Literature Research into Natural Language Generation for the Virtual Storyteller

Marissa Hoek
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
m.d.hoek@student.utwente.nl

Keywords

natural language generation, virtual storytelling, Dutch, emergent narrative

ABSTRACT

This paper presents a literature research into generating stories in natural language based on events created by the Virtual Storyteller, a multi-agent system that simulates actors to create a story. The goal was to come up with recommendations for the system design of an NLG system that can create these stories. This system will be based on an existing subsystem of the Virtual Storyteller called the Narrator.
The Virtual Storyteller communicates with the Narrator by means of a story data model called the Fabula model, which contains the events of the story. With the people currently working on the Virtual Storyteller we formulated a recommendation for a new format for the Fabula model based on GraphML in combination with an OWL-ontology.
I will try to make the generated stories more lively by adding focalization, i.e. telling the story from one character's perspective, and temporal styles, such as flashbacks. Focalization will be done by performing data selection in such a way that only the events the focalized character can perceive will be present. Temporal styles will be made by ordering the content in a non-chronological way that matches the specific style.

1. INTRODUCTION

Researchers at the University of Twente have made a system called the Virtual Storyteller, which can dynamically create story plots based on a multi-agent system which simulates the behaviour of characters (Swartjes and Theune, 2008). This Virtual Storyteller is coupled with a related system called the Narrator, which converts an internal structure called Fabula Model into a story text in natural language (Theune et al., 2007). This Fabula model contains a formal representation of the simulated events, the actions of characters and the internal state of the characters that caused them to do these actions (Swartjes and Theune, 2006).

Currently, a new version of the Virtual Storyteller is being implemented. The goal of this project is to change the system so that it can be used for serious games, such as training police officers (Linssen and de Groot, 2013). This is the perfect time to update the Narrator and to think about how it can be improved.

The goal of this part of the project is to learn how the Narrator can be improved using techniques from the state of the art in natural language generation. Since these improvements could require information that the Fabula model currently does not contain, a second goal of this project is to formulate recommendations for how the Fabula model can be improved.

The research questions are as follows:

1. What are the requirements for a Narrator system for the Virtual Storyteller?
2. How does the current Narrator system work?
3. What does the current internal story model (Fabula model) look like?
4. Is this Fabula model sufficient for the new Narrator system or can it be improved?
5. What is the state of the art in the field of natural language generation in relation to story generation?
6. What techniques can be used to improve the new Narrator system with regards to the current Narrator system?

The paper starts by describing the current Virtual Storyteller and the serious game that uses the new Virtual Storyteller in section 2. In section 3 I describe the general requirements of the Narrator by listing the desired functionality and describing a few use cases. Section 4 describes existing literature that describes the architecture of Natural Language Generation (NLG) system.

In section 5 I describe how the current Narrator system works. Section 6 describes related work, in particular NLG systems that create stories. What we have learned from the previous sections results in section 7. Here I discuss the design choices for the new Narrator. Finally, I conclude my results in section 8.

2. BACKGROUND

This section describes the background of the project. I will describe the Virtual Storyteller in general, the system that generates the stories that the Narrator must convert to natural language. I will also describe the new serious game that uses the Virtual Storyteller.

2.1 Virtual Storyteller

The Virtual Storyteller is a multi-agent system that is used to generate stories using an 'emergent narrative' approach (Swartjes and Theune, 2008). Emergent narrative means in this case that the system simulates the behaviour of characters as agents, and that the story *emerges* from the interaction between these agents.

The Virtual Storyteller has multiple types of agents. The first type is the Plot Agent, which leads the simulation and tries to steer the story into an interesting direction.

The Plot Agent uses a World Agent to keep track of the state of the story world. The characters are 'played' by Character Agents. The job of these agents is to react to situations in the story world as if they are the character they play (Swartjes, 2010, Chapter 6). A Character Agent can also be controlled by a human.

The Virtual Storyteller uses turn-based simulation. Each round the Character Agents are asked in turn to select an action. The system then calculates the effects of its action on the story world, and asks the next Character Agent to selection an action.

To make the story more interesting, the Virtual Storyteller uses improvisation techniques. Two of these techniques are out-of-character communication and late commitment. Out-of-character communication refers to a technique where characters can communicate 'behind the scenes'. The aim is then not to achieve the character's goal, but to make the story more interesting. (Swartjes, 2010, Chapter 6). Late commitment refers to the idea of filling in story details as late as possible, to keep all options open for the characters. For example, using late commitment it's possible to decide at the last moment that a chest contains a bottle of rum, so that the captain can take the bottle from the chest and drink it.

2.2 Loitering Youth Scenario

A new version of the Virtual Storyteller is being developed in cooperation with the Dutch National Police Force. The goal is to create a serious game for training social skills for police officers (Linssen and de Groot, 2013). A scenario developed for this project is called "Loitering Juveniles", which is about a police officer dealing with young people causing trouble on the street. Because the purpose of this game is to learn from the interaction with the game, it is useful to have a report in natural language to reflect on the experience afterwards. This report should be in as fluent Dutch as possible, and this is the job of the new Narrator.

To improve the learning factor from the reflection on the story, the Narrator could tell the story from the perspective of one of the characters. This is called focalization. In theory, telling a story from the perspective of one of the loitering youth could help the training police officer see the point of view of the youth.

3. REQUIREMENTS

In this section I describe the requirements of the project. I describe the role of the Narrator, and list the desired functionality of the software. I also describe the ways a user or client program can interact with the Narrator in the form of use cases.

The Narrator is a software system that generates a text in natural language from data generated by the Virtual Storyteller. This data can be split in two parts. The first part is the 'setting' of the story: the characters, the places and all concepts that can be used in the story. This is for example the scenario of loitering youth, or the setting of Little Red Riding Hood. This part can be used for several stories. The second part is the story plot: everything that has happened in the storyteller through the interaction of agents. This part is unique for each story, and relies on the setting.

The following list describes the required functionality of the Narrator. This is divided among three levels of importance. The first level describes functionality that the Narrator *must* contain. This is the basic functionality, without this the Narrator wouldn't function. The second level

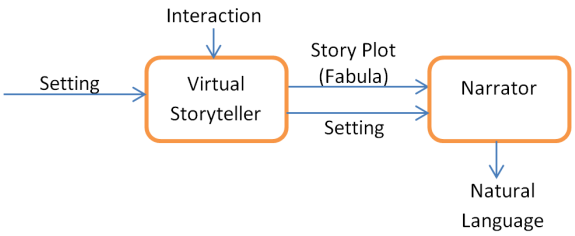

Figure 1. A schematic overview of the job of the Narrator

is functionality that the Narrator *should* contain. This functionality greatly improves the usefulness, but is not absolutely necessary. The third level describes functionality that is not at all necessary for the Narrator, but that *could* be implemented to make the system even better.

1. ***Must*** The Narrator system must be compatible with the Virtual Storyteller through the use of a story plot and setting.
2. ***Must*** The Narrator must create texts in Dutch describing the events from the serious game that will used for training police officers.
3. ***Should*** The Narrator should be easy to use.
4. ***Should*** It should be easy to adapt the narrator system to a new domain.
5. ***Could*** The Narrator could support other languages, like English.
6. ***Could*** The system could support focalization (telling the story from the point of view of a specific character).
7. ***Could*** The system could support different style figures for a more interesting text.

Requirements 6 and 7 may need some clarification. Focalization has been mentioned before as a way to put the police officer playing the serious game in the shoes of one of the other characters in the story. Different style figures are a way of making the story more interesting to read. This could improve the quality of the generated text.

3.1 Use Cases

This section describes the different ways a user can interact with the Narrator. The use cases define a number of steps the user and the system take to achieve a certain goal.

U1: Create the story text from within another program

For example: display the text in the Virtual Storyteller.

Preconditions: The Client Program has a Fabula, for example from interaction with an end user.

Postconditions: -

Primary Actor: Client Program

Related requirements: R1, R2

1. The client program sends the Fabula to the Narrator.
2. The Narrator creates the story text from the Fabula.
3. The Narrator sends the story text to the client program.
4. The client program displays the story text

U2: Create the story text from a Fabula file

Preconditions: The Fabula file is already made, for example by exporting it from the Virtual Storyteller.

Postconditions: -

Primary Actor: End User

Related requirements: R1

1. The user opens the Narrator and selects the option for creating a story text from a Fabula.
2. The user selects the Fabula file using a file selection dialog.
3. The Narrator creates the story text from the Fabula.
4. The Narrator displays the story text.
5. The user can export the story to a text file.

U3: Create the story text for a focalised story

Preconditions: The Fabula file is already made, for example by exporting it from the Virtual Storyteller.

Postconditions: -

Primary Actor: End User

Related requirements: R6

Related use cases: U2

1. The user opens the Narrator and selects a Fabula file, as in U2.
2. The Narrator displays a list of characters from which the story can be focalised.
3. The user selects a character.
4. The Narrator creates a story that's focalised from this character and displays the text, as in U2.

4. LANGUAGE GENERATION

This section describes the basic literature regarding the architecture of NLG systems. It also describes existing systems for linguistic realization, which could be used in the Narrator.

4.1 Architecture

Reiter and Dale

The book by Reiter and Dale (2000) is seen as the standard work about creating natural language generation systems. It discusses the specific tasks a NLG system should perform, and proposes a three-stage architercture for such a system. The focus of the book is on making simple and practical systems that can be built with relative ease.
Reiter and Dale's architecture is based on a pipelined structure with three parts: A high level text planner, a sentence planner and a linguistic realizer. The text planner performs content determination, which is selecting the information that should be present in the final text. Often the data that should be converted to text has a lot of excess information, in this step the NLG system selects what should be communicated. The text planner also performs discourse planning. In this step the system decides how the data should be structured in the text. The output is a text plan, which contains information about the organization and order of the data.
This text plan is then used as input for a sentence planner. This part has three subtasks: sentence aggregation, lexicalization and referring expressions generation. During the sentence aggregation phase the system the system decides how to combine pieces of information into a sentence. The output is a new text plan, made of sentences which contain one or more pieces of information. The planner also decides what 'syntactic mechanism', for example conjunction, ellipsis, relative clause, is used to combine the pieces of information.
The next step is lexicalization. During this phase the NLG system decides what specific words are going to be used for each entitiy. It chooses for example which verbs are used, and how relations are realised. During this step the NLG system also has to create referring expressions, which are the descriptions for entities. This is harder than it sounds at first, because we have to make sure that the reader understands which entity is referred to, while making the description fluent, and not too complicated and hard too read at the same time.
The final step is linguistic realisation. At this step the NLG system takes the words, verbs and descriptions made in the last step and turns them into grammatically correct text. This step inflects the verbs, chooses the right article, makes sure everything is in the right grammatical case and that the words grammatically agree with each other.

Hayes and Flower

Hayes and Flower (1986) provide a different architecture for natural language projects that is based on the way humans write. It is originally intended for teaching students how to write better, but it could be applied to computer systems as well. Hayes and Flower define three processes: planning, sentence generation and revising. In the planning part, a hierarchy of goals is created. The writer first defines the main goals of the story, and then defines subgoals. These subgoals can then also have subgoals, which creates the hierarchical structure.
Although the subprocesses are similar to the three subsystems listed by Reiter and Dale, there is a big difference between the two approaches in the sense that Hayes and Flower propose an interwoven structure. This is more similar to the way a human writer would work. Sometimes a writer will create part of a story, and then another part, doing each process twice or more often. And other times, a writer would add content in the middle of a story and then recreate the goals and plans for the entire story.
The related projects that I have found (see: section 6) all use an architecture similar to the one proposed by Reiter and Dale. This makes sense, as Hayes and Flower describe a method that's more fit for a creative process where the content still has to be created. When the content is already known, it makes more sense to use the systematic pipeline approach described by Reiter and Dale. This is the case for the Narrator as well, since it does not generate the actual plot of the story: this is done by the Virtual Storyteller.

RAGS

Mellish et al. (2006) (originally in 2000) describe an abstract archictecture for Natural Language Generation Systems, with the goal of enabling sharing and re-use of components of such systems. They found that even though all 19 of the systems they surveyed used Reiter and Dale (2000)'s three-stage model, there were big differences in what task was handled by which stage. The systems also lacked a clear unified definition of what the intermediate data structures look like. This makes sharing of components even more difficult.

These problems are there because Reiter and Dale's architecture is not a strict manual for building NLG systems, but rather a 'consensus architecture', a description of how many NLG systems work. To counter this, Mellish et al. (2006) tried to make a reference architecture that systems could adhere to that has more definition than Reiter and Dale's concensus architecture. This resulted in the following points:

- A high level specification of internal data types.
- A low level reference implementation of an internal data model
- An XML specification of the intermediate data structures
- A generic view of how components should interact with each other
- A few reference implementations of the architecture.

Cahill et al. (2001) introduced a more concrete architecture as an implementation of the RAGS specification. This is illustrated with a system called RICHES, an actual NLG system built according to their architecture.

4.2 Linguistic realizers

The literature mentions a few surface realizers that have already been made. Because this task is usually similar between NLG systems, it can be useful and time-saving to use an existing off-the-shelf realizer.

KPML

Bateman (1997) describes KPML, a 'development environment' for Natural Language Generation systems. The goal of the project was to create generic reusable resources and a basic engine for language generation. KPML is interesting because it supports multiple languages. Instead of making different languages the responsibility of different modules, KPML tries to create a shared grammar and resources for multiple languages.
The idea behind KPML is that it uses a single input structure for all languages. For this KPML uses a sentence plan, written in the Sentence Plan Language (Kasper and Whitney, 1989). An example of SPL input can be found in figure 2. SPL uses a dependency tree structure which contains the sentence plan in a semantic way, instead of a syntactic way. This means that it contains the meaning of the sentence structure, instead of the syntax and the words that make up the sentence. This more high-level approach allows KPML to realize a specific SPL input into many different languages.
KPML supports both Dutch and English, as well as many other languages (Bateman, 2003). It is written in Lisp.

RealPro

RealPro (Lavoie and Rambow, 1997) is a surface realizer that is still often used. It is meant as an off-the-shelf implementation of the realization step defined in Reiter and Dale (2000)'s structure. RealPro takes input in the form of a syntactic depency tree, called the Deep Syntactic Structure (DSyntS). This structure stores for each word in the target sentence its uninflected form and its relation with other words. For example, for the phrase 'Mary sees John', the dependency tree stores 'see' as the root node with Mary as the subject and John as the object. It's possible to set features to nodes to define the sentence structure, such as adding a question feature to a verb or a plural feature to a noun. The dependency tree does not store words

```
(S1 / generalized-possession
  :tense past
  :domain (N1 / time-interval
            :lex march
            :determiner zero)
  :range (N2 / time-interval
            :number plural
            :lex day
            :determiner some
            :property-ascription
            (A1 / quality :lex rainy)))
```

Figure 2. An example input for KPML, which produces the sentence 'March had some rainy days'. Based on (Reiter and Dale, 2000, Figure 6.12)

that are only needed for a correct structure, but have no meaning by themselves. Since the structure specifies the literal uninflected word, RealPro cannot choose between alternatives for words. An example of a dependency tree can be found in figure 3.
RealPro uses data structures called Linguistic Knowledge Bases (LKBs) for each language. It has two base LKBs for the grammar structure, which can be used for all languages that have a similar grammar to English. For each language, RealPro also has a word LKB that stores morphology rules and inflections of irregular verbs. The paper only describes an English LKB, with one for French being in development.
RealPro is made in C++, and includes a Java API. RealPro is currently owned by CoGenTex, Inc. (2010).

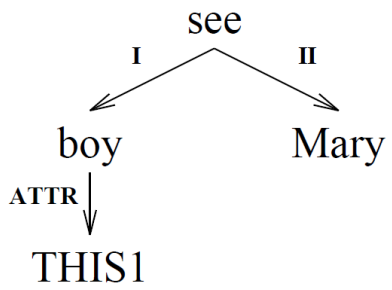

Figure 3. An example of a dependency tree for RealPro, which produces the sentence 'This boy sees Mary' (Lavoie and Rambow, 1997, Figure 1).

SimpleNLG

As an alternative to complex realizers like KPML and RealPro, Gatt and Reiter (2009) created SimpleNLG. SimpleNLG is according to its developers a *realisation engine* instead of 'normal' wide coverage linguistic realizer. The difference is that for KPML and RealPro the input structure is more high-level, it contains the words and the relation these words have with each other. The linguistic realizer then deals with the structure of the text. SimpleNLG has a more low-level input structure where you define the structure of the text yourself, and simpleNLG only deals with inflection and turning the structure you defined into a linear sentence. This gives users of SimpleNLG more control over the final text, as they can specify the words and phrase structure yourself. It's also possible to insert a piece of canned text into a phrase. These features make SimpleNLG useful for NLG systems that have specific language requirements, like when requiring specific terms or jargon.
SimpleNLG has a simple lexicon, that stores only five features: adjective and adverb position, aggregation type for

nouns (mass or count), verb type, and verb complement type. For each phrase, it's possible to set features such as tense, question type and whether the phrase is passive. SimpleNLG only supports English, and is made in Java. An example of Java code used to create a sentence with SimpleSNLG can be found in figure 4.

```
Phrase s1 =
  new SPhraseSpec('leave');
s1.setTense(PAST);
s1.setObject(
  new NPPhraseSpec('the','house'));
Phrase s2 =
  new StringPhraseSpec('the boys');
s1.setSubject(s2);
```

Figure 4. Example code for SimpleNLG, which creates the sentence 'The boys left the house'.

Alpino's realizer

The systems above are all made for the English language, but there are also realizers that are made for Dutch. De Kok and van Noord (2010) describe such a system, which is based on Alpino. Alpino is a grammar and parser that transforms a Dutch sentence into a dependency structure. The linguistic realizer uses the grammar in the opposite direction: it starts with a dependency structure and transforms it into possible sentences that fit this dependency structure.

The realizer uses the same dependency structure as the Alpino parser (Van der Beek et al., 2002), only without information about word order and original word inflection. The dependency structure contains a syntactical representation of the goal sentence in the form of a tree. Each node contains information about its main word, for example the verb in a noun phrase, and its dependents, such as subject, object and determiner. Each dependent can by itself contain a dependency structure. When it's possible to have multiple dependents of the same type, Alpino uses a list structure. An example of the dependency structure of Alpino can be found in figure 5.

Alpino's realizer also includes a fluency ranker, which tries to select the most fluent target sentence from multiple possibilities. This is done by storing the multiple options as partial representations, and combining them using a maximum entropy model.

Alpino's realizer is part of the larger Alpino parser. It is written in TCL and only supports Dutch.

$$\begin{bmatrix} \text{HD} & [\text{LEX} \quad \textit{beklijf}] \\ \text{CAT} & \textit{smain} \\ \text{SU} & \begin{bmatrix} \text{HD} & [\text{LEX} \quad \textit{advies}] \\ \text{CAT} & \textit{np} \\ \text{DET} & \langle [\text{HD} \quad [\text{LEX} \quad \textit{de}]] \rangle \end{bmatrix} \end{bmatrix}$$

Figure 5. An example dependency structure for Alpino, which produces the sentence 'De adviezen beklijven' (The advices endured) (De Kok and van Noord, 2010, Figure 1).

Narrator

The current Narrator also has its own surface realizer, which was created by Hielkema (2005) and creates Dutch surface text. In the next section I describe it in more detail.

5. CURRENT NARRATOR

This section describes the Narrator system for the current Virtual Storyteller. It is based on Nanda Slabbers' Master thesis (Slabbers, 2006) and the system description by René Zeeders (Zeeders, 2008). Here I will describe the three subsystems present in the Narrator, as well as the Fabula model which is used as input.

5.1 Basic Structure

The Narrator uses a structure based on Reiter and Dale's pipeline architecture (Reiter and Dale, 2000). The three submodules are present, here called document planner, microplanner and surface realizer. These modules transform the Fabula into an intermediate output, which is then transformed into the final text. The Narrator is written in Java.

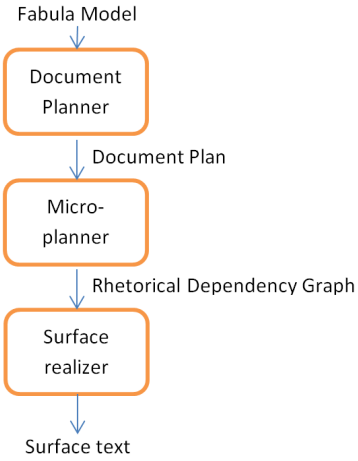

Figure 6. The basic structure of the Narrator.

5.2 Fabula Model

The Fabula model is a formal representation of the events in the story generated by the Virtual Storyteller (Swartjes and Theune, 2006). The name comes from Bal (1997)'s definition of a Fabula, which is 'a series of logically and chronically related events'. The Fabula model follows the structure of a causal network. This is based on a theory (Trabasso et al., 1984), which states that humans understand the relation between events in a story in the form of a network in which some events cause other events.

Trabasso proposed a model called the General Transition Network. This GTN consists of six different story elements (Setting, Event, Internal Response, Goal, Attempt and Outcome) which are connected by four different causal relationships: physical causality, psychological causality, motivation, and enablement.

Although the Fabula model is based on this model, it does not follow it exactly (Swartjes, 2010, Chapter 7). One difference is that Trabasso's model is different for each character in the story, whereas the Fabula model stores one 'objective reality' of all events. Telling the story from the point of view of one character is then the responsibility of the Narrator. Another difference is that the Fabula model does not contain the Setting (e.g. characters and locations) of the story. The Fabula model also adds Perceptions, which denote that a certain character has perceived an event. This was not necessary in the GTN as

that model was different for each character. Because the Fabula model makes one global structure, it's important to store what events each character has perceived.

This leaves the Fabula with six element types. The Goal, which is the drive for a character to act. The Action, a world change caused by a character. The Outcome, which is the result of a goal. It is personal for each character, and based on the character's beliefs instead of the actual state of the world. An Event, which is a world change not caused by a character. Perception, which is a character perceiving another story element. The last type of element is the Internal Element. This is something that happens within a character, such as cognitions, emotions, feelings and beliefs. If a character is made angry by a certain action or event, this 'angry' emotion is an example of an Internal Element.

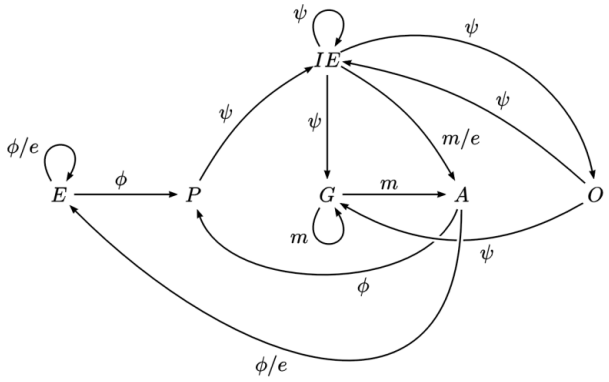

Figure 7. An example of a Fabula graph. The capital letters represent the different types of story events, and the arrow represent the causal relationships between these events. (Swartjes and Theune, 2006)

The four causal relationships are as follows. The Physical casuality (ϕ) denotes that an Action or Event physically caused an Event. For example, in the story of Little Red Riding Hood, if the wolf is killed by the hunter's knife, the death of the wolf is physically caused by the action of the hunter cutting the wolf. If a Perception is caused by an Action or Event, this is also a physical causality. The second causal relation is the Motivation (m), and occurs when one story element motivates another. There are three ways in which this can happen. A Goal motivating a subgoal, which has to be performed before the main goal can be achieved. A Goal motivating an Action, this happens when a character performs a certain action to achieve their goal. And finally an Action motivated by an Internal Element, for example a character running away because they are scared.

The third causal relationship is the Psychological Causality (ψ). This is the reasoning in the minds of the characters. It can be a Perception causing an Internal Element, an Internal Element causing another Internal Element, or an Internal Element causing a Goal. The last causal relationship is Enablement (e). This means that a certain element enables the occurance of another element, but is not the cause of it. For example, if a pirate wants to drink rum, he can't drink until he has a rum bottle. The belief that he is holding a bottle of rum *enables* the Action of drinking rum, but it is caused by the Goal of wanting to drink rum.

The Fabula is saved as a set of RDF (Resource Description Framework) Named Graphs (Carroll et al., 2005). The set consists of a main graph which contains the fabula elements and the causal relationships, saved as RDF triples (*subject:predicate:object*). The other graphs contain the contain of the elements, which is linked from the main graph using the predicate `fabula:hasContent`.The graphs are saved in the TriG format, which is a way of storing RDF Graphs in plain text using Turtle, a syntax made for RDF. The TriG syntax can be found on Bizer (2007)'s website.

5.3 Document Planner

The responsibility of the Document Planner is to define the structure of the story. It does this by creating a Document Plan, which serves as input for the microplanner. The document planner consists of five submodules, as presented in figure 8.

The Document Planner starts by creating a first version of the Document Plan. This is done by a module called InitialDocPlanBuilder, which uses a recursive algorithm to convert the Fabula's graph structure into a dependency tree structure. The InitialDocPlanBuilder is responsible for the content determination, information ordering, and assigning the correct rhetorical relations. Content determination is done by adding all plot elements from the fabula, with a few exceptions. If an action results in the perception that the action was performed, this perception element is not added. Also, when a subgoal exists that is a more specific form of a greater goal, only the subgoal is mentioned. For example, if a character has a goal to drink something, and a subgoal to drink rum, only the goal to drink rum will be mentioned. Both these omissions are there to remove redundancy.

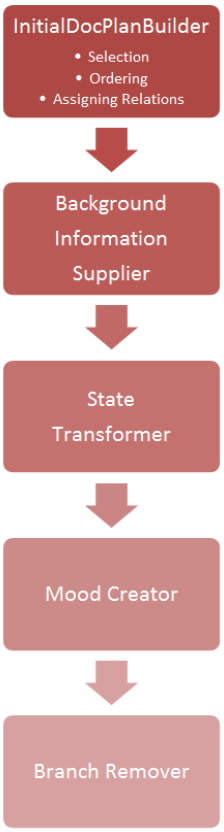

Figure 8. The structure of the DocumentPlanner, based on Chapter 6 of Slabbers (2006).

Information ordering is mostly done in a chronological way. An exception is the case where two characters are doing some actions at the same time. In this case the actions are ordered per character, to avoid switching back and forth. In the third step, the algorithm chooses the right rhetorical relation for the causal relations in the Fabula. Slabbers (2006) also describes the task to add paragraph boundaries. This is currently not performed because the stories created were quite short, but it's an idea for a future version that has to create longer stories.

After creating an Initial Document Plan, the Document Planner performs three transformations. The first is called Background Information Supplier (BGIS). The BGIS adds information from the story world and information about the characters to supply the reader with background information. The BGIS stores a discourse history to en-

sure that it does not tell the same piece of information twice. The second transformation is done by the State Transformer. This part decides how plot elements which described the Internal Element of a character can be described best. It can describe the Internal Element as a seperate sentence ('the princess was scared'), but it can also describe the element as an adjective or relative clause. Finally, it can use a piece of canned text to describe the internal element as an action ('her heart was pounding from fear'). The third transformation is the Mood Creator. This part adds sentences to describe the mood of the story, to make it more attractive for the reader.

The Branch Remover performs the final transformation of the document plan. This part tries to change the document plan so that it doesn't contain long branches which cannot be combined by the surface realizer. This would result in simple, short sentences, so making these branches shorter would result in better surface text.

5.4 Microplanner

This part uses the document plan to create the sentence structure of the story. The output is the Rhetorical Dependency Graph. This graph has the same tree structure as the document plan, the difference being that the Rhetorical Dependency Graph contains dependency trees instead of plot elements. The microplanner must thus convert the plot elements into dependency trees.

The microplanner does this by using a template for each kind of plot element (actions and events, internal states, perceptions and beliefs, goals, and settings). The dependency tree contains information about the plot element: what characters and objects were involved, what action was performed, and in some cases surface words or verbs. This part is comparable to the 'sentence aggregation' task in Reiter and Dale's model. The templates that are used can be found in figure 9.

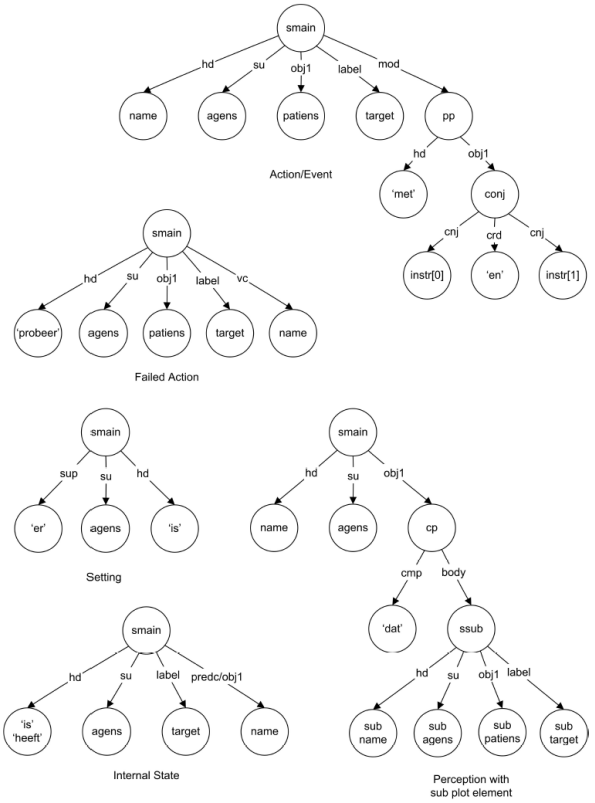

Figure 9. The templates that are used in the Sentence Planner (Slabbers, 2006, Figure 7.2).

After filling the templates of the dependency tree the microplanner performs lexicalization. The Narrator has a lexicon for this, which maps the concepts, such as characters, objects and actions, to actual words. It also stores the previous words used for each concept, to avoid repetition. These two pieces of information are then used to choose the word that will be used.

In Reiter and Dale's structure, the microplanner also performs referring expression generation. In the Narrator however, this is done by the surface realizer.

5.5 Surface Realizer

The surface realizer converts the dependency graph created by the last step into the surface text. It's based on the surface realizer made by Hielkema (2005). First, it transform the rhetoric relations in the dependency tree into their final form. It processes the tree depth-first, and tries to combine the subtrees. It has a conjunctor for doing the actual combining, and an elliptor for syntactic aggregation.

The second step is creating referring expressions for each entity. It uses an algorithm based on a modification of Dale's Incremental algorithm (Krahmer and Theune, 2002). The last component is the surface form generator, which takes the final tree that was combined by the first step, and transforms it into the final text. It uses the referring expressions created in the previous step, decides the final order of sentences, inflects words and adds punctuation.

6. RELATED WORK

This section describes related systems that perform a role similar to the Narrator or the Virtual Storyteller. This means natural language generation systems that focus on creating stories in natural language, or artificial intelligence systems that focus on creating the events of a story.

Curveship

Curveship (Montfort, 2011) is a framework for generating interactive fiction. This framework is interesting because it is capable of changing the narrative style. Curveship supports focalization of different actors, and telling events out of order. Curveship has two main parts: the 'Simulator' and the 'Teller'. The Simulator is the part where the story happens: it maintains the state of the world and determines whether the player's actions succeed. It then passes on a first-order representation of the events that have happened, which the Teller then uses to narrate the story. This split is similar to how the Virtual Storyteller creates the story plot, and the Narrator creates surface text from this plot.
To achieve focalization, Curveship makes a distinction between the 'actual world' and the 'concept', the world as perceived and believed by each actor. Using this technique, it's possible to either tell the story from the point of an all-knowing narrator, or from the point of view of one of the actors. The Teller is based on the standard three stages structure that was discussed before. In the high-level planner, the order of narration of actions is determined. By changing this order, it's possible to tell the story in different styles. For example, chronological order, reverse order, flashbacks and flash-forwards are supported using this method.

McIntyre & Lapata

McIntyre and Lapata (2009) describe a different method of creating stories. Their system creates random stories by combining concepts from a database, without generating

a plot beforehand.
To increase the chance of creating an interesting story, McIntyre and Lapata have made a method for ranking stories. Their method tries to differentiate between enganging and boring stories, and between smooth and incoherent stories. This story ranker is especially fit for their method of creating stories. Since their stories are created randomly, they can more easily make many stories, but there's less guarantee that their stories are good.
The ranking method is based on an interest model and a coherence model. The interest model is learned on existing training data consisting of stories (Aesop's fables) labeled for 'interestingness' and coherence. Their model then tries to find a correlation between specific lexical features and interestingness. Using an SVM model, this method achieved a Kendall's tau correlation of 0.948.
Their coherence model tries to find 'local coherence', which is the coherence from sentence to sentence. This is done by using the Entity Grid approach (Barzilay and Lapata, 2005). This model was trained on original fairy tales from the Andrew Lang collection, and randomly shuffled versions of these fairy tales as examples of incoherent stories. This method found the original stories to be more coherent 67% of the time.
When evaluated by humans for fluency, coherence and interest, their ranking method produced significantly better stories according to all three measurements.

MAKEBELIEVE

MAKEBELIEVE is a system by Liu and Singh (2002) which generates stories using a common sense knowledge base. The knowledge base describes how one event causes another event, for example 'drinking alcohol causes you to be drunk'. The system starts with a first line, which is supplied by the user. MAKEBELIEVE then uses the cause and effect descriptions from knowledge base to generate a new event. This is done by matching the 'effect' of the last event to the 'cause' of an event described in the knowledge base. The system uses fuzzy matching, which means that the cause and effect do not have to match up exactly. This makes it more likely that there's a match, and simulates creativity.

John became very lazy at work. John lost his job. John decided to get drunk. He started to commit crimes. John went to prison. He experienced bruises. John cried. He looked at himself differently.

Figure 10. An example story from Makebelieve.

The result of generating stories this way is a cause and effect structure that is similar to the Fabula model of the Narrator. The difference is that Makebelieve creates a one-dimensional chain of cause and effect, in which each event is the cause of only one other event that happens directly after the event that caused it. This is different from the Fabula model, which has a more complex causal network structure in which events can cause multiple other events. This simple structure is sufficient for Makebelieve because it uses a simple way of creating stories that only uses the last event as the cause of the next event.

Mimesis

Another way of generating stories is described by Young (2007). His paper describes a system called Mimesis, which can create interactive virtual worlds. They propose a bipartite model, which has a distinction between the structure of the story and the structure of the narrative discourse, that is, the actualling telling of the story.

The story structures are created by something called the Decompositional Partial-Order Causal Link (DPOCL) planner (Young and Moore, 1994). This planner creates something called DPOCL plans which store the structure of the story. These plans are similar to the the Narrator's Fabulas. DPOCL plans contain a number of actions, which can be abstract, like a goal, or primitive, like a concrete action. Each action has a set of preconditions, which are the conditions that must be true in the world before the action can succeed. Each action also has a set of effects, which are the conditions of the world that are altered by succesfully executing it. The DPOCL plan also contains a set of what they call *temporal constraints*, which define in what order certain steps should be executed, and causal links, which connect pairs of steps when an effect of one event matches a precondition of the other.

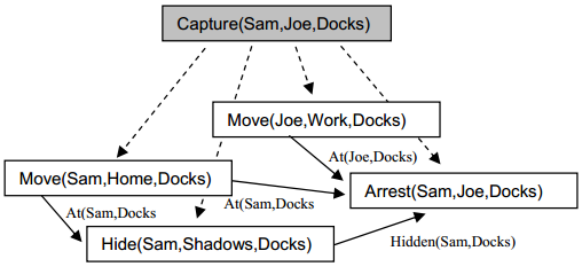

Figure 11. A graphical representation of a DPOCL story plan.

In figure 11 you can see a representation of a part of a story in DPOCL. This story is about Sam wanting to capture Joe at the docks. The grey box belongs to an 'abstract action', in this case the goal of Sam wanting to capture Joe. To attain this goal, Joe and Sam move to the docks. Sam then hides in the shadows. The preconditions for the action Arrest are now filled, and Sam can arrest Joe.
The structure of the DPOCL plans was evaluated in a research by Christian and Young (2004), which compared it to the way humans understand a story.

7. PROPOSED SYSTEM DESIGN

In this section I propose the design of the new Narrator. I describe the general architecture, the Fabula data structure, and the three stages of the language generation process.

7.1 Architecture

The Narrator will be written in Java. There are two reasons for this. The current Narrator is already written in Java, so this would make it easier to reuse code. The second reason is that I am more proficient in Java than in other languages. The new Narrator will use the three-stage model defined by Reiter and Dale (2000). As mentioned before, this is the most practical approach for when the content of the text is already known. It's also the same structure as the current Narrator, this makes it also a bit easier to reuse existing code. In section 4.1 I discussed RAGS (Mellish et al., 2006), a more standardized architecture based on Reiter and Dale (2000). While I think standardization is a good idea, RAGS is not used very often. This lack of other projects that follow the standard makes RAGS not useful enough for this project.

To make interaction with the Narrator easy, there will be one 'Main' class that handles all interaction. Interaction with the Narrator can happen in one of three ways:

1. Call the Narrator as a method from a Java program.

2. Run the Narrator from a command-line interface, so that it writes text to the standard output.

3. Run a graphical interface of the Narrator. This is especially useful for testing.

The Main class thus receives the Fabula data, which it then passes on to a class called Pipeline. This class handles the three stages from Reiter and Dale (2000). I will follow the current Narrator in the sense that all stages are seen as transformations of the previous data, until the final transformation which creates text in natural language. All intermediate data will be in the form of the Rhetorical Dependency Graph.

A simple representation of the basic architecture can be found in figure 12.

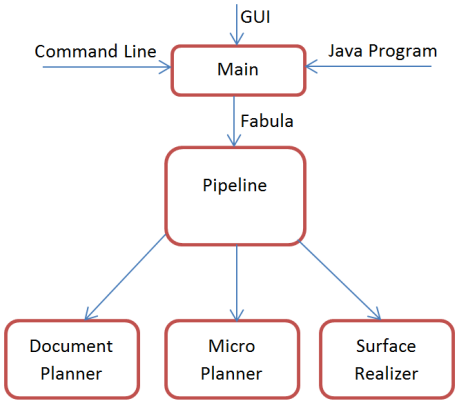

Figure 12. A diagram representing the basic architecture of the new Narrator.

7.2 Fabula Model

The Fabula model is the data structure of the Virtual Storyteller that stores all the events that happened in the story as a causal network. In this section I look at alternatives to the Fabula model, and see if improvements can be made.

In section 6 I discussed several alternatives to the Fabula model. MAKEBELIEVE (Liu and Singh, 2002) uses a knowledge base to create a single causal chain, where each action causes only the action happening directly after it. This simple solution works well for MAKEBELIEVE, because this is the way they create stories. However, this is not how things would happen in the real world. Usually, an event can be caused by more than one event, or be caused by an event that happened in the past. The Virtual Storyteller simulates this kind of causality, so we need a way of storing stories that supports this.

An alternative that supports a causal network instead of a chain is the DPOCL story plan. DPOCL uses an approach from planning theory to save the actions that occur in a story to achieve specific goals. The result is similar to the Fabula model: a network graph that stores the events of the story. There are however differences. Because DPOCL is based on a plan structure, it sometimes has characters perform actions just to achieve a main story goal. For example, in figure 11, Joe moves to the docks so that he can be captured by Sam. In the Virtual Storyteller characters could reason out of character to achieve the same effect, but there would also be an in-character reason for Joe to move to the docks, for example that he was hungry and there was a restaurant at the docks.

Because of its focus on a global plan, instead of characters, the DPOCL plan also does not contain the emotions of the characters, like the Fabula model has with the Internal Element. These differences make DPOCL less suitable for the character-based approach of the Virtual Storyteller.

So far it looks like the Fabula model is the best way of expressing the events of the Virtual Storyteller. But that does not mean the current way of storing the Fabula graph is perfect. The current Fabula model is saved in the TriG format, which is a format made specifically for RDF. To make the Fabula model easier to import, a different format could be used. XML is a good choice, since it is standardized and used often. Expressing RDF in XML is possible by using a format such as TriX or RDF/XML (Worldwide Web Consortium, 2004). However, it's also possible to use something entirely different, as the new Virtual Storyteller does not have to use RDF.

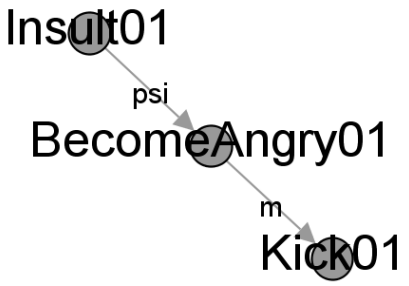

Figure 13. A minimal Fabula, as rendered by Gephi.

We held a meeting with all the people currently involved with the Virtual Storyteller to decide on a data format. We proposed that the Fabula structure will stay the same, but will be stored in the GraphML format. GraphML is a standardized format for graphs, and can be opened in existing graph editing software such as Gephi (Bastian et al., 2009). GraphML allows us to add information to each node and edge. For the nodes, this can be used to store information regarding a story event, such as the subject and object of an action. For the edges, it will be used to store the type of causality this edge represents, i.e. physical, psychological, motivation or enablement. A (minimal) example of a Fabula in GraphML format, as rendered by Gephi, can be found in figure 13. This graph represents a story that involves one character insulting another character. This makes the second character angry, which results in him retaliating by kicking the first character. Example GraphML code for a node and edge of this story can be found in figure 14.

Currently, the Fabula model only contains information about the events in the story. However, some information has to be stored about the concepts, such as characters, places and actions, that the Fabula tells us about. We also should store the words the Narrator should use to describe them. The current Narrator has multiple files to store this. There is a file which describes the setting-specific concepts, and a seperate lexicon. Characters themselves can also have different files. Then there are also files called the Story World Core and Common Sense, that store concepts that are not specific to a certain domain, but that can be used for all domains.

During the meeting we proposed to instead have a single

```
<node id="Insult01">
  <data key="EventType">Action</data>
  <data key="Type">Insult</data>
  <data key="Agens">Henk</data>
  <data key="Patiens">Harry</data>
</node>


<edge id="Psychological01" source="Insult01"
    target="BecomeAngry01">
  <data key="RelType">Psychological</data>
</edge>
```

Figure 14. Example GraphML code for a node and edge of the Fabula graph.

OWL ontology with all the concepts in the story world, for example objects, events, places, characters and actions. The ontology has a tree structure, where a parent node represents an abstraction of its child nodes. The leaves of the structure represent the concrete concepts that can be used in the story. The lexicon information could be stored in two ways. One option is making a different file that contains all the lexicon information. The other option is storing the lexicon information in the ontology. Currently, we have not made a decision on how lexicon information should be stored.

7.3 Document Planner

The document planner is the place where focalization and support of different temporal orders should be added, since these two changes deal with the broad structure of the text.

Focalization will be added in a way that's similar to Montfort (2011), by creating a 'focalized Fabula' from the global Fabula. This Fabula is comparable to Montfort (2011)'s 'concept', and should contain only the story events a certain character can perceive, as well as the Internal Element for that character. It should also contain which character this fabula is focalised from, so the realizer knows from which character's point of view the story is told.

Montfort (2007) describes how to apply different temporal styles to interactive fiction. He describes algorithms in pseudocode which can be used to order a part of story in different ways: chronological, retrograde, zigzag, analepsis (flashback), prolepsis (flash forward), syllepsis (category ordering) and achrony (random order).

Although Montfort (2007) describes *how* temporal styles can be expressed, he does not say *when*. It's still difficult to know when a deviation from the normal chronological order makes the story better. Montfort (2007) describes only a few simple heuristics for a flashback: "If we encounter a character for the second time, select the most salient event from the first time we encountered them" and "If we enter a room, select the most salient things that happened in this room". I think the most practical way of using temporal styles in the Narrator would be to find simple heuristics like that, and see if the change in style improves the story.

If the Fabula model is changed, part of the Document Planner would have to be rewritten to deal with the new input. This change should be made in the InitialDocPlanBuilder. The State Transformer currently contains Dutch canned text. If the Narrator should support multiple languages, this part has to be changed to contain translations of the canned text. To enable reuse of code for the sentence planner, the output of the Document Planner should be in the same format as the current Narrator.

7.4 Sentence Planner

The Sentence Planner from the current Narrator (Slabbers, 2006, Chapter 7) follows a simple design that transforms each specific plot element into a sentence template. This design is sufficient for the current Narrator, and it might also be sufficient for the new Narrator. The templates are already mostly language-independent. If the new Narrator should support English, we should take extra care that the Sentence Planner really generates a language-independent dependency graph.

To support focalization, the template structure would not have to be changed. There should however be a change in the lexicalisation part. This part could lexicalise the focalizing character of the story as 'I' instead of the character's name, if we want to create a first person story.

7.5 Surface Realizers

In this section I explore how well the different surface realizers that I have found fit for this project. I discuss practical considerations, such as programming languages and input formats, that affect the amount of work that would be needed to adapt a specific realizer to work with the Narrator.

Ideally, a surface realizer for the Narrator should:

- Be written in Java, or be easily usable from a Java program
- Have an easy input format that does not take a lot of time to learn
- Support Dutch and English
- Create correct surface text, even for complex sentences

SimpleNLG is the obvious choice for English, as it's written in Java and aims to be easy to use. Unfortunately, it does not support Dutch. It is possible to translate SimpleNLG to different languages: this has already been done for French (Vaudry, 2011) and German (Bollmann, 2011). However, this probably is a lot of a work and does not fall within the scope of this project.

RealPro is another option for English. However, it's a lot harder to use. Reiter and Dale (2000) mention that using RealPro effectively requires you to spend a lot of time learning about the realizer and the theory behind it. It's also not written in Java, and is under a commercial license.

KPML is the only surface realizer that supports both Dutch and English. However it is made in Lisp and has a very complex input structure. Running Lisp code from within a Java program is not straight-forward. The input structure is powerful, but complex and hard to learn.

Alpino's realizer supports only Dutch. Alpino is originally a grammar parser, and the realizer is mostly a proof of concept, instead of an off-the-shelf product like RealPro or SimpleNLG. It's written in TCL, so modifying and using the code from a Java perspective would be more difficult too. It does however have a good grammar model and has some nice features like using an entropy model to create the most fluent sentence from different choices.

Finally, there is Hielkema (2005)'s realizer. It supports Dutch, is written in Java and obviously works well with the Narrator, since it was made for it. A downside is that it might not support complex structures. It's a good idea to keep Hielkema (2005)'s realizer for Dutch, and use SimpleNLG for English.

7.6 Evaluation

Rowe et al. (2009) describe StoryEval, a framework for evaluating automatically generated stories. They describe four different ways of evaluating a story.

The first is evaluation by narrative metrics, this refers to the evaluation of the style, readability, grammar and diction. This evaluation is interesting for this project, because all these things are the job of the Narrator. The second method is a cognitive-affective study, which refers to measuring the cognitive reaction of the reader to a story. Specifically, this can be done by evaluating the presence (i.e. sense of being part of the story), engagement and emotional experience. The third method, called director centric evaluation has a director agent evaluate the story while it is being told. This is not applicable to the Narrator, as this would be part of the simulation in the Virtual Storyteller.

The last method, extrinsic narrative evaluation, is about evaluating the goal of the narrative. This is very important for systems that are used for education and training. This is the case for the Narrator when used with the police training game. However, it can be hard to evaluate during the course of this project, as the new serious game version of the Virtual Storyteller is still under development.

McIntyre and Lapata (2009) use an automatic method of evaluating stories, by using lexical features to measure interestingness and an entity grid to measure coherence. However, their lexical features method was made for English. These would have to be recreated for Dutch, which falls outside the scope of this project. It might be possible to use their coherence model to evaluate Dutch stories, but the different method of generating stories in the Narrator makes stories that are already a lot more coherent than the randomly generated stories made by McIntyre and Lapata (2009).

I propose to have a small user experiment, where human test subjects judge the narrative metrics and cognitive affect as described by Rowe et al. (2009). Testing for the learning goals is important too, but I think this should be part of an evaluation of the entire Virtual Storyteller system when the new version is complete.

8. CONCLUSION

The goal of this paper was to investigate the design of a new version of the natural language generation subsystem of the Virtual Storyteller called the Narrator. The Virtual Storyteller is a multi-agent system that simulates interaction between characters by having agents play the roles of these characters. The Narrator then uses the events from this simulation and converts them into natural Dutch text. The current Virtual Storyteller and Narrator use a data format called the Fabula Model to communicate the contents of the story. A secondary goal of this project was to investigate the Fabula model to find out if it can be improved.

Currently a new Virtual Storyteller is being developed as part of a serious game that trains police officers to deal with loitering youth. The new Narrator should be made for this new serious game. I also propose ways in which the Narrator can be improved for this project. Focalization can be added to put the training police officer in the shoes of the other characters, such as the loitering youth. Style figures such as flashbacks could be added to make the generated text more interesting to read. The Narrator should also be easy to use, and it can be improved further by adding support for a different language such as English.

The architecture of the Narrator will follow Reiter and Dale (2000)'s three-stage model, which specifies a pipeline structure for an NLG system containing a high level document planner, a low level sentence planner and a surface realizer that creates the final text. An alternative architecture from Hayes and Flower (1986) was examined, but I found Reiter and Dale's model more fit for this project.

The document planner is the place where focalization can be added. To do this I will use Montfort (2011)'s method of making a 'concept', a specific story model that only contains the story events that involve the character from whose point of view the story is told. For adding temporal styles such as flashbacks I will draw inspiration from another paper by Montfort (2007) that describes how to order a document plan in these different styles. To find out *when* for example flashbacks are useful to the story, I will try to find simple heuristics. The sentence planner does not have to be changed much, for this part I will try to reuse code from the existing Narrator (Slabbers, 2006).

For the surface realization I examined several off-the-shelf realization systems. From these systems, SimpleNLG (Gatt and Reiter, 2009) was the most fit for the English language, as it is easy to use and learn, and is written in Java so it fits well with the old code. For the Dutch language I will use the existing realizer from the current Narrator (Hielkema, 2005).

To discuss changes to the Fabula model, we held a meeting with the people currently working on the Virtual Storyteller. We proposed a new format for the Fabula model based on GraphML in combination with an OWL-ontology, but at this point the specific format has not been decided.

References

M. Bal. *Narratology: Introduction to the Theory of Narrative*. University of Toronto, 1997.

Regina Barzilay and Mirella Lapata. Modeling local coherence: an entity-based approach. In *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*, ACL '05, pages 141–148, Stroudsburg, PA, USA, 2005.

Mathieu Bastian, Sebastien Heymann, and Mathieu Jacomy. Gephi: An open source software for exploring and manipulating networks, 2009.

John Bateman. What is KPML?, January 2003. URL `http://www.fb10.uni-bremen.de/anglistik/langpro/kpml/README.html`.

John A. Bateman. Enabling technology for multilingual natural language generation: the KPML development environment. *Natural Language Engineering*, 3(1):15–55, March 1997.

Christian Bizer. The TriG syntax, July 2007. URL `http://wifo5-03.informatik.uni-mannheim.de/bizer/trig/`.

Marcel Bollmann. Adapting SimpleNLG to German. In *Proceedings of the 13th European Workshop on Natural Language Generation*, ENLG '11, pages 133–138. Association for Computational Linguistics, 2011.

Lynne Cahill, John Carroll, Roger Evans, Daniel Paiva, Richard Power, Donia Scott, and Kees van Deemter. From rags to riches: exploiting the potential of a flexible

generation architecture. In *Proceedings of the 39th Annual Meeting on Association for Computational Linguistics*, ACL '01, pages 106–113, Stroudsburg, PA, USA, 2001.

Jeremy J. Carroll, Christian Bizer, Pat Hayes, and Patrick Stickler. Named graphs. *Web Semantics*, 3(4):247–267, December 2005.

David B. Christian and R. Michael Young. Comparing cognitive and computational models of narrative structure. In *Proceedings of the 19th national conference on Artifical intelligence*, AAAI'04, pages 385–390. AAAI Press, 2004.

CoGenTex, Inc. CoGenTex, Inc. - RealPro, 2010. URL `http://www.cogentex.com/technology/realpro/index.shtml`.

Daniël De Kok and Gertjan van Noord. A sentence generator for Dutch. In *Proceedings of the 20th Meeting of Computational Linguistics in the Netherlands*, pages 75–90. University of Groningen, 2010.

Albert Gatt and Ehud Reiter. SimpleNLG: A realisation engine for practical applications. In *ENLG-2009*, pages 90–93, 2009.

John R. Hayes and Linda S. Flower. Writing Research and the Writer. *American Psychologist*, 41(10):1106–1113, 1986.

Feikje Hielkema. Performing syntactic aggregation using discourse structures. MSc thesis, University of Groningen, Groningen, The Netherlands, 2005. URL `http://wwwhome.cs.utwente.nl/~theune/VS/FeikjeHielkema_verslag.pdf`.

R. Kasper and R Whitney. SPL: A sentence plan language for text generation. University of Southern California, 1989.

E. Krahmer and M. Theune. Efficient context-sensitive generation of referring expressions. In K. van Deemter and R. Kibble, editors, *Information Sharing: Reference and Presupposition in Language Generation and Interpretation, Center for the Study of Language and Information-Lecture Notes*, volume 143 of *CSLI Lecture Notes*, pages 223–263. CSLI Publications, Stanford, USA, 2002.

Benoit Lavoie and Owen Rambow. A fast and portable realizer for text generation systems. In *Proceedings of the fifth conference on Applied natural language processing*, ANLC '97, pages 265–268, Stroudsburg, PA, USA, 1997.

J.M. Linssen and T. de Groot. AGENT: Awareness game environment for natural training. In *Proceedings of the International Conference on the Foundations of Digital Games*, FDG '13, pages 433–434, 2013.

Hugo Liu and Push Singh. Makebelieve: using commonsense knowledge to generate stories. In *Eighteenth national conference on Artificial intelligence*, pages 957–958, Menlo Park, CA, USA, 2002. American Association for Artificial Intelligence.

Neil McIntyre and Mirella Lapata. Learning to tell tales: a data-driven approach to story generation. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP: Volume 1 - Volume 1*, ACL '09, pages 217–225, 2009.

Chris Mellish, Donia Scott, Lynne Cahill, Roger Evans, Daniel Paiva, and Mike Reape. A reference architecture for natural language generation systems. *Natural Language Engineering*, 12:2006, 2006.

Nick Montfort. Ordering events in interactive fiction narratives. In *Intelligent Narrative Technologies, Papers from the 2007 AAAI Fall Symposium*, pages 87–94, 2007.

Nick Montfort. Curveship's automatic narrative style. In *Proceedings of the International Conference on Foundations of Digital Games*, FDG '11, pages 211–218, 2011.

E. Reiter and R. Dale. *Building Natural Language Generation Systems*. Studies in Natural Language Processing. Cambridge University Press, 2000.

Jonathan P. Rowe, Scott W. McQuiggan, Jennifer L. Robison, Derrick R. Marcey, and James C. Lester. Storyeval: An empirical evaluation framework for narrative generation. In *Intelligent Narrative Technologies II*, pages 103–110, 2009.

Nanda Slabbers. Narration for virtual storytelling. M.Sc. thesis, University of Twente, Enschede, The Netherlands, March 2006. URL `http://essay.utwente.nl/56935/`.

I. Swartjes and M. Theune. The virtual storyteller: Story generation by simulation. In *Proceedings of the 20th Belgian-Netherlands Conference on Artificial Intelligence (BNAIC)*, 2008.

Ivo Swartjes. *Whose story is it anyway? : how improv informs agency and authorship of emergent narrative*. PhD thesis, Enschede, May 2010.

Ivo Swartjes and Mariët Theune. A fabula model for emergent narrative. In S. Göbel, R. Malkewitz, and I. Iurgel, editors, *Technologies for Interactive Digital Storytelling and Entertainment*, volume 4326 of *Lecture Notes in Computer Science 4326*, pages 49–60, Heidelberg, November 2006. Springer Verlag.

Mariët Theune, Nanda Slabbers, and Feikje Hielkema. The automatic generation of narratives. In *Proceedings of the 17th Conference of Computational Linguistics in the Netherlands*, pages 131–146, Enschede, The Netherlands, 2007. University of Twente.

T. Trabasso, T. Secco, and Van Den Broek. Causal cohesion and story coherence. In *Learning and comprehension of text*. Hillsdale, NJ: Erlbaum, 1984.

Leonoor Van der Beek, Gosse Bouma, Rob Malouf, and Gertjan Van Noord. The Alpino dependency treebank. *Language and Computers*, 45(1):8–22, 2002.

Pierre-Luc Vaudry. SimpleNLG bilingue - l'ajout du Français à un réalisateur de texte Anglais, 2011. URL `http://www-etud.iro.umontreal.ca/ vaudrypl/snlgbil/SNLGbil.pdf`.

Worldwide Web Consortium. RDF primer, February 2004. URL `http://www.w3.org/TR/rdf-primer/`.

R. Michael Young and Johanna D. Moore. Dpocl: A principled approach to discourse planning. In *In Proceedings of the 7th International Workshop on Natural Language Generation*, pages 13–20, 1994.

R.M. Young. Story and discourse: A bipartite model of narrative generation in virtual worlds. *Interaction Studies*, 8(2):177–208, 2007.

René Zeeders. Narrator - system description (internal document), 2008.