.realiseSentence(sentence);
System.out.println(output);

// OUTPUT: Marie runs.
```

Listing 3.1: Example of the minimum input required for SimpleNLG. It requires a subject (e.g. *Marie*) and a verb (e.g. *run*). These are added to a clause created with `factory.createClause()` Excluded are the import statements that import the library.

A slightly more advanced example of input code involves using temporary variables of types `NPPhraseSpec` and `VPPhraseSpec`, see Listing 3.2. These temporary variables allow the user to alter the phrase after initialisation by adding features, modifiers or other elements. In this example, the noun phrase *experiment* was extended with a determiner using the `NPPhraseSpec` method `setDeterminer`. In the example, a clause, a noun phrase as subject, a verb and another noun phrase as object are created. By using methods like `setSubject()`, `setVerb()` and `setObject()`, the relationships between the elements can be established. More phrase types are available, such as prepositional phrases. The `setFeature()` method provides the opportunity to apply extra features to a sentence, phrase or other element. An example of a feature is the `TENSE` feature, which can be used to set the tense of the sentence. Note that all sentence element classes are extensions of the `NLGElement` class. Methods available to `NLGElement`s are available to all sentence elements. Methods such as `setFeature` are such methods that can be applied to any element. The way the realiser handles features often depends on the type of element is was applied to.

```
// Import the library
import simplenlg.framework.*;
import simplenlg.lexicon.*;
import simplenlg.realiser.english.*;
import simplenlg.phrasespec.*;
import simplenlg.features.*;

// These first lines are required regardless of the sentence
final static Lexicon lexicon = new XMLLexicon();
final static NLGFactory factory = new NLGFactory(lexicon);
final static Realiser realiser = new Realiser();

// Start the creation of a new clause
SPhraseSpec sentence = factory.createClause();

NPPhraseSpec thisExperiment = factory.createNounPhrase("experiment"
    ↪ );
thisExperiment.setDeterminer("this");

VPPhraseSpec verb = factory.createVerbPhrase("be");

NPPhraseSpec object = factory.createNounPhrase("a success");

// Add the subject, verb and object to the phrase and set the
    ↪ features
sentence.setSubject(thisExperiment);
sentence.setVerb(verb);
sentence.setObject(object);
sentence.setFeature(Feature.TENSE, Tense.PAST);

// Realise the sentence and print the results
String output = realiser.realiseSentence(sentence);
System.out.println(output);

// OUTPUT: This experiment was a success.
```

Listing 3.2: Example input for SimpleNLG.

3.2 Technical structure

In version 4 of SimpleNLG, its architecture was divided into modules with each their function. This allows external developers to build their own modules more easily. The other reason for the change was the previous use of components that had proprietary licenses. In the new version, all components are licensed in a way that allows commercial use, too (Westwater, 2009).

SimpleNLG is divided into the following modules:

- **aggregation** applies aggregation according to specified rules;
- **features** houses the many possible features, such as sentence type, tense, negation, etc.;
- **format.english** formats the text by adding indentation, headers and titles, etc.;
- **framework** keeps all element types that are used throughout, such as PhraseElement, WordElement and NLGFactory;
- **lexicon** contains the lexica, as well as conversion tools to import lexica;
- **morphology** applies morphology rules that it contains, such as inflection rules;
- **morphophonology** applies morphophonology rules, which focus mainly on the sound changes and spelling changes when words are combined, e.g. in French ”de” + ”le” = ”du”;
- **orthography** adds orthographical elements like capitals, periods and question marks;
- **phrasesspec** defines the types of phrases;
- **realiser** contains the realiser class itself;
- **syntax** contains helper functions which create the syntax.

When a realisation request is made, every word is looked up in the lexicon and, based on the findings, inflected and positioned.

3.3 Other languages

SimpleNLG has been adapted for other languages. The following sections provide an overview of the adaptation efforts that were described in papers. The papers describe methods used for adapting the software, gathering lexicon data and evaluating the performance of the system. These methods were considered for the creation and evaluation of SimpleNLG-NL.

3.3.1 French

(SimpleNLG-EnFr) The most noteworthy adaptation is the one in French, named SimpleNLG-EnFr. Not only did this project implement French, but it also rebuilt some of SimpleNLGs architecture in a way that makes the system multi-lingual. It was created by Vaudry and Lapalme (2013) at the Universit de Montral, Canada, which explains the need for a bilingual system.

The code was based on version 4.2 of SimpleNLG, with its new license and internal architecture. Vaudry and Lapalme rewrote some modules and classes by creating abstract classes, which are extended by classes for the selected language. This abstraction allows the use of more than just two languages and, with that, formed a base for some other adaptations, as described in later sections.

Lexicon To create the lexicon, the researchers created the closed part (e.g. determiners and pronouns) manually. The lexicon of the English SimpleNLG uses alphanumerical identifiers from the NIH lexicon, which act as an extra way of retrieving the right lexicon entry if the sentence planner has knowledge of these identifiers. The French kept the identifiers of the words English counterparts. This allows a sentence planner to plan for bilingual surface realisation. The open part (e.g. nouns and verbs) were taken from *L'chelle Dubois-Buyse d'orthographe usuelle franaise*, an overview of the most important and most commonly used French vocabulary. This resulted in a lexicon with 3871 entries.

Evaluation There was no mention of an evaluation process. However, the code did include jUnit tests that were used to test each of the implemented classes (mostly by generating example sentences).

3.3.2 German

(SimpleNLG for German) The German adaptation was started by Bollmann (2011). It was based on SimpleNLG version 3.8, with its non-modular architecture. Bollmann calls the grammatical coverage of his system already considerable, although far from being complete. It has the following features (quoted from the paper):

- Morphological operations, including handling of imperfection classes, separable verb prefixes, compounding, and preposition-article-contraction;
- modal verb clusters and perfect formation;
- relative clauses and relative clause extraposition; and
- constituent reordering.

As Dutch is considered a Germanic language, SimpleNLG for German may seem an obvious choice as a starting point for SimpleNLG-NL. However, it was unsuitable due to SimpleNLGv4s new architecture.

Evaluation To evaluate the performance of the system, five Wikipedia articles were recreated, with the result of 115 out of 152 sentences (75.66%) being recreated correctly. No efficiency evaluation was performed.

3.3.3 Italian

(SimpleNLG-IT) Years after the German adaptation, an Italian version was written by Mazzei et al. (2016). Unlike the French and German adaptations, this system was not based on the main SimpleNLG. Instead, it made use of the multilingual structure of SimpleNLG-EnFr, version 1.1.

Some statistics give an indication of the amount of code (re)written: ten new packages were written and 28 existing classes were modified in order to accommodate 33 new lexical features.

Lexicon The Italian lexicon was created by combining two sources. One provided a large list of words with their lexical properties. The second source contained a shorter list of words. Only the words present in both sources were used. To get information about the type of verbs (regular or irregular), verb lists on Wikipedia were used.

Similar to the lexicon of SimpleNLG-EnFr, the closed part of SimpleNLG-ITs lexicon was created manually. The final lexicon contained 6690 words.

Evaluation The evaluation method was related to that of SimpleNLG for German: the features were tested by generating 96 sentences, partially from the SimpleNLG-EnFr jUnit tests, partially from the grammar reference book used[3]. All tests were passed successfully, while keeping track of the total time it took to execute them. The loading of the lexicon into memory tot 1,433 ms and the generation of the sentences finished in 3,145 ms.

A second test was performed to determine its usability in an application context. A total of 20 sentences was selected from the Universal Dependency Treebank[4]. Of those sentences, 10 were declarative and 10 interrogative. The test was to recreate the sentences in the treebank. The generated, declarative sentences differed from the treebank sentences in several ways: word order, clitics, missing lexicon entry and an irregular verb being handled as regular verb. The interrogative sentences also had problems: word order, missing lexicon entry and certain questions that were not handled.

3.3.4 Brazilian Portuguese

(SimpleNLG-BP) Similar to SimpleNLG-IT, SimpleNLG-BP was based on SimpleNLG-EnFr. This was chosen mainly because both French and Brazilian Portuguese feature preposition contraction, e.g. *de* + *le* becomes *du* in French, and *de* + *a* becomes *da* in Brazilian Portuguese.

SimpleNLG-BP was built by De Oliveira et al. (2014) at the University of Aberdeen, home of the original SimpleNLG.

[3]Patota, G., *Grammatica di referimento dellitaliano comtemporaneo*, Guide linguistiche, 2006.

[4]`http://universaldependencies.org/`

Lexicon The lexicon comprised of a selection from a large (880.000) lexicon of inflected words. As SimpleNLG loads the lexica into memory, the full lexicon turned out to be too large for processing at a workable speed (it took 2.5 seconds to build the lexicon in memory), which was reason to pick 57 irregular verbs, which took only 0.17 seconds to build.

Evaluation Besides the ability to build 80 different forms for the same verb, person and number (of which 22 seem not to be used in the language), there was no evaluation presented.

3.3.5 Filipino

(FilSuRe) As the Filipino language differs much from Western languages, Ong et al. (2011) used SimpleNLG only as a pattern for their Filipino surface realiser. The system handles the following features:

- Verb phrases;
- Noun phrases;
- Prepositional phrases;
- Sentences with the so-called 'karaniwan' word order structure;
- Sentences with the so-called 'di-karaniwan' word order structure.

Evaluation The realiser was linked to a story generator developed earlier (Solis et al., 2009), which used the original SimpleNLG. The results were assessed by a linguist. Several problems were noted in the paper.

3.3.6 Telegu

Like Filippino, Telugu, a language spoken in India, differs much from Western languages. Dokkara, Penumathsa and Sripada (Dokkara et al., 2015) decided to build a realiser from scratch, using the structure of SimpleNLG.

Evaluation For evaluating the system, a batch mode was used. Similar to the SimpleNLG-BP, the realiser for Telugu was tested by generating many forms of the same input words. Only the grammatical features were changed. As a second test, attempts were made to recreate 738 sentences from examples in the used grammar reference book[5]. Out of 738 sentences, 419 (57%) were recreated exactly. Acceptable mismatches were not mentioned.

3.3.7 Spanish

(SimpleNLG-ES) The most recently published adaptation is, like SimpleNLG-PB and SimpleNLG-IT, based on the bilingual SimpleNLG-EnFr (Soto et al., 2017). It replaced the French part, keeping the result bilingual. The lexicon was a ready-built dictionary containing 550,000 forms of 76,000 lemmas, which was converted to XML.

Evaluation The jUnit tests present in SimpleNLG-EnFr were adapted for Spanish and ran. Secondly, SimpleNLG-ES was linked to a weather report writing, template-based NLG system. Both the original system and SimpleNLG-ES generated 76 weather forecasts, of which only 7 showed small, non-relevant differences between the two systems. As a third test, the library is currently being used in three running projects in the fields of weather warnings, official statistical data and business intelligence.

3.4 Conclusion

Dutch is derived from both Latin and German languages. Because of the many remaining similarities between Dutch and German, SimpleNLG for German may seem an obvious choice to use as a base for SimpleNLG-NL. However, because it was based on an older version of SimpleNLG with a different architecture, it was unsuitable. Instead, the bilingual adaptation for English and French was more suitable. SimpleNLG-EnFr offers

[5]Krishnamurti, B. & Gwynn, J.P.L., A Grammar of Modern Telugu, Oxford University Press, 1985.

a clear separation between language dependent and language independent classes and methods, which makes it easier to add a third language. The bilingual SimpleNLG-EnFr has already been used to create those for Italian, Brazilian-Portuguese and Spanish.

For evaluation, several methods were used. One method is to use unit testing, which is already built into SimpleNLG. The sentences that are tested would have to be adapted for a new language, as well as checked their grammatical coverage.

Gathering sentences to recreate is often done from the grammar reference book that is referenced during the project. These books often provide example sentences, which, altogether, should cover many (common) grammatical concepts. Besides reference books, Wikipedia articles have been used. Although it is unlikely that a small number of articles cover many grammatical rules, it can show that the system is capable of generating real world sentences.

Other evaluation measures found in the papers are the time it takes to perform a set of realisation tasks and the number of inflections that the system in able to generate.

All these methods were considered for use. SimpleNLG-NL was iteratively built and evaluated using Wikipedia sentences, unit tests and a proof-of-concept, as described in Chapters 4 and 6.

Chapter 4

Method

The method of adapting SimpleNLG for Dutch differs from those used for the other adaptations described in Section 3.3. Whenever the method was explained, the researchers seemed to use grammar references to determine what grammar rules to implement in their version of SimpleNLG. For example, Vaudry and Lapalme (2013) used a reference containing fundamental French grammar, of which “almost all” grammar points are covered by SimpleNLG-EnFr.

The Italian SimpleNLG-IT and SimpleNLG for German used sentences to evaluate the system. For SimpleNLG-IT, a set of unit tests was adapted from sentences in a grammar reference book and the sentences were generated using SimpleNLG-IT (Mazzei et al., 2016). Secondly, real world sentences coming from a treebank were tested. SimpleNLG for German picked Wikipedia articles from which sentences were recreated.

SimpleNLG-NL also used target sentences, but extended their use to the development phase. Real world sentences from a Wikipedia corpus were used. For each sentence, the SimpleNLG-NL input was written manually and resulting generated sentence was compared with the target sentence. Any differences were analysed and the corresponding grammar rules were corrected. To implement correct rules, the following grammar references were used: Taalportaal (Landsbergen et al., 2014), E-ANS (Haeseryn et al., 1997) and web articles of the Dutch Language Union (Nederlandse Taalunie)[1] and Genootschap Onze Taal[2]. Using this iterative approach, SimpleNLG-NL's coverage of Dutch grammar increased

[1]http://taalunie.org/

[2]https://onzetaal.nl/

with each sentence. This approach had the advantage that the subset of Dutch grammar covered was based on real world sentences. Also, it provided the possibility to base the code on an existing adaptation of SimpleNLG. Instead of starting with no grammar rules and search for and reuse implementations in other languages, the grammar rules were copied and then adapted or removed when necessary.

As a starting point, SimpleNLG-EnFr was used. All French parts were cloned and renamed to Dutch. French was chosen over English, because its more complex features seemed more related to Dutch than the English features, especially the morphology. Because of that, the realisation of the first sentences used French grammar. With each sentence, the grammar showed more aspects of Dutch grammar.

Implementing the Dutch grammar in SimpleNLG-NL was done in four rounds. The rounds are described in Section 4.1. Rounds 1 and 3 used target sentences from a corpus, while Rounds 2 and 4 used newly written unit tests.

Sentence selection The target sentences used in Rounds 1 and 3 were randomly selected, based on the word count needed for the round, from the Wikipedia corpus available in Dact[3], a viewer for Alpino corpora. Alpino[4] (Bouma et al., 2001) is a dependency parser for Dutch. The corpus contains 100,000 sentences. After a sentence was picked, it was tested for two requirements. First, the sentence had to be grammatically correct. Some 'sentences' were just headings of articles, while some others were only part of a sentence. To meet the second requirement, the sentence could not include embedded direct speech. For example: *"That was nice", said John*. While the spoken sentences themselves can be generated, the English SimpleNLG does not support properly embedding them within another sentence. The same applies to SimpleNLG-NL.

After a sentence was selected, the corresponding input code for SimpleNLG-NL was written manually. To aid in this process, the Alpino parse tree already present in Dact was used as reference. The parse tree was converted to SimpleNLG-NL code as closely as possible, in a process not dissimilar to the one later used for a proof-of-concept as described in Sec-

[3]`https://rug-compling.github.io/dact/`

[4]`http://www.let.rug.nl/vannoord/alp/Alpino/`

tion 6.2. It was made sure not to use canned text (except for names and fixed expressions) and no information on word order or punctuation was provided. After the input was written, SimpleNLG-NL would realise the sentence. The result would be compared with the target sentence and any differences would be analysed. The grammar rule or lexicon entry related to the problem would then be corrected and the whole cycle would repeat with the next target sentence.

The second round consisted of unit tests designed to test individual features and structure, as described in Section 4.1.2.

The four rounds are described in Section 4.1, followed by the criteria by which the generated sentences would be judged in Section 4.2.

4.1 Four rounds

4.1.1 Round 1

In the first round, twelve sentences were picked from the Wikipedia corpus. These sentences had an increasing word count, which was assumed to also increase the complexity of the sentence structure. The sentences were iteratively generated, compared and corrected until the desired result was reached (see Section 4.2). After completing a sentence, the input for the next sentence was written and SimpleNLG-NL would be iteratively improved. The sentences (and their results) can be found in Appendix Table A.1

4.1.2 Round 2

In the second round, simple unit tests were written. These consisted of short sentences that were specifically written for testing SimpleNLG-NL's features in isolation. This has the advantage that processes such as syntax and morphology could be tested without potentially inducing problems due to complex sentence structures.

Ten sentences tested the inflection of both regular and irregular verbs, and adjectives and their inflection. Multiple verb types were tested: verbs that required modifications to their stem or specific suffixes (for reasons described in Section 5.1.4), as well as irregular verbs, both with and without a lexicon entry. Irregular verbs without a lexicon entry were expected to fail by being inflected as if they were regular verbs. Adjectives were tested for proper detection of the person and number of its noun, based on the lexicon or the article.

Five sentences tested syntax of negated sentences. Two sentences contained different kinds of verbs, which should have no influence on the negation, such as position of the negation adverb *niet* 'not'. One sentence added an object and adjectives, which added more possibility of failure. The last two sentences replace the negation adverb *niet* 'not' with *geen* 'no/none'.

The next 12 sentences use different tenses to test the morphology of different verb groups.

The unit tests also provided the opportunity to test features that were not present in the first round of test sentences. One such feature was that of interrogative sentences (10 sentences). The choice of interrogative types was based on the types available for French and English. These were translated into Dutch.

All unit tests, including short descriptions and the results, can be found in Appendix Table A.2.

4.1.3 Round 3

The third round consisted of twenty-one more Wikipedia sentences. The first eleven sentences were of a length of up to thirteen words, which roughly corresponds to the sentence length recommended for writing texts suited for the end of primary education, which is found to be twelve words (Taalunieversum, 2018). The next ten sentences had a word count between fourteen and twenty. In this third round, the input for all twenty sentences was created first, to test the state of the system at that point in time. It was afterwards that SimpleNLG-NL was corrected and improved for each sentence.

4.1.4 Round 4

The fourth round was added after the prototype was built as described in Chapter 6. The prototype revealed incorrect placement of objects in the future and conditional tenses and in the perfect form. The unit tests of Round 2 that tested those features did not include an object, without which the results were correct. The mistakes were deemed important enough to correct. Therefore, an extra set of unit tests was written. The sentence *Marie gooit de bal.* ‘Marie throws the ball.’ was realised in 16 different variants: for each of the four tenses (present, past, future, conditional) four different sets of features were applied, namely active-simple, active-perfect, passive-simple and passive-perfect.

4.2 Correctness criteria

A sentence generated by SimpleNLG-NL was deemed correct if the output matched at least one of the following criteria. These criteria are ordered by level of tolerance, with the first being the preferred outcome. A generated sentence was successfully generated if:

- the output matched the target sentences exactly; or
- the output differed from the target in terms of punctuation in the form of commas and quotation marks, but kept the same meaning to readers; or
- the output differed from the target in terms of word order, but kept the same meaning to readers.

It has to be noted that commas, quotation marks and word order can change the stress of a sentence. These cases had to be evaluated individually.

Chapter 5

Implementing Dutch grammar

This chapter outlines the largest changes made to SimpleNLG-EnFr, summarised by subject, and the results of the implementation. It also lists the files that had to be created, copied and/or edited to create SimpleNLG-NL.

5.1 Grammar rules implemented

The following sections describe the grammar rules implemented in SimpleNLG-NL.

5.1.1 General spelling rules

Dutch spelling aims to keep the spelling as consistent as possible between forms and inflections ('gelijkvormigheid')[1]. However, there are a few rules that limit this consistency. These rules apply to all word categories and are implemented in SimpleNLG-NL per category. The leidraad (Nederlandse Taalunie, 2015) mentions three rules as the most important:

- No *v* or *z* at the end of a syllable.
- No repeated consonant at the end of a word.
- If a word ends in a sibilant (*s*, *sh*, *sj*, etc.), the next suffix will drop the *s*, e.g. *nerveus-st* ('most nervous') becomes *nerveus-t*.

[1] `http://woordenlijst.org/leidraad/1/2`

5.1.2 Lexicon

Lexicons used for SimpleNLG are written in XML. During the preparation phase of this project, a simple lexicon was built based on the Open Dutch WordNet (Postma et al., 2016). This lexicon provided word category, plural forms, articles, auxiliary verbs, past tense forms and past participles. However, it did not include critical information like irregular verb forms. While there are lexicons available for Dutch, most don't allow commercial use. Commercial use should be possible if the lexicon is to be released under the same license as the rest of SimpleNLG-NL. This license was also one of the advantages of version 4 of SimpleNLG. Therefore, the information missing in the current lexicon will have to be sourced elsewhere. This lexicon is still work in progress.

In order to test the capabilities of SimpleNLG-NL during the implementation, a new lexicon was started. It included only the manual part that would be added to the final, generated lexicon. This closed part consists of the irregular auxiliary verbs *zijn* ('be') and *hebben* ('have'), prepositions, articles and pronouns. During the four rounds of implementing the Dutch grammar rules, the lexicon was appended with the necessary information when simply the base form of a word was not enough to determine its correct syntax or morphology. For example, the very first verb encountered (*vrijkomen* 'to be released') was an irregular verb, which means that the regular inflection rules do not apply. An example entry for the lexicon can be found in Listing 5.1. Adding the correct forms to the lexicon gave SimpleNLG-NL the desired solution, helping with generating the sentence correctly.

```
<word>
  <base>vertrekken</base>
  <category>verb</category>
  <past>vertrok</past>
</word>
```

Listing 5.1: Basic example entry in the lexicon for the verb *vertrekken* 'leave'. This verb is irregular, which is why the `past` field is set.

5.1.3 Nouns

In SimpleNLG-NL, the morphology of nouns consists of pluralisation. Dutch grammar also has four grammatical cases (nominative, genitive, dative and accusative), but these are used rarely in modern Dutch (Nederlandse Taalunie, 2018a). Therefore, cases were not implemented in SimpleNLG-NL.

Pluralisation of nouns

The short rule for pluralisation is simple: add the suffix *-en* if the last syllable of the noun is stressed, add *-s* if it is not. Since the lexicon does not contain information on stress, the suffix was determined by matching the ending of the noun with a list, found in the e-ANS grammar reference[2]. When adding suffixes, other morphological rules may apply to ensure the correct form.

One of these rules is to keep the sound consistent: *rijkdom* ('richness') is pronounced /ˈrɛikdɔm/, but if the plural form would just add -en *rijkdomen*, its pronunciation would change to /ˈrɛikdomən/. Instead, the /ɔ/ sound should be kept by repeating the consonant following it: *rijkdommen* /ˈrɛikdɔmmən/.

Following the same sound consistency rule, the opposite can also be needed: if the singular form ends in a repeated vowel followed by a single consonant, the plural form can remove one of the vowels. Technically speaking, this is due to the stem having only one vowel, which needs to be repeated in the singular form in order to keep the sound. In the case of SimpleNLG-NL, however, the lexicon and/or user input contains the singular form, not the stem, so the double vowel will have to be reduced. For example, *Italiaan* ('Italian', noun) only needs one *a* in plural to keep the /a/ sound: *Italianen*.

A test of pluralisation after the new implementation can be found in Table 5.1. These results show two cases in which the system fails: *kopie* and *bacterie*. While most words ending in *-ie* are appended by an *-s*, these words should get *-en* instead, both with different morphological rules. If the last syllable was stressed, the resulting ending would be *-ieën* (e.g. *kopie*

[2] `http://ans.ruhosting.nl/e-ans/03/05/03/body.html`

$\rightarrow$ *kopieën*). If it was unstressed, one *e* would be left out (e.g. *bacterie* $\rightarrow$ *bacteriën*).

The default suffix for nouns ending in *-ie* is *-s*. Special cases, such as the previous examples, should be specified in the lexicon.

The input for pluralisation is kept the same as for English and French: `nounPhrase.setFeature(Feature.Number, NumberAgreement.PLURAL)`.

5.1.4 Verbs

This section describes the morphology of verbs and the syntax in verb phrases used in SimpleNLG-NL. The first sections describe regular and irregular verbs. A separate section is dedicated to so-called 'Separable Comlex Verbs', which require different rules. Most changes were made in the `VerbPhraseHelper.java` and the `MorphologyRules.java` files, the former containing mostly syntax rules for verb phrases, the latter describing morphology rules for all word groups.

Tenses

When inflecting verbs, the morphology part of SimpleNLG-NL first looks in the lexicon for matching forms. If it does not find the form it is looking for, based on tense, number and person, it resorts to rules for regular verbs. Inflections are based on the stem of the verb. This stem can be deduced from either the infinitive or the first person singular. Having some exceptions, the stem is the infinitive minus the trailing *-en*. It can then be inflected by adding prefixes or suffixes, sometimes requiring extra rules to keep the sound consistent.

In any tense, the letters *v* and *z* at the end of a the stem will be replaced by *f* and *s*, respectively.

Present simple The *tegenwoordige tijd* ('present simple') requires little inflection. The first person singular uses the bare stem, the other singular form get a *-t* as suffix, the plural forms all use the infinitive. Some morphological rules apply, such as not appending the *-t* if the stem already ends in a *t*. Also, similar to nouns, vowels may need to be repeated to keep the

Singular	**Output**	**Expected**
oma	oma’s	oma’s
radio	radio’s	radio’s
bureau	bureaus	bureaus
metselaar	metselaars	metselaars
grijsaard	grijsaarden	grijsaarden
wijzer	wijzers	wijzers
reservoir	reservoirs	reservoirs
leeuw	leeuwen	leeuwen
rijkdom	rijkdommen	rijkdommen
opponent	opponent	opponent
stommerik	stommerikken	stommerikken
prinses	prinsessen	prinsessen
bijzonderheid	bijzonderheden	bijzonderheden
maatschappij	maatschappijen	maatschappijen
Italiaan	Italianen	Italianen
travestiet	travestieten	travestieten
boerin	boerinnen	boerinnen
woning	woningen	woningen
violist	violisten	violisten
calamiteit	calamiteiten	calamiteiten
leerling	leerlingen	leerlingen
hindernis	hindernissen	hindernissen
wetenschap	wetenschappen	wetenschappen
apotheek	apotheken	apotheken
fabriek	fabrieken	fabrieken
zeef	zeven	zeven
lijf	lijven	lijven
duif	duiven	duiven
reis	reizen	reizen
kopie	**kopies**	kopieën
bacterie	**bacteries**	bacteriën

Table 5.1: Plural forms generated after applying most rules for regular plurals. Marked in bold are two results that were incorrect, as those nouns have irregular plural. Such irregularities can be accounted for by providing the correct form in the lexicon.

sound consistent. For example, the first person of *dromen* ('to dream') is *droom*, so the /o/ sound is preserved.

Several types of verbs were tested (see Table 5.2).

Past simple The *onvoltooid verleden tijd* ('past simple') of regular verbs is formed by adding one of the suffixes *-de* or *-te* to the stem. And extra *-n* should be added in the plural case. Which of these suffixes to use depends on the ending sound of the stem: if it ends in an unvoiced consonant, the suffix will be *-te*, in all other cases it's *-de*. The list of unvoiced consonants can be remembered using the mnemonic *'t kofschip*. Because this mnemonic is based on sound, it is extended to *'t kofschiptaxietje* (Onze Taal, 2018), which adds the /ks/ as in 'flex' and the /ʃt/ as in 'match', as well as similar sounds. SimpleNLG-NL tests the string for one of the following endings: *t, k, f, s, ch, p, x, sj, c*. If this test fails to apply the correct suffix, it can be corrected by adding the `past` field or the more specific `past<person><number>` such as `past2p` to the verb's lexicon entry.

Irregular verbs have to be specified in the lexicon. The general inflection rules are to use the past stem for a singular number, and the past stem + *-en* for the plural case. For example: *lopen* ('to walk') has a present stem of *lop*, but a past stem of *liep*. The first person singular would therefore be *ik liep*, whereas the first person plural is inflected as *wij liepen*.

Past perfect Past participles of regular verbs are built use the prefix *ge-* and the suffix *-d* or *-t*. The suffix is determined using the same mnemonic as used for the past tense. In some cases, the prefix *ge-* is replaced with *be-*. This is not implemented in SimpleNLG-NL, but mistakes can be prevented with a lexicon entry, similar to irregular verb with irregular past participles.

Future tense The future tense in the second round, because the sentences form round one did not require its implementation. This tense uses the auxiliary verb *zullen* 'will'. This auxiliary verb is inflected according to the person and number of the subject. Its position is the same as that of other auxiliary verbs. As can be seen in the results of Round 4 (Section 5.3), there are cases in which the other auxiliary verbs need to be

Verb	**Tense**	**Output**	**Expected**
vrijkomen	present	komt vrij	komt vrij
	past	kwam vrij	kwam vrij
rennen	present	rent	rent
	past	rende	rende
zetten	present	zet	zet
	past	zette	zette
vrezen	present	vreest	vreest
	past	vreesde	vreesde
leven	present	leeft	leeft
	past	leefde	leefde
lopen	present	loopt	loopt
	past	liep	liep
komen	present	komt	komt
	past	kwam	kwam
dromen	present	droomt	droomt
	past	droomde	droomde
weglopen	present	***wegloopt***	loopt weg
	past	liep weg	liep weg
weg|lopen	present	loopt (...) weg	loopt (...) weg
	past	liep (...) weg	liep (...) weg
wegrennen	present	***wegrent***	rent weg
	past	***wegrende***	rende weg
weg|rennen	present	rent (...) weg	rent (...) weg
	past	rende (...) weg	rende (...) weg

Table 5.2: Verb inflections of the third person singular. Verbs were chosen to reflect most inflection rules. *Weglopen* was in the lexicon, but only included the past form. *Wegrennen* was not in the lexicon at all, explaining the assumption of it being a normal, regular verb. Adding a pipe between the two parts of the word solves the problem, but also places the preverb at the end of the phrase.

pushed to a different position. This is not yet implemented in SimpleNLG-NL. The main verb is used in its infinitive form.

Conditional tense The implementation of the conditional tense is very similar to that of the future tense. The difference is in the tense of the auxiliary verb. Where the future tense uses the present tense auxiliary, the conditional tense uses the past tense auxiliary *'zouden'* 'would'.

Separable Complex Verbs

Perhaps the most challenging aspect of adapting SimpleNLG for Dutch was the morphology and syntax *scheidbaar samengestelde werkwoorden* ('Seperable Complex Verbs' or SCVs). SCVs are a class of verbs that consist of a main verb prefixed by another word, called a 'preverb' (Booij & Audring, 2018). Preverbs are not to be inflected or changed, which causes the inflection of SCVs to be complex. In this section, *toekennen* 'assign' will be used as an example. In this SCV, the preverb is *toe* 'to'. This section describes the difficulties of detecting and inflecting SCVs.

Firstly, in the present simple, SCVs are split into their preverb and main verb and their order is reversed. The main verb is inflected as it would if it were on its own. For example: *ik ken toe* ('I assign').

The position of the preverb is often very flexible: direct objects, indirect objects, prepositional phrases and even entire subclauses can be placed between the main verb and its preverb. However, it can be the stylistic preference to move the preverb to immediately after the main verb. In SimpleNLG-NL, it was decided to be positioned at the end of the sentence. Positioning is different in subordinate clauses. The preverb is kept attached to the main verb and objects are often placed between the subject and the verb. The main verb is still inflected normally, but the preverb is prefixed to it after inflection. This would result in, as per the example, *..., dat ik hem toeken* ('..., that I assign to him').

The verb *vrijkomen* ('to get released') is an SCV that occurred in the very first sentence of the first round. Its main verb is *komen* and its preverb is *vrij*. After implementing the rules for SCVs,the user input can either be *vrijkomen* or *vrij—komen*. The first input looks for *vrijkomen* in the lexicon. It will find it and use its predefined forms. The second input will split the verb *komen* from the preverb *vrij*. It will then look for *komen* in the lexicon and inflect it appropriately (either from lexicon data or regular verb rules).

Komen happens to be a special irregular verb. While most irregular verbs only are irregular in their past forms, *komen* is also irregular in the present tense. Compare: *komen* and *ik kom* with *dromen* and *ik dro**o**m*. As *komen* is irregular, SimpleNLG-NL needs more information from the lexicon.

SCVs are again proving difficult in the case of past participles. With SCVs, the *ge*-prefix is added between the preposition and the stem. However, the prefix is left out of verbs that are prefixed by a preposition, which look exactly like SCVs with the difference being the stress. For example, *doorboren* can be stressed in two ways: */ˈdoorˌboren/* ('to go on drilling') and */ˌdoorˈboren/* ('to perforate')[3]. The first would make the participle *doorgeboord*, while the second would make *doorboord*. It is this kind of cases where the sense of the word is important. However, the difference should be reflected in the lexicon.

An attempt was made to detect SCVs. Five methods were used sequentially. If the first method failed, the next was tried. SCVs are detected using the following methods (using *voor* as an example preverb):

1. The preverb being set in the lexicon: `<preverb>voor</preverb>`

2. Using a pipe ('|') in the input to separate the preverb from the main verb: `sentence.setVerb("voor|komen")`

3. Detected based on prefixes: bij, in, na, uit, op, af, mee, tegen, tussen, terug, toe

5.1.5 Adjectives

Dutch adjectives can be used attributively or predicatively (Audring, 2018). SimpleNLG currently only supports attributive use. A variable has been added to the features for predicative use, but it is not implemented yet. For now, SimpleNLG-NL will also only support attributive adjectives.

There are four cases in which the adjective gets the suffix *-e*:

[3]`http://taalportaal.org/taalportaal/topic/pid/topic-13998813296768009#section_svl_rtr_rk`

1. the noun is plural;
2. the noun has the COMMON gender;
3. the noun is preceded by a possessive pronoun; or
4. the noun is preceded by a definite determiner (*de* or *het*).

In total, seven different adjectives were tested. These were selected to cover the inflection rules and the results of combining those rules. An overview can be found in Table 5.3.

If the adjective requires the suffix, inflection is based on the pronunciation. As with other word groups, there are many exceptions to some general adjective inflection rules. For instance, if the adjective ends in a double vowel followed by one consonant, one of the vowels has to be removed (e.g. *duur* 'expensive' becomes *dure*).

Adjectives that match the pattern for so-called 'regular doubles' have their trailing single consonant repeated. Regular doubles are adjectives (or verbs) that end in a vowel and a single consonant, of which the consonant has to be repeated in order to keep the sound of the vowel consistent. For example, *snel* 'quick' is pronounced /snɛl/, but simply adding the suffix *-e* would result in a change in the pronunciation of the vowel: /snelə/. To follow the spelling rules, the last consonant is repeated (*snelle*), which results in /snɛlə/. If the pattern matching returns a false negative, the user can set the `feature.PATTERN` feature to `Pattern.REGULAR_DOUBLE`. An example result would be *snel* being inflected as *snelle*.

If the base form already ended in a *-e*, the suffix should not be added. For example, *stupide* 'stupid' stays *stupide*.

As with other word categories, an *f* or *s* as ending consonant after a vowel should be replaced by a *v* or *z*, respectively, as seen in *grijs* and *grijze*[4].

[4]The *ij* is a digraph: two characters forming one sound. Even though the *j* is not a vowel, the digraph is traditionally considered a vowel in Dutch. In this example, it should be treated as a vowel in order to comply with the sound consistency guidelines.

Adjective	***-e* suffix**	**Comparative**	**Superlative**
duur	dure	duurder	duurst
snel	snelle	sneller	snelst
stupide	stupide	stupider	meest stupide
grijs	grijze	grijzer	grijst
komisch	komische	komischer	komischte (or *meest komisch*)
erg	erge	erger	ergst
accentloos	accentloze	accentlozer	accentloost

Table 5.3: Adjectives and their different inflections. The *italic* forms are exceptions that can to be added to the lexicon.

Comparatives The comparative form of an adjective is generally the adjective appended by *-er*. If the positive form already ends with /ə/ (or simplified: *-e*), it only requires an *-r* (Haeseryn et al., 1997, section 6.4.3.1). As with the positive form, the comparative may receive an extra *-e* as suffix. The comparative of *snel* becomes *sneller* or *snellere*.

A second way of using comparative adjectives is to add the adverb *meer* 'more' to the phrase. The use of *meer* is similar to the use of *more* in English. Still, the suffix *-e* may need to be applied to the adjective. The choice between adding *-er* or using *meer* is often based on pronunciation difficulties or style. The preferred method is adding the suffix, but *meer* is often used with:

- predicative adjectives (not supported in SimpleNLG-NL);
- adjectives formed from a participle (actually requires *meer*);
- adjectives ending in *-st*, *-sd*, *-s*, *-sch*, *-sk* or *-de*;
- adjectives that create an odd or difficult pronunciation (e.g. *gebruikelijkere* /ɣə'brœykələkərə/ (Nederlandse Taalunie, 2018b)

As predicative adjectives are treated as attributive, and participles and difficult pronunciation are not detected, only the word endings are implemented in SimpleNLG-NL. The other cases can be corrected by adding a lexicon entry.

Superlatives Very similar to comparatives, superlatives can be either appended by a suffix (*-st* + *-e* if necessary) or accompanied by the adverb *meest* 'most'. The same ending matching is implemented to choose to use *meest*. Inflection is based on the sound: if the adjective ends in *-s* or *-sch* (both sounding as /s/), then only a *-t* is a appended. In all other cases, *-st* is used.

5.1.6 Word order

Dutch word order uses multiple constructions (Kooij, 1978, pp. 33-36). It does not adhere to a standard like *subject-verb-object* (SVO) or *subject-object-verb*. Luckily, SimpleNLG is fairly flexible. It provides the user three types of modifier: 'premodifier', 'modifier' and 'postmodifier', which place the selected element before or after its parent element. An example of a premodifier being used in the input can be seen in Listing 5.2. In this example, the premodifier is treated as an adjective, because that is the default behaviour if the word category is not specified. However, other elements, such as prepositional phrases, can also be set as premodifier, which will then not be inflected as if it were an adjective. This flexibility allows the user to position sentence elements in many different ways. It has to be noted, though, that if an NLG system wants to take advantage of this flexibility, the sentence planner will have to be built in a way that uses these types of modifiers, even though it is strictly not the task of a sentence planner.

```
NPPhraseSpec np = factory.createNounPhrase("aap");
np.addPreModifier("snel");
np.setSpecifier("een");

// Result after realisation: "Een snelle aap."
```

Listing 5.2: Example usage of the premodifier method.

While Dutch word order is flexible, SimpleNLG-NL requires some defaults. Based on the target sentences, it was chosen to use the SVO order for main clauses and the SOV order for subordinate clauses. For example, the target sentence *De Sectie 3 is de staf-afdeling die zich bezighoudt met het functiegebied Operaties.* matches that structure: *subject(de Sectie 3)* *verb(is) object(de afdeling), subject(die) object(zich) verb(bezighoudt) (...)*. This decision has allowed for the generation of the target sentences to an extent described in Section 5.3.

To-infinitives Dutch has three types of phrases with infinitive verbs: *te-infinitive*, *om + te-infinitive* and the bare infinitive without any additional words (Broekhuis, 2018).

Bare infinitive verb phrases consist of only the infinitive verb and its modifiers and complements. In comparison to English, Dutch infinitive verbs are not always accompanied by *te* 'to'. For example, the sentence *Ik wil rennen.* 'I want *to* run.' does not include *te*.

Te-infinitive phrases consist of the infinitive verb, modifiers and complement, as a whole preceded by *te* which is similar to the English 'to'.

The positioning of *te-infinitive* verb phrases is a stylistic choice. They can be positioned directly after the object (*Marie probeert de bal* ***te gooien*** *naar de aap.*, literally *Marie tries the ball to throw to the monkey* 'Marie tries to throw the ball to the monkey.') or at the end of the sentence (*Marie probeert de ball naar de aap* ***te gooien***, literally *Marie tries the ball to the monkey to throw* 'Marie tries to throw the ball to the monkey'). From personal observations, it seems that it is preferred to position it closer to the main verb phrase if the remaining sentence is 'too long'. However, this is a stylistic choice, which is not built into SimpleNLG-NL. By default, infinitive phrases are positioned at the end of the sentence (e.g. *Het schip ...* ***werd*** *op 7 juni 1998 naar een dok van het Maritieme Museum in de haven van La Rochelle* ***overgebracht***).

Adding a te-infinitive or a bare infinitive can be done by creating a verb phrase (`factory.createVerbPhrase("verb")`) to be added to the sentence using `sentence.addComplement()`. The choice between te-infinitive and bare infinitive should be made by the sentence planner, which can add one of two features to the verb phrase: bare infinitive can be used by setting the `FORM` feature to `INFINITIVE`, while for te-infinitives, a new feature was added. The sentence planner can set `DutchFeature.TE_INFINITIVE` to `true` in order to set the verb to its infinitive form and add *te* to the verb group.

The third form of infinitive phrases, *om + te-infinitive*, precedes the verb phrase with the complementiser *om* 'to' or 'in order to', e.g. *Marie heeft* *zin* ***om te zingen*** ‘Marie feels like singing’. It is the task of the sentence planner to decide whether or not to add *om*, as it can be omitted depending on the context. For instance, when accompanying the verb *proberen* ‘try’, *om* can be omitted, but can also be included: *Marie probeert te zingen.* ‘Marie tries to sing’ and *Marie probeert om te zingen.* are both correct. To add this complementiser in SimpleNLG-NL, the sentence planner can use set the `Feature.COMPLEMENTISER` feature of the verb phrase to `"om"`. The support of *om + te-infinitives* in SimpleNLG-NL is limited when objects are added. Because the verb phrase element is realised as a whole, any objects added to the sentence element are placed before it. This leads to, for example, the incorrect *Marie probeert het lied om te zingen*, literally ‘Marie tries the song to sing’. To correct this, the object can be added to the verb phrase element itself, which causes SimpleNLG-NL to position the object in the verb phrase, such as *Marie probeert om het lied te zingen.* ‘Marie tries to sing the song’. This would have to be done by the sentence planner.

Past participles Past participles are positioned similar to te-infinitives. They can be positioned directly behind the main verb or at the end of the sentence. In the target sentences, past participles were often placed at the end of the sentence by SimpleNLG-NL. For example, in sentence wik_part0229/333218-12-7.xml (*Het schip ... werd ... naar een dok ... overgebracht.* ‘The ship was transported to the dock.’), the past participle *overgebracht* was placed at the end of the sentence. Similar to te-infinitives, past participles seem to be moved to the front in ‘long’ sentences, as may be the case in this sentence, with 17 words between the auxiliary verb and the participle. The result would become: *Het schip ... werd ... overgebracht naar een dok....* However, this is a stylistic choice and it is not implemented in SimpleNLG-NL.

Reflexive pronouns Pronominal verbs need a reflexive pronoun, like *zich*, added to the front of the verb group. In SimpleNLG-EnFr, reflexive pronouns are added by setting them as objects. However, the positioning of reflexive pronouns and that of objects is not the same in Dutch. Instead, reflexive pronouns are set as object of the verb. The default position for objects is right after the verb. Reflexive pronouns as objects are detected

Person	Number	Personal pronoun	Reflexive pronoun
first	singular	ik	me
second	singular	jij	je
third	singular	hij/zij/het/u	zich
first	plural	wij	ons
second	plural	jullie	je
third	plural	zij/u	zich

Table 5.4: Reflexive pronouns.

as reflexive pronouns because of the lexicon and are inflected based on the number of the subject, according to Table 5.4.

Prepositions Verbs with predetermined prepositions require the preposition to be chosen by the sentence planner. The lexicon of the sentence planner can include the preposition for each sense.

5.1.7 Aggregation

SimpleNLG supports some basic aggregation. If two aggregated clauses have the same subject, SimpleNLG will not render it the second time. Equality of subject is detected if the user created a new noun phrase with the same word or directly reused the first noun phrase variable. And example input for aggregation can be found in Listing 5.3.

```
ClauseCoordinationRule coord = new ClauseCoordinationRule();
List<NLGElement> elements = Arrays.asList(clause1, clause2);
List<NLGElement> result = coord.apply(elements);
NLGElement aggregated = result.get(0);

String result = realiser.realiseSentence(aggregated);
```

Listing 5.3: Example input for the aggregation of two clauses: clause1 and clause2.

SimpleNLG-EnFr (and now SimpleNLG-NL) also supports relative clauses. The user can set the `DutchFeature.RELATIVE_PHRASE` to be a noun phrase mentioned earlier (either the subject or an object of the previous clause). An example can be found in Listing 5.4. Notice that the second clause has to be set as an complement of the target element (either subject, object or clause). The gender and number of the target element will be used to determine the proper relative pronoun. If it is neuter and singular, *dat* is used, otherwise *die* is chosen.

```
SPhraseSpec clause1 = factory.createClause();
clause1.setSubject("Marie");
clause1.setVerb("hoort");
NPPhraseSpec clause1_object = factory.createNounPhrase("jongen");
clause1_object.setSpecifier("een");
clause1.setObject(clause1_object);

SPhraseSpec clause2 = factory.createClause();
clause2.setFeature(DutchFeature.RELATIVE_PHRASE, clause1_object);
clause2.setVerb("roepen");
clause1_object.addComplement(clause2);
System.out.println(realiser.realiseSentence(clause1));

// Result after realisation: "Marie hoort een jongen die roept"
```

Listing 5.4: Example input for a sentence with a relative clause.

Relative clauses

In the last sentence of the first round, a conjunction phrase was used. In SimpleNLG, coordinated phrases are used to create conjunction phrases, but the `CoordinatedPhraseElement` class can not be used directly as a clause by itself. The coordinated phrase has to be part of a regular clause element, which requires the regular input of a subject and a verb. This is where the problem becomes apparent. In one of the target sentences, the conjunction phrase is a mere enumeration, which does not include a verb, just multiple subjects (*zoals probleemtaken, actietaken, studietaken en discussietaken* 'like problem tasks, action tasks, study tasks and discussion tasks'). SimpleNLG does not support this kind of sentences.

Interrogative type	**Dutch keyword**	**Example**
yes-no	-	*Fietst Marie?*
why	waarom	*Waarom fietst Marie?*
where	waar	*Waar fietst Marie?*
how-many	hoeveel	*Hoeveel ballen gooit Marie?*
who-subject	wie	*Wie fietst?*
who-object	wie	*Wie zoent Marie?*
who-indirect-object	wie	*Naar wie gooit Marie een bal?*
what-object	wat	*Wat gooit Marie?*

Table 5.5: The eight types of interrogative sentences supported by SimpleNLG-NL.

5.1.8 Interrogative sentences

SimpleNLG and SimpleNLG-NL support basic interrogative sentences, see Table 5.5. The input for interrogative sentences is similar to that for regular sentences, with the addition of having to specify the interrogative type by setting the feature `Feature.INTERROGATIVE_TYPE` on the clause to `InterrogativeType.<sometype>`. Currently, SimpleNLG-NL does not check for neuter gender, which requires the keyword for who-type questions to be *wat* 'what' instead of *wie* 'who'. The word order for interrogative sentences is changed from SVO to VSO.

Who-indirect-object questions use the preposition given to the indirect object. For example, *Naar wie gooit Marie een bal?* 'Who does Marie throw the ball to?' uses the preposition *naar* from the indirect object in the original sentence *Marie gooit een bal* ***naar de aap****.* 'Marie throws a ball to the monkey.'.

5.1.9 Punctuation

The punctuation remains mostly similar to French. Some small changes were made. Clauses within clauses (e.g. starting with *maar* 'but') should start with a comma appended to the beginning. The `orthographyHelper` class now checks after every element whether the next element will be of the type CLAUSE, in which case it will append a comma to the end of the current element. This works for coordinating clauses. However, this is not the only reason to add a comma when using subclauses. Relative clauses get surrounded by two commas, but only based on the semantic meaning[5]. As of now, these cases are not handled.

5.2 Files changed

SimpleNLG is divided into multiple folders. These folders correspond with a module as described in Section 3.2. Inside the folders are files, which each describe one object class. Building SimpleNLG-NL using SimpleNLG-EnFr as a basis required the copying of all 12 French-specific files and changing the 8 language independent files to include Dutch as a new addition. That formed the initial setup for SimpleNLG-NL. After that, implementing Dutch grammar as described earlier resulted in the editing of 16 Java files, 7 of which were language independent files. When changing these language independent files, extra care had to be taken to not change the behaviour of the realiser for other languages. Nine files only affected Dutch realisation. The changed files and a summary of the changes are listed in Table 5.6.

5.3 Results

The final coverage of SimpleNLG-NL after all four rounds can be found in Table 5.7. Each sentence was checked for correctness based on the criteria described in Section 4.2. The use of automated evaluation metrics such as BLEU (Papineni et al., 2002) was considered, but because the number of sentences was small, it seemed unnecessary. More importantly, such metrics would not take the relatively free word order in Dutch into account. Therefore, it was chosen to evaluate the sentences manually.

[5] http://taaladvies.net/taal/advies/vraag/459/komma_bij_beperkende_en_uitbreidende_bijvoeglijke_bijzinnen/

Results Round 1 Out of 12 sentences, 11 were generated correctly (91.7%). One of the three incorrect sentences that were accepted placed the past participle at the end of the sentence, which is a stylistic difference. A second one used topicalisation, but without it, the sentence was still accepted as having the same meaning. The third added non-mandatory commas. After the first round of sentences, SimpleNLG-NL generated 8 exact matches (66.7%).

The results found in Table A.1 in Appendix A are the result after all four rounds. While the second and third round were performed with the intention to not break working solutions, the rounds could improve results from the earlier rounds. In this case, the incorrect positioning of the negation auxiliary adverb *niet* was corrected in a later round.

Results Round 2 This round consisted of 37 short sentences as a means of testing individual features. The first ten sentences were aimed at the basic verb inflection, both regular and irregular. It also tested adjectives and their inflection. Next were five negated sentences. During these sentences, the positioning of the negation auxiliary adverb *niet* was corrected. Another twelve sentences tested verb tenses. And finally, ten sentences were set to interrogative types. In Round 2, all 37 short test sentences were generated correctly (100%).

Results Round 3 To test the capabilities of the system, 21 new sentences were selected from the Wikipedia corpus. Eleven sentences had a word count between seven and thirteen, ten more sentences had a length of fourteen to twenty words. The results can be found in Tables A.3 and A.4 in Appendix A.

Nine out of 11 medium long sentences were generated correctly (81.8%). These were all exact matches. Seven out of 10 long sentences were generated successfully (70%). This includes two sentences that did not match exactly, but are grammatically correct and have not changed in meaning.

Results Round 4 The last round generated 16 different variants of the same sentence. Ten variants were generated exactly and there were no mismatches accepted as correct. The future perfect (e.g. *...zal hebben* *gegooid.*) and conditional perfect (e.g. *...zou hebben gegooid*) incorrectly placed the auxiliary verb *hebben* 'have' before the object: *Marie zal hebben de bal gegooid.* 'Marie will have the ball thrown.', instead of *Marie zal de bal gegooid hebben.*. This is due to all auxiliary verbs (in this example: *zal* and *hebben*) being realised immediately after each other. The same problem causes mistakes in the perfect passive sentences. For example, the future perfect passive *De bal zal zijn geweest gegooid door Marie.* contains three auxiliary verbs followed by the main verb. Instead, the main verb should be placed after the first auxiliary verb, forming *De bal zal gegooid zijn geweest door Marie.*. A stylistic choice could be to swap the last auxiliary verbs *zijn* and *geweest*. From empirical experience, perfect passive sentences are not common and building support for them in SimpleNLG-NL was considered future work.

Overall results After four rounds, 74 out of 86 sentences were generated correctly (86.0%). 69 (80.2%) were exact matches. Leaving out the unit tests from Rounds 2 an 4, 22 out of 33 Wikipedia sentences matched exactly (66.7%). When including the accepted mismatches, 27 sentences were generated correctly (81.8%).

5.4 Known issues

There are some known issues. Some originate from SimpleNLG and its architecture, others were created in SimpleNLG-EnFr and SimpleNLG-NL. This section describes problems that came up.

Topicalisation is the restructuring of a sentence element to put more stress on said element. In Dutch, as in English, this is done by moving the element to the front of the sentence. In the first sentence *Op 1 oktober 1966 kwam hij vrij.*, the element is *op 1 oktober*. Normally, it would be placed at the end of the sentence, but to stress its importance, it is moved to the front. The bilingual SimpleNLG-EnFr does not include a proper method for this kind of structure. The generated sentences will still have the same meaning, but lack the stress on the element.

Using the `ClauseCoordinationRule` to aggregate clauses having the same subject, requires that not only the subject, but also premodifiers and postmodifiers are equal between the clauses. That means that the clauses can not have different modifiers added and, instead, elements such as prepositional phrases should be added as complements. The downside to using complements is that they appear in the order they were added. This might also be seen as an advantage, as it gives the user more control over word order.

When using a relative phrase, the debug mode built into SimpleNLG (activated using `realiser.setDebugMode(true)`) results in a StackOverflowError. Also printing the relative phrase in other places results in the same error. This is due to a bug in toString() for relative clauses. Java's Hashmap.toString() can not print Hashmaps that contain a self reference. This was the case when setting an object in the relative phrase feature `DutchFeature.RELATIVE_PHRASE`.

In aggregated sentences, modifiers and postmodifiers must match between the two clauses. If not, the aggregation will fail and only the first clause will be returned. Postmodifiers can be added by adding them to the verbs directly.

Known error: in Sentence wik_part0461/958283-5-1.xml, the word order in the om-te-infinitive is incorrect. The correct order would require splitting the verb group *elders in Amerika wonen*, which the current architecture of SimpleNLG makes difficult.

5.5 Conclusion and discussion

The iterative development process consisted of four rounds and resulted in 86.0% of all sentences being generated correctly. However, that number does not necessarily represent SimpleNLG-NL's coverage of the Dutch grammar. It represents the coverage of the grammar rules used in the target sentences and unit tests. The idea of the iterative approach was that it would provide a set of rules that would cover real world sentences and increase the capabilities of SimpleNLG-NL. The more target sentences were used, the larger the set of rules covered would be. However, this also means that grammar rules may not get implemented if the target sentences did not require the rule. Also, there is no end to the process of adding more target sentences, unless a coverage goal is set, for example by choosing a set of sentences or unit tests. A grammar reference would have provided a finite set of rules to implement and with that, a goal. On the other hand, grammar references can also be incomplete or may contain a rule set too large to be feasible for implementation, which may cause the developer to define his own subset of rules.

Then there is the potential of (small) differences between the official grammar and the common written language. Over time, languages change and evolve, but official grammar may not keep up with that. One has to choose between following the official, potentially archaic rules and the more up-to-date, yet potentially more chaotic or random language. That decision may depend on the NLG system's use case. For example, the second person singular of the verb *kunnen* 'can' is *kunt* in formal language, but the informal form has evolved to be *kan*. When generating conversational texts to simulate natural conversations, it may be suitable to base the system on commonly used grammar. For generating business reports, a more formal style is probably preferable.

Dutch grammar has shown to contain many exceptions and much flexibility. This is especially true for word order.

Word order has shown to be the most difficult to implement. Often a verb group had to be split apart and its objects had to be repositioned. In the case of Separable Complex Verbs, the separation of the preverb posed extra challenges, such as preverb positioning and extra steps during inflection.

Other challenging aspects were the pronunciation-based morphology rules. The lexicon does not contain information about pronunciation and SimpleNLG-NL and other versions of SimpleNLG do not support it. Instead, SimpleNLG-NL uses generalised rules based on spelling. While these rules seem to cover many cases, their coverage may not be complete. A system based on pronunciation data may be more complete, assuming that rules regarding Dutch pronunciation are more consistent than those for spelling.

In Chapter 9, future work on SimpleNLG-NL is described. One proposal would be to create a different, more flexible approach to word order. SimpleNLG for German (Bollmann, 2011) uses an extra feature in the input to determine the order. This allows more variation and control, but it has to be taken into consideration that that approach would move the choice of word order to the sentence planner. Another possibility would be to expand the analysis of the input structure and pick a different word order based on that. SimpleNLG-NL could have more word order rules for more situations. This could be achieved using the iterative development approach taken to build SimpleNLG-NL. However, it has to be made sure not to make large generalisations that cause other sentences to be generated incorrectly. Preventing generalisations requires many variations and situations to be covered. This approach can have the advantage of not having to alter the use of SimpleNLG-NL compared to that of SimpleNLG any more. Currently, the only Dutch-specific use is the addition of the feature for *te-infitives*. Keeping the use as consistent as possible would make multilingual generation easier, as the sentence planner would require fewer or no Dutch-specific rules.

File	Summary of changes
res/small-test.xml	created the minimal lexicon used during development
features/Gender.java	added `COMMON` gender
features/Pattern.java	updated Javadoc
features/dutch/DutchFeature.java	added `PREVERB` and `TE_INFINITIVE`
features/dutch/DutchLexical-Feature.java	added past forms and `PREVERB`
features/dutch/PronounType.java	added `REFLEXIVE` type
framework/LexicalCategory.java	added (unused) `NUMERAL` type
framework/NLGFactory.java	detect words with hyphens as words
lexicon/Lexicon.java	defined Dutch default words, e.g. co-ordination conjunction *en* ‘and’
lexicon/dutch/default-dutch-lexicon.xml	created a default lexicon (work in progress)
morphology/dutch/Morphology-Rules.java	added Dutch morphology rules
morphophonology/dutch/Morpho-phonologyRules.java	removed vowel elision rules
orthography/dutch/Orthography-Helper.java	added Dutch punctuation rules
phrasespec/SPhraseSpec.java	added check for Dutch `RELATIVE_PHRASE` feature
syntax/AbstractVerbPhrase-Helper.java	added handling of `PREVERB`
syntax/dutch/ClauseHelper.java	changed interrogative sentences and detection of subordinate clauses
syntax/dutch/NounPhrase-Helper.java	changed handling of adjective phrases
syntax/dutch/VerbPhraseHelper.java	changed syntax for verb phrases

Table 5.6: The files that were changed to implement Dutch grammar. Only the files changed after the initial copying of French grammar files and updating language independent files to include a third language are shown.

Sentence set	**Exact match**		**Accepted as correct**		**# total**
	#	% of total	#	% of total	
Round 1	8	66.7%	11	91.7%	12
Round 2	37	100.0%	37	100.0%	37
Round 3 (medium)	9	81.8%	9	81.8%	11
Round 3 (long)	5	50.0%	7	70.0%	10
Round 4	10	62.5%	10	62.5%	16
Total	69	80.2%	74	86.0%	86

Table 5.7: The final results of implementing SimpleNLG-NL. A generated sentence was considered an exact match only if every character, including punctuation, matched the target sentence. The sentence was 'accepted as correct' if it followed the criteria described in Section 4.2.

Chapter 6

Using SimpleNLG-NL with parse trees

6.1 Introduction

After the development of SimpleNLG-NL had ended, a small proof-of-concept was built to demonstrate a possible use case for SimpleNLG-NL. SimpleNLG-NL is just a surface realiser. Adhering to Reiter and Dale's (Reiter & Dale, 2000) steps of NLG described in Section 2.2, a surface realiser needs input from a sentence planner. When using an existing sentence planner, its output has to be converted into input suitable for the surface realiser.

Sentence planner In the case of SimpleNLG-NL, an attempt was made to link it to the Narrator, a story generator developed at the University of Twente (Theune et al., 2007). The Narrator contains it own basic surface realiser. The goal was to replace that surface realiser with SimpleNLG-NL. However, converting the sentence plans generated by the system proved difficult due to the Narrator's architecture being not clearly separable at the stage where SimpleNLG-NL could take over its surface realisation. Another attempt was made with a newer version of the Narrator, which has a slightly different architecture, however, that attempt was also stopped for similar reasons.

Other use cases One of the use cases for SimpleNLG-NL would be a content writer for (serious) games. The goal of this proof-of-concept was to enable content writers to write one sentence and get multiple variations of that sentence in return. The variations could differ in tense or any of the other features of SimpleNLG-NL. The writer would then pick one or more of the variations to use in the game. It is assumed that the system would make writing with more variety easier and speed up the writing process.

This method could also prove useful in other situations, like teachers writing learning material. Additional research may find it suitable for paraphrasing and text adaptation, which should increase the readability of the text, e.g. by reducing the number of modifiers or using easier synonyms. It could also quickly rewrite texts into a different tense, for example based on the end time of an event: a news website could write a report about a current event and, after the event has ended, generate it in the past tense. That would automate the manual rewriting of articles.

SimpleNLG-NL could also be used in a mix between template-based and rule-based NLG. It would then perhaps only realise parts of a sentence, or even just be used for inflection. This is how SimpleNLG-NL will probably be used in the POSTHCARD[1] project. This project is currently developing a simulation of Alzheimer patients to provide caregivers with realistic situations that can help them in their interaction with patients. The simulations will be built in English, French and Dutch. For the Dutch part, SimpleNLG-NL will be used.

For this proof-of-concept, the use case of the content writer for games will be used.

Parsers Instead of using a full NLG system, it was chosen to use a parse tree as input. A parse tree is the result of a dependency parser and contains information about the words and phrases in a sentence and their relations (dependencies). The information it contains is similar to that of a sentence planner. This information can then be converted for SimpleNLG-NL. Three parsers capable of parsing Dutch were considered. The Stanford POS Tagger supports Dutch (Toutanova et al., 2003). However, it only assigns parts-of-speech without dependency structures, which makes it

[1]`http://posthcard.eu/`

unsuitable for this project. Sadly, the sibling project Stanford Parser does not support Dutch. A second option would be Frog (Bosch et al., 2007). Frog is a project that consists of multiple modules, one of which is a dependency parser. The outcome of this parser would be usable for conversion. The third option is using Alpino. As Alpino was also used when manually writing the input code for the target sentences while developing SimpleNLG-NL, it had already shown to be useful. The parse trees it generates contain detailed information about the constituents of a sentence and can be exported in XML, which is easy to use in software. Note that the Frog dependency parser was trained on the Alpino treebank. Alpino was chosen as the parser to generate parse trees that can be converted to SimpleNLG-NL input.

Converter As part of the proof-of-concept, a converter was written. This converter would take the parse tree from Alpino and generate the input for SimpleNLG-NL. The method of conversion is described in Section 6.2. The following features were used for the alternative sentences: tenses, perfect form and active or passive form.

This chapter describes the process of developing the proof-of-concept and the subset of Dutch grammar it covers. The system has no graphical user interface, yet.

6.2 Method

The proof-of-concept consists of three modules: the parser, the converter and the realiser. The Alpino parser reads the input sentence and writes the parse tree it generated to an XML file. This file is read by the converter. The converter then calls SimpleNLG-NL functions corresponding to the tree structure it finds in the XML file. This would result in what Simple-NLG calls the ‘initial tree’ of SimpleNLG elements, before any realisation is performed. Lastly, SimpleNLG-NL realises multiple variants based on the initial tree. An overview of these steps can be found in Figure 6.1.

To write the converter, an iterative process similar to that of the development of SimpleNLG-NL was used. Simple sentences were parsed by Alpino, of which the results were manually checked to confirm their cor-

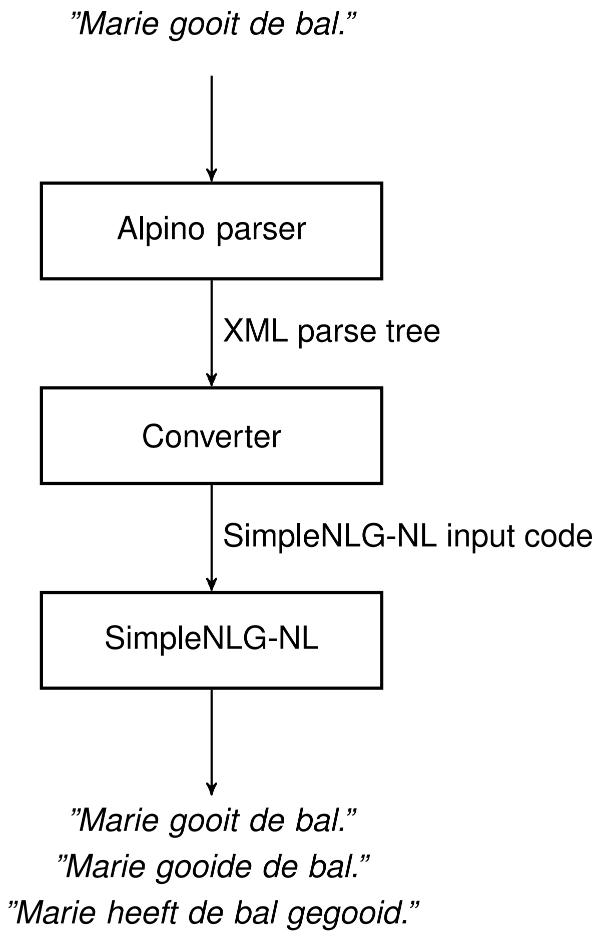

Figure 6.1: An overview of the three modules that comprise the proof-of-concept system modules and their interaction.

rectness. The code architecture of the converter was loosely based on that of 'nlgserv'[2], a server written in Python that converts JSON objects into SimpleNLG input. It was written as part of the PhD project of data scientist Darren Richardson[3] and the project forks indicate it is being used by business intelligence company Nugit and Zato Novo, which focuses on financial data and social media analysis. After the basic methods of the converter were written, they were extended when a new node structure was found. A match had to be made between the XML structure and the tree structure required for SimpleNLG-NL. In the following subsections,

[2]`https://github.com/mnestis/nlgserv`
[3]`https://mnestis.net/`

this conversion is explained.

6.2.1 Alpino XML structure

An XML parse tree consists of `<node></node>` elements which can have child nodes. Each node has attributes that provide information about the node. For example, the `rel` attribute contains the dependency label, such as `mod` for 'modifier' or `su` for 'subject'. An example tree can be found in Listing 6.1.

```
<?xml version="1.0" encoding="UTF-8"?>
<alpino_ds version="1.6">
  <parser .../>
  <node cat="top" rel="top">
    <node cat="smain" rel="--" ...>
      <node cat="np" rel="su">
        <node lemma="mijn" infl="pron" rel="det" .../>
        <node lemma="zoon" getal="ev" pos="noun" rel="hd" .../>
      </node>
      <node lemma="zitten" pos="verb" rel="hd" .../>
      <node cat="pp" rel="ld">
        <node lemma="in" pos="prep" rel="hd" .../>
        <node cat="np" rel="obj1 ...">
          <node lemma="die" pos="det" rel="det" .../>
          <node lemma="zaal" pos="noun" rel="hd" .../>
        </node>
      </node>
    </node>
  </node>
  <sentence sentid="27">mijn zoon zit in die zaal</sentence>
  <comments>
    <comment>Q#27|mijn zoon zit in die zaal|1|1|-3.550817951410001</
        ↪ comment>
  </comments>
</alpino_ds>
```

Listing 6.1: Example of an Alpino XML dependency tree. Unused attributes have been left out for readability.

The dependency label `rel` attribute has shown to be useful for the conversion process, as it is the attribute that has values similar to the names of the elements and methods used in SimpleNLG-NL. The most straight-

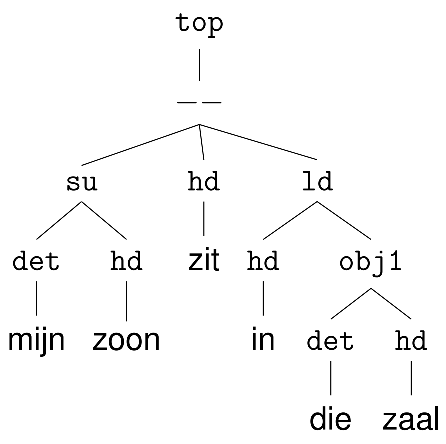

Figure 6.2: The parse tree for the sentence *Mijn zoon zit in die zaal.* 'My son is in that room.' as Alpino returns it. Every node in this graph is a representation of a `node` element in the XML file and is named after the `rel` attribute of the node.

forward example of this is the node with the value of `rel` being `su`: this node contains the subject, which could be set for SimpleNLG-NL using `sentence.setSubject("<subject>")` where `<subject>` is retrieved from the parse tree by reading the value of a different attribute of the node, in this case the `lemma` attribute. This small example demonstrates the process of writing the converter. For each possible value of `rel`, determine what element of SimpleNLG-NL it corresponds to and then find the data required by SimpleNLG-NL in the dependency tree. In some cases, this was simply a case of reading a different attribute from the same element, in other cases it required traversing the tree to look for the required attribute of child elements. In this example, the process was straightforward, as can be seen by comparing the graph structure coming from Alpino (6.2) with the input tree for SimpleNLG-NL (6.3). The tree structure is the same when looking at the SimpleNLG-NL variable classes. However, building the elements and populating them takes a slightly different approach. The tree in Figure 6.4 shows the methods (functions) used to build the example sentence. It shows that SimpleNLG-NL variables are highly dependent on each other.

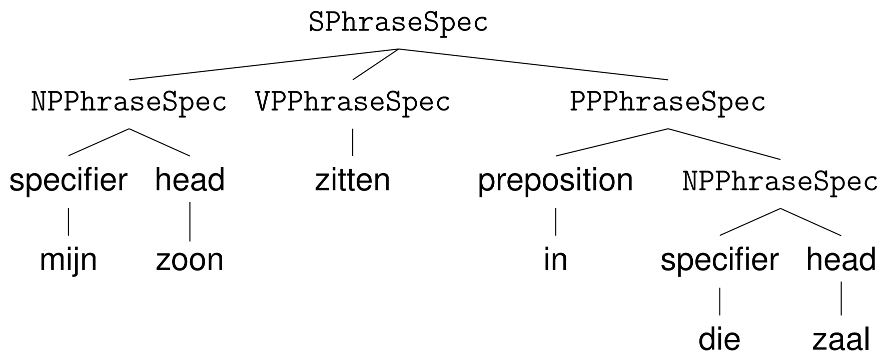

Figure 6.3: The dependency tree for the sentence used in Figure 6.2 using SimpleNLG-NL element types. Every node in this graph represents an element used in SimpleNLG-NL.

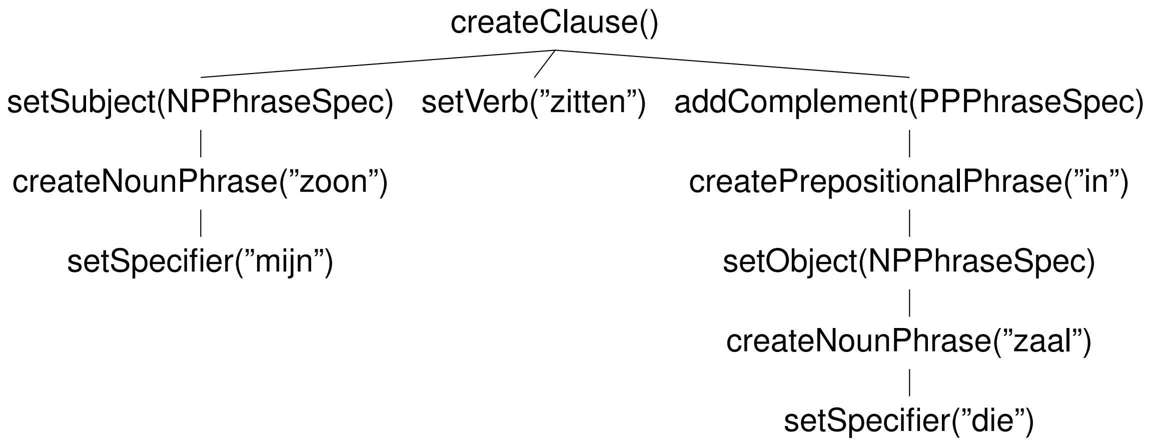

Figure 6.4: The tree for the sentence used in Figure 6.2 in the form of SimpleNLG-NL methods. While this strictly not a dependency tree, it provides a visual comparison between the structure of a sentence and the structure of SimpleNLG-NL input. Every node in this graph shows the Java method used to create the required element. Methods in child nodes are applied to the variable created by their parent. If a class name is used as a parameter, it means that the parameter is of that type and is created using the method in its child node.

6.2.2 Subjects

Every sentence has to have a subject. As stated earlier, retrieving the subject from the XML tree is potentially very straightforward: simply navigate the tree until a node with attribute `rel="su"` is found and then read its `lemma` attribute. However, as can be seen in the example of Listing 6.1, that node has child nodes and is parsed as being of category `np` (noun phrase). Such child nodes make the code for SimpleNLG-NL more complex than simply e.g. `sentence.setSubject("hij")`. Instead, a `NPPhraseSpec` noun phrase element is created and populated with the available data: the head noun and the specifier/determiner, as can be seen in Listing 6.2. The head noun and specifier have to be collected from the children with `rel="det"` or `rel="hd"`. Note that the code also removes any underscores in lemma, which were added in compound words by Alpino, but are not wanted in the input for SimpleNLG-NL.

Lastly, any modifiers have to be applied. If one of the children has the dependency label `mod`, its lemma is used as input for `np.addModifier()`. While the parse tree can provide information about the position of a modifier (e.g. `positie="prenom"`), this information is not used, as it should not be the task of a sentence planner to determine the correct word order. Instead, it is the task of SimpleNLG-NL to position it accurately. Note that modifiers are not necessarily mere adjectives, but can also be prepositional phrases or other complex structures. Because of this fact, each `rel="mod"` node has to be checked for children, which have to be handled according to their own dependency labels, as described in Section 6.2.5.

6.2.3 Verb phrases

The second input that is required by SimpleNLG-NL, is the verb. The verb node can be identified by the `rel="hd"` ('relation=head') attribute in combination with the `pos="verb"` attribute. However, in cases of perfect, future or conditional tenses, that node is just the auxiliary verb. Dutch verb phrases can have multiple auxiliary verbs. A proper analysis of the sentences should result in the same initial SimpleNLG-NL input tree, regardless of any syntax and inflection based on tenses. Features like tense and

```
// Variable elem contains the node with attribute rel="su"

// Create the SimpleNLG-NL noun phrase element
NPPhraseSpec np = factory.createNounPhrase();

GetHeadElementReturn headElement = getHeadElement(elem);
// Rejoin lemma that was split into original words by Alpino
String head = headElement.lemma.replaceAll("_", "");

// Set lemma
np.setNoun(head);
// Set determiner
np.setSpecifier(getNPSpecifier(elem));

// Add copied feature
np.setFeature(Feature.NUMBER, headElement.number);

processModifiers(np, elem);
```

Listing 6.2: Source code for converting a Alpino subject element (`rel="su"`) to a SimpleNLG-NL noun phrase. First, an empty noun phrase is created, which is then populated with the head noun and a specifier. Lastly, the number (singular or plural) is copied from the parse tree element to the new noun phrase.

form can be reapplied later, which should result in the auxiliary verbs being added back by SimpleNLG-NL. Verb phrases can not only be found directly under the root of the sentence, but also within complements. This is described in Section 6.2.6.

While traversing the tree, an element with the attributes `rel="hd"` and `pos="verb"` will be found. This is the head verb node. Usually, this node is the main verb, of which the `lemma` attribute contains the verb that can be sent to SimpleNLG-NL's `sentence.setVerb()`. However, in some cases this node will be found to be just an auxiliary verb indistinguishable from the main verb based on the attributes added by Alpino. This is the case if one of the nodes found later in the tree traversal is a passive/perfect participle (`cat="ppart"`), present participle (`cat="ppres"`) or a verbal complement (`rel="vc"`). This particle or verbal complement indicates that the previously found verb was only an auxiliary verb. To find the main verb, the converter has to read the `lemma` attribute of the participle node or one of its child nodes, see the pseudo-code given in Listing 6.3. For all cases, the converter looks for the node with the `rel="hd"` attribute and use its lemma in `sentence.setVerb(<lemma>)`.

```
for node in tree:
  if node has rel=hd and pos=verb:
    // This node has a head verb. For now, assume it is the
        ↪ main verb. A later loop may find this verb was only
        ↪ an auxiliary verb.
    set e = getVerbElementUsingNodeLemma()

  else if node has cat=ppart, cat=ppres or rel=vc:
    // The node is a participle or verbal complement, so the
        ↪ earlier found verb was only an auxiliary verb. Find
        ↪ the main verb in this node or its children instead.
    set e = getVerbElementFromChildNodes()

  // Use the generated verb phrase element as verb
  sentence.setVerb(e)


function getVerbElementFromChildNodes():
  if node has no children:
    return getVerbElementUsingNodeLemma()

  else:
    // The node has children, so find the head element.
    create empty verb phrase
    get head verb from child lemma
    add head verb to verb phrase
    return verb phrase
```

Listing 6.3: Pseudo-code describing how the main verb of a phrase is found.

6.2.4 Objects

Objects are identified by `rel="obj1"` (direct object) or `rel="obj2"` (indirect object). Either of those can have child nodes. If they do not, the value of the `lemma` attribute of the current node is taken and used as a string in `sentence.setObject(<lemma>)` or `sentence.setIndirectObject(<lemma>)`, respectively. If the node does have children, the node is unpacked as a noun phrase, similar to noun phrases used as subjects. This allows the object to have a specifier and modifiers.

6.2.5 Modifiers

Modifiers can be applied to `NPPhraseSpec`, `VPPhraseSpec` and the prepositional phrase `PPPhraseSpec`, but also to `SPhraseSpec`, the main clause. Nodes are marked as modifiers using the `rel="mod"` attribute. While SimpleNLG only differentiates between adverbs and adjectives, Alpino parse trees can have more complex structures. Based on the `cat` attribute, the converter only supports single adjectives (`adj`), single adverbs(`adv`) and prepositional phrases (`pp`) as modifiers. Multi-word adjectival or adverbial phrases are not supported, as those are not properly supported by SimpleNLG. Other more complex structures, such as noun phrases as modifiers, are also not handled by the converter.

6.2.6 Complements

Similar to modifiers, complements come in many shapes in Alpino parse trees. Based on their `rel` and `cat` attributes, the types currently supported are adjective phrases (with both attributes `rel="predc"` and `cat="ap"`), noun phrases (`rel="predc"` and `cat="np"`), locative or directional phrases (`rel="ld"` and `cat="pp"`) as prepositional phrases, other prepositional complements (`rel="pc"` and `cat="pp"`) and te-infinitive verbal complements (`rel="vc"` and `cat="ti"`). Each of these types can be expanded if the node has child nodes.

Two structures are particularly interesting: prepositional complements require the unpacking of the child nodes to find the preposition node and use its `lemma` in SimpleNLG-NL's `pp.setPreposition(<lemma>)`. They can also contain a direct object, which can be either a single noun or a noun phrase.

The other interesting structure is the te-infinitive. Even though it has the attribute `rel="vc"`, this type of complement is excluded from the search for the main verb described in Section 6.2.3, as the infinitive should not be used as the main verb in SimpleNLG-NL, but as a complement. Te-infinitive structures consist of a main verb that is followed by *to* and an infinitive. For example: *probeert te gooien* 'tries to throw'. The main verb can be found in the same place as other main verbs in Alpino parse trees and the *te* is marked as the complementiser of a verbal complement. However, the infinitive is buried within a node with `rel="body"`. This node can also contain noun phrases as objects, such as *de ball* 'the ball' in *Marie probeert de bal te gooien.* 'Marie tries to throw the ball.'. An example of this structure can be found in Listing 6.4. Currently not supported are om-te-infinitives, which contain the extra adverb *om* ('to') to be set as a complementiser.

```
...
  <node cat="ti" rel="vc" ...>
    <node lemma="te" rel="cmp" .../>
    <node cat="inf" rel="body" ...>
      <node rel="su" .../> <!-- Empty node. -->
      <node cat="np" rel="obj1" ...>
        <node lemma="de" pos="det" rel="det" .../>
        <node lemma="bal" pos="noun" rel="hd" .../>
      </node>
      <node lemma="gooien" pos="verb" wvorm="inf" rel="hd" .../>
    </node>
  </node>
...
```

Listing 6.4: Part of the Alpino XML dependency tree for the sentence *Marie probeert de bal te gooien.* 'Marie tries to throw the ball', containing the te-infinitive phrase *te gooien*. The number of attributes per node has been limited for readability.

6.3 Evaluation

The proof-of-concept consists of three modules: the Alpino parser, the converter and SimpleNLG-NL. Each of these modules can make mistakes. SimpleNLG-NL is evaluated as described in Section 5.3. The now following section describes the results of the converter. It assumes that the Alpino parser is perfect and will provide a dependency tree that can be used to generate the sentence, given that the conversion and SimpleNLG-NL are correct, too.

Grammatical coverage The grammatical coverage of the converter can be described using a tree structure. The order of siblings does not matter, but the relations between parent and child do. The tree can be found in Listing 6.5.

The conversion does not cover the entirety of Alpino's parse options. While the converter aims to use as general attributes as possible, it may come across structures that use more specific attributes or unexpected branch structures. Another possibility is that SimpleNLG and SimpleNLG-NL do not support the grammatical features. One example are *als..., dan...* 'if..., then' sentences. There is currently no feature in SimpleNLG-NL that allows such sentences. Therefore, input sentences with such structures are not analysed by the converter.

A structure that is supported by SimpleNLG-NL, but not by the converter, is the multi-word unit. For instance, names or dates consisting of multiple words can be used as input for SimpleNLG-NL in the form of canned text. The converter currently has no method built-in to handle such nodes, but such a method could easily be implemented by concatenating the `word` attribute values of the node that make up the multi-word unit.

Verb phrases A simple test was performed using eight different forms of the same sentence as input for the converter, see Table 6.1. All parsings and conversions result in the same SimpleNLG-NL input tree, which was manually checked and compared with the tree of a manual input. One difference between manual and generated trees occurring in specific cases, is the extra nested `VPPhraseSpec` that could be added by SimpleNLG itself.

```
subject:
  noun
  OR
  noun phrase:
    specifier
    noun
    modifiers
modifiers
verb:
  verb
  OR
  verb phrase:
    object: (noun OR noun phrase)
    verb
modifiers:
  adjective:
    comparative/superlative
  adverb
  prepositional phrase:
    preposition
    object: (noun OR noun phrase)
    modifiers
  predicative adjective:
    comparative/superlative
direct object: (noun OR noun phrase)
indirect object: (noun OR noun phrase)
complements:
  adjective phrase:
    adjective
  noun phrase: (noun OR noun phrase)
  directional complement:
    preposition
    OR
    prepositional phrase: ...
  te-infinitive:
    verb
    object (np)
    modifiers
    complements (first level)
```

Listing 6.5: The grammatical coverage of the converter. The structure of a noun phrase and a prepositional phrase is expanded only once in this notation. The converter reuses methods to generate the elements.

Verbs in SimpleNLG can be passed to the `sentence.setVerb()` method as a parameter in the form of a text string or a `VPPhraseSpec` object.

When a string is used, see Listing 6.6, SimpleNLG creates a `VPPhraseSpec` object and adds a child element of type `WordElement` (the word itself) to it. In the sentence tree of SimpleNLG, just these two elements are added.

```
// Create the clause element
SPhraseSpec sentence = factory.createClause();

// Set the subject
sentence.setSubject("Marie");

// Set the main verb using a string
sentence.setVerb("gooien");

System.out.println(realiser.realiseSentence(sentence));
// Prints: Marie gooit. ‘Marie throws.’
```

Listing 6.6: Example of using a string as input for `setVerb()`.

The sentence planner (or, in this case, the converter) can also want to pass a `VPPhraseSpec` object, instead of a string, because it allows for the addition of modifiers, complements and features to the verb itself. This is needed, for example, when setting the verb form of this specific verb phrase to infinitive. To take this approach, the converter creates a new `VPPhraseSpec` object using `factory.createVerbPhrase("gooien")` and storing it in a variable. That variable is then passed as parameter to `sentence.setVerb(verbPhrase)`. An example of this is shown in Listing 6.7. The resulting tree in SimpleNLG contains a `VPPhraseSpec` (created by default) with a child in the form of the `VPPhraseSpec` created by the converter, which in turn has a `WordElement` as child. The automatic creation of the `VPPhraseSpec` by SimpleNLG should not be necessary if the verb phrase created by the sentence planner can be used or copied, but currently, this is strange behaviour of the original SimpleNLG which can be classified as a bug. SimpleNLG-NL currently is not able to handle this extra layer under some circumstances.

When using a `VPPhraseSpec` as input, SimpleNLG-NL behaves in an unexpected way. At the end of the verb phrase, the main verb is repeated in its infinitive form (e.g. *zal hebben **gooien gooien*** instead of the expected *zal hebben* ***gegooid***). This seems to be caused by an error in the `VerbPhraseHelper.java`, where both the parent and child `VPPhraseSpec`s are realised. This is an odd bug in SimpleNLG-NL that only happens for future and conditional tenses and should be resolved. It can possibly also be solved by solving the bug in the `setVerb()` method of the original SimpleNLG mentioned earlier.

```
// Create the clause element
SPhraseSpec sentence = factory.createClause();

// Set the subject
sentence.setSubject("Marie");

// Create the verb phrase
VPPhraseSpec verbPhrase = factory.createVerbPhrase("gooien");

// Add modifiers, complements and features directly to this verb
gooien.addModifier("hard");

// Use the verb phrase as the main verb
sentence.setVerb(verbPhrase);

System.out.println(realiser.realiseSentence(sentence));
// Prints: Marie gooit hard. ‘Marie throws fast.’
```

Listing 6.7: Example of using a `VPPhraseSpec` as input for `setVerb()`.

As shown in Appendix B Table B.1, there were some problems when using a string as input for the future and conditional tenses. While building the proof-of-concept, it became clear that a fourth round of development of SimpleNLG-NL was needed, because of an error in the positioning of objects in the future and conditional tenses. The object was placed in the SVO position, while these tenses require SOV ordering. The future and conditional tenses did not appear in the Wikipedia sentences, which is why the word order changes were not accounted for. Note that the short test sentences of Round 2 did include those tenses, but the sentences did not include objects, without which the order seemed correct. Round 4 solved the problem. However, there are still problems related to SimpleNLG positioning auxiliary verbs incorrectly, but the input for it is generated as expected.

The final output results can be found in Appendix B Table B.3. This test after Round 4 included 16 combinations of the following features: `Feature.TENSE` (present, past, future and conditional), `Feature.PASSIVE` (true and false) and `Feature.PERFECT` (true and false). The features chosen for this proof-of-concept were: four tenses (present, past, future and conditional), two voices (active and passive) and both simple and perfect form. This resulted in 16 variants of the same sentence.

Another feature SimpleNLG-NL found to be lacking, is the gerund form (e.g. *het gooien* 'the throwing'). It was not part of the Wikipedia sentences, and did not get implemented.

Testing more complex sentences To be able to generate variants of a sentence, the base sentence has to be extracted from the input sentence written by the human author. To test that functionality of the converter, simple sentences were written with increasing complexity as a form of unit tests. Each sentence was written with different combinations of tenses and perfect forms. Currently, passive voice is not supported, but it is detected and marked by the Alpino parser, so support can be added to the converter in future work.

The first set of input variations was based on the basic sentence *Marie gooit.* 'Marie throws', see Table 6.1. These sentences were fed into the converter and should all result in the same SimpleNLG tree. If this is the case, the human writer can write the input sentences in any of these forms and the system will generate variations of them. For these tests, only the present simple was generated, which confirms that all input variations were converted correctly.

The second set of input sentences include an object: *Marie gooit **de bal**.* 'Marie throws the ball.' As shown in Table 6.2, all eight sentences were converted correctly. Internally, SimpleNLG and SimpleNLG-NL add objects as children to the verb phrase, while Alpino sees the object node as siblings of the main verb node. The converter uses `sentence.setObject()` as input (not `verbPhrase.setObject()`), which is the method described on the SimpleNLG Wiki[4]. This demonstrates one of the differences between the structure of Alpino and SimpleNLG-NL.

[4]`https://github.com/simplenlg/simplenlg/wiki/Section-V-%E2%80%93-Generating-a-simple-sentence`

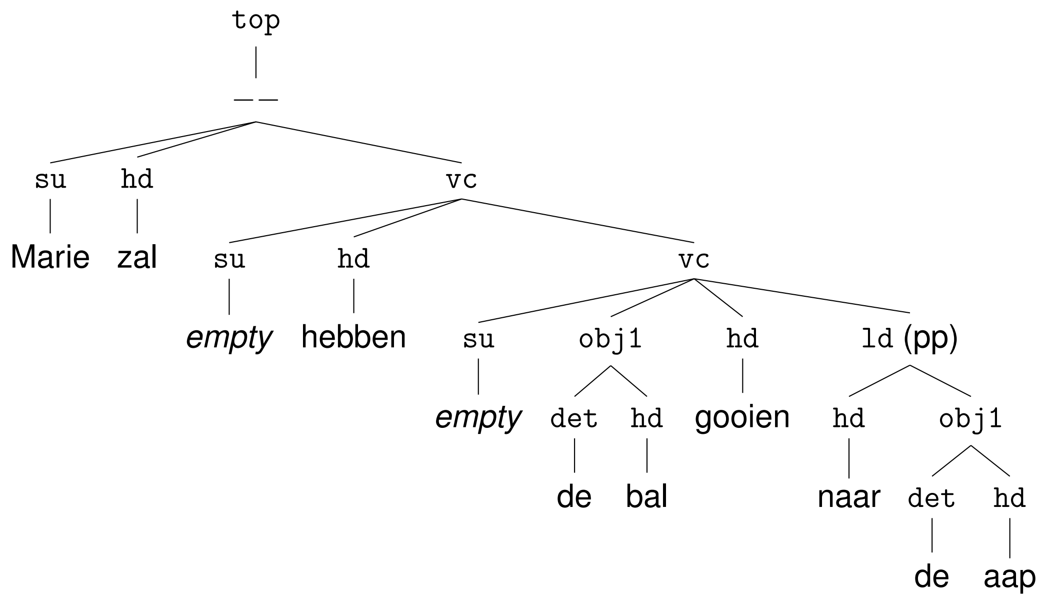

Figure 6.5: Parse tree for the future perfect sentence *Marie zal de bal gegooid hebben naar de aap* 'Marie will have thrown the ball to the monkey'. This structure was found to be too complex for the converter.

The third set added a prepositional phrase: *Marie gooit de bal **naar de aap**.* 'Marie thows the ball to the monkey', see Table 6.3. It also tested two different positions for the prepositional phrase, which requires Alpino to detect the phrase properly and use similar markings as with the other positioning, so the converter can handle it. The future perfect and conditional perfect sentences fail, because the converter is not able the extra complexity in Alpino's output. Alpino uses a structure that includes nested verbal complements, of which the deepest has its own complement, see Figure 6.5. This last complement (the prepositional phrase) is not handled by the converter.

The fourth set tested the addition of modifiers to the object (*de **zware** bal* 'the heavy ball'), the subject (***Sterke** Marie* 'Strong Marie') and the object of the prepositional phrase (*de **snelle** aap* 'the quick monkey'), respectively. The results are similar to those of the fourth set, see Appendix Table B.4.

Lastly, an adverb was added to the sentence. Two sentences contained the adverb *hard* 'hard'/'fast' in different positions, which should result in the same input for SimpleNLG-NL. The adverb was placed either before or after the object. The converter handled both cases correctly and applied the adverb to the SimpleNLG-NL sentence element. SimpleNLG-NL generated the sentences with the adverb being placed before the object: *Sterke Marie gooit hard de zware bal naar de snelle aap* 'Strong Marie throws the heavy ball to the monkey fast.'

Input sentence	**Tense**	**Form**	**Output sentence**
			present simple
Marie gooit	present	simple	Marie gooit.
Marie heeft gegooid	present	perfect	Marie gooit.
Marie gooide	past	simple	Marie gooit.
Marie had gegooid	past	perfect	Marie gooit.
Marie zal gooien	future	simple	Marie gooit.
Marie zal gegooid hebben	future	perfect	Marie gooit.
Marie zou gooien	cond.	simple	Marie gooit.
Marie zou gegooid hebben	cond.	perfect	Marie gooit.

Table 6.1: Eight different input sentences resulting in the same tree structure when converted. The present simple is generated as *Marie gooit*, which is the desired outcome.

6.4 Conclusion and discussion

The proof-of-concept shows that it is possible to create a converter that converts Alpino parse trees into input code for SimpleNLG-NL. The converter developed for this proof-of-concept is capable of reading basic tree structures in multiple tenses and forms and convert them into input code for SimpleNLG-NL. It also applies a set of features to the generated initial tree, which result in SimpleNLG-NL realising multiple variants of the same input sentence. The features chosen for this proof-of-concept were:

Input sentence	**Tense**	**Form**	**Output sentence**
			present simple
Marie gooit de bal	present	simple	Marie gooit de bal
Marie heeft de bal gegooid	present	perfect	Marie gooit de bal
Marie gooide de bal	past	simple	Marie gooit de bal
Marie had de bal gegooid	past	perfect	Marie gooit de bal
Marie zal de bal gooien	future	simple	Marie gooit de bal
Marie zal de bal gegooid hebben	future	perfect	Marie gooit de bal
Marie zou de bal gooien	cond.	simple	Marie gooit de bal
Marie zou de bal gegooid hebben	cond.	perfect	Marie gooit de bal

Table 6.2: Eight different input sentences with added objects resulting in the same tree structure when converted.

four tenses (present, past, future and conditional), two voices (active and passive) and both simple and perfect form. This resulted in 16 variants of the same sentence. Note that perfect passive sentences and perfect active senses in the future and conditional tenses were not generated correctly by SimpleNLG-NL, see Appendix B Table B.3. A total of 10 different variants were generated correctly.

Rather than simply creating many variants of a sentence at once, other use cases may require specific features to be chosen, for instance when rewriting text in a different tense. One case to which this could be applied is news articles that describe events that have completed after the original text was written in the future tense.

The converter is very basic. Not all sentence structures provided by the Alpino parser can be handled by the converter. More work can be done to increase the overlap. However, even if the converter could read all sentences structures properly, the structures may not be supported by SimpleNLG-NL. SimpleNLG-NL can also be improved upon. Because none of these modules is perfect, there is a risk of multiplying errors if the next module receives incorrect input.

While it needs additional work to support more than just basic sentences, the proof-of-concept shows that the approach is suitable for reading parse trees and converting them into input for SimpleNLG-NL, which can then generate 10 sentence variants.

Input sentence	**Tense**	**Form**	**Output sentence**
			present simple
Marie gooit de bal naar de aap	pres.	simple	Marie gooit de bal naar de aap.
Marie heeft de bal gegooid naar de aap	pres.	perfect	Marie gooit de bal naar de aap.
Marie heeft de bal naar de aap gegooid	pres.	perfect	Marie gooit de bal naar de aap.
Marie gooide de bal naar de aap	past	simple	Marie gooit de bal naar de aap.
Marie had de bal gegooid naar de aap	past	perfect	Marie gooit de bal naar de aap.
Marie had de bal naar de aap gegooid	past	perfect	Marie gooit de bal naar de aap.
Marie zal de bal gooien naar de aap	fut.	simple	Marie gooit de bal naar de aap.
Marie zal de bal naar de aap gooien	fut.	simple	Marie gooit de bal naar de aap.
Marie zal de bal gegooid hebben naar de aap	fut.	perfect	*Marie gooit de bal.*
Marie zal de bal naar de aap gegooid hebben	fut.	perfect	*Marie gooit de bal.*
Marie zou de bal gooien naar de aap	cond.	simple	Marie gooit de bal naar de aap.
Marie zou de bal naar de aap gooien	cond.	simple	Marie gooit de bal naar de aap.
Marie zou de bal gegooid hebben naar de aap	cond.	perfect	*Marie gooit de bal*
Marie zou de bal naar de aap gegooid hebben	cond.	perfect	*Marie gooit de bal*

Table 6.3: Fourteen different input sentences with prepositional phrases which should have resulted in the same tree structure when converted. The difference between two sentences of the same tense and form is the word order, which is a stylistic choice in these cases. The sentences in *italic* are missing the prepositional phrase. This is due to the extra complexity in Alpino's output, which is currently not supported by the converter.

Chapter 7

Discussion

Developing SimpleNLG-NL in an iterative manner was a novel approach. This approach had the advantage of resulting in real-world sentences being realised. Another advantage was the ability to re-use existing French grammar rules as a base. With every sentence, the grammatical coverage of SimpleNLG-NL increased.

The downside of this approach is that there is no predetermined end to the process. While languages themselves are of an ever-changing nature, working through a (basic) grammar reference would have set a clear goal and increased the confidence that the result is a usable product. Instead, the limit was set by the number of sentences, their complexity and time. Grammar references were only used to determine the rules for the problem encountered. The current approach shows there is room for improvement of the grammatical coverage, but it also provides a method that can be used to continue the work.

Writing unit tests for an NLG system has shown to be difficult. There are many small components that can be tested individually, but it is unpractical to test all possible combinations. Grammatical references could have provided example sentences to be adapted. This could have provided more small, specific unit tests. Ideally, they would also include more complex examples. Also, the unit tests used for other languages could have been adapted more literally as far as the tested features were not specific for that particular language. Currently, the unit tests are only loosely based on those for the English SimpleNLG. While those tests can not guarantee to test all combinations either, future work may be to adapt them to Dutch more literally.

While other adaptations of SimpleNLG used real world sentences for evaluation, SimpleNLG-NL was evaluated during the development iterations. This was part of the iterative approach, but it may not give a statistic representation of the subset of grammar covered because of the relatively small number of sentences. A good test would be to generate another, large set of sentences without changing SimpleNLG-NL.

The proof-of-concept demonstrated that it is possible to use parse trees to generate SimpleNLG-NL input from. Currently, only basic sentences can be generated using this method. Alpino can parse sentences in great detail and it is the task of the converter to recognise all structures and translate it into a structure suitable for SimpleNLG-NL. However, since SimpleNLG-NL has a more simplified structure, elements can be lost in that translation. Alpino was probably the best choice of parser because of its completeness, but an extra project could be dedicated to making a converter that can handle more sentence structures.

Chapter 8

Conclusions

This project resulted in SimpleNLG-NL and an Alpino to SimpleNLG-NL converter. SimpleNLG-NL provides surface realisation in Dutch. Using an iterative development process, the grammatical coverage was increased to over 80% of the test sentences, with a few differences in punctuation and word order. However, this number only represents the coverage of grammar used in the target sentences. The way the user or sentence planner can manipulate word order can probably be improved upon, as described in Chapter 9.

The iterative development process of SimpleNLG-NL answered research questions R1.1, R1.2 and R1.3. The development method showed a way to implement Dutch grammar by evaluating the results intermediately based on correctness criteria. After a round of development, unit tests were written to evaluate the state of the system and improve upon it. A third round of medium and long sentences acted as another form of evaluation and revision. After building the converter, a fourth round corrected some mistakes in the future and conditional tenses. In total, 16 files had to be changed to implement Dutch grammar after initially copying 12 French-specific files and changing 8 language independent files to add the support for the third language. Research question R2 was answered by describing a use case and building a prototype for it. The prototype made use of parse trees by reading its structure and converting it to input code for SimpleNLG-NL. This allows the user to enter a sentence and receive 16 variations on that sentence in return, based on tense, perfect and active/passive form. This can speed up the process of writing scenarios, dialog or other texts used in serious games.

There are many possible uses for Natural Language Generation. For example, other NLG systems are used for generating reports on weather or business intelligence. SimpleNLG-NL will be used in the POSTHCARD project, performing surface realisation for the simulation of an Alzheimer patient. This project will give SimpleNLG-NL the opportunity to prove its usefulness for society.

SimpleNLG-NL will be released under the Mozilla Public License [1], which is the same as SimpleNLG. The source code is accompanied by comments and Javadoc information. Additionally, the SimpleNLG Wiki [2] will be adapted for Dutch. The converter built for the proof-of-concept will be available on request.

A short paper describing SimpleNLG-NL has been submitted to a conference and is awaiting approval.

Hopefully, the tools developed during this project will be useful to researchers, companies or anyone else interested in Natural Language Generation.

[1] `https://www.mozilla.org/en-US/MPL/`

[2] `https://github.com/simplenlg/simplenlg/wiki`

Chapter 9

Future work

The iterative process of improving SimpleNLG-NL will never truly end. More work needs to be done to make sure all features in SimpleNLG work correctly for Dutch. After that, new features can be added to further enhance its capabilities. One feature that could be added is topicalisation.

Another grammatical aspect which could be handled differently is that of word order. Currently, the sentence planner can alter word order using premodifiers, postmodifiers and complements. While this does make many structures possible, it is not very flexible when changing the order afterwards, because it would require restructuring the input tree itself, rather than just the realisation order. A better approach may be that of SimpleNLG for German, in which Bollmann (2011) provided a feature to choose the word order. This feature would have to be adapted for the new architecture of SimpleNLG version 4. When giving the sentence planner the ability to change the word order, more variants can be generated. This would provide interesting research opportunities on stylistic choices or affective language. Note that this feature would have to be set by the sentence planner, essentially moving the decision of word order to the sentence planner, which it should not do.

There are many uses for NLG systems. More research could be done for the use in paraphrasing and text adaptation. Also, SimpleNLG-NL could be linked to an existing sentence planer, perhaps the Narrator.

The current converter used in the proof-of-concept is very basic. Improvements could be made by adding support for more Alpino categories and dependency labels. If the sentence rewriting process is to be used, a graphical user interface would make it easier to use.

The use of SimpleNLG-NL in the POSTHCARD project may add new requirements for the system. New features may be added based on such requirements.

Acknowledgements

I would like to express my greatest gratitude towards dr. Mariët Theune. As my main supervisor she provided me with many insights and feedback. We had many interesting discussions on unusual grammatical features and other linguistic issues. While this was my first encounter with Natural Language Generation, she sparked my interest in the field and helped me get familiar with it. My second committee member, prof. dr. D.K.J. Heylen has provided direct and constructive feedback on the draft version for this thesis, for which I am thankful.

I would also like to thank my mother, the rest of my family and my friends. After losing my father at the beginning of this project, they provided me with the confidence and strength to finish my thesis and graduate successfully, which is a milestone my father often steered me towards. A year ago, I knew very little about NLG, but I am very happy that I chose this subject for my thesis. It has been a great learning opportunity, as well as a provider for my first job in the POSTHCARD project. I hope to have contributed to the scientific community and perhaps even society by building SimpleNLG-NL.

References

Androutsopoulos, I., Lampouras, G., & Galanis, D. (2013). Generating Natural Language Descriptions from OWL Ontologies: the NaturalOWL system. *Journal of Artificial Intelligence Research*, *48*, 671–715.

Audring, J. (2018, March 15). *Adjectival Inflection.* Retrieved from `http://www.taalportaal.org/taalportaal/topic/pid/topic-13998813296919801`

Bateman, J. A. (1997). Enabling Technology for Multilingual Natural Language Generation: the KPML Development Environment. *Natural Language Engineering*, *3*(1), 15–55.

Bollmann, M. (2011). Adapting SimpleNLG to German. In *Proceedings of the 13th european workshop on natural language generation* (pp. 133–138).

Booij, G., & Audring, J. (2018, March 15). *Separable Complex Verbs (SCVs).* Retrieved from `http://www.taalportaal.org/taalportaal/topic/pid/topic-13998813296768009`

Bosch, A. v. d., Busser, B., Canisius, S., & Daelemans, W. (2007). An Efficient Memory-based Morphosyntactic Tagger and Parser for Dutch. *LOT Occasional Series*, *7*, 191–206.

Bouma, G., Van Noord, G., & Malouf, R. (2001). Alpino: Wide-Coverage Computational Analysis of Dutch. *Language and Computers*, *37*, 45–59.

Broekhuis, N., Hans; Corver. (2018, March 15). *4.4. Three Main Types of infinitival Argument Clauses.* Retrieved from `http://www.taalportaal.org/taalportaal/topic/link/syntax__Dutch__vp__V4_41_dependent_clauses_introduction__V4_41_dependent_clauses_introduction.4.4.xml`

Deemter, K. V., Theune, M., & Krahmer, E. (2005). Real versus Template-based Natural Language Generation: A False Opposition? *Compu-*

tational Linguistics, *31*(1), 15–24.

De Oliveira, R., & Sripada, S. (2014). Adapting SimpleNLG for Brazilian Portuguese Realisation. In *Proceedings of the 8th international natural language generation conference (inlg)* (pp. 93–94).

Dokkara, S. R. S., Penumathsa, S. V., & Sripada, S. G. (2015). A Simple Surface Realization Engine for Telugu. In *Proceedings of the 15th european workshop on natural language generation (enlg)* (pp. 1–8).

Ell, B., & Harth, A. (2014). A Language-Independent Method for the Extraction of RDF Verbalization Templates. In *Proceedings of the 8th international natural language generation conference (inlg)* (pp. 26–34).

Gatt, A., & Krahmer, E. (2018). Survey of the State of the Art in Natural Language Generation: Core tasks, applications and evaluation. *Journal of Artificial Intelligence Research*, *61*, 65–170.

Gatt, A., & Reiter, E. (2009). SimpleNLG: A Realisation Engine for Practical Applications. In *Proceedings of the 12th european workshop on natural language generation* (pp. 90–93).

Goldberg, E., Driedger, N., & Kittredge, R. I. (1994). Using Natural-Language Processing to Produce Weather Forecasts. *IEEE Intelligent Systems*, *9*(2), 45–53.

Haeseryn, W., Romijn, K., Geerts, G., de Rooij, J., & van den Toorn, M. (1997). *Algemene Nederlandse Spraakkunst.* Groningen/Deurne: Martinus Nijhoff uitgevers/Wolters Plantyn. Retrieved from `http://ans.ruhosting.nl/`

Kooij, J. (1978). *Aspekten van Woordvolgorde in het Nederlands.* Leiden: Publikaties van de Vakgroep Nederlandse Taal- en Letterkunde.

Landsbergen, F., Tiberius, C., & Dernison, R. (2014, may). Taalportaal: an online grammar of dutch and frisian. In N. C. C. Chair) et al. (Eds.), *Proceedings of the ninth international conference on language resources and evaluation (lrec'14).* Reykjavik, Iceland: European Language Resources Association (ELRA).

Mazzei, A., Battaglino, C., & Bosco, C. (2016). SimpleNLG-IT: Adapting SimpleNLG to Italian. In *Proceedings of the 9th international natural language generation conference* (pp. 184–192).

Nederlandse Taalunie. (2015). *Leidraad.* Retrieved from `http://`

woordenlijst.org/leidraad

Nederlandse Taalunie. (2018a). *Naamvallen (Algemeen).* Retrieved from http://taaladvies.net/taal/advies/tekst/30/naamvallen_algemeen/

Nederlandse Taalunie. (2018b). *Omschreven Trappen van Vergelijking (Algemeen).* Retrieved from http://taaladvies.net/taal/advies/tekst/92/omschreven_trappen_van_vergelijking_algemeen/

Ong, E., Abella, S., Santos, L., & Tiu, D. (2011). A Simple Surface Realizer for Filipino. In *Proceedings of the 25th pacific asia conference on language, information and computation.*

Onze Taal. (2018). *t Kofschip.* Retrieved from https://onzetaal.nl/taaladvies/t-kofschip/

Papineni, K., Roukos, S., Ward, T., & Zhu, W.-J. (2002). BLEU: a Method for Automatic Evaluation of Machine Translation. In *Proceedings of the 40th annual meeting on association for computational linguistics* (pp. 311–318).

Postma, M., van Miltenburg, E., Segers, R., Schoen, A., & Vossen, P. (2016). Open Dutch WordNet. In *Proceedings of the eight global wordnet conference.* Bucharest, Romania.

Reiter, E. (1995). NLG vs. Templates. *arXiv preprint cmp-lg/9504013*.

Reiter, E. (2016, 12). *The Story of SimpleNLG.* Retrieved from https://ehudreiter.com/2016/12/15/the-story-of-simplenlg/

Reiter, E., & Dale, R. (2000). *Building Natural Language Generation Systems.* Cambridge university press.

Reiter, E., Sripada, S. G., & Robertson, R. (2003). Acquiring Correct Knowledge for Natural Language Generation. *Journal of Artificial Intelligence Research*, *18*, 491–516.

Solis, C. J., Siy, J. T., Tabirao, E., & Ong, E. (2009). Planning Author and Character Goals for Story Generation. In *Proceedings of the Workshop on Computational Approaches to Linguistic Creativity* (pp. 63–70). Stroudsburg, PA, USA: Association for Computational Linguistics. Retrieved from http://dl.acm.org/citation.cfm?id=1642011.1642020

Soto, A. R., Gallardo, J. J., & Diz, A. B. (2017). Adapting SimpleNLG to Spanish. In *Proceedings of the 10th international conference on natural language generation* (pp. 144–148).

Taalunieversum. (2018). *Toelichting bij het Keurmerk Makkelijk Lezen.* Retrieved from http://taalunieversum.org/algemeen/toelichting-bij-het-keurmerk-makkelijk-lezen

Theune, M., Slabbers, N., & Hielkema, F. (2007, 6). The Narrator: NLG for Digital Storytelling. In S. Busemann (Ed.), *Enlg-07 11th european workshop on natural language generation* (pp. 109–112). DFKI (Deutsches Forschungszentrum für Künstliche Intelligenz GmbH).

Toutanova, K., Klein, D., Manning, C. D., & Singer, Y. (2003). Feature-rich Part-of-Speech Tagging with a Cyclic Dependency Network. In *Proceedings of the 2003 conference of the north american chapter of the association for computational linguistics on human language technology-volume 1* (pp. 173–180).

Vaudry, P.-L., & Lapalme, G. (2013). Adapting SimpleNLG for Bilingual English-French Realisation. In *Proceedings of the 14th european workshop on natural language generation* (pp. 183–187).

Westwater, D. (2009). *SimpleNLGv4.* Retrieved from https://github.com/simplenlg/simplenlg/blob/master/docs/SimpleNLGv4%20Architecture.pdf

Appendix A

Sentence generation results

The sentence IDs correspond with those used in the Wikipedia corpus in Dact.

Sentence ID	**Target result** **Final result**
wik_part0001/1-19-5.xml	Op 1 oktober 1966 kwam hij vrij. *Hij kwam vrij op 1 oktober 1966.*
wik_part0232/340263-9-5.xml	Zij maakten hem tot mikpunt van spot. Zij maakten hem tot mikpunt van spot.
wik_part0239/357277-5-2.xml	Zijn echte naam is niet precies bekend. Zijn echte naam is niet precies bekend.
wik_part0009/4-30-4.xml	Veel boeren hebben zich toegelegd op de veeteelt. Veel boeren hebben zich toegelegd op de veeteelt.
wik_part0239/358263-13-8.xml	Ik moest me concentreren op de komende ontmoeting. Ik moest me concentreren op de komende ontmoeting.
wik_part0395/760363-11-4.xml	Dit was, tot nu toe, haar succesvolste album. *Dit was tot nu toe haar succesvolste album.*
wik_part0230/336374-2-2.xml	De race startte in Lissabon en eindigde in Dakar. De race startte in Lissabon en eindigde in Dakar.
wik_part0225/325191-4-2.xml	De Spanjaarden hadden in Groningen een solide uitvalsbasis voor verschillende plundertochten. De Spanjaarden hadden in Groningen een solide uitvalsbasis voor verschillende plundertochten.
wik_part0004/764-25-8.xml	Hij werkte aan een aantal romans, maar wist deze niet te voltooien. Hij werkte aan een aantal romans, maar wist deze niet te voltooien.
wik_part0223/320068-12-3.xml	De coureurs van de autos hebben ook contact met de personen in de volgautos. De coureurs van de auto's hebben ook contact met de personen in de volgauto's.
wik_part0229/333218-12-7.xml	Het schip verbleef daar een tijdje en werd op 7 juni 1998 overgebracht naar een dok van het Maritieme Museum in de haven van La Rochelle. *Het schip verbleef daar een tijdje en werd op 7 juni 1998 naar een dok van het Maritieme Museum in de haven van La Rochelle overgebracht.*
wik_part0229/334066-5-1.xml	Het onderwijsmateriaal bestaat uit een reeks van taken, zoals probleemtaken, actietaken, studietaken en discussietaken, waardoor studenten op verschillende manieren de te behandelen stof dienen te benaderen.

Sentence ID	Target result Final result
	Het onderwijsmateriaal bestaat uit een reeks van ’taken’, waardoor studenten dienen de stof te behandelen te benaderen op verschillende manieren.

Table A.1: Results for the first round of sentences, after having completed all four rounds. The sentence IDs correspond with those used in the Wikipedia corpus in Dact. Sentences in *italic* were accepted as correct, even though there was no exact match; sentences in *bold* were marked as incorrect. Sentence wik_part0004/764-25-8.xml originally contained the incorrect (non-existent) *‘voltooiten’*. This is corrected in the target.

#	Feature	Target result Final result
	Verb group types and adjectives	
01	verb: remove double consonant to get stem	Marie rent. Marie rent.
02	verb: repeat vowel from stem	Marie loopt. Marie loopt.
03	verb: ’t kofschip	Marie fietst. Marie fietst.
04	verb: irregular, not in lexicon	Marie moogt.[1] Marie moogt.
05	verb: irregular, in lexicon	Marie heeft een aap. Marie heeft een aap.
06	detect person & number from subject (singular)	Zij heeft een aap. Zij heeft een aap.
07	detect person & number from subject (plural)	Wij hebben een aap. Wij hebben een aap.
08	indefinite article; neuter adjective not in lexicon	Wij hebben een snel aap.[2] Wij hebben een snel aap.
09	definite article *de*; neuter adjective not in lexicon	Wij hebben de snelle aap. Wij hebben de snelle aap.
10	definite article *het*; neuter adjective not in lexicon	De snelle aap rent naar het vuile raam. De snelle aap rent naar het vuile raam.
	Negation	
11	verb: remove double consonant + negation	Marie rent niet. Marie rent niet.
12	verb: repeated vowel + negation	Marie loopt niet. Marie loopt niet.
13	definite article, neuter adjective not in lexicon, negation	De snelle aap rent niet naar het vuile raam.

[1]Both the target and the result are grammatically incorrect, because the verb is an irregular verb being inflected as if it were a regular verb. This is the expected result for irregular verbs that are not in the lexicon.

[2]Both the target and the result are grammatically incorrect. This tests the adjectival inflection when the noun is accompanied by a neuter determiner and the gender of the noun is not in the lexicon. This is the expected result.

#	Feature	Target result Final result
		De snelle aap rent niet naar het vuile raam.
14	use *geen* as negation auxiliary	De aap heeft geen zin. De aap heeft geen zin.
15	repeated vowel plural + *geen* as negation auxiliary	De aap heeft geen poten. De aap heeft geen poten.
	Tenses	
16	past simple regular	Marie rende. Marie rende.
17	past simple ('t kofschip)	Marie fietste. Marie fietste.
18	present perfect	Marie heeft gerend. Marie heeft gerend.
19	present perfect ('t kofschip)	Marie heeft gefietst. Marie heeft gefietst.
20	past perfect	Marie had gerend. Marie had gerend.
21	past perfect ('t kofschip)	Marie had gefietst. Marie had gefietst.
22	future	Marie zal rennen. Marie zal rennen.
23	future ('t kofschip)	Marie zal fietsen. Marie zal fietsen.
24	future perfect	Marie zal hebben gerend. Marie zal hebben gerend.
25	future perfect ('t kofschip)	Marie zal hebben gefietst. Marie zal hebben gefietst.
26	conditional	Marie zou rennen. Marie zou rennen.
27	conditional ('t kofschip)	Marie zou hebben gefietst. Marie zou hebben gefietst.
	Interrogative sentences	
28	yes/no interrogative	Fietst Marie? Fietst Marie?
29	how interrogative	Hoe fietst Marie?

#	Feature	Target result **Final result**
		Hoe fietst Marie?
30	what (object) interrogative	Wat gooit Marie? Wat gooit Marie?
31	who (subject) interrogative	Wie fietst? Wie fietst?
32	who (indirect object) interrogative	Naar wie gooit Marie een bal? Naar wie gooit Marie een bal?
33	why interrogative	Waarom gooit Marie een bal naar de aap? Waarom gooit Marie een bal naar de aap?
34	where interrogative	Waar gooit Marie een bal naar de aap? Waar gooit Marie een bal naar de aap?
35	how many interrogative	Hoeveel apen geven de bal aan Marie? Hoeveel apen geven de bal aan Marie?
36	how many interrogative + separable compound verb	Hoeveel apen gooien de bal terug naar Marie? Hoeveel apen gooien de bal terug naar Marie?
37	who (object) interrogative	Wie zoent Marie? Wie zoent Marie?

Table A.2: Results for the unit tests. The double horizontal lines divide the tests into sections. All tests were successful.

Sentence ID	Target result Final result
wik_part0043/23522-2-3.xml	Zij werd vervangen door Justyna Majkowska. Zij werd vervangen door Justyna Majkowska.
wik_part0109/98833-14-3.xml	Het kan enkel via het water en via de lucht bereikt worden. **Het kan bereikt enkel via het water en via de lucht worden.**
wik_part0230/337851-3-4.xml	De verhalen blijven nog lange tijd onder supervisie van Willy Vandersteen. De verhalen blijven nog lange tijd onder supervisie van Willy Vandersteen.
wik_part0317/530745-4-1.xml	De zaal was ook speelzaal van de vorstin. De zaal was ook speelzaal van de vorstin.
wik_part0377/709379-13-2.xml	Hij en Captain America zetten hun persoonlijke oorlog voort. Hij en Captain America zetten hun persoonlijke oorlog voort.
wik_part0428/864874-11-7.xml	Bijna alle Vlamingen werden gedood of gevangengenomen. **Alle bijna Vlamingen werden gedood of gevangengenomen.**
wik_part0521/1120494-4-3.xml	Hij wierp in Athene een nieuw persoonlijk record van 84,95. Hij wierp in Athene een nieuw persoonlijk record van 84,95.
wik_part0569/1284172-17-2.xml	Het stinkende walvisstation en de slachterijen werden gesloten. Het stinkende walvisstation en de slachterijen werden gesloten.
wik_part0665/1624550-2-7.xml	Hij speelde ook mee in een aantal films en TV-producties. Hij speelde ook mee in een aantal films en TV-producties.
wik_part0667/1633610-4-6.xml	De toelatingsperiode gaat in op 1 april 2010 en duurt 10 jaar. De toelatingsperiode gaat in op 1 april 2010 en duurt 10 jaar.
wik_part0691/1706818-19-1.xml	De Sectie 3 is de staf-afdeling die zich bezighoudt met het functiegebied Operaties. De Sectie 3 is de staf-afdeling die zich bezighoudt met het functiegebied Operaties.

Table A.3: Results for medium long sentences (7-13 words) in round 3. Sentences in *italic* were accepted as correct, even though there was no exact match; sentences in **bold** were marked as incorrect.

Sentence ID	**Target result** **Final result**
wik_part0028/13662-11-2.xml	Carmen 7 is een lofrede op zijn schoonvader Avitus ter gelegenheid van diens inauguratie als keizer. Carmen 7 is een lofrede op zijn schoonvader Avitus ter gelegenheid van diens inauguratie als keizer.
wik_part0073/50154-12-3.xml	Spyker kreeg op 27 mei 2004 een notering aan de Euronext Amsterdam; de introductiekoers was 15,50. **Spyker kreeg op 27 mei 2004 een notering aan de Euronext Amsterdam.**
wik_part0117/111546-173-2.xml	De huidige Afrikaners stammen af van de Nederlandse kolonisten die zich daar in de loop der tijd vestigden. De huidige Afrikaners stammen af van de Nederlandse kolonisten die zich daar in de loop der tijd vestigden.
wik_part0156/181464-26-2.xml	De stemgerechtigden kozen er op 3 november voor dat hun grondwet moest worden aangepast om het homohuwelijk te verbieden. *De stemgerechtigden kozen er op 3 november voor, dat hun grondwet moest worden aangepast om het homohuwelijk te verbieden.*
wik_part0221/315278-5-1.xml	De naam van het aangrenzende sportpark Parkschouwburg herinnert aan het theater dat hier tot 1911 stond. De naam van het aangrenzende sportpark Parkschouwburg herinnert aan het theater dat hier tot 1911 stond.
wik_part0289/464781-6-2.xml	Ze worden echter aangevallen door de Barban die koste wat het kost de komst van de nieuwe Gingaman willen voorkomen. **Ze worden echter aangevallen door de Barban die de komst van de nieuwe Gingaman willen voorkomen koste wat het kost.**
wik_part0352/625806-2-3.xml	Het verwerkt 83 miljoen reizigers per jaar en behoort daarmee tot de drie drukste stations van Parijs. Het verwerkt 83 miljoen reizigers per jaar en behoort daarmee tot de drie drukste stations van Parijs.
wik_part0406/794098-6-1.xml	De plant komt voor op in de zomer droogvallende plaatsen in heidegebieden en kalkarme duinen bij en in vennen.

	De plant komt op droogvallende plaatsen in de zomer in heidegebieden en kalkarme duinen bij en in vennen voor.
wik_part0461/958283-5-1.xml	Crain vertrok daarop om elders in Amerika te gaan wonen, maar stierf binnen korte tijd door verdrinking. **Crain vertrok daarop om wonen elders in Amerika te gaan, maar stierf binnen korte tijd door verdrinking.**
wik_part0553/1224231-2-3.xml	Hij vertolkt zowel het Nederlandse lied, als het Gronings en is regelmatig te zien op het Mega Piraten Festijn. Hij vertolkt zowel het Nederlandse lied, als het Gronings en is regelmatig te zien op het Mega Piraten Festijn.

Table A.4: Results for long sentences (14-20 words). Sentences in *italic* were accepted as correct, even though there was no exact match; sentences in **bold** were marked as incorrect.

Appendix B

Proof-of-concept results

Output tense	**Output form**	**Voice**	**Output sentence**
present	simple	active	Marie gooit de bal.
past	simple	active	Marie gooide de bal.
future	simple	active	**Marie zal gooien de bal.**
conditional	simple	active	**Marie zou gooien de bal**.
present	perfect	active	**Marie heeft gegooid de bal.**
past	perfect	active	**Marie had gegooid de bal.**
future	perfect	active	**Marie zal hebben gegooid de bal.**
conditional	perfect	active	**Marie zou hebben gegooid de bal.**
present	simple	passive	De bal is gegooid door Marie.
past	simple	passive	De bal was gegooid door Marie.
future	simple	passive	De bal zal zijn gegooid door Marie.
conditional	simple	passive	De bal zou zijn gegooid door Marie.
present	perfect	passive	**De bal is geweest gegooid door Marie.**
past	perfect	passive	**De bal was geweest gegooid door Marie.**
future	perfect	passive	**De bal zal zijn geweest gegooid door Marie.**
conditional	perfect	passive	**De bal zou zijn geweest gegooid door Marie.**

Table B.1: Initial results of the tenses, forms and voices of a sentence passing a string to `setVerb()`. While the input was written manually, it could be generated automatically with a small tweak of the converter. Sentences in **bold** are incorrect. The results have been improved during Round 4, of which the results can be found in Table B.3

Output tense	**Output form**	**Voice**	**Output sentence**
present	simple	active	Marie gooit de bal.
past	simple	active	Marie gooide de bal.
future	simple	active	***Marie zal gooien gooien de bal.***
conditional	simple	active	***Marie zou gooien gooien de bal.***
present	perfect	active	Marie heeft gegooid de bal.
past	perfect	active	Marie had gegooid de bal.
future	perfect	active	***Marie zal hebben gooien gooien de bal.***
conditional	perfect	active	***Marie zou hebben gooien gooien de bal.***
present	simple	passive	De bal is gegooid door Marie.
past	simple	passive	De bal was gegooid door Marie.
future	simple	passive	***De bal zal zijn gooien gooien door Marie.***
conditional	simple	passive	***De bal zou zijn gooien gooien door Marie.***
present	perfect	passive	De bal is geweest gegooid door Marie.
past	perfect	passive	De bal was geweest gegooid door Marie.
future	perfect	passive	***De bal zal zijn geweest gooien gooien door Marie.***
conditional	perfect	passive	***De bal zou zijn geweest gooien gooien door Marie.***

Table B.2: Results of the tenses and forms of a sentence passing a `VPPhraseSpec` to `setVerb()`. The input was generated using the converter. Marked in *italic* are the sentences that differ from the sentences generated using a string input, shown in Table B.1. Sentences in **bold** are incorrect.

Output tense	**Output form**	**Voice**	**Output sentence**
present	simple	active	Marie gooit de bal.
past	simple	active	Marie gooide de bal.
future	simple	active	Marie zal de bal gooien.
conditional	simple	active	Marie zou de bal gooien.
present	perfect	active	Marie heeft de bal gegooid.
past	perfect	active	Marie had de bal gegooid.
future	perfect	active	**Marie zal hebben de bal gegooid.**
conditional	perfect	active	**Marie zou hebben de bal gegooid.**
present	simple	passive	De bal is gegooid door Marie.
past	simple	passive	De bal was gegooid door Marie.
future	simple	passive	De bal zal zijn gegooid door Marie.
conditional	simple	passive	De bal zou zijn gegooid door Marie.
present	perfect	passive	**De bal is geweest gegooid door Marie.**
past	perfect	passive	**De bal was geweest gegooid door Marie.**
future	perfect	passive	**De bal zal zijn geweest gegooid door Marie.**
conditional	perfect	passive	**De bal zou zijn geweest gegooid door Marie.**

Table B.3: Final results of the tenses, forms and voices of a sentence passing a string to `setVerb()` after Round 4. While the input was written manually, it could be generated automatically with a small tweak of the converter. Sentences in **bold** are still incorrect.

Input sentence	**Tense**	**Form**
Sterke Marie gooit de zware bal naar de snelle aap	present	simple
Sterke Marie heeft de zware bal gegooid naar de snelle aap	present	perfect
Sterke Marie heeft de zware bal naar de snelle aap gegooid	present	perfect
Sterke Marie gooide de zware bal naar de snelle aap	past	simple
Sterke Marie had de zware bal gegooid naar de snelle aap	past	perfect
Sterke Marie had de zware bal naar de snelle aap gegooid	past	perfect
Sterke Marie zal de zware bal gooien naar de snelle aap	future	simple
Sterke Marie zal de zware bal naar de snelle aap gooien	future	simple
Sterke Marie zal de zware bal gegooid hebben naar de snelle aap	future	perfect
Sterke Marie zal de zware bal naar de snelle aap gegooid hebben	future	perfect
Sterke Marie zou de zware bal gooien naar de snelle aap	cond.	simple
Sterke Marie zou de zware bal naar de snelle aap gooien	cond.	simple
Sterke Marie zou de zware bal gegooid hebben naar de snelle aap	cond.	perfect
Sterke Marie zou de zware bal naar de snelle aap gegooid hebben	cond.	perfect

Table B.4: Fourteen different input sentences with the modifiers *sterk* ‘strong’ added to the elements which should have resulted in the same tree structure when converted. The difference between two sentences of the same tense and form is the word order, which is based on a stylistic choice. The expected result was: *Sterke Marie gooit de zware bal naar de snelle aap.* The conversion results of the sentences in **bold** missed the prepositional phrase and therefore the input for SimpleNLG-NL was incomplete. This is due to the extra complexity in Alpino’s output, which is currently not supported by the converter.