University of Hamburg
Department Informatics

Template-based incremental natural language generation in dynamically evolving domains

Higher Seminar Paper

Alexis Engelke
Matr.Nr. 6269552

September 30, 2014

Abstract

Natural language generation typically employs templates or grammars. Grammars can be overly complex if they need to cover only limited domains in which a template-based approach may be sufficient. In addition, derivation from grammars may be infeasible in incremental use-cases in which utterances can grow very long.

We implemented a template-based incremental natural language generation component that is able to take available timing into account in order to deliver as much information (with as low delay) as possible given the timing constraints of a highly dynamic domain. We present a simulation framework that uses types for template messages, which can be open-ended, and be concatenated with continuations later on. Our system produces more elegant sentences than the non-incremental baseline within a small (but extensible) domain.

Contents

Chapter 1

Introduction

Auditive output of a computer system can support humans in many cases. Many people know navigation systems, which tell the car driver when to turn in order to reach a target defined by the car driver himself. Through the auditive output, the car driver can focus on the street and is not required to know the way or to look at a map.

This paper is about a contrary system: the CarChase system, as described by Baumann and Schlangen (2013). Such a system can be used to guide blind people (Lohmann, 2012) or to comment on soccer games, as shown by Kilger and Finkler (1995). This system comments the path which is driven by the car. It is non-interactive and reads the path of the car as well as the appropriate comments from the same file. The aim of this paper is to give interative possibilities of control over the path and speed of the car to the user, where a dynamic comment is spoken parallely.

To realize speech from domain-specific data, two components are necessary: One component that generates the text that will be spoken, is called the *natural language generation* component. This component produces the text based on the *concept* of what has to be spoken. The other component articulates the text, i.e. generating and playing waveforms based on the given text, and is called the *text-to-speech* component. This process is called *speech synthesis*.

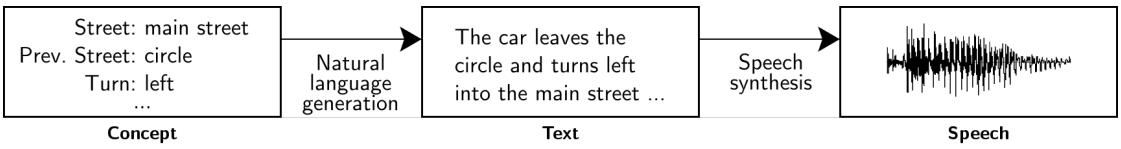


Figure 1.1: Process of generating auditive output from a concept

However, this approach is inflexible: Once a text is produced, it will be articulated, and we cannot interrupt it, if the situation changes. Therefore, *incremental speech synthesis* is preferred. Following Baumann (2013), incremental speech synthesis means a progressive articulation of the input text. Incremental speech synthesis allows to change the text dynamically. This opens several possibilities: We can insert another text if the situation requires it, or we can start articulating

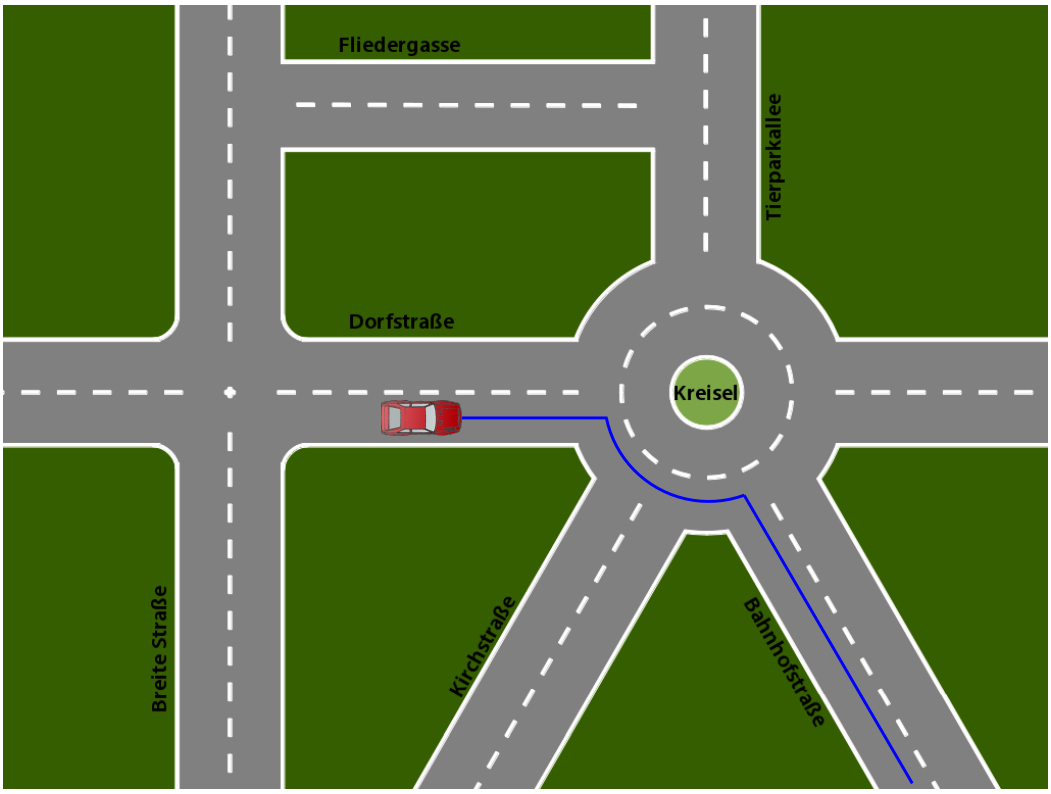


Figure 1.2: One car path in the orginal CarChase system.

about a concept, even if the information is incomplete. Furthermore, incremental speech synthesis allows the use of hesitations.

There are basically two ways to realize a natural language generation component: a template-based and a grammar-based approach. The template-based approach has some hand-written texts (called *canned texts*), which contain gaps. These gaps get filled with the information from the concept. Using this approach, the number of possible texts to describe the concept is finite. A grammar is described b y a set of rules. Using a grammar-based approach, complex concepts can get articulated with a few grammar rules. This way of modelling seems to be more simple compared to the set of canned texts of the template-based approach. However, in incremental use-cases, where the utterances can grow long, this approach might be impracticable. Therefore, in simple domains, the template-based approach can be considered as sufficient.

One system that uses incremental speech synthesis is the *CarChase system*, as described by Baumann and Schlangen (2013). This system comments on the path of a car, which drives on a map. It describes the street the car is currently driving on and announces junctions including the way it takes afterwards. Incremental speech synthesis is used to start commenting about the car turning at the next junction, if the car is likely to turn. Once the car actually turns, the new direction is appended. If the car did not turn after the text was spoken, a hesitation ("ugh")

gets introduced in order to bridge some time. In some situations, an action is predicted incorrectly. In this case, a hesitation is used to show that the articulated prediction is wrong, and the spoken utterance gets corrected.

As the CarChase system reads all the texts and its appropriate timings from a hand-written configuration file, it cannot react on any change of the car path. Thus, a natural language generation component is needed to allow a dynamic commenting of the user's actions. In this paper, one implementation of a natural language generation component will be presented, which features a low timespan between the action and the spoken information.

In this paper, a software system named CarChase 2 that comments on the path of a driving car will be presented. The path and speed is defined by the user interactively. This system has a natural language generation component, which features a low timespan between the car action and the description in the comment.

Overview

In chapter 2, existing approaches for a natural language generation component will be presented. Then the existing approaches will be discussed and it will be explained why a new design is neccessary. In chapter 3, two existing systems that are related to the CarChase system will be described: one other commenting system and one system that is used for the implementation. Then, in chapter 4, a design approach to solve the resulting problem of the missing natural language generation component will be presented. In chapter 5, one possibility to model a natural language generation component for the extended CarChase domain (named CarChase 2) will be described. The architecture of the resulting CarChase 2 application and the implementation of the natural language generation component that is described in chapter 4 and 5 will be explained in chapter 6 in detail. The implementation will be evaluated and discussed in chapter 7, including a survey conducted to compare the new implementation with a non-incremental baseline system. In chapter 8 the results will be summarized and an outlook on future work as well as possible extensions will be given.

Chapter 2

Motivation

The process of natural language production consists of three parts (Reiter and Dale, 2000): the planning, the realisation and the synthesis. In a simple domain, as the CarChase domain, a planning component is not needed. Thus, we focus on the realization (the natural langugage generation) and the synthesis in the following.

As mentioned in the introduction, natural language generation typically employs templates or grammars. In this section, both possibilites for natural language generation will be presented shortly.

2.1 Template-based approach

One simple approach for a natural language generation component is to use templates. (Reiter, 1996) A template is a text with gap, which gets filled with the appropriate information. Here is an example for a template:

> The car drives into the [Street Name].

The gap, annotated with the information that the name of the street where the car just drove into, will get filled. At runtime, the text states, for example: "The car drives into *main street*".

This approach is simple and efficient, as there is nearly no effort to generate the language. In limited domains this approach can be considered as sufficient, as the number of templates to be written is limited, too. Some flexibility can be added to this approach by defining different template for each situation, and by choosing one randomly.

However, a template-based approach is limited to concepts of a known size, and therefore inflexible: if we do not know the size of concept at the time of writing the templates, we cannot ensure that the whole concept gets articulated. Furthermore, as templates have no semantic information, it is impossible to append continuations.

2.2 Grammar-based approaches

More flexibility can be gained by using grammars. Grammars allow a recursive embedding, and therefore offer more flexibility. To ensure that the semantics of the produced sentence is correct, it is necessary that the grammar rules have semantic information. The generation has to happen with respect to the semantics of the rules. There are different possibilities for grammars to generate natural language, in the following natural language generation through a constraint-based and a tree-adjoining grammar will be explained shortly. (Neumann, 2002)

2.2.1 Constraint-based Grammars

Natural language can be generated using a constraint-based grammar, e.g. the functional unification grammar (FUG). Information is generated through unification of descriptions of a sentence. Originally designed for parsing, where the information is specified at the beginning, and the syntactic and semantic information is generated through unification, it can be used for natural language generation: starting at the semantic information, a text can be generated through unification.

2.2.2 Tree-Adjoining-Grammars

Another approach for natural language generation through grammars is the use of *Tree-Adjoining-Grammars* (TAG). (Joshi and Schabes, 1991) A tree-adjoining grammar consists of a set of terminal symbols, a set of non-terminal symbols, a starting non-terminal symbols and two sets of trees. One set contains the so-called initial trees. An initial tree has a non-terminal as root node and terminals as well as non-terminals in the frontier. The other type of tree is the auxilary tree: in contrast to the initial tree, one non-terminal element with the same type as the root node at the frontier is marked, and called the *foot node*.

The natural language is generated by starting from the specified starting symbol. The generation process involves two types of operations: The substitution and the adjoining. In the process of substitution, a non-terminal symbol gets replaced with an initial tree. The adjoining process is performed on any tree and an auxilary tree: a node with the non-terminal of the auxilary tree gets moved as a replacement to the foot node of the auxilary tree. The auxilary tree then gets moved to the place where the node we moved was.

A tree-adjoining grammar can be used for a dynamic appending of continuations by starting the synthesizing with having an open non-terminal, which gets replaced as more information is available.

A grammar-based approach for natural language generation offers more flexibility than a template-based approach. However, using a grammar involves much more computation effort to generate the natural language, and requires the use of open nodes to allow continuations.

2.3 Discussion

A template-based approach for natural language generation has the advantage that is is efficient, and can be considered as sufficient in limited domains. As the messages only realize small concepts, it is possible to replace or remove some messages later on. However, it is inflexible and limited to concepts of a known size, and, as the templates have no metadata, they does not support continuations. As a consequence, there are many short and mostly redundant sentences, because each template itself represents a complete sentence.

A grammar-based approach for natural language generation offers more flexibility, which is useful in large domains with complex sentences. In contrast to a template-based approach, it requires more computation effort, and it is not simple to change or remove some parts of a message afterwards: For any change, we need to recompute the sentence grammar while paying attention to not change the parts of the sentence that are already synthesized. Furthermore, continuations do not seem intended, and open nodes at the end are no elegant way to allow continuations.

As the CarChase domain is a limited, but dynamic domain, neither a template-based nor a grammar-based approach seems feasible. However, if the templates were extended with meta information about the grammatical structure, continuations would be possible. This would lead to an efficient approach, which offers the flexibility of continuations while being able to dynamically remove or shorten the messages. Continuations avoid redundant information, which is especially important in fast-paced domains, as the CarChase domain.

Summary

For the CarChase domain, the approach is a template-based approach where the templates have metadata, leading to a flexible and efficient approach for natural language generation. In the next chapter, one other commenting system and one system for incremental speech synthesis will be presented. In chapter 4, the design approach to realise the extension with metadata will be presented.

Chapter 3

Existing systems

In this section, two existing systems which are related to the CarChase system will be described. The first system is the VAVETaM, which is a dynamic commenting system, as the CarChase system. The second system is InproTK, which is the toolkit used for the speech synthesis in the original CarChase system and will be used for the implementation of the new approach later on.

3.1 VAVETaM

The VAVETaM (Verbally Assisted Virtual Environment Tactile Map) system is a commenting system on the position of the user in a virtual tactile map. (Lohmann, 2012; Lohmann et al., 2012) Users explore a tactile map with a three dimensional pointer device, while the system comments on the left and right area as well as on the upcoming area in the direction of movement.

As the user can move the pointer quickly and uncontinuously through the map, the commenting domain is highly dynamic. Featuring incremental speech synthesis, a dynamic adaption of new utterances is possible. If the situation changes, the previous utterances which are not completely spoken are handled in the following way: (Eichhorn, 2013)

1. The system computes formulations for the new situation, where the new formulation matches the old formulation to the actual progress of speaking. I.e., that possible formulations which do not have the same beginning as the actual utterance cannot be chosen as dynamic adaption.

2. The unspoken words get revoked and replaced with the not spoken part of the new formulation. To show the difference to the user, an "hmm" gets inserted at the point of difference.

The VAVETaM system uses a template-based approach for natural language generation (Lohmann, 2012). The generated semantic information (so called *preverbal messages*) about the surrounding of the user's position gets filled into different templates, where one template is chosen randomly. All preverbal messages

and spoken utterances get tracked by a *memory component*. This component can mark some messages as repetition of previous spoken utterances. In this case, a word that indicated the repetion (e.g. *again*) gets placed.

However, VAVETaM does not support continuations of already uttered sentences: whenever a new preverbal message gets formulated, the previous utterance does not *directly* influence the beginning of the new formulation. Continuations are useful to refer on already spoken references, e.g. (longer) street names, with a pronoun. This has the advantage of delivering the same amount of information with a lower offset from the new situation to the actual content of the assisting formulation.

3.2 InproTK

The CarChase commenting system described in section 1 uses InproTK as the text-to-speech (TTS) component for the incremental speech synthesis. InproTK is an incremental processing toolkit for both, speech synthesis and speech recognition (Baumann, 2013), and follows the IU model described by Schlangen and Skantze (2009). It is based on incremental units and incremental modules. The final process of speaking is performed by a dispatcher.

3.2.1 Incremental Units

Incremental units (IUs) are chunks that store information about text and can be used in both cases, speech recognition and speech synthesis. There are two types of connections between IUs: same-level links and grounded-in links. Same-level links are typically used to connect IUs that follow on each other. Hierachical structures are implemented by using *grounded-in links*. These links are generated by processors, which are part of an incremental module.

InproTK offers several types of IUs: at the top level, there are chunk IUs. A chunk IU has word and hesitation IUs as grounded-ins. A hesitation IU is an "ugh", which gets removed, if another IU gets appended after the hesitation. Both, hesitation and word IUs have syllable IUs as grounded-ins.

3.2.2 Incremental Modules

An incremental module consists of three parts: a left buffer, a processor and a right buffer. The left buffer contains IUs, which can be connected with same-level links. The processor reacts on a changing left buffer and generates new IUs, which are connected with the IUs in the left buffer, e.g. via grounded-in links. If there are IUs in the right buffer, which are outdated, they get revoked and replaced by new IUs

As an incremental module is connected with other modules, i.e., that the right buffer of a module is the left buffer of at least one other module, the processors

of other modules get activated. However, as the computed results of the modules can change, the result is called a *hypothesis*.

One special type of incremental module are the *IU sources*: their left buffer is not connected, and the IUs in the left buffer are ignored. A source only produces IUs, their processors get triggered by other events, e.g. a user input.

3.2.3 Incremental Speech Synthesis

To support dynamic incremental speech synthesis, the right buffer of an IU module supports three types of so-called edit types:

- **Add:** With this edit type, we append a new IU to the right buffer of a module.
- **Revoke:** We revoke the last IU of the hypothesis. In combination with the add operation, it is also possible to revoke any other IU in the right buffer: We simply revoke all IUs up to the IU we want to revoke, and re-add the others.
- **Commit:** If an IU is commited, it cannot get revoked with future edits. This allows more stability in other IU modules. InrpoTK commits IUs automatically when they get synthesized, because an IU that is already in progress or finished with synthesizing is impossible to get revoked.

Summary

The VAVETaM system currently uses a simple template-based approach for natural language generation. The approach that will be presented in the next chapter can be adapted to this system and improve the produced language. The incremental units and modules of InproTK will be used for the incremental speech synthesis in implementation of the new approach, presented in chapter 6.

Chapter 4

Design

In this section, a design approach for a natural language generation component will be presented. This approach is implemented in the CarChase 2 system, which will be described in section 6. As described in section 1, the interactive CarChase 2 system exposes the control over the speed and path of the car to the user.

The natural language generation component uses a template-filling approach for natural language generation. This approach is simple and efficient: the generation of natural language takes only some milliseconds, which is important for a responsive application. Furthermore, a template-based approach seems to be sufficient for the simple CarChase 2 domain.

4.1 Patterns

The natural language generation component uses patterns for text generation. A pattern consists of conditions, some template messages and an *optional* flag.

A condition is a check of a variable against a value or another variable. This is used to detect a change of the street or mentioning an upcoming junction. It is also possible to detect changes to the speed and the environment of the carHere are some examples for a condition:

- Detect a street change: Street $\neq$ Previous Street
- Detect an upcoming junction: Upcoming Junction = yes
- Whether the car is driving slowly: Speed $\leq$ 2 px/s
- Whether the street the car is driving on is bidirectional: Bidirectional = yes

A template message is consists of an *information level*, a *start type*, an *end type* and the actual text to speak. The text is already formulated and can contain named gaps (variables). The variables will be replaced with the appropiate and specified information, e.g. the name of the street the car is currently driving on. Here is an example for such a text:

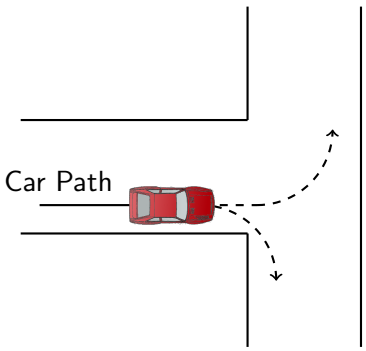


Figure 4.1: Typical situation: the car reaches a junction, and is likely to turn. However, we do not know when the car will turn and whether it will turn left or right. We can only assume that the speed does not change, and choose the information level appropriatly.

- The car turns [Left/Right] into the [Street].
- The car drives into the [Street].

4.2 Information level

The information level is a measure for the number of details of a message. The information level is a value in the range of one to three. A low information level means that the message has less details and is shorter, whereas a high information level is used to describe long utterances with many of details.

The information level is used to estimate the length of a message. If the car is driving with a high speed, the system will try to select a message with a lower information level. However, if the car is driving slowly, the system will try to choose a utterance with a higher information level.

This system of message selection tries to maximize the number of details given while trying to reduce the offset between the actual action of the car and the spoken utterance. Furthermore, if the car is driving slowly, a longer message will be used to reduce the eventual break between two actions, e.g. the car is driving towards the junction and turns. (See figure 4.1.)

Here are some examples for the three information levels:

Level	**Example sentence in the situation of figure 4.1**
1	The car reaches the junction...
2	The car is driving towards the junction...
3	The car is driving slowly towards the junction with the main street...

Table 4.1: The information levels with examples

4.3 Type system

As mentioned above, each template message in a pattern has two types, one for the start and one for the end of the possible utterance. A *type* consists of two properties: the completeness and the manner. The completeness property describes whether the beginning/ending is a valid beginning/ending of a sentence: The utterance "The car drives into the main street" is a complete sentence and does not require another message to be uttered to complete the current sentence. However, the text "and turns right into the circle" requires a matching utterance before this text can be articulated.

The second property describes the manner of the beginning/ending of the utterance and is used for appending continuations. For example, the utterance "and drives into the circle" should have another manner at the beginning as the utterance "turns left", because both cannot be replaced with each other.

For each use-case, the types defined can be varied. In the following, the completeness property will be described by an F(*ull*) for complete sentences and an R(*equires*) for incomplete sentences. The manner is a positive number.

4.3.1 Type matching

As mentioned in the previous section, the types are used to append continuations. There are three different cases if the system needs to append a new utterance:

- The dispatcher is not speaking
- The dispatcher is speaking and the previous utterance has a complete ending
- The dispatcher is speaking and the previous utterance requires another message at the end

In the first case, we can just start a new sentence with a message which start type is *complete*.

In the second case, we can either start a new sentence as in the first case or try to append a *type-matching* continuation. A continuation is a message, which start type is marked as incomplete sentence beginning. A continuation is type-matching, if the manner of the ending of the previous utterance is the same as the manner of the start type of the continuation. However, if a type-matching continuation is available, it will be preferred over the beginning of a new sentence.

In the third case, we want try to find a type-matching continuation, as defined above. If we are unable to find a continuation, a new sentence will be started.

4.3.2 Avoiding repetitions

However, the manner can be also used for avoiding repetitions. As figure 4.3 shows, the pattern that described the upcoming junction has two messages: One that mentions the street name and one that does not. As both have different manners,

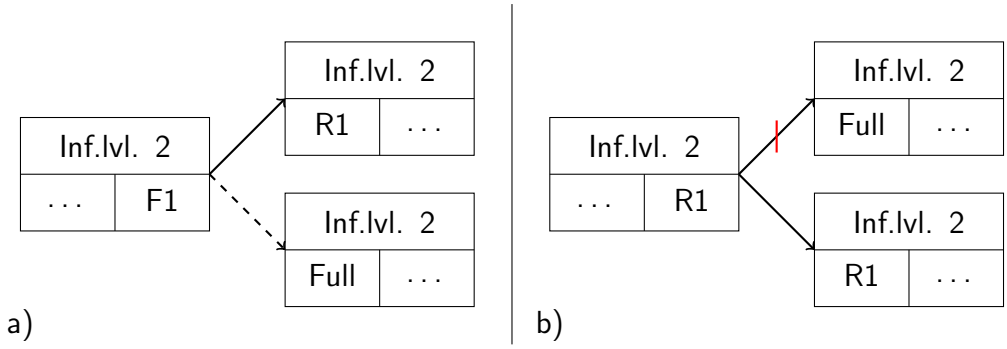


Figure 4.2: Appending continuations. Each block represents a message, with an information level (2), and two types. a) visualizes that continuations are preferred over starting a new sentence, while both options are feasible. b) visualizes that starting a new sentence after an open phrase is not feasible. If no other type-matching continuation were available, we would have to choose that option.

only those messages are possible continuations in the pattern that describes the situation after the junction, where the name of the street is mentioned exactly once in both utterances.

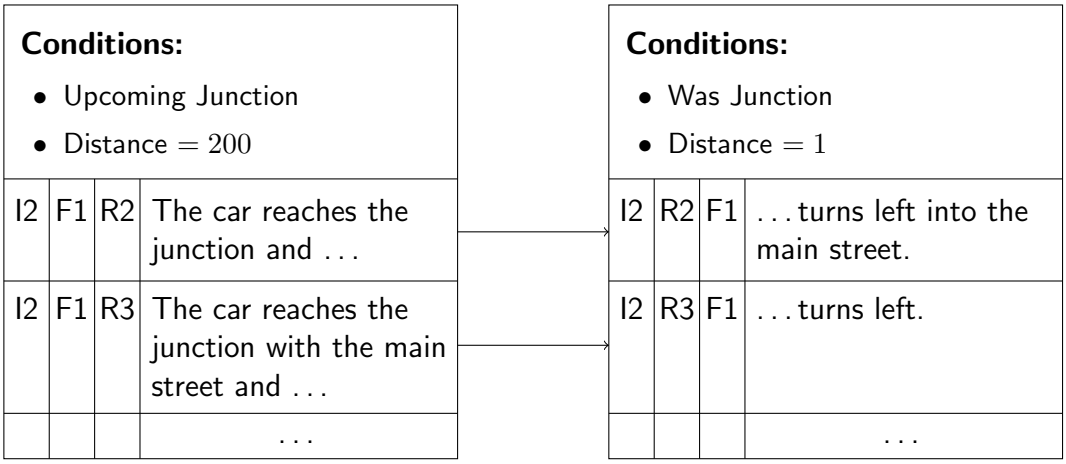


Figure 4.3: Avioding repetitions using types: The left pattern described the situation when the car reaches a junction. The other pattern describes the situation after the junction. The first has two messages with different end types. For each message there is exactly one type-matching continuation in the right pattern, so that the name of the street is always mentioned only once.

4.4 Incremental articulation

Through InproTK we can do incremental speech synthesis. If a new message has to be articulated and there is an offset, we try to reduce the offset by either revoking

previous utterances, or by shortening them. If an optional utterance is outdated, we can revoke the utterance with respect to the type of the previous and the following utterance.

However, if the utterance is not optional or the system cannot remove it, because the types do not match, the system tries to use a shorter utterance with the same types.

This way of dynamic shortening and removing allows to reduce the offset between the action of the car and the speaking of the information.

Summary

The presented approach allows a dynamic appending of continuations to already articulated sentences, due to a type system that distinguishes in full and incomplete sentences. Through the different information levels, we take timing into account, and try to reduce the eventual breaks while delivering as much information as possible.

Chapter 5

Modelling

Based on the design approach presented in chapter 4, there are many possibilities to model the CarChase 2 domain. In this section, one modelling approach of the natural language component for the CarChase 2 domain will be presented.

5.1 World description

The world used in the CarChase system is loaded from two files: one image, which gets rendered, and one metadata file that contains information about the streets (called `world.txt`). In the metadata file, there is a list of points, where each point has a name and a coordinate in the image, followed by a list of streets. A street is a list of points described earlier, and onedirecional or bidirectional. To model a circle, it is necessary to add the first point from the list again as the last point.

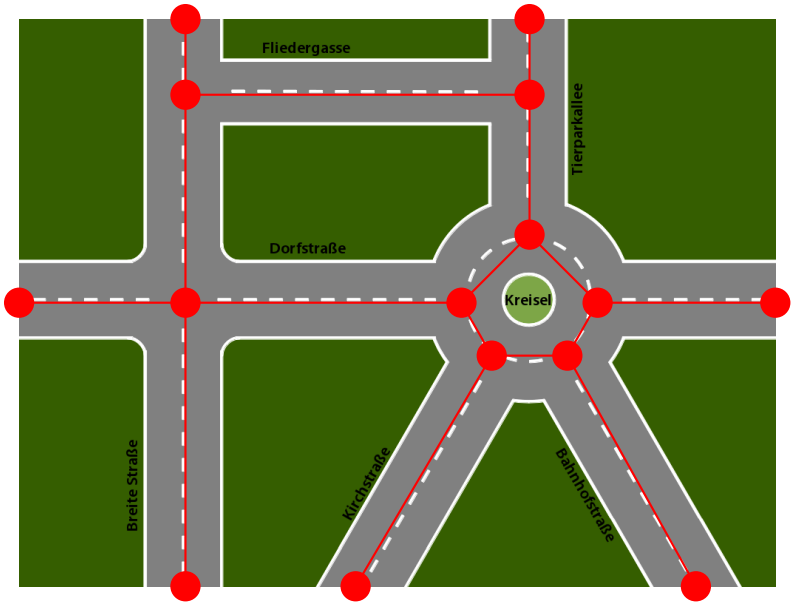


Figure 5.1: The points and streets of one map. The names of the points and streets are not shown in this visualisation

5.2 Types

As described in section 4.3, a type consists of a completeness property and a manner. Table 5.1 lists all types used in the application. The types R3 and R5 as well as R4 and R6 actually have the same manner in the utterance. The only difference is to avoid the repetition of the street name: As described in section 4.3.2, one type mentions the new street before the car turns (R5) while the other type mentions the new street after the car turned (R3).

Type	**Example**
F1 (start)	The car drives ...
F1 (end)	... drives into the the main street.
R1 (start)	**and** drives into the circle ; **and** reaches the junction.
R3 (end)	... driving towards the junction **and**
R3 (start)	... turns left into the main street.
R4 (end)	... driving towards the junction **and turns**
R4 (start)	... left into the main street.
R5 (end)	... reaches the main street **and**
R5 (start)	... turns left.
R6 (end)	... reaches the main street **and turns**
R6 (start)	... left.

Table 5.1: All types used in the application (with examples)

5.3 Pattern definition

To allow more flexibility, the patterns are loaded from a file (called `patters.txt`). This file contains a list of patterns, as described in section 4.1. Furthermore, it includes the values used for left and right and the definition of inflectional forms of the steet names.

5.4 Inflectional forms of street names

In German, it is necessary to choose the inflectional forms appropriatly. The system offers support for inflectional forms of the street names through variables: one flexion form to match with the preposition “in” and another flexion form to match with the preposition “out of”. There are defaults for the prepositions of the street names, for some streets these defaults need to be overwritten. This is specified in the definition of the patterns. Here is an example how it is used:

- The car drives out of [Previous Street, flexion out of] into [Street, flexion into].

5.5 Variables

Table 5.2 lists all variables, which value is computed and stored in the dictionary, combined with the description and an example value. This set of variables includes all necessary information about the car and its environment to generate an appropriate description of the car path.

Variable Name	**Description**	**Example (German)**
street	name of the current street	Fliedergasse
prevstreet	name of the previous street	Kreisel
junctionstreet	name of the crossing street at the next junction	Breite Straße
prevjunction-street	name of the crossing street at the last junction	Tierparkallee
flex1*street	flexion form 1, to match with preposition "in"	die Fliedergasse, den Kreisel
flex2*street	flexion form 2, to match with preposition "out of"	der Fliedergasse, dem Kreisel, der Breiten Straße
int*street	internal name	Dorfstrasse-Ost
distance	Distance to the next waypoint in pixels. *This variable is not instanced, see section 6.4.3*	–
direction	Direction of the car	1
prevdirection	Previous direction of the car	2
bidirectional	Street is bidirectional, 0 = no, 1 = yes	0
isjunction	Is the next waypoint a junction? (0 = no, 1 = X-Junction, 2 = T-Junction)	2
wasjunction	Previous waypoint was a junction? Same values as *isjunction*	0
numstreets	Before a junction: number of streets crossing	2
leftright	After a junction: whether the car turned left or right	links
speed	Current speed of the car in speed units, values are in the range of $0-3$	1
prevspeed	On speed change: previous speed of the car	2

Table 5.2: All variables instantiated at runtime.

Chapter 6

Implementation

To implement the CarChase 2 system based on the original system, various changes and additions are necessary. In this section, the architecture of the original CarChase system will be presented first. After the presentation of the original CarChase system, the changes and extensions of the implementation for building the CarChase2 system will be explained in detail.

6.1 CarChase system

The original CarChase system mainly consists of three components: The *CarChaseExperimenter*, the *CarChaseViewer* and the *Articulator*.

6.1.1 Experimenter

The experimenter is the main component of the CarChase system. It is responsible for loading and executing the configuration, which is loaded from a file, as well as for setting up the window for the viewer component.

6.1.2 Viewer

The viewer is the component that draws the map and the car on the screen. It is based on *timelines*: For each path the car is driving, a timeline is set up, which interpolates the car position and angle over time and repaints the image on the screen for the given duration.

6.1.3 Articulator

The class `Articulator` specifies one abstract method: the method `say` which gets called when something should get articulated. The original system has two articulators: the *standard* and the *incremental* articulator. The standard articulator follows a very simple strategy: if a new message is available, we stop the current message, if any, and start dispatching the new message. However, the behavior

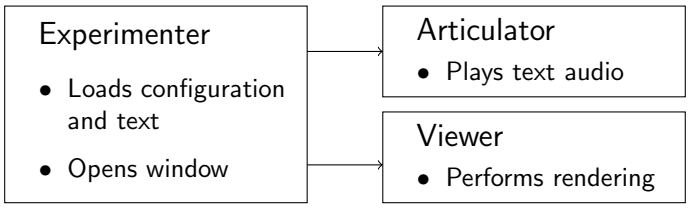


Figure 6.1: Components of the original CarChase system

was changed later to the following strategy: all messages get enqueued, except optional messages when the dispatcher is currently speaking.

The incremental articulator follows a different strategy: if the dispatcher is currently speaking, we append a continuation, otherwise the new text gets uttered. If no continuation is specified, the application crashes.

For the final articulation, both articulators use a modified installment IU, which is named "HesitatingSynthesisIU". This installment appends hesitations if the given text ends with `<hes>` and removes them if a continuation gets appended.

6.1.4 Configuration

The experimenter loads all *actions* from a configuration file at the beginning. Each action has a start time when it should get triggered at the execution. There are four types of actions:

- World Start Action: this action gets triggered once at the beginning to initialize the position and the angle of the car.
- World Action: this action gets triggered when the car should drive another path, e.g. change the street at a junction. This action has two additional properties: the duration and the target point.
- TTS Action: this action gets triggered when a message should be uttered. This action has three additional properties: an optional flag, the text to speak and optionally a continuation.
- Shutdown Action: this action gets triggered when the application should quit.

6.1.5 Execution

After the configuration file was parsed, all TTS and world actions get precomputed: for each TTS action, a new HesitatingSynthesisIU is set up, which precomputes the IU network for the text and, if available, the continuation. For each world action, the viewer computes a timeline for the animation.

Once all actions are precomputed, they get executed according to the start time of the action. A world start action is executed by the viewer in changing

Figure 6.2: Components and their relations for the generation of auditive output

the position and angle of the car, a world action by playing the precomputed timeline. To execute a TTS action, the method `say` gets called on the articulator. A shutdown action is performed by just exiting the application.

As a consequence, the only use of InproTK is in the precomputation phase. The articulators just play the precomputed audio data.

6.2 New Architecture

In the new architecture, there are three new components: a world component, a component for managing the interactive control (called *interactive configuration*) and a natural language generation component. The three existing components, the experimenter, the viewer and the articulators, are still existent in modified form. For the incremental articulation, InproTK will be used, as the original CarChase system does.

The general architecture for producing the output is as follows: the natural language generation component gets information from the viewer about the position of the car and emits so called *articulatables*. These are processed by an articulator to IUs, which are synthesized by InproTK.

In the following, all components and their relations will be presented in detail.

6.2.1 World description

In contrast to the original system, where to configuration described the car path in pixels, we need a description of the world to define where the car can drive. This is implemented by putting a set of `WorldPoints` on the map, where each point has a name and a coordinate in pixels. These points are connected with streets. Each street has a name, an internal identification name, a list of points and a bidirectional flag.

The world description is loaded from a file and parsed by the class `World`, which also stores the information.

6.2.2 Configuration

The configuration described the path and speed of the car. As the configuration is interactive, the class `InteractiveConfiguration` is a key listener and processes the events to the car path and the speed. On a change to the speed, the experimenter gets notified.

The car path is selected with the number keys on the keyboard: given a key press with number n, at each junction the n-th street is chosen as the next path. The speed is an integer value in the range of 0 to 3, while 0 means that the car stopped, and can be controlled via the `+` and `-` keys on the keyboard.

However, the configuration is still scriptable and can be parsed from a configuration file, which includes the car path and times for speed changes.

6.2.3 Experimenter

The experimenter is still responsible for setting up the window for the viewer. In contrast to the original system, there are no actions to dispatch. To be able to react on user inputs, the new experimenter is *event-driven*: if the car finished driving one path, the experimenter gets notified via an event, and sends a new path to the viewer. When the experimenter receives a speed change event, the new path with the new speed and duration and starting offset is sent to the viewer.

6.2.4 Viewer

In contrast to the original viewer, the new CarChase viewer is not based on timelines but on a PApplet. This allows a permanent redrawing at a lower frame rate of 30 frames per second and makes user interaction easier.

A fundamental change in the architecture is that not the experimenter notifies the viewer about a new path but the viewer notifies the experimenter when the car finished driving the specified path. To allow speed changes, the rendering of a path can be interrupted.

To improve the rendering of the car in a curve, the car path consists of *segments*. There are two types of segments: circle and line segments. The first type is used to render the changing angle of the car in a curve or street change. This is done by calculating a circle with a radius of 50 pixels to match the given angles of the streets. Line segments describe the car path along the street.

As there is no script for the TTS actions, the matching patterns have to be detected, and the detection has to get triggered. This is the second task of the viewer: On every drawing, the natural language generation component gets informed about the change of the car position. This is done by wrapping all data about the car position (e.g. the street name, the direction, the position) into an instance of the class `CarState`. This data also includes the position from the previous rendering, so that the range the car drove in the time is part of the state.

In order to keep high rendering frame rates, the dispatching of these updates is done in the special `DispatcherThread`. This thread checks for all patterns whether they match. The produced text of the first matching pattern will be sent to the articulator.

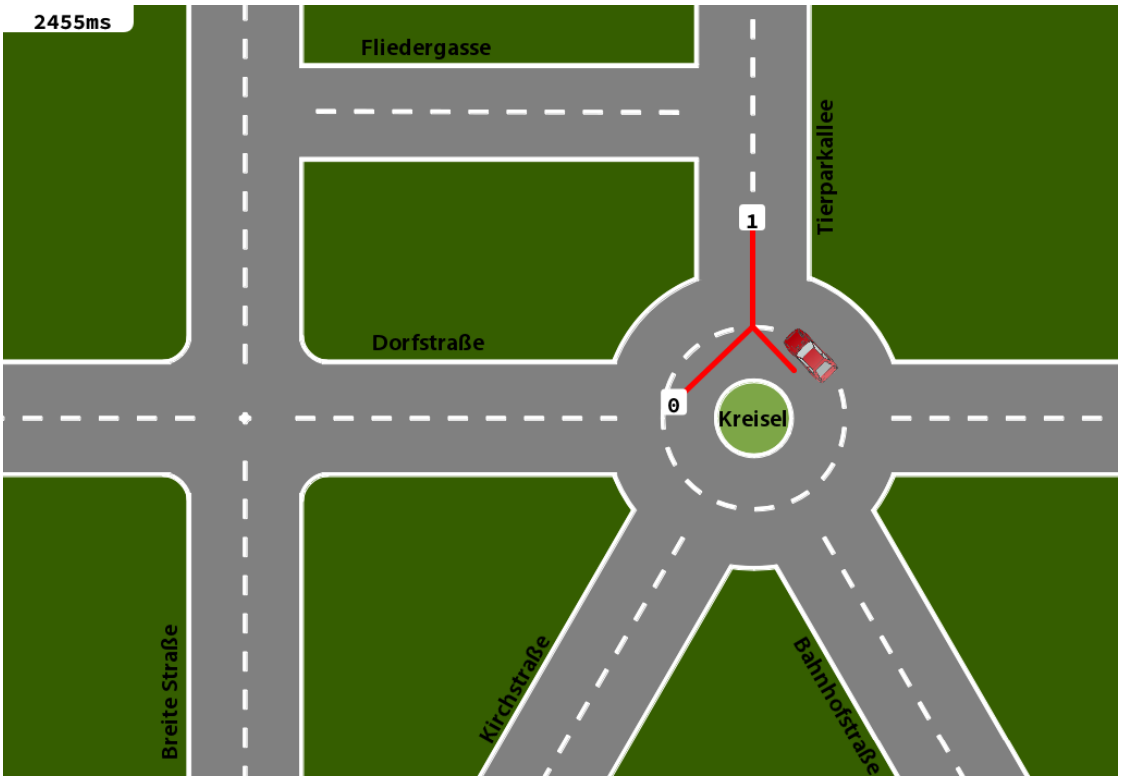


Figure 6.3: The CarChase 2 system in a typical situation: The numbers indicate the different possibilities of where to drive, the user has to choose the path.

6.3 Articulator

An articulator is the interface between the natural language generation component, which produces The abstract class `Articulator` is now based on so called `Articulatable`s. The method `say` does not require a text with a possible continuation but an articulatable. To offer more flexibility to the natural language generation component, three more methods have to be implemented by the articulators:

- `getLast`: The last upcoming or ongoing articulatable
- `isSpeaking`: Whether the dispatcher is speaking
- `reduceOffset`: Reduce offset to the last enqueued articulatable

6.3.1 Articulatables

An articulatable has two messages: a preferred and a short message and has to implement these four methods:

- `getPreferredText`: The preferred text to be articulated.
- `getShorterText`: The shorter text, which will be eventually articulated.

- `isOptional`: Whether the articulatable can get removed.
- `setUseOfShorterText`: To set the use of the shorter variant.

As in the original system, there are two articulators implemented: a standard and an incremental articulator.

6.3.2 Standard Articulator

The standard articulator follows the same strategy as the original, i.e. optional articulatables do not get uttered if the dispatcher is speaking. In the implementation, some changes are necessary to match the new articulator interface and the changed architecture of InproTK.

The biggest change is that a `SysInstallmentIU` is no longer created. Instead, an IU source is implemented, which wraps the preferred text of the articulatables in a `ChunkIU` and appends it to the right buffer.

The four additional methods are implemented by returning a default value: We never expose whether the dispatcher is speaking, the method `isSpeaking` returns `false`. As the standard articulator does not save the articulatables, we cannot get the last articulatable, and we cannot remove any upcoming articulatable.

6.3.3 Incremental Articulator

The incremental articulator is the articulator which supports incremental speech synthesis. However, this articulator has no real strategy, every articulatable is enqueued. The natural language generation component is the component that actually manages the articulatables.

As the for standard articulator, an IU source is implemented. In each `ChunkIU` the articulatable is stored as user data. The method `say` of the IU source takes two other parameters next to the articulatable: a boolean whether the shorter or the preferred variant shall be used and a boolean whether the IU should get committed. The method itself just triggers the IU source to append a `ChunkIU`, and a `HesitationIU`, if the text ends with `<hes>`, and commits the `ChunkIU` if wanted. The method `isSpeaking` of the IU source returns whether the dispatcher is currently speaking.

The method `getLast` gets the last `ChunkIU` in the right buffer of the IU source and returns the articulatable. If there is no such IU, it returns `null`.

To be able to avoid a repetition of the same sentence structure, the chosen articulatables, and whether the short version is used, will be stored in a list, named `articulated`.

The required method `reduceOffset` first revokes all upcoming and not-commited IUs and stores them in a list. Now, for each `ChunkIU` in that list, we get the articulatable. If it is optional, we first check for the next articulatable object in the `articulated`-list. If it has no follow-up or the type of the follow-up is the same as the articulatable we try to remove, we do not re-add it it. Otherwise, we try to use the shorter variant.

If we can use a shorter variant, we use it and commit the change, as we cannot reduce the offset for this IU in future calls. Otherwise, we use the preferred variant and commit it. In order to not trigger many notifications of the listeners, we only notify the listeners when we are done with the changes. In the source code, the procedure looks like this:

Listing 6.1: Reducing the duration of upcoming chunks

```
public void reduceOffset() {
  ccIUSource.beginChanges();
  ArrayList<IU> ius = ccIUSource.revokeUpcoming();
  for (IU iu : ius) {
    if (!(iu instanceof ChunkIU)) continue;
    Articulatable art = (Articulatable) iu.getUserData("articulatable");
    if (articulatable.isOptional()) {
      boolean canRemove = true;
      int index = articulates.indexOf(art);
      if (index >= 0 && index < articulates.size() - 1) {
        Articulatable next = articulates.get(index + 1);
        canRemove = next.canReplace(art);
      }
      if (canRemove) {
        articulates.remove(art);
        continue;
      } // Otherwise, we try to use the shorter variant, and commit.
    }
    if (art.getShorterText() != null) {
      art.setUseOfShorterText(true);
      ccIUSource.say(art, true, true);
    } else
      ccIUSource.say(art, false, true);
  }
  ccIUSource.doneChanges();
}
```

6.4 Natural Language Generation

The natural language generation (NLG) component is the most important part of the CarChase 2 system. This component is responsible for reading the pattern definitions, check for matching patterns, filling the templates and instructing the articulator. Thus, the NLG component consists of three parts: a parser, a pattern representation, which is also responsible for matching and template filling, and a dispatcher thread. In the following the last two parts will be presented.

6.4.1 Pattern representation

A pattern is represented by the class `Pattern`. It stores the conditions as tuple of two string, and an operator. Here is the implemented structure for a condition:

$$VariableName \begin{Bmatrix} = \\ \neq \\ \leq \\ \geq \end{Bmatrix} \begin{Bmatrix} VariableName \\ Constant \end{Bmatrix}$$

A message includes the two types, the information level and the actual message, and is stored in instances of the class `TTSAction`. The information level is described by a *T(ime)* followed by a number in the range of 1 (low, short message) to 3 (high, more details). For the types, the completeness property is described by an *F(ull)* or an *R(equires)*, the manner is a number. A complete list of types with examples used in the application, with example, is shown in table 5.1.

6.4.2 Variable instantiation

The variable instantiation happens inside the method `instaniateVariables` of the class `Pattern`, which takes a `CarState`, as produced by the viewer, as input. It returns a dictionary (`StringDict`) of variable names and their instantiations. The variables instanced and their meaning is shown in table 5.2

In order to produce syntactic correct sentences, the CarChase 2 system is built to support two inflectional forms. This happens in the following way: At the time of the start up, two default prefixes are loaded from the pattern definition file. However, for some streets both flexion forms are defined explicitly. Otherwise, the default prefix gets prepended to the street name.

6.4.3 Pattern matching

The main method of the class is the `match` method, which returns an Articulatable if the pattern matches and takes a `CarState` and the last Articulatable as input. If the Articulatable is `null`, we assume that there is no previous articulatable, where a continuation can be appended. In the method, all instantiations for the variables get calculated at first. Then, all conditions of the pattern are checked: If the operator of the is $=$ or $\neq$, we evaluate both sides to strings and compare them. Otherwise, we try to parse both sides to integers, and compare them appropiately. However, as the viewer gives a span for the distance to the next point, it is necessary to check whether the given distance is in this range. To implement this in an efficient manner, it is necessary to forbid other expressions than $Distance = Number$. For this number, we can check whether it is in distance the car moved since the last update.

6.4.4 Message selection

If a pattern is matching, we first have to select all type-matching messages. We do this by checking the types for each message. In the code, the procedure looks like this:

Listing 6.2: Finding type-matching messages

```
// ...
for (Message message : matches)
  if (last == null)
    if (!message.typeStart.requiresSentence())
      messages.get(message.ilevel).add(message);
    else;
  else if (last.typeEnd.requiresSentence())
    if (message.typeStart.getManner() == last.typeEnd.getManner() &&
        message.typeStart.requiresSentence())
      messages.get(message.ilevel).add(message);
    else;
  else if (message.typeStart.requiresSentence())
    if (message.typeStart.getManner() == last.typeEnd.getManner())
      messages.get(message.ilevel).add(message);
// ...
```

If there is no previous utterance (i.e. `last == null`), we can only start a new sentence. Otherwise, if the last message requires a continuation, we check whether the manner is matching, as described in section 4.3.1. If the last message has a complete ending, we can still append a continuation, if the manners are equal. The possible messages are stored in a dictionary, where the information levels are mapped to a list of messages.

From the type-matching messages, we finally have to select a preferred and a shorter message which will be articulated. Therefore, we compute an ideal information level according to the speed according to this formula:

$$\text{Ideal Information Level} = \begin{cases} 1 & \text{if } speed \leq 1 \\ speed & \text{otherwise} \end{cases}$$

If we found messages with the ideal information level, we select one random message of these messages. However, it can happen that there are no messages of the ideal information level. If this is the case, we check whether there are messages in the neighbouring information level lists, while preferring lower information levels. The implementation is shown in listing 6.3.

We first compute the ideal information level. If we have messages with this level, we found a set of preferred messages. Otherwise, we iterate through the neighbouring information levels. If we do not find any message in all of the information levels, we cannot apply this pattern. Otherwise, we find the set of messages with the lowest information level.

Once we have these two sets, we construct a mapping from a preferred to a shorter message, where both have the same types. We do this by finding a shorter

Listing 6.3: Finding a preferred and a shorter message

```
// ...
// Compute ideal information level
MessageInformationLevel informationLevel =
    MessageInformationLevel.fromInteger(4 - s.speed);
ArrayList<Message> posPreferred = null, posShorter = null;

// Find messages of the preferred inf. level
if (messages.get(informationLevel).size() > 0)
  posPreferred = messages.get(informationLevel);
else
  for (int d = 1; d < MessageInformationLevel.values().length; d++) {
    MessageInformationLevel lowerLevel =
        MessageInformationLevel.fromInteger(4 - s.speed - d);
    MessageInformationLevel higherLevel =
        MessageInformationLevel.fromInteger(4 - s.speed + d);
    if (messages.get(lowerLevel).size() > 0) {
      posPreferred = messages.get(lowerLevel);
      break;
    } else if (messages.get(higherLevel).size() > 0) {
      posPreferred = messages.get(higherLevel);
      break;
    }
  }
if (posPreferred == null) return null;

// Find messages with lowest inf. level
for (int i = 1; i < MessageInformationLevel.values().length; i++)
  if (messages.get(MessageInformationLevel.fromInteger(i)).size() > 0) {
    posShorter = messages.get(MessageInformationLevel.fromInteger(i));
    break;
  }
if (posShorter == null) posShorter = new ArrayList<Message>();

// now we find a pair of preferred and shorter message with the same types.
HashMap<Message, Message> mapping = new HashMap<Message, Message>();
for (Message pref : posPreferred) {
  boolean put = false;
  for (Message m : posShorter) {
    if (pref.typeStart != m.typeStart || pref.typeEnd != m.typeEnd)
      continue;
    mapping.put(pref, m);
    put = true;
    break;
  }
  if (!put) mapping.put(pref, null);
}
if (mapping.size() == 0) return null;

int chosen = random.nextInt(mapping.size());
Map.Entry<Message, Message> result = (Map.Entry<Message, Message>)
    mapping.entrySet().toArray()[chosen];
```

message of the same types for each preferred message. If we do not find a shorter message, we set the corresponding shorter message to `null`.

Finally, we choose a random mapping and return it wrapped as an Articulatable.

Summary

The presented implementation is an extension of the orginal CarChase system. The incremental speech synthesis is performed by InproTK, which is connected using one articulator. The dynamic shortening or removing of articulatables allows a flexible offset reduction. The dynamic selection of the information level results in less breaks.

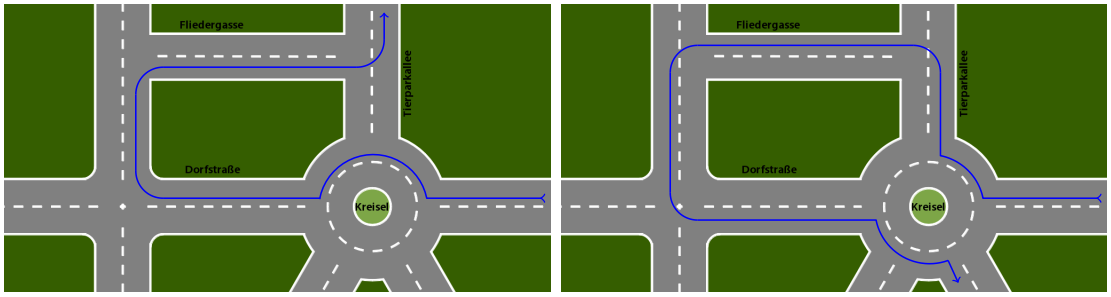
Chapter 7

Evaluation

In this section, we will evaluate the CarChase 2 system with four criteria: the offset between the action and the comment, the correctness of the language generated, the completeness of the description and the elegance of the comment. We performed a survey to compare the impression on the incremental CarChase 2 system in contrast to a baseline system. The baseline system, also used to evaluate the offset and the elegance, only uses full sentences, is non-incremental and does not support a dynamic revoking or shortening and the appending of continuations.

7.1 Offset

One important criterion for a commenting system is the offset between the action of the car and the comment about that action. There are two ways to measure the offset: The first way is the physical offset. That is the offset between the action and the start of the message that describes the action. The second way is the semantic offset: this describes the offset between the action of the car and the part of the message, where the action really gets described. In the following, the relevant part is the name of the new street after a street change, or, if it was mentioned before, the fact that the car actually turned.



(a) Car path 1 with a normal speed **(b)** Car path 2 with a high speed

Figure 7.1: The evaluation paths

We will measure the offset in two situations: in one situation, the car is driving at a normal speed along the path depicted in figure 7.1a. In the other situation,

the car is driving at a high speed along the path shown in figure 7.1b. In each situation, we measure the offset once with the incremental articulator, and once with the standard (baseline) articulator. In the end we will compare the results.

7.1.1 Evaluation using a Slow Speed

In the path depicted in figure 7.1a, the duration of the upcoming part of the current hypothesis is shown in the graph of 7.2 for both articulators. Figure 7.3 shows the points where some text is added to the current hypothesis. The incremental articulator mentions upcoming junctions in contrast to the baseline articulator, which only mentions the upcoming circle at the beginning.

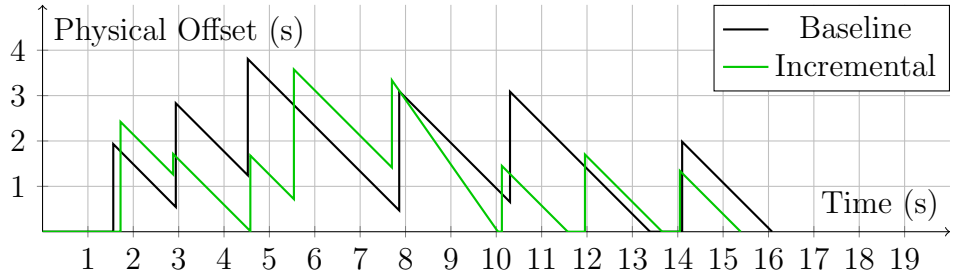


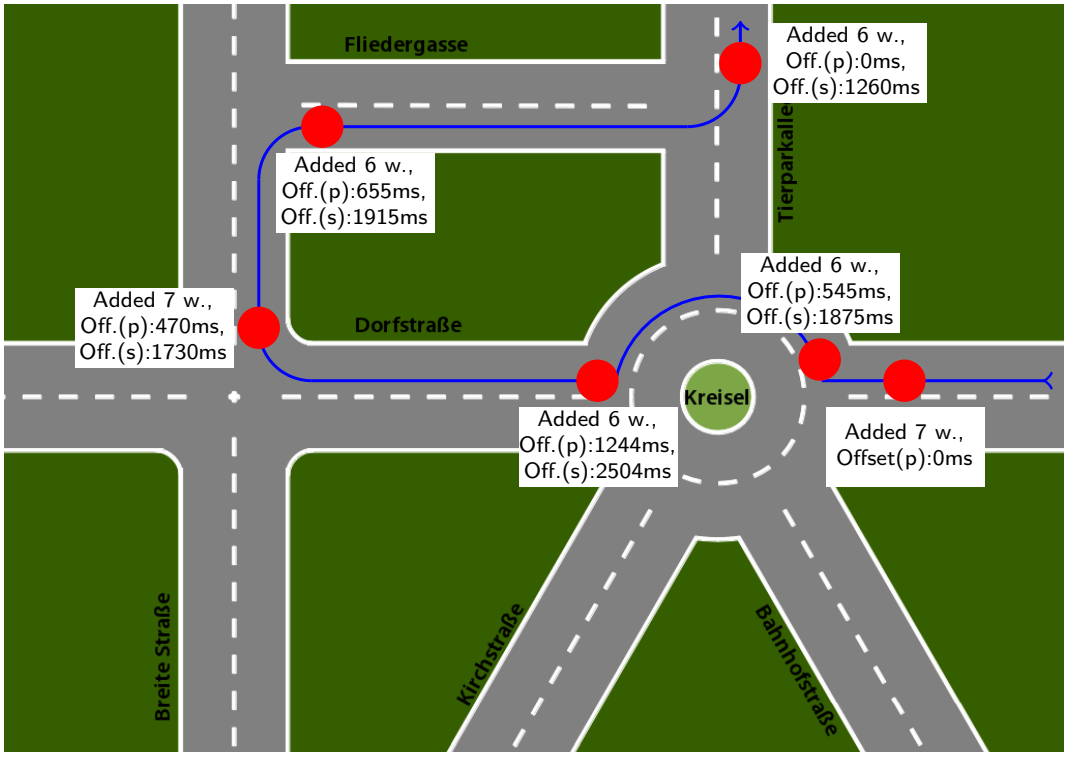
Figure 7.2: Duration of the upcoming and ongoing text in the current hypothesis in the first evaluation path

The main difference between the articulators is that the standard articulator starts a new sentence on every matching pattern. All these sentences have a beginning of the form "The car...". This increases the offset between the messages as well as the semantic offset to the street name. As the frequence of appending messages is higher than the frequence of synthesizing, the offset increases steadily in the first part of the path. The average physical offsets is 485.7ms, the mean of the semantical offsets more than one second larger, with 1856.8ms.

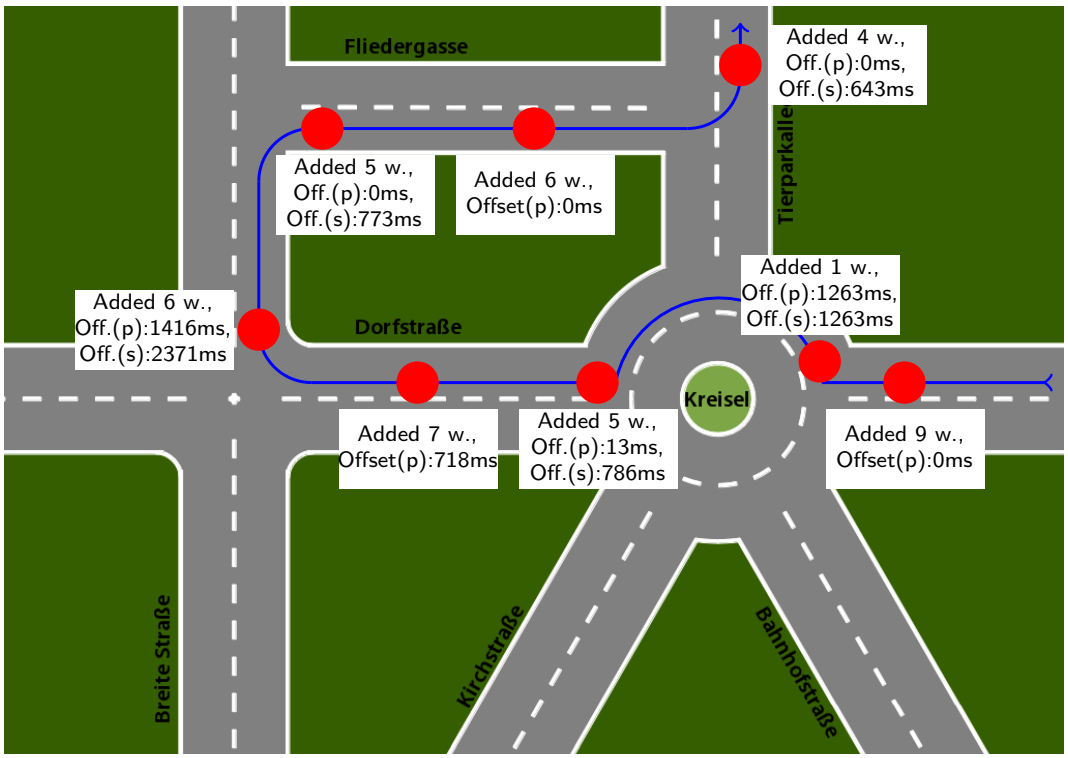
The incremental articulator omits the redundant information about the car by using continuations. In contrast to the baseline, which articulates a noun phrase, the incremental articulator just uses a conjunction to concatenate the verb phrases. As a consequence, the offsets are lower, although there is a notice about the upcoming junction. The average physical offset is not much lower as with the standard articulator, with 426.25ms. The difference in the average semantical offsets is larger, with 1167.2ms using the incremental articulator.

All in all, if the car is driving slowly through the streets, there is nearly no difference in the physical offset between both articulators. A physical offset in the region of 0.5 seconds can be considered as acceptable, compared to an average word duration of 0.38 seconds.[1] However, the incremental articulator yields an average semantic offset that is 0.7 seconds lower than the average semantic offset from the standard articulator. In the next part, we will see how the result of the offset changes, if the car is driving with a higher speed.

[1]This duration is the mean of all different words used, including street names.



(a) Evaluation of the physical and semantical offset with the standard articulator



(b) Evaluation of the physical and semantical offset with the incremental articulator

Figure 7.3: Evaluation of the first car path

7.1.2 Evaluation using a High speed

We will evaluate the offset of the articulators when the car is driving fast using the path shown in figure 7.1b. The duration of the upcoming and ongoing part of the hypothesis is shown in figure 7.4. Figure 7.5 shows the points where some text is added to the hypothesis, including the physical and semantical offsets, for both articulators.

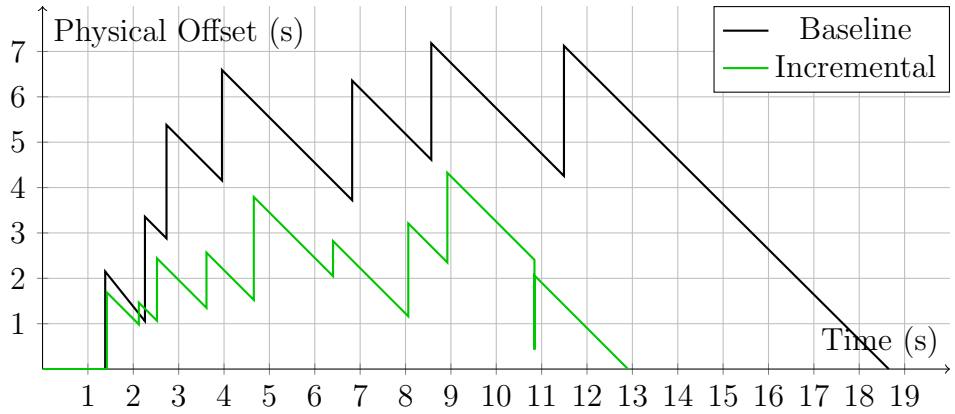
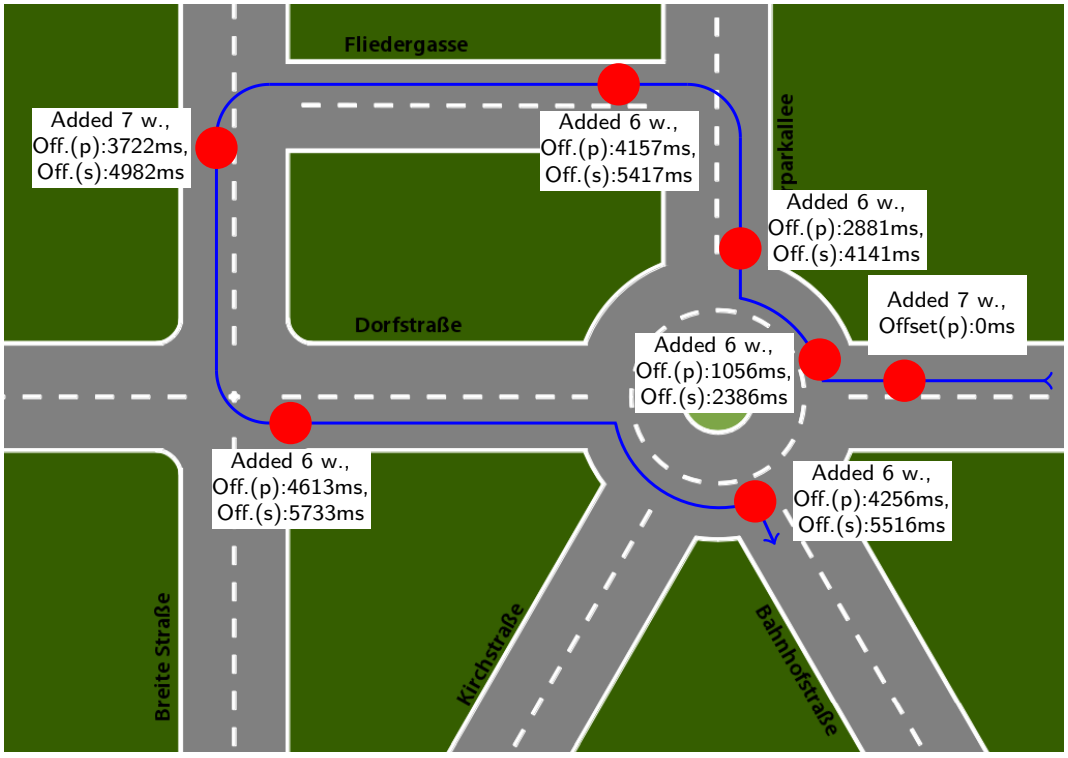


Figure 7.4: Duration of the upcoming and ongoing text in the current hypothesis. Note that after 10.85 seconds an IU gets revoked, and the duration of the hypothesis decreases abruptly.

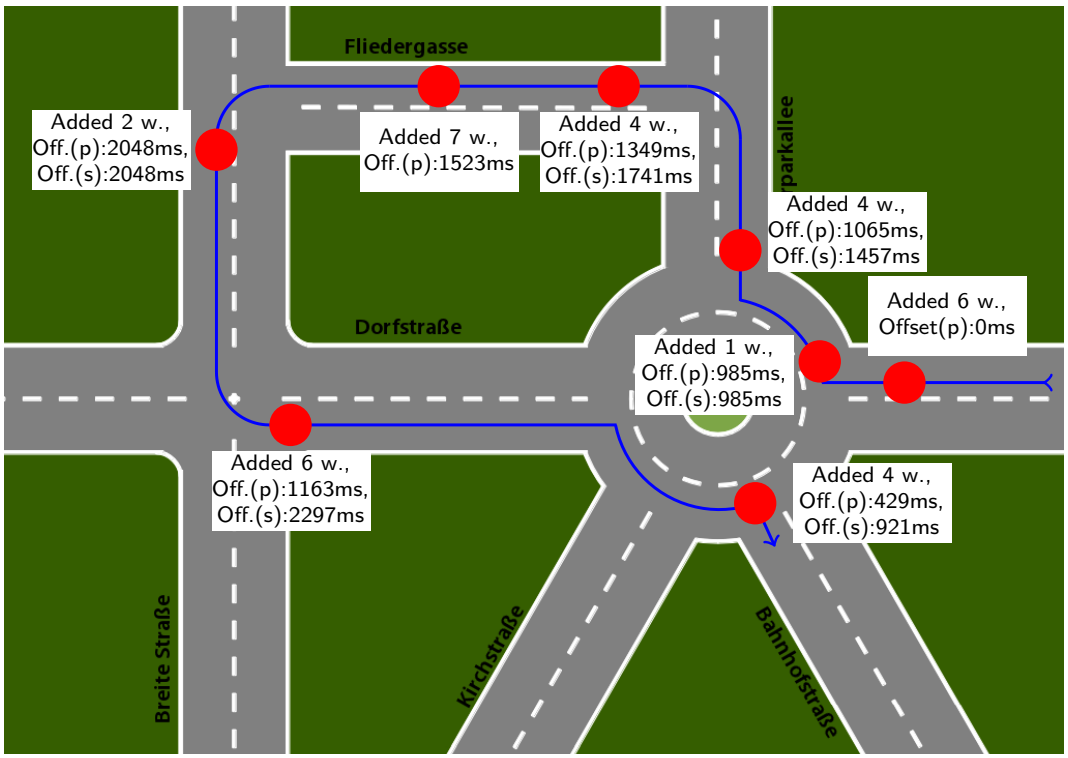
Using the non-incremental articulator, the offset increases steadily at the beginning. This is caused by the long sentences that start with "The car drives...". Even though upcoming junctions are not reported, the offset does only decrease at the end, where no more sentences get spoken. The physical offset is nearly always larger than 3.5 seconds, except for the first three sentences. The average physical offset is at 3.1 seconds. Compared to an average word duration of 0.38 seconds, such an offset is fairly large. The semantical offset is even higher due to the repetition of the sentence beginning: The maximum semantical offset is 5.7 seconds, the average is 4.53 seconds.

The incremental articulator does not have the problem with the high frequency of relevant information at the beginning by using shorter continuations. As described above, the incremental articulator replaces the noun-phrase of a full sentence with a conjunction. Through mentioning the street name of the other street at a junction before the car turns, a lower semantical offset is achieved. Also, the dynamic revoking of an information about the circle reduces the offset at the end. The average physical offset with 1.1 seconds is much lower than in the baseline, and the mean semantical offset is with 1.57 seconds about 2.8 times lower than the average semantical offset of the baseline.

The baseline achieves good results, if the car is driving slowly. However, if the car drives fast, the offset of the baseline increases drastically. As the baseline does not support continuations, but starts a new sentence for every information, the semantical offset is even larger. The incremental articulator reduces the offset



(a) Evaluation of the physical and semantical offset with the standard articulator



(b) Evaluation of the physical and semantical offset with the incremental articulator

Figure 7.5: Evaluation of the second car path

by using continuations and dynamic shortening or removing the chunks. Through continuations, the semantical offset is lower, too.

7.2 Elegance

In order to evaluate the elegance of the formulation, we need to define some criteria first:

- A repetition of the same sentence structure is not elegant, especially when the sentences are short.
- Continuations that avoid redundant information, e.g. the same subject, are preferred.
- The same structure of continuations, e.g. “and ... , and ...”, is not elegant.
- Pause fillers like “ugh” or “ähmm”, or chunks without any relevant information are not elegant, but acceptable if they reduce the redundant information given or a repetition of the sentence structure.

To perform this evaluation, we use the text generated for the paths shown in figure 7.1. The text produced for the first path by both articulators are the following:

> **Baseline:** Das Auto fährt auf den Kreisel zu. Das Auto fährt in den Kreisel. Das Auto fährt in die Dorfstraße. Das Auto fährt in die Breite Straße. Das Auto fährt in die Fliedergasse. Das Auto fährt in die Tierparkallee.
>
> **Incremental:** Das Auto fährt auf den Kreisel zu und biegt hinein und fährt in die Dorfstraße und erreicht die Kreuzung und biegt rechts in die Breite Straße und in die Fliedergasse und erreicht die Kreuzung und biegt links in die Tierparkallee.

The baseline starts a new sentence for each information, and these sentences are short and always have the same sentence structure. As continuations are not supported by the baseline, the redundant information about the car is given to produce syntactically correct sentences. Pause fillers are not supported, and would not be adequate either: The baseline variant always starts a new sentence.

The incremental articulator supports continuations, and uses them extensively. The structure of the sentence is more elegant than in the baseline, although the continuations have the same type, namely “and ...”. The incremental variant does not use hesitations, but uses other types of pause fillers, as “and reaches the junction”. However, these not just reduce the offset, but also avoid the beginning of a new sentence.

The texts uttered for the second path, where the car was driving faster, by both articulators are the following:

> **Baseline:** Das Auto fährt auf den Kreisel zu. Das Auto fährt in den Kreisel. Das Auto fährt in die Tierparkallee. Das Auto fährt in die Fliedergasse. Das Auto fährt in die Breite Straße. Das Auto fährt in die Dorfstraße. Das Auto fährt in die Bahnhofstraße.
>
> **Incremental:** Es erreicht den Kreisel und biegt hinein und in die Tierparkallee und in die Fliedergasse und erreicht die Breite Straße und biegt links hinein und biegt links in die Dorfstraße und in die Bahnhofstraße.

The baseline also starts a new sentence for each street change. As for the other path, the sentences always have the same structure and length, and give redundant information about the driving car.

The incremental articulator puts all information in one sentence. The sentence structure does not repeat, though the structure of the continuations does. There is no redundant information articulated. Also, there are no pause fillers.

The sentences produced by using the incremental articulator are more elegant, as they use continuations to avoid short sentences of the same structure. Furthermore, the continuations have different formulations, in contrast to the baseline.

7.3 Correctness

Another evaluation criteria for the system is whether a valid language is produced, that means a syntactical correctness of the generated sentences. If the sentences are not syntactically correct, it should be at least possible to follow the car path by just listening to the comment.

Using the default configuration, valid German is produced in the most cases. However, if the car is driving slowly and there is a break of one second, the system will start a new sentence, even if the previous message requires a continuation. This behavior has two reasons: First, there is a lack of type combinations in the different patterns. This problem can be easily solved by adding more messages for the different combinations and by adding more types for different continuation possibilities.

Second, there is no break detection. That means that if a sentence gets started, but the car is not driving as fast as we implicitly assume by choosing the information level, there is a break between the end of the first utterance and the next action of the car. When the next message gets triggered after the next car action and there is no ongoing or upcoming IU, the system will start a new sentence. However, this only happens if the break is larger than 0.5 seconds, as the last IU is still stored in the right buffer of the IU source for a short time after the IU is finished.

This problem can be solved by storing the last IU seperately, and using the absolute end time of that IU to detect small breaks. Another way to solve this problem is to append a `HesitationIU` after each incomplete message. However,

this is not implemented due to some bugs in InproTK at the time of development. If the bug in InproTK was fixed, hesitations would allow an appending of continuations after a break of up to 1.2 seconds.

7.4 Completeness

To evaluate the completeness of the description, we first have to describe the criteria. We differentiate this in two sections: necessary information, which has to be given in every case, and optional information, which is nice to have. The necessary information to follow the car is the street name at the beginning and after a street change. If the street name is not mentioned, except for short transitioning streets as circles, it is impossible to say where the car is, without knowing the map.

Optional information can be referred as the following:

- Whether the car turns left or right
- The speed and stops of the car
- That the car is reaching a decision point, e.g. a junction
- Indicate whether the car drives the same way many times, e.g. in a circle, or that the car stopped for a longer time
- Important landmarks and other information that is not related to the path of the car, if there is no other information about the car path

The CarChase 2 system mentions the street name always after or right before a street change. This is ensured by a non-optional pattern that permanently checks for a change of the street name. The only exception is the beginning, that street name will not be mentioned at any time.

From the optional informations, some are mentioned: At a junction, it is mentioned whether the car drives straight-forward, or turns left or right. Also, if the car is near a junction, it is mentioned, and at a higher information level, the street of the junction is uttered, too. At a high information level, a relative speed (e.g. slowly) is said if the car reaches a junction. This is not the case if the car is driving faster, in order to save time. To mention speed changes directly is possible and implemented. However, as the information is not that important, it increases the offset between car action and the next information near a junction, although marked as optional. The only speed change that is mentioned is a stop and a continuation.

As there is no memory component of the car path yet, it is impossible to mark a repetition of the same way. Also, mentioning important landmarks, as the VAVE-TaM system does, as well as other non-car-path information is not implemented due to a lack of a semantic memory component.

7.5 Survey

Beside the evaluation with the criteria described above, it is also important to see how users see the incremental system using continuations in comparison to the non-incremental baseline. To evaluate this, we performed an online survey, similar to the experiment conducted by Baumann and Schlangen (2013).

7.5.1 Experiment

We used three different scenarios, in each scenario the car was driving along another path with a variable, but predefined, speed. Both articulator strategies were used to generate the videos, resulting in six videos. Note that the development at the time of the survey was not finished and two minor issues are included in the videos from the incremental strategy: the name of the street the car turns in was not always mentioned and there was only a simple revoking procedure without shortining of already articulated text parts implemented.

In the first scenario, the car is driving slowly at the beginning and is increasing the speed after turning right into the "Breite Straße". This scenario should represent a usual behavior. Using the incremental strategy, all information is put into three sentences using continuations. At the end, the name of the street the car turns into is not mentioned. The baseline strategy performs the commenting in six sentences. One issue with the baseline articulator is that it states "The car reaches the junction" when it actually reaches the circle, which is no junction.

In the second scenario, the car drives fast through the streets. This extreme case shall find out the impact of the offset in both articulation strategies. The incremental articulator uses continuations all the time, except at the beginning, and has an offset of approximatly 2.5 seconds in the worst case. The baseline leaves out any information about the circle, resulting in a maximum offset of approximatly 4 seconds.

In the third scenario the car is driving slowly. At the end the incremental articulator starts commenting about the car reaching the junction with the "Tierparkallee". As this open-ended messages ends, the car did not turned yet and therefore, an hesitation is inserted at the end. This allows the use of a continuation at the end. The baseline strategy does not use hesitations, there is a break after the comment that the car reaches the junction.

Path	Car Speed	Intention
1	Increasing	Continuations vs. Short complete sentences
2	Fast	Offset: 2.5 seconds vs 4 seconds
3	Slow	Hesitation and open-ended utterance vs. pause and complete sentences

Table 7.1: Car Speed and Intention for each path

For each participant, from these six videos were four chosen randomly and

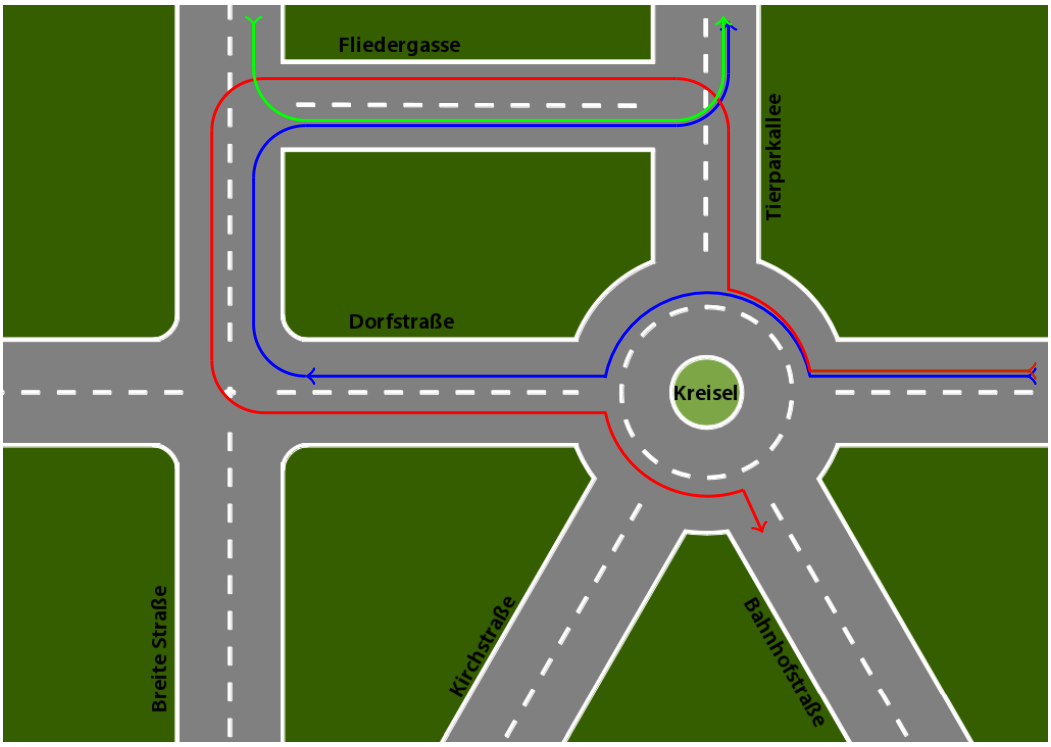


Figure 7.6: The paths from the survey. The first path is blue, the second path is red and the third path is green.

played to the participator in a random order. This is necessary, as the participants might communicate. In total 22 people, who are not involved into the development, participated. They were told that the videos were generated by many different systems and asked, after the video was finished, to rate the impression of the comment on a four-point Likert-scale.

We assumed that the incremental system is preferred, because the offset between the car action and the comment is lower and the produced sentences are more elegant.

7.5.2 Results

For a statistical evaluation, the four choices get translated in a numeric value first. [2] This is done by defining "positive" as 1.5 and the other values as decrements of 1, so that "negative" has a value of -1.5. As the ordering of the videos is defined randomly and different for each participant, and as no introduction video was shown, the values got normalized first. This is performed by shifting the values in that way that the first given answer is 0. The result will be still correct, as the first video was chosen randomly. The resulting in a value range is the range from -3 to 3.

The overall and path-specific mean ratings are shown in figure 7.7. In the first

[2] The raw and the normalized data can be found in appendix C

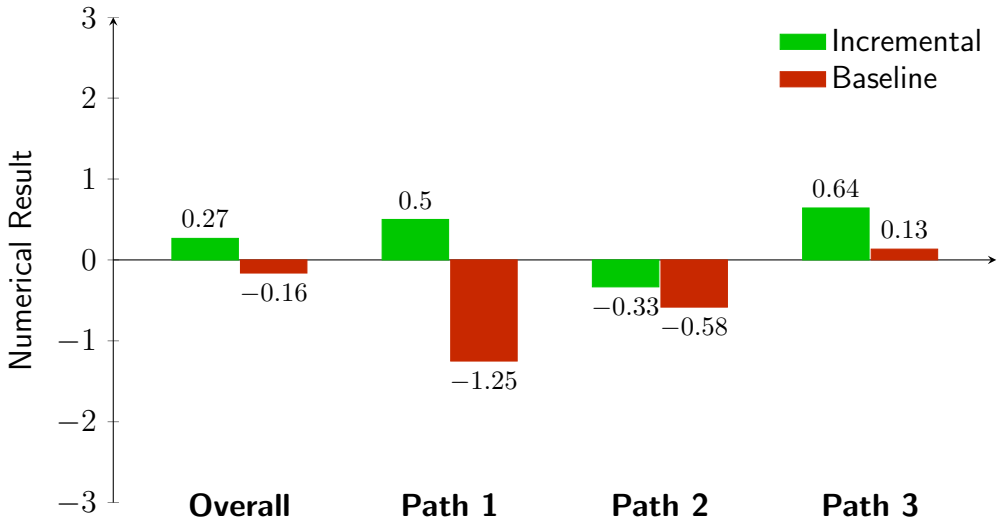


Figure 7.7: Normalized mean ratings of both articulators for the three paths

path, the incremental version is rated significantly better than the baseline (t-test, $p < .00001$). This indicates that continuations are preferred over short sentences.

The incremental version commenting on the second path was rated worse than the incremental version in path 1 (t-test,$p < .02$), and is not significantly better than the baseline (t-test,$p < .37$). This is a clear indicator that a high offset is not tolerated, independent of the elegance, and should be avoided.

For the third path, the incremental version is rated better than the baseline, the difference is not significant, though (t-test,$p < .8$). However, this indicates that hesitations in cooperation with a lower semantical offset are tolerated over short sentences and breaks.

The overall result shows that the incremental strategy is preferred, though not significantly (t-test,$p < .026$). However, as the offset reduction was not implemented at the time of the survey, the results for path 2 may differ from the results the current version would achieve. Excluding path 2, the results from the incremental strategy are significantly better (t-test,$p < .0118$).

7.5.3 Consequences

The results of the survey show that the continuations of the incremental strategy are preferred over the short sentences of the baseline. However, the preference for a low offset is stronger: if the offset is considerably long (2.5 seconds), the elegance of the produced sentences has only a minor priority. Hesitations are tolerated, if short sentences and pauses are avoided.

7.6 Discussion

The comment produced by the incremental articulator is a complete description of a car path, although speed changes and repetitions of the same sub-path are not reported. The produced text is in the most cases syntactically correct. However, if the car drives slowly, the produced text is incomplete, due to a missing break detection. The sentences produced by the incremental articulator are more elegant, and are preferred by the users. In contrast to the baseline strategy, the incremental articulator uses continuations instead of short sentences. The survey indicates that the incremental variant is preferred for a lower semantical offset, as a result of the continuations. The survey indicates that the stronger preference to the elegance is the offset between the car action and the comment.

The advantage of the implemented system is that more types can be added easily in order to allow more variants of formulations. This might increase the elegance of the produced text. However, realizing complex sentences is only possible with a big effort, as the type system itself is a flat structure. In this context, complex sentences might be not have a huge advantage, though. Furthermore, the users already accepts the simple continuations. A future survey can elaborate this further.

Although the survey indicates a clear direction towards the incremental strategy, it is necessary to ask a broader audience outside of the university to get more comprehensive results. Also, the users were not able to control the car by themselves. In a future survey the difference between the incremental and the baseline strategy when the user controls the car can be distinguished.

Chapter 8

Summary and Outlook

An interactive version of the CarChase application needs a natural language component for a dynamic commenting on the car path and speed. The implemented natural language generation component uses templates with different information levels to allow a dynamic selection of the duration of the appended text, depending on the speed of the car. By adding types to the beginning and ending of a template, a flexible use of continuations is possible. This improves the elegance of the produced sentences in contrast to a non-incremental baseline system. It is possible to gain more flexibility by adding more types, which can be done easily. As a result of allowing open-ended messages, the time between the action of the car and the articulation of that action can be reduced. Using incremental speech synthesis based on InproTK, the presented CarChase 2 implementation reacts dynamically on the actions of the user. It performs a dynamic and flexible elaborating, shortening and removing of the text parts, which also reduces the offset between action and comment. The changing of the sentences does not influence the completeness of the description, as only messages that are marked as optional can be removed. Also, the CarChase 2 system produces syntactically correct sentences in most cases.

The results of a conducted survey show that the continuation of the incremental strategy are preferred over short sentences of a non-incremental baseline system. A high offset between action and comment is not tolerated. Pause-fillers are tolerated, if pauses can be avoided.

Outlook

The current implementation shows that the flexible generation and altering of the text is accepted by the users. In the following, some points to improve the system will be presented shortly.

To improve the variation in the sentence structure, **more types and variations** can be added. The current implementation will choose one variation randomly. Furthermore, the system should be tested with a **larger map** to see how the system behaves in other situations. Also, a **survey** to evaluate the reaction on an interactive control of the user should be conducted.

The completeness of the description can be improved by mentioning **speed changes**. Furthermore, a **memory component** about the car path allows identifing a repetition of the same car path, and acting appropriately. The syntactical correctness can be improved by using an explicit **break detection**.

It is possible to **predict actions** of the user, with the result of a lower offset. If the prediction was wrong, it would be possible to revoke the non-spoken part and insert an indicator word, like “no”, and to append the correct description.

Another way to reduce the offset is to change the **speed of speaking** dynamically, according to the speed of the car. This is an extension of the incremental speech synthesis component and helps to avoid pause fillers and a reduction of the offset.

Appendix A

Example output of the application

Here is an example for the dynamic output of the application for the path depicted in figure 7.1.

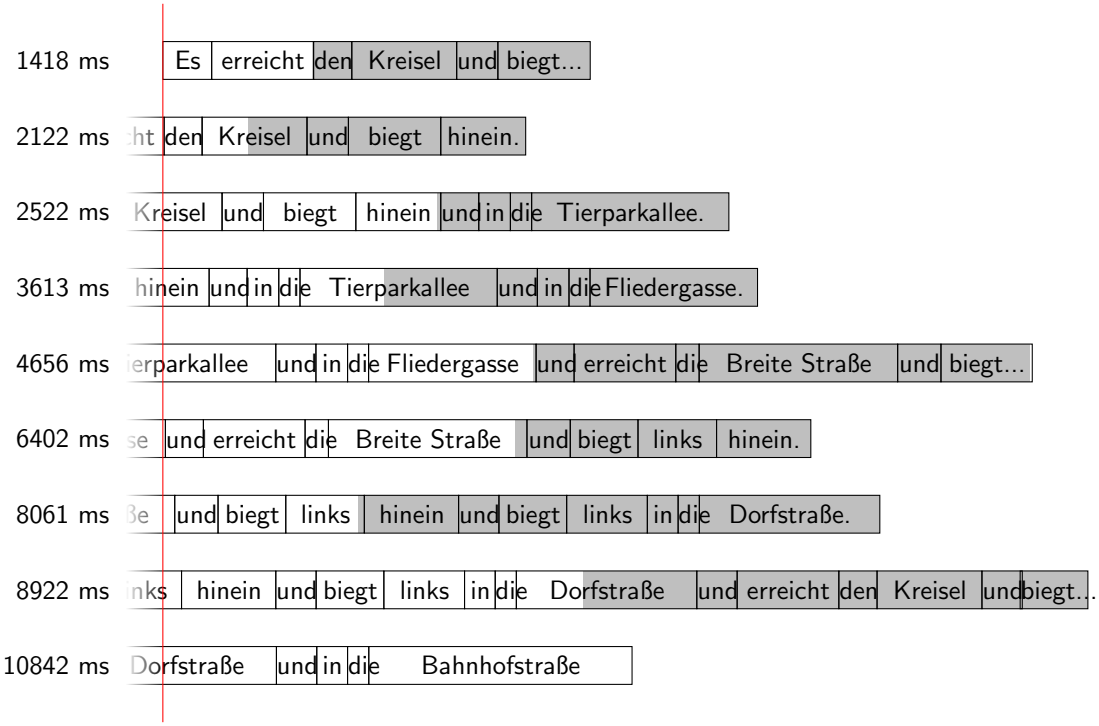


Figure A.1: Example for the dynamic development of a hypothesis. The red line denotes the current progress of synthesizing.

Appendix B

Configurations used for evaluation

Here are the configuration file for the evaluation paths. The point `E-R` denotes the right exit of the "Dorfstraße", the internal streetname `DorfstrasseR` represents the part of the "Dorfstraße" which is right to the circle. The speed command takes two arguments, the time in milliseconds after start and the new speed. See section 7.1 and figure 7.1 for a detailed description and visualisation of the paths.

Listing B.1: Configuration for the first evaluation path

```
# CarChase 2 Configuration file
#
Start: E-R, DorfstrasseR, -, 0
--- direction, 0
--- direction, 1
--- direction, 2
--- direction, 1
--- direction, 1
```

Listing B.2: Configuration for the second evaluation path

```
# CarChase 2 Configuration file
#
Start: E-R, DorfstrasseR, -, 0
--- speed, 0, 3
--- direction, 1
--- direction, 0
--- direction, 0
--- direction, 2
--- direction, 1
--- direction, 0
```

Appendix C

Survey results

Path 1		Path 2		Path 3	
Base.	**Incr.**	**Base.**	**Incr.**	**Base.**	**Incr.**
(0) –	(2) +			(3) –	(1) –
	(3) –	(0) – –	(1) – –		(2) +
(1) ++		(0) +		(2) ++	(3) ++
(0) +	(1) ++	(3) –			(2) +
(1) –		(2) – –	(0) –		(3) –
(3) – –	(0) –		(2) –	(1) – –	
(0) –	(3) ++			(1) –	(2) +
	(2) –		(0) +	(1) –	(3) ++
(2) – –		(1) –	(3) –	(0) +	
(0) –	(2) ++		(1) +	(3) +	
(2) –	(3) ++		(0) –		(1) +
(3) –	(2) +	(1) – –	(0) –		
(0) –		(3) –	(1) – –	(2) +	
	(1) – –		(3) +	(2) – –	(0) –
(2) +	(0) –			(1) +	(3) +
	(1) – –		(2) – –	(3) +	(0) –
(3) +		(1) – –		(2) +	(0) –
(1) +	(2) ++		(3) –	(0) ++	
(2) +		(1) +	(3) –	(0) +	
	(1) +	(0) –	(2) – –		(3) ++
	(2) –	(3) – –	(0) –	(1) – –	
(1) +	(0) ++	(3) +			(2) ++

Table C.1: Raw input data from the 22 participants. The number in parenthesis denotes the index of the video, the video with number 0 was shown first.

Path 1		Path 2		Path 3	
Base.	**Incr.**	**Base.**	**Incr.**	**Base.**	**Incr.**
0	1			0	0
	1	0	0		2
1		0		1	1
0	1	-1			0
0		-1	0		0
-1	0		0	-1	
0	2			0	1
	-1		0	-1	1
-2		-1	-1	0	
0	2		1	1	
0	2		0		1
0	1	-1	0		
0		0	-1	1	
	-1		1	-1	0
1	0			1	1
	-1		-1	1	0
1		-1		1	0
-1	0		-2	0	
0		0	-1	0	
	1	0	-1		2
	0	-1	0	-1	
-1	0	-1			0
Mean					
-1.25	0.5	-0.583	-0.333	0.133	0.643
Variance					
0.609	1	0.243	0.622	0.649	0.515

Table C.2: Normalized input data from the 22 participants, including the mean and the variance. See section 7.5.2 for details about the procedure.

Appendix D

Source code

The full source code and configuration files is available on GitHub: `https://github.com/aengelke/carchase2`

Listing D.1: The CarChase 2 Natural Language Generation Component

```
package inprotk.carchase2;

import inpro.apps.SimpleMonitor;
import inpro.audio.DispatchStream;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;
import java.util.Random;
import java.util.Vector;

import processing.data.StringDict;
import inprotk.carchase2.Articulator;
import inprotk.carchase2.CarChase;
import inprotk.carchase2.Articulator.Articulatable;
import inprotk.carchase2.Configuration.CarState;
import inprotk.carchase2.IncrementalArticulator;
import inprotk.carchase2.World.Street;
import inprotk.carchase2.World.WorldPoint;

public class CarChaseTTS {
  private ArrayList<Pattern> patterns;
  private HashMap<String, StreetReplacement> streetNames;
  private Articulator articulator;
  private DispatchStream dispatcher;
  private DispatcherThread dispatchThread;

  private String flexForm1, flexForm2, left, right;

  public CarChaseTTS(String patternsFilename) {
```

```
  try {
    dispatcher = SimpleMonitor.setupDispatcher();
    if (CarChase.getSuperConfig("baseline").equals("true"))
      articulator = new StandardArticulator(dispatcher);
    else
      articulator = new IncrementalArticulator(dispatcher);
    parsePatterns(patternsFilename);
    dispatchThread = new DispatcherThread();
    dispatchThread.start();
  } catch (Exception e) {
    e.printStackTrace();
  }
}

public MyCurrentHypothesisViewer getHypothesisViewer() {
  return articulator.getHypothesisViewer();
}

private void parsePatterns(String filename) throws Exception {
  patterns = new ArrayList<Pattern>();
  streetNames = new HashMap<String, StreetReplacement>();
  Pattern currentPattern = null;
  String[] lines = CarChase.readLines(filename);
  int index = 0;
  for (String line : lines) {
    if (index++ == 0) continue;
    line = line.trim();
    if (line.startsWith("#")) continue;
    if (line.startsWith("--msg")) {
      String[] args = line.substring(6).split("=");
      String[] meta = args[0].split("#");
      MessageInformationLevel type = MessageInformationLevel.valueOf(
↪ meta[0]);
      MessageType typeStart = MessageType.valueOf(meta[1]);
      MessageType typeEnd = MessageType.valueOf(meta[2]);
      String key = meta[3];
      currentPattern.addTemplate(key, args[1], typeStart, typeEnd,
↪ type);
    }
    else if (line.startsWith("--cond")) {
      currentPattern.addCondition(line.substring(7));
    }
    else if (line.startsWith("++")) {
      patterns.add(currentPattern);
      currentPattern = null;
    }
    else if (line.startsWith("pt")) {
```

```
      String[] args = line.substring(3).split(",");
      for (int i = 0; i < args.length; i++)
        while (args[i].startsWith("␣")) args[i] = args[i].substring
↪ (1);
      boolean optional = args.length > 1 && args[args.length - 1].
↪ equals("y");
      currentPattern = new Pattern(optional);
    }
    else if (line.startsWith("street")) {
      String[] args = line.substring(7).split(",");
      for (int i = 0; i < args.length; i++)
        while (args[i].startsWith("␣")) args[i] = args[i].substring
↪ (1);
      String key = args[0];
      String name = args[1];
      if (name.equals("%")) name = key;
      String flex1 = args[2];
      String flex2 = args[3];
      flex1 = flex1.replace("%", key);
      flex2 = flex2.replace("%", key);
      streetNames.put(key, new StreetReplacement(name, flex1, flex2))
↪ ;
    }
    else if (line.startsWith("flex")) {
      if (line.startsWith("flex1")) flexForm1 = line.substring(6);
      if (line.startsWith("flex2")) flexForm2 = line.substring(6);
    }
    else if (line.startsWith("leftright")) {
      String[] values = line.split("␣");
      left = values[1];
      right = values[2];
      CarChase.log("LR", left, right);
    }
    else if (line.equals("")) continue;
    else throw new RuntimeException("Illegal␣line:␣" + line + "␣in␣
↪ file␣" + filename);
  }
}

public void matchAndTrigger(CarState state) {
  dispatchThread.addDispatchTask(state);
}

private class DispatcherThread extends Thread {
  private Vector<CarState> actions;
  public DispatcherThread() {
    super("CarChaseTTS␣Dispatcher");
```

```
    actions = new Vector<CarState>();
  }
  public void run() {
    while (true) {
      Vector<CarState> toDispatch = new Vector<CarState>();
      synchronized (this) {
        toDispatch.addAll(actions);
        actions.clear();
      }

      for (CarState action : toDispatch) {
        dispatch(action);
      }

      synchronized(this) {
        try {
          wait();
        } catch (InterruptedException e) {
        }
      }
    }
  }
  private void dispatch(CarState state) {

    CarChaseArticulatable startAction = null;
    CarChaseArticulatable continuationAction = null;
    CarChaseArticulatable lastIU = (CarChaseArticulatable)
↪   articulator.getLast();
    boolean continuationPossible = articulator.isSpeaking();

    ArrayList<Pattern> matchings = new ArrayList<Pattern>();
    for (Pattern p : patterns) {
      if (p.doesMatch(state) || p.doesMatch(state.alternative))
        matchings.add(p);
    }

    if (matchings.size() == 0) return;

    // Important to do this here, otherwise
    // we can't be sure with the types.
    articulator.reduceOffset();

    for (Pattern p : matchings) {
      if (startAction == null)
        startAction = p.findMatch(state, null);
      if (continuationAction == null && continuationPossible)
        continuationAction = p.findMatch(state, lastIU);
```

```
      if (startAction != null && (!continuationPossible ||
  ↪ continuationAction != null)) break;
    }
    if (startAction == null)
      return;

    CarChaseArticulatable finalAction = startAction;
    if (continuationPossible && continuationAction != null)
      finalAction = continuationAction;

    articulator.say(finalAction);
  }
  private void addDispatchTask(CarState state) {
    synchronized (this) {
      actions.add(state);
      notify();
    }
  }
}

public static class Message {
  public MessageType typeStart, typeEnd;
  public MessageInformationLevel ilevel;
  public String text;
  public boolean optional;

  private Message(MessageType typeStart, MessageType typeEnd,
  ↪ MessageInformationLevel type, String text, boolean optional) {
    this.typeStart = typeStart;
    this.typeEnd = typeEnd;
    this.ilevel = type;
    this.text = text;
    this.optional = optional;
  }

  public String toString() {
    return "[TTSAction␣text=" + text + ",level=" + ilevel.toString()
  ↪ + "]";
  }
}

public static enum MessageInformationLevel {
  T1, // Information Level 1, high speed
  T2, // Information Level 2, normal speed
  T3; // Information Level 3, low speed
  public static MessageInformationLevel fromInteger(int level) {
    int intInformationLevel = Math.min(3, Math.max(1, level));
```

```
    return MessageInformationLevel.valueOf("T" + intInformationLevel)
  ↪ ;
  }
}


// Example: Das Auto faehrt in den Kreisel . (Begin: S2, End: S1)
// Example: und faehrt in den Kreisel . (Begin: R1, End: S1)
// Example: Das Auto faehrt auf die Kreuzung zu und (Begin: S1, End: R2)
public static enum MessageType {
  F1(false),
  F2(false),
  F3(false),
  R1(true),
  R2(true),
  R3(true),
  R4(true),
  R5(true),
  R6(true);

  // Moeglich ist : [ F1 F1 ] [ R1 F2 ] [ R2 F1 ] [ F1 F1 ]
  // R benoetigt einen Satz davor, der gerade gesprochen wird;
  // F geht immer, wenn der vorige Satz keinen nachfolgenden braucht.

  private boolean requiresSentence;
  private int manner;
  private MessageType(boolean requiresSentence) {
    this.requiresSentence = requiresSentence;
    this.manner = Integer.parseInt(toString().substring(1));
  }

  /**
   * Is a sentence of the same type required at the beginning / ending? If
  ↪ no, it is a valid beginning / ending of a sentence.
   * @return
   */
  public boolean requiresSentence() {
    return requiresSentence;
  }

  public int getManner() {
    return manner;
  }
}

public class Pattern {
  protected boolean optional;
```

```
public HashMap<String, Message> templates;
public ArrayList<Condition> conditions;
public Pattern(boolean optional) {
  templates = new HashMap<String, Message>();
  conditions = new ArrayList<Condition>();
  this.optional = optional;
}

// Requires: XY#OP#XY
public void addCondition(String conditionLine) {
  String[] condParts = conditionLine.split("#");
  conditions.add(new Condition(condParts[0], condParts[2],
↪ condParts[1]));
}

public void addTemplate(String key, String value, MessageType
↪ sortStart, MessageType sortEnd, MessageInformationLevel type) {
  if (templates.containsKey(key)) CarChase.log("WARNING:␣Duplicate␣
↪ key", key);
  templates.put(key, new Message(sortStart, sortEnd, type, value,
↪ optional));
}

private StringDict instantiateVariables(CarState s) {
  World w = CarChase.get().world();
  Street currentStreet = w.streets.get(s.streetName);
  Street prevStreet = w.streets.get(s.prevStreetName);
  WorldPoint nextPoint = w.points.get(s.nextPointName);
  WorldPoint prevPoint = w.points.get(s.prevPointName);
  StringDict replace = new StringDict();
  replace.set("*INTSTREET", s.streetName);
  replace.set("*INTPREVSTREET", s.prevStreetName);
  StreetReplacement streetRpl = streetNames.get(s.streetName);
  if (streetRpl == null) streetRpl = new StreetReplacement(s.
↪ streetName);
  StreetReplacement prevStreetRpl = streetNames.get(s.
↪ prevStreetName);
  if (prevStreetRpl == null) prevStreetRpl = new StreetReplacement(
↪ s.prevStreetName);
  replace.set("*STREET", streetRpl.name);
  replace.set("*PREVSTREET", prevStreetRpl.name);
  replace.set("*FLEX1STREET", streetRpl.flex1);
  replace.set("*FLEX1PREVSTREET", prevStreetRpl.flex1);
  replace.set("*FLEX2STREET", streetRpl.flex2);
  replace.set("*FLEX2PREVSTREET", prevStreetRpl.flex2);
  replace.set("*DIRECTION", s.direction + "");
  replace.set("*PREVDIRECTION", s.prevDirection + "");
```

```
    replace.set("*POINTNAME", s.nextPointName);
    replace.set("*PREVPOINTNAME", s.prevPointName);
    replace.set("*SPEED", "" + s.speed);
    replace.set("*PREVSPEED", "" + s.prevSpeed);
    replace.set("*BIDIRECTIONAL", "" + (currentStreet.bidirectional ?
↪  1 : 0));
    replace.set("*NUMSTREETS", "" + nextPoint.streets.size());
    replace.set("*LEFTRIGHT", "" + (s.lr == 1 ? left : right));
    // Junctions
    applyJunction(nextPoint, currentStreet, replace, s.direction,
↪ nextPoint.streets, false);
    applyJunction(prevPoint, prevStreet, replace, s.prevDirection,
↪ prevPoint.streets, true);

    return replace;
  }

  public boolean doesMatch(CarState s) {
    StringDict replace = instantiateVariables(s);
    for (Condition cond : conditions) {
      String instancedLeftSide = instanciate(cond.leftSide, replace);
      String instancedRightSide = instanciate(cond.rightSide, replace
↪ );
      if (cond.isDistance) {
        int distance = Integer.parseInt(instancedRightSide);
        if (cond.operator == '=') {
          if (s.previousDistance > s.currentDistance) {
            if (distance >= s.previousDistance || distance < s.
↪ currentDistance) return false;
          }
          else if (distance < s.previousDistance || distance >= s.
↪ currentDistance) return false;
        } else if (cond.operator == '>') {
          if (s.currentDistance <= distance) return false;
        }
      } else {
        if (cond.operator == '=' && !instancedLeftSide.equals(
↪ instancedRightSide)) return false;
        else if (cond.operator == '!' && instancedLeftSide.equals(
↪ instancedRightSide)) return false;
        else if (cond.operator == '<' || cond.operator == '>') {
          int left = Integer.parseInt(instancedLeftSide);
          int right = Integer.parseInt(instancedRightSide);
          if (cond.operator == '<' && left >= right) return false;
          if (cond.operator == '>' && left <= right) return false;
        }
      }
```

```
    }
    return true;
  }

  public CarChaseArticulatable findMatch(CarState s,
↪ CarChaseArticulatable lastArticulatable) {
    Message last = lastArticulatable != null ? lastArticulatable.
↪ preferred : null;
    StringDict replace = instantiateVariables(s);

    ArrayList<Message> instancedMessages = new ArrayList<Message>();
    for (Map.Entry<String, Message> entry : templates.entrySet())
      instancedMessages.add(new Message(entry.getValue().typeStart,
↪ entry.getValue().typeEnd, entry.getValue().ilevel, instanciate(
↪ entry.getValue().text, replace), entry.getValue().optional));
    Message[] matches = instancedMessages.toArray(new Message[0]);
    if (matches == null) return null;
    HashMap<MessageInformationLevel,ArrayList<Message>> messages =
↪ new HashMap<MessageInformationLevel,ArrayList<Message>>();
    for (MessageInformationLevel level : MessageInformationLevel.
↪ values())
      messages.put(level, new ArrayList<Message>());
    for (Message message : matches)
      if (last == null)
        if (!message.typeStart.requiresSentence())
          messages.get(message.ilevel).add(message);
        else;
      else if (last.typeEnd.requiresSentence())
        if (message.typeStart.getManner() == last.typeEnd.getManner()
↪  && message.typeStart.requiresSentence())
          messages.get(message.ilevel).add(message);
        else;
      else if (message.typeStart.requiresSentence())
        if (message.typeStart.getManner() == last.typeEnd.getManner()
↪ )
          messages.get(message.ilevel).add(message);

    Random random = new Random();

    // Compute ideal information level
    MessageInformationLevel informationLevel =
↪ MessageInformationLevel.fromInteger(4 - s.speed);
    ArrayList<Message> posPreferred = null, posShorter = null;
    if (messages.get(informationLevel).size() > 0) {
      posPreferred = messages.get(informationLevel);
    } else {
      for (int distance = 1; distance < MessageInformationLevel.
```

```
↪ values().length; distance++) {
      MessageInformationLevel lowerLevel = MessageInformationLevel.
↪ fromInteger(4 - s.speed - distance);
      MessageInformationLevel higherLevel = MessageInformationLevel
↪ .fromInteger(4 - s.speed + distance);
      if (messages.get(lowerLevel).size() > 0) {
        posPreferred = messages.get(lowerLevel);
        break;
      }
      else if (messages.get(higherLevel).size() > 0) {
        posPreferred = messages.get(higherLevel);
        break;
      }
    }
  }

  for (int i = 1; i < MessageInformationLevel.values().length; i++)
↪  {
    if (messages.get(MessageInformationLevel.fromInteger(i)).size()
↪  > 0) {
      posShorter = messages.get(MessageInformationLevel.fromInteger
↪ (i));
      break;
    }
  }

  if (posPreferred == null) return null;
  if (posShorter == null) posShorter = new ArrayList<Message>();

  // now we find a pair of preferred and shorter message with the same
↪  start and end types.
  HashMap<Message, Message> mapping = new HashMap<Message, Message
↪ >();
  for (Message pref : posPreferred) {
    boolean put = false;
    for (Message m : posShorter) {
      if (pref.typeStart != m.typeStart || pref.typeEnd != m.
↪ typeEnd)
        continue;
      mapping.put(pref, m);
      put = true;
      break;
    }
    if (!put) mapping.put(pref, null);
  }

  if (mapping.size() == 0) return null;
```

```
  int chosen = random.nextInt(mapping.size());

  @SuppressWarnings("unchecked")
  Map.Entry<Message, Message> result = (Map.Entry<Message, Message
↪ >) mapping.entrySet().toArray()[chosen];

  return new CarChaseArticulatable(result.getKey(), result.getValue
↪ (), optional);
}

private void applyJunction(WorldPoint point, Street street,
↪ StringDict replace, int direction, ArrayList<String>
↪ streetNamesCrossNextPoint, boolean was) {
  String prefix = was ? "WAS" : "IS";
  String prevPrefix = was ? "PREV" : "";
  boolean isEndOfStreet = street.fetchNextPoint(point, direction)
↪ == null;

  int isJunction = 0;
  if (streetNamesCrossNextPoint.size() == 2 &&
↪ streetNamesCrossNextPoint.indexOf(street.name) >= 0){
    isJunction = isEndOfStreet ? 2 : 1;

    int indexOfOther = 1 - streetNamesCrossNextPoint.indexOf(street
↪ .name);
    assert streetNamesCrossNextPoint.indexOf(street.name) >= 0 : "
↪ Something␣somewhere␣went␣terribly␣wrong:␣" +
↪ streetNamesCrossNextPoint.toString() + street.name;
    String streetName = streetNamesCrossNextPoint.get(indexOfOther)
↪ ;
    Street crossStreet = CarChase.get().world().streets.get(
↪ streetName);
    int indexInCross = crossStreet.streetPoints.indexOf(point.name)
↪ ;
    if (indexInCross <= 0 || indexInCross >= crossStreet.
↪ streetPoints.size() - 1)
      isJunction = 0;
    else {
      StreetReplacement streetRpl = streetNames.get(streetName);
      if (streetRpl == null) streetRpl = new StreetReplacement(
↪ streetName);
      replace.set("*INT" + prevPrefix + "JUNCTIONSTREET",
↪ streetName);
      replace.set("*" + prevPrefix + "JUNCTIONSTREET", streetRpl.
↪ name);
      replace.set("*FLEX1" + prevPrefix + "JUNCTIONSTREET",
```

```
    ↪ streetRpl.flex1);
          replace.set("*FLEX2" + prevPrefix + "JUNCTIONSTREET",
    ↪ streetRpl.flex2);
        }
      }
      replace.set("*" + prefix + "JUNCTION", "" + isJunction);
    }

    private class Condition {
      public String leftSide;
      public String rightSide;
      public char operator;
      public boolean isDistance;

      public Condition(String left, String right, String op) {
        this.leftSide = left;
        this.rightSide = right;
        this.operator = op.charAt(0);
        this.isDistance = leftSide.equals("*DISTANCE");
      }
    }

    private String instanciate(String original, StringDict replace) {
      String newString = original;
      for (String key : replace.keyArray())
        newString = newString.replace((CharSequence) key, replace.get(
    ↪ key));
      return newString;
    }
  }

  public static class CarChaseArticulatable extends Articulatable {
    private Message preferred, shorter;
    private boolean optional;

    public CarChaseArticulatable(Message preferred, Message shorter,
    ↪ boolean optional) {
      this.preferred = preferred;
      this.shorter = shorter;
      this.optional = optional;
    }

    public String getPreferredText() {
      return preferred.text;
    }

    public String getShorterText() {
```

```
      if (shorter == null) return null;
      return shorter.text;
    }

    public boolean isOptional() {
      return optional;
    }

    public boolean canReplace(Articulatable other) {
      if (other == null || !(other instanceof CarChaseArticulatable))
    ↪ return false;
      CarChaseArticulatable cca = (CarChaseArticulatable) other;
      return cca.preferred.typeStart == preferred.typeStart;
    }

    public void setUseOfShorterText(boolean value) {}

    public String toString() {
      return "----\n--pr-" + preferred.text + "\n--sh-" + (shorter ==
    ↪ null ? "null" : shorter.text);
    }
  }

  private class StreetReplacement {
    public String name, flex1, flex2;

    public StreetReplacement(String name, String flex1, String flex2) {
      this.name = name;
      this.flex1 = flex1;
      this.flex2 = flex2;
    }

    public StreetReplacement(String name) {
      this(name, flexForm1 + "␣" + name, flexForm2 + "␣" + name);
    }
  }
}
```

Listing D.2: The class Articulator represents the interface for a articulator and an articulatable

```
package inprotk.carchase2;

import inprotk.carchase2.CarChase;
import inpro.audio.DispatchStream;
import inpro.incremental.processor.SynthesisModule;

public abstract class Articulator {
```

```
  protected DispatchStream dispatcher;
  protected SynthesisModule synthesisModule;
  public Articulator(DispatchStream dispatcher) {
    this.dispatcher = dispatcher;
  }

  public MyCurrentHypothesisViewer getHypothesisViewer() {
    MyCurrentHypothesisViewer v = new MyCurrentHypothesisViewer();
    synthesisModule.addListener(v);
    return v;
  }

  public abstract void say(Articulatable action);

  public final static int getGlobalTime() {
    return CarChase.get().getTime();
  }

  public abstract Articulatable getLast();
  public abstract boolean isSpeaking();
  public abstract void reduceOffset();

  public static abstract class Articulatable {
    public abstract String getPreferredText();
    public abstract String getShorterText();
    public abstract boolean isOptional();
    public abstract void setUseOfShorterText(boolean value);
    public abstract boolean canReplace(Articulatable articulatable);
  }
}
```

Listing D.3: The Baseline Artculator

```
package inprotk.carchase2;

import java.util.Collection;
import java.util.List;

import inprotk.carchase2.Articulator;
import inpro.audio.DispatchStream;
import inpro.incremental.IUModule;
import inpro.incremental.processor.SynthesisModule;
import inpro.incremental.unit.ChunkIU;
import inpro.incremental.unit.EditMessage;
import inpro.incremental.unit.IU;

public class StandardArticulator extends Articulator {
  private final CarChaseIUSource ccIUSource;
```

```
  public StandardArticulator(DispatchStream dispatcher) {
    super(dispatcher);
    synthesisModule = new SynthesisModule(dispatcher);
    ccIUSource = new CarChaseIUSource();
    ccIUSource.addListener(synthesisModule);
  }

  public MyCurrentHypothesisViewer getHypothesisViewer() {
    MyCurrentHypothesisViewer v = new MyCurrentHypothesisViewer();
    ccIUSource.addListener(v);
    return v;
  }

  public void say(Articulatable action) {
    ccIUSource.say(action);
  }

  public Articulatable getLast() {
    return null;
  }

  public boolean isSpeaking() {
    return false;
  }

  public void reduceOffset() {}

  private class CarChaseIUSource extends IUModule {
    protected void leftBufferUpdate(Collection<? extends IU> ius,
        List<? extends EditMessage<? extends IU>> edits) {
      throw CarChase.notImplemented;
    }
    public void say(Articulatable action) {
      if (action.isOptional() && dispatcher.isSpeaking())
        return;
      String text = action.getPreferredText();
      rightBuffer.addToBuffer(new ChunkIU(text));
      rightBuffer.notify(iulisteners);
    }
  }
}
```

Listing D.4: The Incremental Artculator

```
package inprotk.carchase2;

import java.util.ArrayList;
```

```
import java.util.Collection;
import java.util.List;

import inpro.audio.DispatchStream;
import inpro.incremental.IUModule;
import inpro.incremental.processor.SynthesisModule;
import inpro.incremental.sink.CurrentHypothesisViewer;
import inpro.incremental.unit.ChunkIU;
import inpro.incremental.unit.EditMessage;
import inpro.incremental.unit.EditType;
import inpro.incremental.unit.HesitationIU;
import inpro.incremental.unit.IU;
import inpro.incremental.unit.IU.Progress;
import inprotk.carchase2.CarChase;
import inprotk.carchase2.StandardArticulator;

public class IncrementalArticulator extends Articulator {
  private final CarChaseIUSource ccIUSource;

  private ArrayList<Articulatable> articulates;

  public IncrementalArticulator(DispatchStream dispatcher) {
    super(dispatcher);
    this.dispatcher = dispatcher;
    synthesisModule = new SynthesisModule(dispatcher);
    ccIUSource = new CarChaseIUSource();
    ccIUSource.addListener(synthesisModule);
    articulates = new ArrayList<Articulatable>();
  }

  public MyCurrentHypothesisViewer getHypothesisViewer() {
    MyCurrentHypothesisViewer v = new MyCurrentHypothesisViewer();
    ccIUSource.addListener(v);
    return v;
  }

  @Override
  public void say(Articulatable action) {
    // I now assume that the action is chosen intelligent .
    // Therefore, here the continuation WILL NOT be appended,
    // but rather the caller has to check whether he can.

    if (action == null) return;
    articulates.add(action);
    ccIUSource.say(action, false, false);
  }
```

```
public Articulatable getLast() {
  ChunkIU iu = ccIUSource.getLast();
  if (iu == null) return null;
  return (Articulatable) iu.getUserData("articulatable");
}

public boolean isSpeaking() {
  //spokeInLastSeconds(2);
  return dispatcher.isSpeaking();
}

public boolean spokeInLastSeconds(double seconds) {
  ChunkIU iu = ccIUSource.getLast();
  if (iu == null) return dispatcher.isSpeaking();
  double timeDelta = iu.endTime() - CarChase.get().
 ↪ getInproTimeInSeconds() + seconds;
  CarChase.log(iu.startTime(), iu.endTime(), timeDelta);
  if (true) throw CarChase.notImplemented;
  return dispatcher.isSpeaking() || timeDelta >= 0;
}

public void reduceOffset() {
  ccIUSource.beginChanges();
  ArrayList<IU> ius = ccIUSource.revokeUpcoming();
  for (IU iu : ius) {
    if (!(iu instanceof ChunkIU)) continue;
    Articulatable articulatable = (Articulatable) iu.getUserData("
 ↪ articulatable");
    if (articulatable.isOptional()) {
      boolean canRemove = true;
      int index = articulates.indexOf(articulatable);
      if (index >= 0 && index < articulates.size() - 1) {
        Articulatable next = articulates.get(index + 1);
        canRemove = next.canReplace(articulatable);
      }
      if (canRemove) {
        articulates.remove(articulatable);
        continue;
      } // Otherwise, we try to use the shorter variant.
    }
    if (articulatable.getShorterText() != null) {
      articulatable.setUseOfShorterText(true);
      ccIUSource.say(articulatable, true, true);
    } else {
      ccIUSource.say(articulatable, false, true);
    }
  }
```

```
    ccIUSource.doneChanges();
  }

  private class CarChaseIUSource extends IUModule {
    private boolean changing;

    protected void leftBufferUpdate(Collection<? extends IU> ius,
        List<? extends EditMessage<? extends IU>> edits) {
      throw CarChase.notImplemented;
    }

    public ChunkIU getLast() {
      for (int i = rightBuffer.getBuffer().size() - 1; i >= 0; i--) {
        IU lastIU = rightBuffer.getBuffer().get(i);
        if (!(lastIU instanceof ChunkIU)) continue;
        return (ChunkIU) lastIU;
      }
      return null;
    }

    public ArrayList<IU> revokeUpcoming() {
      ArrayList<IU> upcoming = new ArrayList<IU>();
      ArrayList<IU> all = new ArrayList<IU>();
      for (IU lastIU : rightBuffer.getBuffer())  {
        if (lastIU.getProgress() != Progress.UPCOMING) continue;
        if (lastIU.isCommitted()) continue;
        all.add(lastIU);
        if (!(lastIU instanceof ChunkIU)) continue;
        upcoming.add(lastIU);
      }
      for (int i = all.size() - 1; i >= 0; i--)
        rightBuffer.editBuffer(new EditMessage<IU>(EditType.REVOKE, all
    ↪ .get(i)));
      if (!changing)
        rightBuffer.notify(iulisteners);
      return upcoming;
    }

    public void say(Articulatable action, boolean shorter, boolean
    ↪ commit) {
      String text = shorter ? action.getShorterText() : action.
    ↪ getPreferredText();
      boolean addHesitation = false;
      if (text.matches(".*<hes>$")) {
        text = text.replaceAll("␣<hes>$", "");
        addHesitation = true;
      }
```

```
      ChunkIU iu = new ChunkIU(text);
      iu.setUserData("articulatable", action);
      rightBuffer.addToBuffer(iu);
      if (addHesitation)
        rightBuffer.addToBuffer(new HesitationIU());
      if (commit)
        rightBuffer.editBuffer(new EditMessage<IU>(EditType.COMMIT, iu)
    ↪ );
      if (!changing)
        rightBuffer.notify(iulisteners);
    }

    public void beginChanges() {
      changing = true;
    }

    public void doneChanges() {
      changing = false;
      rightBuffer.notify(iulisteners);
    }
  }
}
```

Listing D.5: The CarChase 2 Experimenter (main class)

```
package inprotk.carchase2;

import java.awt.BorderLayout;
import java.awt.Rectangle;
import java.awt.event.KeyListener;

import inprotk.carchase2.CarChaseViewer.DriveAction;
import inprotk.carchase2.World.Street;
import inprotk.carchase2.World.WorldPoint;

import javax.swing.JFrame;
import javax.swing.SwingUtilities;

import inprotk.carchase2.CarChase;
import inprotk.carchase2.CarChaseExperimenter2;
import inprotk.carchase2.CarChaseViewer;
import inprotk.carchase2.Configuration;
import inprotk.carchase2.ConfigurationUpdateListener;

public class CarChaseExperimenter2 {
  private CarChaseViewer viewer;
  private JFrame frame;
```

```
private int lastEvent;

private CarChaseExperimenter2() {
  setupGUI();
  lastEvent = -1;
}

private void setupGUI() {
  try {
    SwingUtilities.invokeAndWait(new Runnable() {
      public void run() {
        frame = new JFrame("CarApp");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setUndecorated(true);
        frame.setSize(1024, 768);
        viewer = new CarChaseViewer();
        viewer.init();
        frame.setLayout(new BorderLayout());
        frame.add(CarChase.get().tts().getHypothesisViewer().
↪ getTextField(), BorderLayout.SOUTH);
        frame.add(viewer, BorderLayout.CENTER);
            frame.pack();
        frame.setVisible(true);
        viewer.requestFocusInWindow();
        if (CarChase.get().isInteractive()) {
          viewer.addKeyListener((KeyListener) CarChase.get().
↪ configuration());
        }
        viewer.start();

        if (CarChase.getSuperConfig("recording").equals("awt"))
          RecorderAWT.startRecording(new Rectangle(0, 0, 1024, 768));

      }
    });
  } catch (Exception e) {
    e.printStackTrace();
    throw new RuntimeException(e);
  }

  final CarChaseExperimenter2 self = this;

  CarChase.get().configuration().addListener(new
↪ ConfigurationUpdateListener() {

    @Override
    public void configurationUpdated(int type) {
```

```
      if (type != ConfigurationUpdateListener.PATH_CHANGED) {
        lastEvent = type;
        synchronized(self) {
          self.notify();
        }
      }
    }
  });
}

public void execute() {
  Configuration config = CarChase.get().configuration();

  WorldPoint startPoint = config.startPoint;
  WorldPoint nextPoint = config.nextPoint;
  Street currentStreet = config.currentStreet;
  int direction = config.direction;
  int travelDuration;
  float percent = 0;

  viewer.notifyOnSetup(this);

  synchronized(this) {
    try {
      wait();
    } catch (InterruptedException e) {
    }
  }

  do {

    int time = CarChase.get().getTime();
    travelDuration = (int) ((1 - percent) * (nextPoint.distanceTo(
↪ startPoint) / config.getCurrentSpeed(time) - 10));

    viewer.executeDriveAction(new DriveAction(startPoint, nextPoint,
↪ currentStreet, direction, travelDuration, (float) config.
↪ getCurrentSpeed(time), percent));

    synchronized(this) {
      try {
        wait();
      } catch (InterruptedException e) {
      }
    }

    if (lastEvent == ConfigurationUpdateListener.PATH_COMPLETED) {
```

```
        boolean keepRunning = config.update(time);
        if (!keepRunning) break;

        startPoint = config.startPoint;
        currentStreet = config.currentStreet;
        nextPoint = config.nextPoint;
        direction = config.direction;
        percent = 0;
        lastEvent = -1;
      }
      else if (lastEvent == ConfigurationUpdateListener.SPEED_CHANGED)
    ↪ {
        float interruptionPercent = viewer.interrupt();
        percent = interruptionPercent;
        lastEvent = -1;
      }
    } while (true);
    try {
      Thread.sleep(5000);
    } catch (InterruptedException e) {
    }
    if (CarChase.getSuperConfig("recording").equals("awt")) {
      // This will handle the shutdown.
      RecorderAWT.stopRecording();
    } else {
      System.exit(0);
    }
  }

  public static void main(String[] args) {
    CarChase.get().init("default");

    CarChaseExperimenter2 exp = new CarChaseExperimenter2();
    CarChase.get().start();
    exp.execute();
  }
}
```

Listing D.6: The Configuration class, which is also responsible for finding the next waypoint

```
package inprotk.carchase2;

import inprotk.carchase2.World.Street;
import inprotk.carchase2.World.WorldPoint;
import inprotk.carchase2.World.WorldPointWrapper;

import java.io.IOException;
```

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;

import inprotk.carchase2.CarChase;
import inprotk.carchase2.ConfigurationUpdateListener;
import inprotk.carchase2.World;

public class Configuration {
  public ArrayList<DirectionAction> directionActions;
  public ArrayList<SpeedAction> speedActions;
  public int startDirection;
  public String startPointStr, startStreetStr;

  public int currentDirAction;

  public WorldPoint startPoint, nextPoint;
  public Street currentStreet;
  public int travelDuration;
  public int direction;

  private ArrayList<ConfigurationUpdateListener> listeners;
  private int startSpeed;

  protected Configuration() {
    directionActions = new ArrayList<DirectionAction>();
    listeners = new ArrayList<ConfigurationUpdateListener>();
    speedActions = new ArrayList<SpeedAction>();
  }

  public Configuration(String filename) {
    try {
      String[] lines = CarChase.readLines(filename);
      directionActions = new ArrayList<DirectionAction>();
      speedActions = new ArrayList<SpeedAction>();
      startSpeed = 2;
      for (String line : lines) {
        if (line.startsWith("#")) continue;
        if (line.startsWith("Start:")) {
          String[] args = line.substring(line.indexOf(":")).split(",");
          args[0] = args[0].substring(1); // Remove : at the beginning
          for (int i = 0; i < args.length; i++)
            while (args[i].startsWith("␣")) args[i] = args[i].substring
  ↪ (1);
          startPointStr = args[0];
          startStreetStr = args[1];
          startDirection = Integer.parseInt(args[2] + "1");
```

```
      }
      else if (line.startsWith("---")) {
        String[] args = line.substring(line.indexOf("---") + 3).split
↪ (",");
        args[0] = args[0].substring(1); // Remove : at the beginning
        for (int i = 0; i < args.length; i++)
          while (args[i].startsWith("␣")) args[i] = args[i].substring
↪ (1);
        String name = args[0];
        if (name.equals("direction")) {
          directionActions.add(new DirectionAction(Integer.parseInt(
↪ args[1])));
        }
        if (name.equals("speed")) {
          speedActions.add(new SpeedAction(Integer.parseInt(args[1]),
↪  Integer.parseInt(args[2])));
        }
      }
      else if (line.equals("")) continue;
      else throw new RuntimeException("Illegal␣line:␣" + line + "␣in␣
↪ file␣" + filename);
    }

    World w = CarChase.get().world();
    startPoint = w.points.get(startPointStr);
    currentStreet = w.streets.get(startStreetStr);
    nextPoint = currentStreet.fetchNextPoint(startPoint,
↪ startDirection);
    direction = startDirection;
    listeners = new ArrayList<ConfigurationUpdateListener>();
    Collections.sort(speedActions);
  } catch (IOException e) {
    e.printStackTrace();
  }
}

public void addListener(ConfigurationUpdateListener l) {
  if (l != null)
    listeners.add(l);
}

protected void notifyListeners(int type) {
  for (ConfigurationUpdateListener l : listeners)
    l.configurationUpdated(type);
}

public boolean update(int time) {
```

```
  ArrayList<WorldPointWrapper> possibilities = getPossibilitiesIntern
 ↪ (startPoint, nextPoint);

  int wrapperIndex = possibilities.size() - 1;
  if (possibilities.size() > 1) {
    wrapperIndex = getNextDirection(time);
    directionUsed();
  }
  if (wrapperIndex < 0) return false;
  WorldPointWrapper wrapper = possibilities.get(wrapperIndex);
  startPoint = nextPoint;
  nextPoint = wrapper.point;
  currentStreet = wrapper.street;
  direction = wrapper.direction;
  return true;
}

protected ArrayList<WorldPointWrapper> getPossibilitiesIntern(
 ↪ WorldPoint startPoint, WorldPoint nextPoint) {
  ArrayList<String> streets = nextPoint.streets;
  ArrayList<WorldPointWrapper> possibilities = new ArrayList<
 ↪ WorldPointWrapper>();
  for (String streetName : streets)  {
    Street street = CarChase.get().world().streets.get(streetName);
    WorldPoint point1 = street.bidirectional ? street.fetchNextPoint(
 ↪ nextPoint, -1) : null;
    if (point1 != null && !point1.equals(startPoint)) {
      double theta1 = Math.atan2(nextPoint.y - point1.y, nextPoint.x
 ↪ - point1.x);
      WorldPointWrapper wrapper1 = new WorldPointWrapper(point1,
 ↪ street, theta1, -1);
      if (!possibilities.contains(wrapper1)) {
        possibilities.add(wrapper1);
      }
    }
    WorldPoint point2 = street.fetchNextPoint(nextPoint, 1);
    if (point2 != null && !point2.equals(startPoint)) {
      double theta2 = Math.atan2(nextPoint.y - point2.y, nextPoint.x
 ↪ - point2.x);
      WorldPointWrapper wrapper2 = new WorldPointWrapper(point2,
 ↪ street, theta2, 1);
      if (!possibilities.contains(wrapper2)) {
        possibilities.add(wrapper2);
      }
    }
  }
```

```
  Collections.sort(possibilities, new Comparator<WorldPointWrapper>()
  ↪  {
    @Override
    public int compare(WorldPointWrapper arg0, WorldPointWrapper arg1
  ↪ ) {
      return (int) (100 * arg0.theta - 100 * arg1.theta);
    }
  });
  return possibilities;
}

public ArrayList<WorldPoint> getPossibilities(WorldPoint prev,
  ↪ WorldPoint current) {
  ArrayList<WorldPoint> possibilities = new ArrayList<WorldPoint>();
  for (WorldPointWrapper w : getPossibilitiesIntern(prev, current)) {
    possibilities.add(w.point);
  }
  return possibilities;
}

public ArrayList<WorldPoint> getPossibilities() {
  return getPossibilities(startPoint, nextPoint);
}

public ArrayList<WorldPoint> getComingPath() {
  ArrayList<WorldPoint> path = new ArrayList<WorldPoint>();
  WorldPoint sp = startPoint;
  WorldPoint np = nextPoint;
  int steppedOver = 0;
  path.add(sp);
  path.add(np);
  for (int i = currentDirAction; i <= directionActions.size() +
  ↪ steppedOver; i++) {
    ArrayList<WorldPointWrapper> nextPossibilities =
  ↪ getPossibilitiesIntern(sp, np);
    int index = nextPossibilities.size() - 1;
    if (i >= directionActions.size() + steppedOver && index == 0) {
      steppedOver++;
      i++;
    }
    else if (i >= directionActions.size() + steppedOver) break;
    if (index > 0) {
      index = directionActions.get(i - steppedOver).data;
    } else if (index == 0) {
      steppedOver++;
    } else {
      break;
```

```
    }
    path.add(nextPossibilities.get(index).point);
    sp = np;
    np = nextPossibilities.get(index).point;
  }
  return path;
}

public void pushDirection(int direction) {
  directionActions.add(new DirectionAction(direction));
  notifyListeners(ConfigurationUpdateListener.PATH_CHANGED);
}

public void popDirection() {
  if (directionActions.size() >= currentDirAction && directionActions
  ↪ .size() > 0) directionActions.remove(directionActions.size() -
  ↪ 1);
}

public void setNextDirection(int direction) {
  while (directionActions.size() <= currentDirAction)
  ↪ directionActions.add(null);
  directionActions.set(currentDirAction, new DirectionAction(
  ↪ direction));
  notifyListeners(ConfigurationUpdateListener.PATH_CHANGED);
}

public int getNextDirection(int millisFromStart) {
  if (currentDirAction >= directionActions.size()) return -1;
  return directionActions.get(currentDirAction).data;
}

public void directionUsed() {
  currentDirAction++;
}

public double getCurrentSpeed(int millisFromStart) {
  return getDiscreteSpeed(millisFromStart) / 18.; //0.1; // px/ms
}

public int getDiscreteSpeed(int millisFromStart) {
  int speed = 2;
  for (SpeedAction a : speedActions) {
    if (a.millis <= millisFromStart)
      speed = a.delta;
  }
  return speed;
```

```
  }

  public void checkSpeed(int millis, int currentSpeed) {
    if (getDiscreteSpeed(millis) != currentSpeed)
      notifyListeners(ConfigurationUpdateListener.SPEED_CHANGED);
  }

  public void markDone() {
    notifyListeners(ConfigurationUpdateListener.PATH_COMPLETED);
  }

  public static class DirectionAction {
    public int data;

    public DirectionAction(int data) {
      this.data = data;
    }

    public String toString() {
      return "DirectionAction[data=" + data + "]";
    }
  }

  public static class SpeedAction implements Comparable<SpeedAction> {
    public int millis, delta;

    public SpeedAction(int millis, int delta) {
      this.millis = millis;
      this.delta = delta;
    }

    public String toString() {
      return "SpeedAction[millis=" + millis + "ms,delta=" + delta + "]"
    ↪ ;
    }

    @Override
    public int compareTo(SpeedAction arg0) {
      return millis - arg0.millis;
    }
  }

  public static class CarState {
    public final String streetName;
    public final String prevStreetName;
    public final String nextPointName;
    public final String prevPointName;
```

```
    public final String pointName;
    public final int direction;
    public final int prevDirection;
    public final int previousDistance;
    public final int currentDistance;
    public final int speed;
    public final int prevSpeed;
    public final int lr;
    public final CarState alternative;

    public CarState(String streetName, String prevStreetName,
        String prevPointName, String nextPointName, String pointName,
        int direction, int prevDirection, int previousDistance,
        int currentDistance, int speed, int prevSpeed, int lr,
        CarState alternative) {
      this.streetName = streetName;
      this.prevStreetName = prevStreetName;
      this.nextPointName = nextPointName;
      this.prevPointName = prevPointName;
      this.pointName = pointName;
      this.direction = direction;
      this.prevDirection = prevDirection;
      this.previousDistance = previousDistance;
      this.currentDistance = currentDistance;
      this.speed = speed;
      this.prevSpeed = prevSpeed;
      this.lr = lr;
      this.alternative = alternative;
    }
  }
}
```

Listing D.7: The Interactive Configuration class

```
package inprotk.carchase2;

import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;
import java.util.ArrayList;

import javax.swing.JOptionPane;

import inprotk.carchase2.Configuration;
import inprotk.carchase2.ConfigurationUpdateListener;
import inprotk.carchase2.World.WorldPoint;

public class InteractiveConfiguration extends Configuration implements
    ↪ KeyListener {
```

```
private int currentSpeed, backupSpeed;
private boolean waitForDirection;

public InteractiveConfiguration() {
  super();
  currentSpeed = 2;
  waitForDirection = false;
  World w = CarChase.get().world();

  String[] streetNames = w.streets.keySet().toArray(new String[0]);
  String startStreetStr = (String) JOptionPane.showInputDialog(null,
 ↪ "Choose␣start␣street:␣", "CarChase␣2", JOptionPane.
 ↪ QUESTION_MESSAGE, null, streetNames, streetNames[0]);
  String[] pointNames = w.streets.get(startStreetStr).streetPoints.
 ↪ toArray(new String[0]);
  String startPointStr = (String) JOptionPane.showInputDialog(null, "
 ↪ Choose␣start␣point:␣", "CarChase␣2", JOptionPane.
 ↪ QUESTION_MESSAGE, null, pointNames, pointNames[0]);

  boolean ask = false;
  Object next = w.streets.get(startStreetStr).fetchNextPoint(w.points
 ↪ .get(startPointStr), 1);
  if (next == null) startDirection = -1;
  else {
    next = w.streets.get(startStreetStr).fetchNextPoint(w.points.get(
 ↪ startPointStr), -1);
    if (next == null) startDirection = 1;
    else ask = true;
  }
  if (ask) {
    startDirection = Integer.parseInt((String) JOptionPane.
 ↪ showInputDialog(null, "Choose␣start␣direction:␣", "CarChase␣2",
 ↪ JOptionPane.QUESTION_MESSAGE, null, new String[] {"-1", "1"}, "1
 ↪ "));
  }

  startPoint = w.points.get(startPointStr);
  currentStreet = w.streets.get(startStreetStr);
  nextPoint = currentStreet.fetchNextPoint(startPoint, startDirection
 ↪ );
  direction = startDirection;
}

@Override
public void keyPressed(KeyEvent e) {
}
```

```
@Override
public void keyReleased(KeyEvent e) {
}

@Override
public void keyTyped(KeyEvent e) {
  int number = e.getKeyChar() - 48;
  if (number >= 0 && number < 10) {
    ArrayList<WorldPoint> path = getComingPath();
    WorldPoint last = null;
    if (path.size() < 2) last = path.size() == 1 ? nextPoint :
  ↪ startPoint;
    else last = path.get(path.size() - 2);
    WorldPoint current = null;
    if (path.size() == 0) current = nextPoint;
    else current = path.get(path.size() - 1);
    int possibilityCount = getPossibilities(last, current).size();
    if (possibilityCount > number) {
      pushDirection(number);
      if (waitForDirection) {
        synchronized (this) {
          notify();
        }
      }
    }
  }
  if (e.getKeyChar() == '/') popDirection();
  if (e.getKeyChar() == '+' && currentSpeed < 3) {
    currentSpeed++;
    notifyListeners(ConfigurationUpdateListener.SPEED_CHANGED);
  }

  if (e.getKeyChar() == '-' && currentSpeed > 0) {
    currentSpeed--;
    notifyListeners(ConfigurationUpdateListener.SPEED_CHANGED);
  }
}

public double getCurrentSpeed(int millisFromStart) {
  return currentSpeed / 20.0; // px/ms
}

public int getDiscreteSpeed() {
  return currentSpeed;
}
```

```
  public void markDone() {
    int possibilityCount = getPossibilities(startPoint, nextPoint).size
    ↪ ();
    if (possibilityCount > 1 && getNextDirection(-1) == -1) {
      waitForDirection = true;
      backupSpeed = currentSpeed;
      currentSpeed = 0;
      synchronized(this) {
        try {
          wait();
        } catch (InterruptedException e) {
        }
      }
      waitForDirection = false;
      currentSpeed = backupSpeed;
    }
    notifyListeners(ConfigurationUpdateListener.PATH_COMPLETED);
  }

  public void checkSpeed(int millis, int currentSpeed) {}
}
```

Listing D.8: The Configuration Update Listener interface

```
package inprotk.carchase2;

public interface ConfigurationUpdateListener {
  public void configurationUpdated(int type);

  public static final int PATH_CHANGED = 0;
  public static final int SPEED_CHANGED = 1;
  public static final int PATH_COMPLETED = 2;
}
```

Listing D.9: The parser and representation for the world

```
package inprotk.carchase2;

import java.awt.Point;
import java.io.IOException;
import java.util.ArrayList;
import java.util.HashMap;

import inprotk.carchase2.CarChase;


/**
 * Eine Darstellung der Welt, bestehend aus Straen und Knotenpunkten.
 * Eine Strasse wird durch eine Liste von Punkten beschrieben,
```

```
 * @author Alexis Engelke
 *
 */
public class World {
  public HashMap<String, Street> streets;
  public HashMap<String, WorldPoint> points;
  public World(String filename) {
    try {
      String[] lines = CarChase.readLines(filename);
      points = new HashMap<String, WorldPoint>();
      streets = new HashMap<String, Street>();
      for (String line : lines) {
        if (line.startsWith("#")) continue;
        else if (line.startsWith("Point:")) {
          String[] args = line.substring(line.indexOf(":")).split(",");
          args[0] = args[0].substring(1); // Remove : at the beginning
          for (int i = 0; i < args.length; i++)
            while (args[i].startsWith("␣")) args[i] = args[i].substring
    ↪ (1);
          String name = args[0];
          int x = Integer.parseInt(args[1]);
          int y = Integer.parseInt(args[2]);
          points.put(name, new WorldPoint(name, x, y));
        }
        else if (line.startsWith("Street:")) {
          String[] args = line.substring(line.indexOf(":")).split(",");
          args[0] = args[0].substring(1); // Remove : at the beginning
          for (int i = 0; i < args.length; i++)
            while (args[i].startsWith("␣")) args[i] = args[i].substring
    ↪ (1);
          String name = args[0];
          boolean bidirectional = Integer.parseInt(args[1]) > 0;
          ArrayList<String> streetPoints = new ArrayList<String>();
          for (int i = 2; i < args.length; i++) {
            streetPoints.add(args[i]);
          }
          streets.put(name, new Street(name, bidirectional,
    ↪ streetPoints));
        }
        else if (line.equals("")) continue;
        else throw new RuntimeException("Illegal␣line:␣" + line + "␣in␣
    ↪ file␣" + filename);
      }
    } catch (IOException e) {
      e.printStackTrace();
    }
```

```
}
public class Street {
  public ArrayList<String> streetPoints;
  public String name;
  public boolean bidirectional;

  public Street(String name, boolean bidir, ArrayList<String>
 ↪ streetPoints) {
    assert points != null;
    assert name != null;
    this.name = name;
    this.bidirectional = bidir;
    this.streetPoints = new ArrayList<String>();
    this.streetPoints.addAll(streetPoints);
    for (String p : streetPoints)
      points.get(p).addStreet(name);
  }

  public String toString() {
    String s = "Street[name=" + name + (bidirectional?",bidirectional
 ↪ ":",onedirectional") + ",points=" + streetPoints.toString() + "]
 ↪ ";
    return s;
  }

  public WorldPoint fetchNextPoint(WorldPoint wp, int step) {
    if (streetPoints.indexOf(wp.name) + step < 0 || streetPoints.
 ↪ indexOf(wp.name) + step >= streetPoints.size())
      return null;
    return points.get(streetPoints.get(streetPoints.indexOf(wp.name)
 ↪ + step));
  }
}
public class WorldPoint {
  public ArrayList<String> streets;
  public String name;
  public int x, y;

  public WorldPoint(String name, int x, int y) {
    assert name != null;
    streets = new ArrayList<String>();
    this.name = name;
    this.x = x;
    this.y = y;
  }

  public void addStreet(String s) {
```

```
      streets.add(s);
    }

    public String toString() {
      String s = "Point[name=" + name + ",x" + x + ",y=" + y + ",
    ↪ streets=" + streets.toString() + "]";
      return s;
    }

    public int distanceTo(WorldPoint other) {
      return (int) Math.sqrt( Math.pow(x - other.x, 2) + Math.pow(y -
    ↪ other.y, 2));
    }

    public Point asPoint() {
      return new Point(x, y);
    }
  }

  public static class WorldPointWrapper {
    public WorldPoint point;
    public Street street;
    public double theta;
    public int direction;
    public WorldPointWrapper(WorldPoint p, Street s, double alpha, int
    ↪ dir) {
      point = p;
      street = s;
      theta = alpha;
      direction = dir;
    }

    public String toString() {
      return point.name + "," + street.name + "," + direction + "," +
    ↪ theta;
    }

    public boolean equals(Object another) {
      if (!(another instanceof WorldPointWrapper)) return false;
      WorldPointWrapper w = (WorldPointWrapper) another;
      return point.name.equals(w.point.name) && street.name.equals(w.
    ↪ street.name) &&
            theta == w.theta && direction == w.direction;
    }
  }
}
```

Listing D.10: The CarChase management class

```
package inprotk.carchase2;

import inpro.util.TimeUtil;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.HashMap;

import inprotk.carchase2.CarChase;
import inprotk.carchase2.CarChaseTTS;
import inprotk.carchase2.Configuration;
import inprotk.carchase2.InteractiveConfiguration;
import inprotk.carchase2.World;

public class CarChase {
  public static final RuntimeException notImplemented = new
    ↪ RuntimeException("Not␣Implemented.");

  private static CarChase carchase;

  private String configSet;
  private String configName;
  private World world;
  private Configuration config;
  private CarChaseTTS tts;
  private long startTime;

  private static HashMap<String, String> superConfig;

  static {
    if (superConfig == null) {
      superConfig = new HashMap<String, String>();
      try {
        String[] raw = readLines("inprotk/carchase2/configs2/
    ↪ superconfig.txt");
        int index = 0;
        for (String line : raw) {
          if (index++ == 0) continue;
          line = line.trim();
          if (line.startsWith("#")) continue;
          if (line.equals("")) continue;
          superConfig.put(line.split(":")[0], line.split(":")[1]);
```

```
      }
    } catch (Exception e) {
      log("Error␣reading␣or␣parsing␣Super-Configuration.␣Full␣stop.")
  ↪ ;
      log(e);
      System.exit(-1);
    }
  }
}

public void setWorld(World w) {
  world = w;
}
public World world() {
  return world;
}

public void setConfiguration(Configuration config) {
  this.config = config;
}
public Configuration configuration() {
  return config;
}

public CarChaseTTS tts() {
  return tts;
}
public void setTTS(CarChaseTTS tts) {
  this.tts = tts;
}

public String getConfigSet() {
  return configSet;
}

public String getConfigName() {
  return configName;
}

public String getConfigFilename(String name) {
  return getFilename("configs2/" + configSet + "/" + name);
}

public void init(World w, CarChaseTTS tts, Configuration c) {
  world = w;
  this.tts = tts;
  config = c;
```

```
}

private Configuration makeConfig() {
  this.configName = getSuperConfig("config");
  if (getSuperConfig("interactive").equals("true"))
    return new InteractiveConfiguration();
  return new Configuration(getConfigFilename(configName + ".txt"));
}

public void init(String name) {
  configSet = name;
  world = new World(getConfigFilename("world.txt"));
  config = makeConfig();
  String patternFile = "patterns" + (getSuperConfig("baseline").
  ↪ equals("true") ? "-baseline" : "");
  tts = new CarChaseTTS(getConfigFilename(patternFile + ".txt"));
}

public boolean isInteractive() {
  return config instanceof InteractiveConfiguration;
}

public void start() {
  startTime = System.currentTimeMillis();
}

public float frameRate() {
  if (getSuperConfig("recording").equals("papplet"))
    return 5f;
  return 30f;
}

public int getTime() {
  return (int) ((System.currentTimeMillis() - startTime) / (30.0 /
  ↪ frameRate()));
}

public double getInproTimeInSeconds() {
  return (System.currentTimeMillis() - TimeUtil.startupTime) /
  ↪ TimeUtil.SECOND_TO_MILLISECOND_FACTOR;
}

private CarChase() {}

public static CarChase get() {
  if (carchase == null) carchase = new CarChase();
  return carchase;
```

```
  }

  // Helper methods.

  public static String getFilename(String s) {
    return CarChase.class.getResource(s).getPath().replaceAll("%20", "␣
    ↪ ");
  }

  public static void log(Object ... o) {
    for (Object ob : o) {
      if (ob == null) System.out.print("null");
      else System.out.print(ob.toString());
      System.out.print("␣");
    }
    System.out.println();
  }

  public static String[] readLines(String filename) throws IOException
    ↪ {
        InputStreamReader fileReader = new InputStreamReader(new
    ↪ FileInputStream(filename), "UTF-8");
        BufferedReader bufferedReader = new BufferedReader(fileReader);
        ArrayList<String> lines = new ArrayList<String>();
        String line = null;
        while ((line = bufferedReader.readLine()) != null) {
            lines.add(line);
        }
        bufferedReader.close();
        return lines.toArray(new String[lines.size()]);
    }

  public static String getSuperConfig(String key) {
    return superConfig.get(key);
  }
}
```

Listing D.11: The CarChase 2 Viewer

```
package inprotk.carchase2;

import inprotk.carchase2.Configuration.CarState;
import inprotk.carchase2.World.Street;
import inprotk.carchase2.World.WorldPoint;

import java.util.ArrayList;
import java.util.LinkedList;
```

```
import processing.core.PApplet;
import processing.core.PImage;
import processing.core.PVector;

public class CarChaseViewer extends PApplet {
  private static final long serialVersionUID = 1L;

  private static final float CAR_SCALE = 1f / 4.3f;
  private static final int CURVE_WIDTH = 50;

  private LinkedList<Segment> segments;

  private PImage map;
  private PImage car;

  private WorldPoint start;
  private WorldPoint end;

  private Street previousStreet;
  private Street currentStreet;
  private int speed;
  private int prevSpeed;
  private int direction;
  private int previousDirection;

  private float carAngle;
  private PVector carPosition;
  private int leftright;

  private int startMillis;
  private int duration;

  private boolean animating;

  private float previousTimelinePosition;

  private Object objectToNotifyOnSetup;

  public int sketchWidth() {
    return 1024;
  }
  public int sketchHeight() {
    return 708;
  }
  public String sketchRenderer() {
    return JAVA2D;
  }
```

```
public void setup() {
  map = loadImage(CarChase.get().getConfigFilename("
  ↪ mapWithStreetNames.png"));
  car = loadImage(CarChase.getFilename("data/car.png"));
  textFont(createFont("ArialMT-Bold", 15));
  frameRate(CarChase.get().frameRate());
  prevSpeed = -1;
  carAngle = -10;
  segments = new LinkedList<>();
  synchronized(objectToNotifyOnSetup) {
    if (objectToNotifyOnSetup != null) objectToNotifyOnSetup.notify()
  ↪ ;
  }
  image(map, 0, 0);
  textFont(createFont("Source␣Code␣Pro␣Bold", 20));
}

// Called on update
public void draw(){
  update();
  render();
}

public void update() {
  if (segments.size() == 0) return;
  Segment segment = segments.peek();

  if (segment.getPosition() >= 1 && animating) {
    segments.pop();
    if (segments.size() == 0) {
      CarChase.get().configuration().markDone();
      animating = false;
    }
  }

  if (!animating) return;

  segment.update();

  float position = segment.getPosition();

  CarChase.get().configuration().checkSpeed(CarChase.get().getTime(),
  ↪  speed);

  if (segment instanceof CircleSegment) return;
```

```
  if (previousTimelinePosition > position) previousTimelinePosition =
  ↪  position;
  if (previousTimelinePosition == position) return;

  int prevDistance1 = direction * (int) (previousTimelinePosition *
  ↪ end.distanceTo(start));
  int prevDistance2 = -direction * (int) ((1 -
  ↪ previousTimelinePosition) * end.distanceTo(start));
  int distance1 = direction * (int) (position * end.distanceTo(start)
  ↪ );
  int distance2 = -direction * (int) ((1 - position) * end.distanceTo
  ↪ (start));
  if (prevDistance1 > distance1) {
    int dist1 = distance1;
    distance1 = prevDistance1;
    prevDistance1 = dist1;
  }
  if (prevDistance2 > distance2) {
    int dist2 = distance2;
    distance2 = prevDistance2;
    prevDistance2 = dist2;
  }

  CarChaseTTS tts = CarChase.get().tts();

  CarState state2 = new CarState(currentStreet.name, previousStreet.
  ↪ name, start.name, end.name, end.name, direction,
  ↪ previousDirection, prevDistance2, distance2, speed, prevSpeed,
  ↪ leftright, null);
  CarState state1 = new CarState(currentStreet.name, previousStreet.
  ↪ name, start.name, end.name, start.name, direction,
  ↪ previousDirection, prevDistance1, distance1, speed, prevSpeed,
  ↪ leftright, state2);

  tts.matchAndTrigger(state1);

  prevSpeed = speed;
  previousTimelinePosition = position;
}

public void render() {
  background(255);
  image(map, 0, 0);
  if (segments.size() == 0) {
    renderTime();
    return;
  }
```

```
// Render Car and Path

carAngle = segments.peek().getAngle();
carPosition = segments.peek().getAbsCarPosition();

strokeWeight(5);
stroke(255, 0, 0);
ArrayList<WorldPoint> path = CarChase.get().configuration().
↪ getComingPath();
PVector decisionPoint = wp2vec(path.get(path.size() - 1));
line(carPosition.x, carPosition.y, end.x, end.y);
for (int i = 2; i < path.size(); i++) {
  line(path.get(i - 1).x, path.get(i - 1).y, path.get(i).x, path.
↪ get(i).y);
}

WorldPoint last = null;
if (path.size() < 2) last = path.size() == 1 ? end : start;
else last = path.get(path.size() - 2);
WorldPoint current = null;
if (path.size() == 0) current = end;
else current = path.get(path.size() - 1);

decisionPoint = wp2vec(current);
ArrayList<WorldPoint> possibilities = CarChase.get().configuration
↪ ().getPossibilities(last, current);

for (int i = 0; i < possibilities.size(); i++) {
  PVector p = wp2vec(possibilities.get(i));
  float dist = min(p.dist(decisionPoint), 100);
  float theta = PVector.sub(p, decisionPoint).heading();
  float x = dist * cos(theta) + decisionPoint.x;
  float y = dist * sin(theta) + decisionPoint.y;
  stroke(255, 0, 0);
  line(decisionPoint.x, decisionPoint.y, x, y);
  fill(255);
  if (!possibilities.get(i).equals(path.get(path.size() - 1)) &&
↪ possibilities.size() != 1)
    noStroke();
  rect(x - 12, y - 12, 24, 24, 4);
  fill(0);
  textAlign(CENTER, CENTER);
  text("" + i, x, y);
  stroke(255, 0, 0);
}
```

```
    pushMatrix();
    translate(carPosition.x, carPosition.y);
    rotate(carAngle + HALF_PI);
    translate(-car.width / 2 * CAR_SCALE, -car.height / 2 * CAR_SCALE);
    translate(20, 0);
    scale(CAR_SCALE, CAR_SCALE);
    image(car, 0, 0);
    stroke(0,255,0);
    popMatrix();

    noStroke();
    fill(255);
    rect(0, 0, 120, 25, 0, 0, 10, 0);
    fill(0);
    textAlign(RIGHT, TOP);
    text(CarChase.get().getTime() + "ms", 100, 2);

    if (CarChase.get().frameRate() < 6)
      saveFrame("../processing-recordings/v4/" + CarChase.get().
  ↪ getConfigName() + "/#####.png");

    renderTime();
  }

  private void renderTime() {
    noStroke();
    fill(255);
    rect(0, 0, 120, 25, 0, 0, 10, 0);
    fill(0);
    textAlign(RIGHT, TOP);
    text(CarChase.get().getTime() + "ms", 100, 0);
  }

  public int getTime() {
    return CarChase.get().getTime();
  }

  public void notifyOnSetup(Object o) {
    objectToNotifyOnSetup = o;
  }

  public void executeDriveAction(final DriveAction a) {
    if (a.percent > 0) segments.clear();
    animating = true;
    previousStreet = currentStreet == null ? a.street : currentStreet;
    currentStreet = a.street;
    start = a.start;
```

```
  end = a.end;
  speed = (int) (20 * a.speed);
  previousDirection = direction;
  direction = a.direction;

  final int millisToSkip = a.percent > 0 ? (int) (lineDuration(start,
  ↪  end, a.speed) * a.percent) : 0;
  previousTimelinePosition = a.percent > 0 ? a.percent : 0;

  float carStartAngle = carAngle;
  float carTargetAngle = atan2(end.y - start.y, end.x - start.x);
  if (carAngle == -10) carAngle = carStartAngle = carTargetAngle;

  carTargetAngle = (carTargetAngle + TWO_PI) % TWO_PI;
  carTargetAngle += (indexAbsMin(carTargetAngle - TWO_PI - carAngle,
  ↪ carTargetAngle - carAngle, carTargetAngle + TWO_PI - carAngle) -
  ↪  1) * TWO_PI;

  int rotationDuration = 0;
  if (carStartAngle != carTargetAngle && a.percent == 0) {
    boolean inverse = carStartAngle < carTargetAngle;
    rotationDuration = (int) (2 * Math.abs(carAngle - carTargetAngle)
  ↪  * (20 / a.speed) * (!inverse ? 1.4 : 1.2));
    segments.add(new CircleSegment(start, rotationDuration, getTime()
  ↪  - millisToSkip, carStartAngle, carTargetAngle));
  }

  segments.add(new LineSegment(start, end, a.speed, getTime() -
  ↪ millisToSkip + rotationDuration));

  duration = rotationDuration + segments.peekLast().duration;
  startMillis = getTime() - millisToSkip;

  leftright = carStartAngle > carTargetAngle ? 1 : 2;
}

private static int indexAbsMin(float a, float ... b) {
  float min = abs(a);
  int index = 0;
  for (int i = 0; i < b.length; i++) {
    if (abs(b[i]) < min) {
      min = abs(b[i]);
      index = i + 1;
    }
  }
  return index;
}
```

```
public float interrupt() {
  if (!animating) return 0;
  animating = false;
  float position = map(getTime() - startMillis, 0, duration, 0, 1);
  return position;
}

public static class DriveAction {
  public int /*duration, */direction;
  public float speed;
  public WorldPoint start, end;
  public Street street;
  public float percent;

  public DriveAction(WorldPoint start, WorldPoint end, Street street,
  ↪  int direction, int duration, float speed, float percent) {
    this.start = start;
    this.end = end;
    this.direction = direction;
    //this.duration = duration;
    this.speed = speed;
    this.street = street;
    this.percent = percent;
  }
}

private static PVector wp2vec(WorldPoint p) {
  return new PVector(p.x, p.y);
}

private abstract class Segment {
  protected PVector startPoint;
  protected PVector endPoint;
  protected PVector carPosition;
  protected int duration;
  protected int startTime;
  protected float carAngle;

  public Segment(WorldPoint start, WorldPoint end, int duration, int
  ↪ startTime) {
    startPoint = wp2vec(start);
    endPoint = wp2vec(end);
    this.duration = duration;
    this.startTime = startTime;
    carAngle = -100;
  }
```

```
  public abstract void update();

  public PVector getAbsCarPosition() {
    if (carPosition == null) update();
    return carPosition;
  }

  public float getAngle() {
    if (carAngle == -100) update();
    return carAngle;
  }

  public float getPosition() {
    int millis = getTime() - startTime;
    return map(millis, 0, duration, 0, 1);
  }
}

private class LineSegment extends Segment {
  public LineSegment(WorldPoint start, WorldPoint end, float speed,
  ↪ int startTime) {
    super(start, end, -1, startTime);
    float theta = PVector.sub(endPoint, startPoint).heading();
    float x = -CURVE_WIDTH * cos(theta) + endPoint.x;
    float y = -CURVE_WIDTH * sin(theta) + endPoint.y;
    this.endPoint = new PVector(x, y);
    x = CURVE_WIDTH * cos(theta) + startPoint.x;
    y = CURVE_WIDTH * sin(theta) + startPoint.y;
    this.startPoint = new PVector(x, y);
    carAngle = theta;
    this.duration = lineDuration(start, end, speed);
  }

  @Override
  public void update() {
    this.carPosition = PVector.lerp(startPoint, endPoint, getPosition
  ↪ ());
  }
}

private class CircleSegment extends Segment {
  private PVector mid;
  private float radius;
  private float angleStart, angleEnd;
  private boolean inverse;
```

```
  public CircleSegment(WorldPoint pt, int duration,
      int startTime, float anglePr, float angleNe) {
    super(pt, pt, duration, startTime);
    float x = CURVE_WIDTH * cos(angleNe) + pt.x;
    float y = CURVE_WIDTH * sin(angleNe) + pt.y;
    this.endPoint = new PVector(x, y);
    x = -CURVE_WIDTH * cos(anglePr) + pt.x;
    y = -CURVE_WIDTH * sin(anglePr) + pt.y;
    this.startPoint = new PVector(x, y);

    this.angleStart = anglePr;
    this.angleEnd = angleNe;

    inverse = angleStart < angleEnd;

    radius = cotan(angleDistance(angleStart, angleEnd)/ 2)*
↪ CURVE_WIDTH*2;
    mid = PVector.add(PVector.mult(PVector.fromAngle(angleStart +
↪ HALF_PI * (inverse ? 1 : -1)), radius/2), startPoint);

    if (inverse) {
      angleStart += PI;
      angleEnd += PI;
    }
  }

  @Override
  public void update() {
    carAngle = map(getPosition(), 0, 1, angleStart, angleEnd);
    this.carPosition = PVector.add(PVector.mult(PVector.fromAngle(
↪ carAngle+HALF_PI), radius / 2), mid);
    carAngle -=  inverse ? PI : 0;
  }

}

private float angleDistance(float angle1, float angle2) {
  float dist1 = angle1 - angle2;
  float dist2 = TWO_PI - angle2 + angle1;
  return min(abs(dist1), abs(dist2));
}

private float cotan(float f) {
  return 1/tan(f);
}
```

```
  private int lineDuration(WorldPoint start, WorldPoint end, float
    ↪ speed) {
    PVector startPoint = wp2vec(start), endPoint = wp2vec(end);
    float theta = PVector.sub(endPoint, startPoint).heading();
    float x = -CURVE_WIDTH * cos(theta) + endPoint.x;
    float y = -CURVE_WIDTH * sin(theta) + endPoint.y;
    endPoint = new PVector(x, y);
    x = CURVE_WIDTH * cos(theta) + startPoint.x;
    y = CURVE_WIDTH * sin(theta) + startPoint.y;
    startPoint = new PVector(x, y);
    carAngle = theta;
    return (int) (startPoint.dist(endPoint) / speed);
  }
}
```

Bibliography

Timo Baumann. *Incremental spoken dialogue processing: architecture and lower-level components*. PhD thesis, Bielefeld University, 2013.

Timo Baumann and David Schlangen. Open-ended, extensible system utterances are preferred, even if they require filled pauses. Proceedings of Short Papers at SIGdial 2013, 2013.

Ole Eichhorn. Dynamische anpassung bei der generierung natürlicher sprache, 2013.

Aravind Joshi and Yves Schabes. Tree-adjoining grammars and lexicalized grammars. Technical report, University of Pennsylvania, 1991.

Anne Kilger and Wolfgang Finkler. Incremental generation for real-time applications. Technical report, Saarländische Universitts- und Landesbibliothek, 1995.

Willem Levelt. Monitoring and self-repair in speech. *Cognition*, 14.1:41–104, 1983.

Kris Lohmann. *Verbal Assistance with Virtual Tactile Maps: a Multi-Modal Interface for the Non-Visual Acquisition of Spatial Knowledge*. PhD thesis, Universität Hamburg, 2012.

Kris Lohmann, Ole Eichhorn, and Timo Baumann. Generating situated assisting utterances to facilitate tactile-map understanding: A prototype system. In *Third Workshop on Speech and Language Processing for Assistive Technologies*, 2012.

Günter Neumann. Natural language generation: Grammar formalisms and their processing. `http://www.dfki.de/~neumann/publications/new-ps/nlg-overview.pdf`, 2002.

Ehud Reiter. Building natural-language generation systems. *CoRR*, cmp-lg/9605002, 1996.

Ehud Reiter and Robert Dale. *Building natural language generation systems*. Cambridge University Press, 2000.

David Schlangen and Gabriel Skantze. A general, abstract model of incremental dialogue processing. Proceedings of the 12th Conference of the European Chapter of the Association for Computational Linguistics (EACL 2009), pages 710–718, 2009.