Department of Computer Science
Lund Institute of Technology

Development and Integration of Linguistic Components for an Automatic Text-to-Scene Conversion System

Hans Svensson & Ola Åkerberg

August 2002

Abstract

The purpose of this master's thesis is to implement a text-to-scene converter to visualize accident descriptions. The text description of an accident can be difficult to understand. We believe that misinterpretations can be avoided by visualizing the accident in a three dimensional scene.

The architecture of the CarSim 2.0 system consists of two parts and draws its structure from CarSim (Dupuy et al. 2001). The first part is an information extraction module that gathers the significant data from a text and presents it in a tabular XML format. The second part is a visualization module that interprets the tabular data and displays a three dimensional scene of the accident.

This report describes the linguistic components that we have developed or integrated for the information extraction module. Only a part of the text is relevant to visualize the accident. The objective of the information extraction module is to find the road configuration, the vehicles, their movement, the list of collisions, and static objects that play a role in the accident.

The architecture of the information extraction module is similar to that of the FASTUS system (Hobbs et al. 1997). WordsEye (Coyne and Sproat 2001) that translates written descriptions into non-moving pictures has also influenced us. The report describes the integration of two widespread linguistic resources: Link Grammar and WordNet. Link Grammar is a syntactical parser and WordNet is a semantic database. The report then describes how we use these tools and the algorithms we have implemented to detect information about dynamic objects (vehicles) and static objects (trees, traffic lights etc.).

We tested CarSim 2.0 with a corpus of accident summaries that we collected from the National Transportation Safety Board home page in the United States. We finally report the results we obtained.

Abstrakt

Syftet med detta examensarbete är att visualisera beskrivningar av trafikolyckor. Det kan vara svårt att förstå ett händelseförlopp från en textbeskrivning av en trafikolycka. Genom att visualisera olyckan i en datorgenererad 3D miljö kan man undvika missförstånd.

CarSim 2.0 består av två delar och bygger på CarSim (Dupuy et al. 2001). Syftet med den första delen är att extrahera information från en text och lagra den i ett XML tabellformat. Syftet med den andra delen är att tolka och visualisera händelseförloppet i XML beskrivningen.

Vårt projekt innefattar endast den första delen. Systemet baseras på tekniker liknande de som används i FASTUS (Hobbs et al. 1997). Ett annat projekt att dra lärdom av är WordsEye (Coyne and Sproat 2001). WordsEye är ett system som överför text till statiska bilder. Vi kommer att integrera två lingvistiska program i vårt projekt: Link Grammar och WordNet. Link Grammar är en syntaktisk parser och WordNet är en semantisk ord databas. Systemet kommer att använda de lingvistiska komponenterna för att detektera information om fordon och icke rörliga objekt.

Vi evaluerade CarSim 2.0 med texter tagna från The National Transportation Safety Board i USA. Slutligen redovisar vi resultaten vi fått.

Table of Content

Chapter 1 Introduction

1.1 A Background of CarSim

CarSim (Dupuy et al. 2001) is a program that analyzes texts describing car accidents and visualizes them in a 3D environment. The CarSim architecture is split into two modules. The first module carries out a linguistic analysis of the accident and creates a template – a tabular representation of the text. The second module creates the 3D-scene. The template is designed to contain minimal information but enough to recreate the scene in a symbolic (iconic) way.

The first version of CarSim was designed to process texts in French. The information extraction part was written in Prolog and the graphical module in Java. CarSim was developed and tested on a corpus of insurance reports written by the drivers. These reports were obtained from the MAIF insurance company.

CarSim is one of few projects working with text-to-scene conversion.

1.2 Why Animate Texts

The advantage of text-to-scene conversion can be better explained with an example. Consider the text below and try to understand it.

> *About 10:30 a.m. on October 21, 1999, in Schoharie County, New York, a Kinnicutt Bus Company school bus was transporting 44 students, 5 to 9 years old, and 8 adults on an Albany City School No. 18 field trip. The bus was traveling north on State Route 30A as it approached the intersection with State Route 7, which is about 1.5 miles east of Central Bridge, New York. Concurrently, an MVF Construction Company dump truck, towing a utility trailer, was traveling west on State Route 7. The dump truck was occupied by the driver and a passenger. As the bus approached the intersection, it failed to stop as required and was struck by the dump truck. Seven bus passengers sustained serious injuries, 28 bus passengers and the truckdriver received minor injuries. Thirteen bus passengers, the busdriver, and the truck passenger were uninjured.*

The text describes an accident between a truck and a bus in an intersection. The bus was driving on State Route 30A and the truck on State Route 7. Did you get that? The text is taken from the corpus used when testing the CarSim 2.0 system

1.3 An Overview of the Project

In our Master's project, we designed and implemented a new version of CarSim's information extraction module. In contrast to the first CarSim prototype, which analyzed texts in French, the new language-processing module is designed for English.

We gathered a corpus of car accident descriptions from the National Transportation Safety Board (Highway Accident Reports Publications), referenced as NTSB in the remainder of this report. The NTSB is the accident research organization of the United States government. They survey different scenarios, mostly flight accidents but also incidents regarding boats. We used the summary part of the road accident reports that were collected from the Web (Appendix A). We developed the language processing components to analyze these texts. We worked particularly on semantic components specific to the car accident domain. When available we also tried to use generic tools.

The first version of CarSim used two different executable programs written in Prolog and Java. An intermediate file managed the communications between the two modules (Figure 1.1).

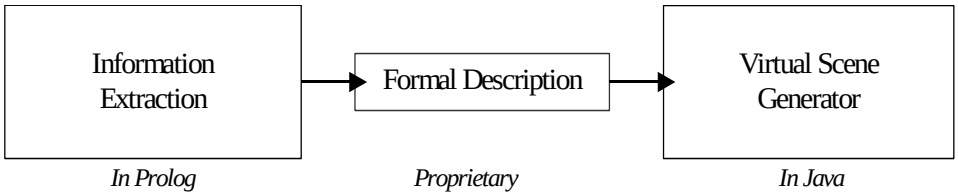

Figure 1.1 The CarSim architecture

In addition to the language-processing module, we rewrote the template formalism in XML and we integrated both parts of CarSim: the information extraction module and the graphical module in the same interface. The user can start the information extraction and the three dimensional display of an accident from the same graphical user interface. She/he can also adjust the settings of the program. Figure 1.2 shows the new architecture. The graphical user interface supports direct editing of the original text file and the XML template. It does not require an external editor as before.

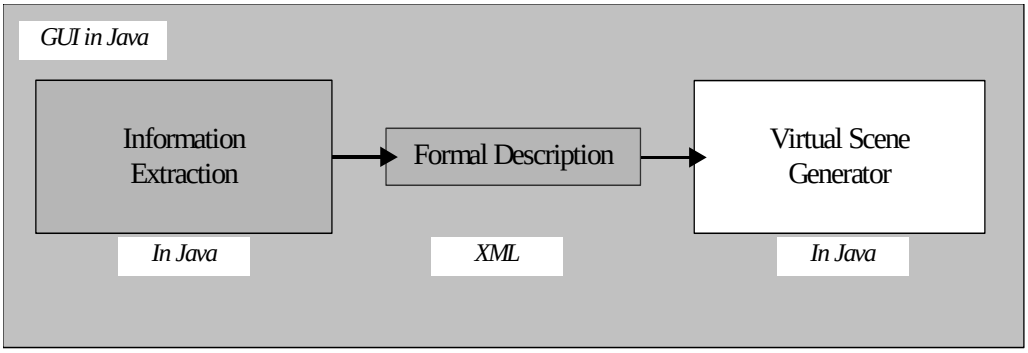

Figure 1.2 The new modules

The information extracted from the text is mapped onto a predefined template that consists of three parts: the static objects, the dynamic objects, and the collision objects. The static objects are the non-moving objects such as trees, obstacles, and road signs. The dynamic objects are moving objects, the vehicles. Examples of dynamic objects are cars and trucks. The collision object structure describes the interaction between dynamic objects and/or static objects.

We use two widespread linguistic resources to analyze English texts: WordNet (Fellbaum and Miller 1998) and Link Grammar (Sleator and Temperley 1991). Link Grammar is a dependency parser that determines the grammatical relationships between the words of a sentence. WordNet is a lexical database and a semantic network. The idea of CarSim is to combine these two resources and use them to understand the meaning of texts. The strategy to fill the templates is to find verbs that are frequently used to describe accidents. CarSim uses regular expressions to search verb patterns in texts. Then CarSim extracts the dependents of the verb. It evaluates the function of the

word groups in the sentence relative to the verb. It examines words and classifies them using WordNet.

We have developed the information extraction module in Java. A program in Java has the advantage of being platform independent. However, the IE module also integrates C libraries and these are not portable. We adapted the program to the Solaris and Windows platforms and that covers a large percentage of the market. We used Gnu tools to build the C libraries so an expansion to the Linux platform should be feasible. The amount of time allocated to this project has not permitted testing on the Linux platform.

1.4 An Overview of the Report

Chapter 1 gives an introduction and an overview to the CarSim system. It contains an example justifying why text-to-scene conversion can be a powerful tool to examine and understand texts.

Chapter 2 covers the analysis of the system. It describes the goals, the purpose and the architecture. It summarizes two similar projects: FASTUS and WordsEye.

Chapter 3 describes the system components and modules. It covers two important modules of the system: the WordNet lexical database and the Link Grammar parser.

Chapter 4 describes how the whole system works. This chapter details the system components and the building steps.

Conclusions are in chapter 5. It describes the results we obtained with our system.

Appendix A contains the texts, which we used as a corpus in this master thesis. They come from National Transportation Safety Board in the United States. Appendix B contains the typographical errors in the texts from National Transportation Safety Board. Appendix C contains the DTD describing the XML format. Appendix D describes the auxiliary tools we used in the development of CarSim 2.0

Chapter 2 Analysis

2.1 General Purpose

The purpose of the CarSim 2.0 project is to write a system to visualize car accidents. More precisely, our objective was to design and implement an information extraction module for texts in English, with a semantic part dedicated to the analysis of accidents.

The architecture of information extraction systems generally consists of a sequence of linguistic modules. Some of these are generic like the parser and some are specific to the application. Unlike the first CarSim version, CarSim 2.0 should integrate generic linguistic resources such as the WordNet semantic database and the Link Grammar parser and contain modules specific to the visualization requirements and the road accident domain. It should produce intermediate templates using the XML standard. CarSim 2.0 should also integrate the information extraction and visualization modules to be accessible from the same interface written in Java.

The application should be a convenient development tool and improve the workflow. If the information extraction module does not produce the desired output, the user must be able to edit the original English text, maybe remove some words, and try again. If there is data missing in the resulting XML template (the intermediate file) the user ought to have the option of filling in the preferred value and view the changes in the three dimensional display of the accident.

As for other software system, CarSim 2.0 must be robust, easy to use and not prone to error. It should accept any type of text without crashing. It should be well documented as someone else may enhance the project.

CarSim 2.0 should be made available to other people interested in natural language processing and visualization. A homepage with instructions and a downloadable CarSim package should be constructed. The different parts of CarSim should be free to use for non-commercial purposes.

2.2 Goals

Information extraction from texts in natural language is not a simple task, so we do not expect to interpret the texts correctly in one hundred percent of the cases. One can also argue about when the scenario in a text is in fact accurately detected. With CarSim we will try to achieve as high throughput as possible but considering the amount of time allocated to this project, the expectations are not exaggerated. The information extraction used in CarSim is very specialized. The texts are analyzed with the template structure in mind i.e. the goal is to fill the template with data. This means that any text, may it be a poem, will be parsed and a template filled with at least the minimum requirements that are set by the graphical module will be produced. No matter what text is loaded in CarSim it will produce a template that can be viewed in the graphical module. If a text that does not describe an accident is analyzed by the system, a default template will be produced. It contains a straight road with a parked car. On the road is a sign displaying the message “No accidents were found”.

2.3 Architecture

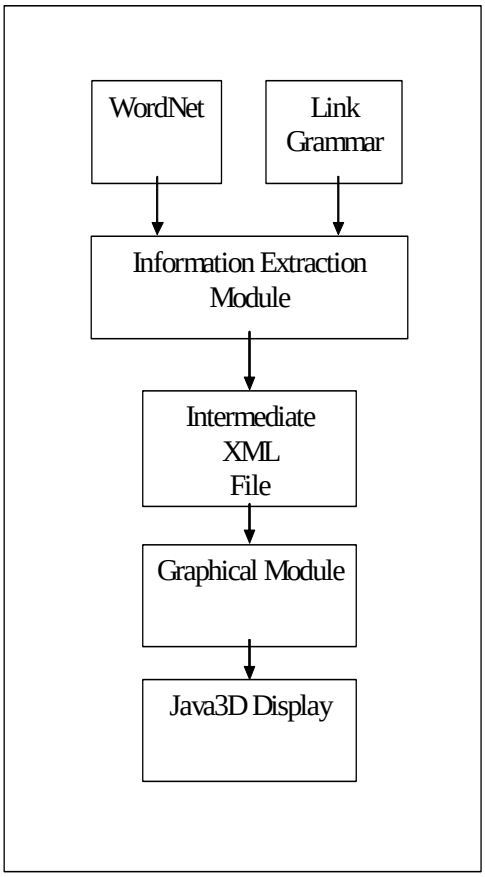

Figure 2.1 The Architecture of CarSim 2.0

The architecture of CarSim 2.0 is basically the same as in the first version. One difference is that both information extraction and the animation tasks can be controlled from the same graphical user interface.

Figure 2.1 shows the logical workflow in the CarSim system. The main components are the information extraction module and the Graphical module. The information extraction module is the workhorse of the program and it carries out the analysis of the texts. Dynamic objects, static objects and accidents are detected. Dynamic objects are entities with the ability to move for example busses, cars or any other vehicles. Static objects are non-moving entities like trees and signposts. An additional task for the information extraction module is road type discovery. Road types are comprised of four categories: `straightroad`, `left_turn`, `right_turn` and `crossing`. The graphical module can only display turns in the east-north or the east-south directions which corresponds to a `left_turn` and a `right_turn` respectively. In reality, all directions are possible. Hence, the detected turns are translated into the available directions.

Furthermore, the straight road exists only in the east-west direction. As shown in Figure 2.1 the information extraction engine uses two resources: WordNet and Link grammar. The main information extraction module is written in the Java programming language and the two resources are written C and C++. These are compiled into

dynamic shared libraries and the connection between the resources and the information extraction module is handled by a technique called the Java Native Interface JNI. The JNI provides a framework for integration of native code in a Java application. JNI is described further below.

The other main component of the system is the graphical module. The graphical module reads the data produced by the information extraction module. It then applies rules and tries to animate the accident. This is done with the Java3D technology.

The box between the information extraction module and the graphical module in Figure 2.1 called "Intermediate XML file" symbolizes the data exchange between the two modules. In the first version, a file was saved and shared between the two separate programs. In CarSim 2.0, it is rather a `String` than a file passed between the modules due to the integration into a common graphical user interface. However, an XML file is still saved to disc in order to make it available for future reference. The template follows a strict convention described by a DTD. As mentioned before the information extraction is done with this structure in mind.

2.4 Related Projects

In this section, we review two systems that are related to our project. One covers information extraction, FASTUS, and the other the visualization of texts, WordsEye. They can be considered as good examples of the state-of-the-art in their respective domain.

2.4.1 FASTUS Project

FASTUS (Hobbs et al. 1997) is an acronym for Finite State Automaton Text Understanding System. FASTUS is an information extraction tool. It uses a cascade of language processing modules based on finite state automata. Its aim is not to fully understand the text but rather to search for a fixed set of information that is described in a template: tabular data. It assumes that only a part of the information is relevant. FASTUS is designed to process vast amounts of text and it is optimized for speed, hence its name.

FASTUS is implemented using a set of tasks where each aims at extracting different interesting information. The four major tasks are complex words, phrase recognition, pattern recognition, and merging:

- Complex words recognition is a task that identifies multi-words and names. Examples on multiword are "joint venture" and "trading house". Names are words like "Manchester United".
- Phrase recognition involves splitting up sentences into smaller, easier handled groups. These groups consist of words that make up a phrase. Such groups include the noun groups, verb groups, and other phrases. The phrase recognition task is divided into two subtasks: basic phrases and complex phrases.
- Pattern recognition: The sequence of phrases found in the phrase recognition task is searched for interesting patterns. The information that is not included in the groups constructed in phrase recognition is discarded. The selected patterns are then mapped into an incident structure. The incident structure is predefined and determines what data the text is scanned for.
- Merging combines several patterns found in the text. Information from the patterns is merged into one single template.

Currently FASTUS supports the English language and the Japanese language but its principles are applicable to other languages as well. The development process has been under way since 1992; the system is implemented in Common Lisp.

2.4.2 WordsEye

WordsEye (Coyne and Sproat 2001) is a system for translating a text into a three dimensional graphic scene. It can currently handle texts in English. The Scene generated is static, not an animation. The people behind WordsEye have chosen to concentrate on the language processing part rather than animation. WordsEye carries out syntactic and semantic analyses of a text. It uses several already existing components for parsing and for the semantic analysis of the text. The tools used are Ken Church's part of speech tagger (1988), Michael Collin's parser (1999) and WordNet. WordsEye has semantic entries for 1,300 English nouns and 2,300 verbs.

WordsEye first parses a sentence and then converts it into a dependency structure. Lexical semantic rules are then used to create the components of the scene description. This description is mapped onto poses, low-level depicters that together make up an image visualizing the text. The image can be the snapshot of an action for example "A man throwing a ball" showing a man in a throwing position.

To be able to show a wide variety of texts as a three-dimensional image, WordsEye uses a large database of three-dimensional objects. The same picture, a hand for instance, can be shown differently, open or closed. This is achieved using inference rules on poses, spatial relations, and color. To resolve conflicts and add implicit constraints, WordsEye has a set of transduction rules that is applied to the result of previous task.

WordEye also uses a co-referencing algorithm that connects personal pronouns to their antecedents. WordsEye considers gender features and number to get an accurate result from the co-reference algorithm. "The cat is sitting on the table. It is black" exemplifies an anaphoric sentence. WordEye connect the pronoun "It" to the antecedent "cat". The resulting picture will display a black cat sitting on a table. In some cases, the information in a sentence is ambiguous. In that case, WordsEye tries to guess what the writer wanted to express in the sentence. Consider the following example "John sits in the car. The vehicle is next to the boat". Using WordNets Hierarchical structure WordsEye connects "car" to "vehicle" and they will be interpreted as one entity. It is not certain that this is a reflection of the writer's intensions. It still produces a better result than interpreting "car" and "vehicle" as two entities.

Chapter 3 Description of the System Components and Modules

3.1 Integrating the Components

The CarSim project integrates available linguistic components. It also uses auxiliary programs to support the software development. The purpose of these programs is to take advantage of quality resources and to decrease the development time. The programs we used are free software downloaded from the Internet and they are described in Appendix D

The main development programming language used in the CarSim project is Java. Java is an established object-oriented language originally developed by Sun. Some parts of the project use C libraries made available through the Java Native Interface JNI.

This section describes the integration of external modules and programs in our Java environment.

3.2 Parsing

3.2.1 Link Grammar

Link Grammar is a syntactic parser used to analyze English sentences. It is based on dependency theories and was developed by Davy Temperley, Daniel Sleator, and John Lafferty at the Carnegie Mellon University. Link Grammar is written in C and versions are available for both UNIX and Windows platforms. The version used in CarSim project comes with an API, which gives access to vital functions of Link Grammar. The API grants access to the dictionary as well as the link structure. Link Grammar can be parametered using options that alters the way it processes a text.

Link Grammar's dictionary contains over 60,000 words and it parses one sentence at a time. Link Grammar can handle typographical errors. The result from a parsed sentence is the dependency structure: links between words, which reflects the grammatical relations between them. The output from Link Grammar can be shown in two ways. The first way shows the dependency links that connect two words (Figure 3.1.)

```
  +-Ds-+--Ss-+--Pa--+
  |    |     |     |
my cat.n is.v black.a
```

Figure 3.1 Sentence with links as displayed by Link Grammar

Link Grammar can also produce a constituent tree. It recursively displays noun phrases, verb phrases, etc. Figure 3.2. shows a constituent tree.

```
(S(NP My cat)
   (VP is
      (ADJP black)))
```

Figure 3.2 Constituent Tree, the other output from Link Grammar

To get a valid linkage in Link Grammar, all words must be connected either directly or indirectly to each other and the links cannot cross. The words' part-of-speech is indicated by suffixes such as `.n` designating a noun, `.v` a verb, `.a` an adjective, and `.e` an adverb. Depending on the type of grammatical relation between the words, different labels annotate the link: subject, object, etc. The word that the S-link connects to the left is a subject (or the headword of the subject phrase). The word to the right is the finite verb. The part-participle link enables the location of the subject and the agent in a passive sentence.

Link Grammar can be configured with many options that improve its robustness. For example, it is possible to run Link Grammar with "allow Null links" and skip words that are not recognized. Link Grammar still gets a correct linkage on the rest of sentence. When Link Grammar does not recognize a word, it displays a trailing `[?]`. If the system is using Null links and Link Grammar cannot find a parse for a word it will ignore it and try to link the rest of the sentence. Brackets surround the word then.

Parsing of a sentence consists of several passes. First, it tries to find a linkage without null links. If it does not succeed, the parser will try to find a complete linkage with one null link and so on. When everything else has failed, Link Grammar enters panic mode (that can be switched off) and tries again with different rules. It will only consider links of a certain length and will allow islands of disconnected words. In this mode, parsing will always be completed in a reasonable time. Notice that Link Grammar is not one hundred percent reliable. It sometimes fails to produce a valid result.

Link Grammar also has a post-processing stage where it checks the linkages found. There are certain phrases that parsing cannot resolve. This is taken care of in a post processing stage. The idea is to subdivide a sentence into blocks of words called domains. Then, it controls the links present in each domain. If they are not valid, it discards this linkage. The corresponding rules are placed in the post-processing file. Post-processing can also be turned off.

Link Grammar uses five different kinds of files to parse sentences: dictionary files, post-processing files, constituent-knowledge files, affix files and word files. The dictionary files contain rules about links. The post-processing file is described above. The constituent file contains the rules to create a constituent tree. The affix file contains rules to handle punctuation and special characters. The word files contains categories of words. Link Grammar comes with standard files but they can be edited or replaced. In the CarSim project, we use the standard files. Link Grammar can also set a verbosity level to determine the amount of output.

3.2.2 Finding the words and the grammatical relations

CarSim identifies the subject and object of a verb by examining the Link Grammar output. Figure 3.1 shows an output example from Link Grammar. The links of the sentence can be seen above the words. The verb is used as a starting point. The pair (*is, cat*) is connected by an Ss link and the pair (*is, black*) by the Pa link. The Ss connects the verb of a sentence to the subject and the Pa link points to the object of the sentence. This is a simple example. Some sentences require traversing much more complex graphs to find the subject, object, etc.

In addition, Link Grammar tags the words with their part of speech. It can be extracted using the word suffix such as .n for nouns and .v for verbs after the parse.

3.2.3 Link Grammar JNI

As mentioned earlier in this document, Link Grammar is written in C and our program is written in Java. To integrate Link Grammar, we used a JNI bridge. JNI is an API to integrate code written in other languages than Java. However, if you use JNI, you loose one of the main concepts in Java, “Write once, run everywhere”. The code becomes system dependent (The C code is system dependent).

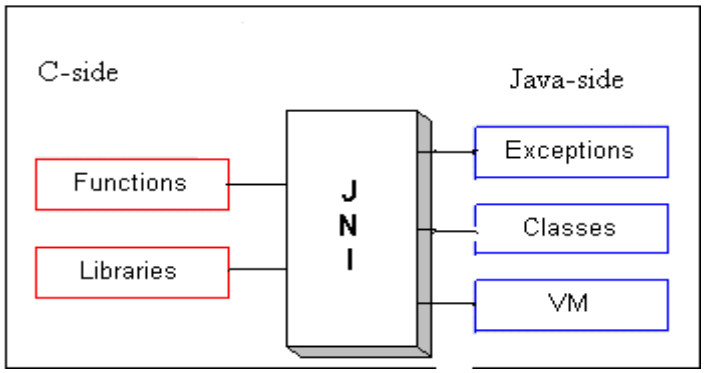

Figure 3.3 Picture shows how the JNI works in a system

Figure 3.3 shows the concept of JNI. JNI enables native methods to use and update Java objects created on the Java side. Java Objects can also be created on the native side. The objects can be passed between the Java and the Native side and thus share data. Applications and legacy programming libraries that are already implemented in a native language can be accessed by Java code. A Java method can call a native method, pass the required parameters, and get the result back. The same is true when native methods call a Java method. You can throw and catch exceptions from the native method. With functions in JNI you can load Java classes and get class information. JNI also supports runtime type checking.

3.3 Lexical Semantics Analysis

3.3.1 WordNet

WordNet is a lexical reference that was developed at the Cognitive Science Laboratory at Princeton University under the direction of Professor Miller. It is designed for English but projects exist to extend the idea to other languages as well. WordNet separates words into nouns, verbs and adjectives. These are organized into synonym sets (synsets). Words in each part of speech class have specific functions. These functions include synonyms, hypernyms, and hyponyms for the noun class.

Many English words have more than one sense. Consider the word “road”. First, it can be the physical sense. The road you walk on. It can also have the meaning of a way to achieve something as in the sentence “the road to success”. When CarSim looks up a word in WordNet, it can select a specific sense and part of speech.

3.3.2 Acquiring classes of words

To find specific information in the text, we use WordNet to create a word database. For example when we try to detect a crash verb in the text, we need a list of all the words that can occur in an accident description. To achieve this, a full hyponym tree is looked up in WordNet. Hypernymy and hyponymy are semantic relations between words. We can rephrase these relations as “more general” and “more specific". The word “birch” is a hyponym of “tree”, which in turn is a hyponym of “plant”. This means that if the system detects the word “birch” in the text it knows that it is a “tree” and it will save it as a static object.

We determined sets of words in the WordNet hierarchy useful to the CarSim road domain with an interactive manual process. We acquired abstract words defining semantic classes applicable to this domain and we called them the super words. A super word is the highest level in the hyponym tree. We tested WordNet with different words to figure out which super word to use and we inspected their hyponym trees. The word that returned the most accurate results in terms of numbers and usefulness was selected. Sometimes, we used several super words to create a group in the word database. Figure 3.4 shows the result produced by WordNet online version for the word “light”, sense 14, noun part-of-speech, and full hyponym tree. Later, we selected “light” as a super word.

```
light
   => traffic light, traffic signal, stoplight
         => green light, go-ahead
         => red light
         => yellow light
```

Figure 3.4 A full hyponym tree for the noun light at sense 14

3.4 Template Structure

The template is the tabular output of CarSim information extraction module. When filled in properly, the system will be able to run the accident in the 3D-simulator. The structure of the template can be divided into three different parts: the static objects, the dynamic objects, and the collision object. These objects have different required parameters. Chapter 4 explains how the data is collected. We discuss here the three parts of the templates.

The static object section is the part where the system stores all the non-moving objects, such as trees, road sign, road type, level crossings, and traffic lights. Detection of other obstacles like a fence and an embankment is feasible but the simulator cannot visualize these types of objects. These objects all end up as a tree. There are four different road configurations and the road type is one of the required parameters in the template. The types are `straightroad`, `crossing`, `turn_left` and `turn_right`. Tree and traffic lights must have an id because there can be more than one entity referring to for instance a tree in a text. The color of the traffic light is also a required value and the types of colors supported are `red`, `orange`, `green`, or `inactive`. The only type of sign that can exist in the simulation is the `stopSign`.

The second part of the template describes the dynamic objects, the vehicles mentioned in the text. The possible types of vehicles are a `truck` or `car`. All the dynamic objects must also have an id. All dynamic objects must also have an initial

direction, so that the simulator knows where the vehicle starts. The possible values are `north`, `south`, `west`, and `east`. This part of the template structure also contains possible road signs if any are detected. A typical road sign could be "*State Route 30A*"[1]. All the vehicles must have an event list that describes their action sequence. The possible values are `driving_forward`, `turn_left`, `turn_right`, `stop`, `overtake`, `change_lane_left`, and `change_lane_right`.

The last part of template describes the collision objects. It contains the information about the objects that have crashed and the list of accidents. The system tries to detect which vehicle is the victim and which is the actor. The required data in the collision object is what part of the vehicle was involved in the accident. The choices are `front`, `rear`, `left_side`, `right_side`, or `unknown`. The `unknown` value is used when the program was unable to detect any vehicle parts from the text.

[1] A better design would probably move the directions road signs to the static object section of the template.

Chapter 4 How the Whole Program Works

4.1 A General Description

The work of developing CarSim in general and the CarSim information extraction engine in particular can be subdivided into steps. At first we studied the field of computational linguistics. To get a notion of what is considered to be the most successful way of accurately interpreting texts with the aim on understanding and displaying a road accident.

The texts in the corpus are collected from the Internet home site of the National Transportation Safety Board. They are naturally real texts (not manually invented). They are typically about two hundred words long. The language used is technical, dense, and hard to read. The CarSim is a new development of the French version. It used initially the original graphical module to display the accident information gathered from the texts.

The division of the project into tasks was as follows:

- Incorporate the widespread linguistic analyzing tools: WordNet and Link Grammar.
- Split texts into sentences.
- Build a dictionary of words indicating an accident situation using WordNet.
- Search the text for static objects and find the attributes that describes them.
- Proceed with the detection of road configuration.
- Extract the names of the roads.
- Integrate the program written by Torbjörn Ekman and Anders Nilsson (2002) connecting a subject and object to a verb.
- Map the information gathered in previous step onto a predetermined accident structure.
- Instead of verbs indicating an accident situation, use verbs describing any kind of movement to determine the sequence of events described in the text. This includes changes of movement such as stops.
- At this point, all the dynamic object structures are created. Some of the objects in a subject-verb-object phrase can however be static objects. Find them and make the connection to static objects that are already extracted. The sentences with the subject-verb-object combination are inspected to find events. Possible events are `driving_forward`, `turn_left`, `turn_right` and `stop`.
- Engineer a more user-friendly interface.
- Integrate the CarSim information extraction module and the graphical module. In parallel with implementation of the information extraction module, Bastian Schulz (2002) upgraded the graphical module. Bastian also introduced a new graphical user interface that gives the user control over both the information extraction task and three dimensional animation tasks.
- With the knowledge of the subject-verb-object chain and their individual locations in the sentence, search in what direction the vehicles (dynamic objects) are moving.
- Interpret the directions and translate them so that they fit the road configuration.
- The text has now gone through all passes of interpretation. The information is formatted and transmitted to the graphical module.

4.2 The Tasks Performed in the CarSim Project

4.2.1 Introduction

The following sections explain each task in depth. The approach is to divide the project into smaller chunks that are simpler to manage. The pros of using this method are that you divide a very difficult problem into smaller pieces that each has a more straightforward solution. The negative impact on development using this problem solving technique is the loss of a general view. The interaction between the subprojects can conflict and resources may not be shared at an optimum level. The integration of the subprojects can prove to be a big challenge. The English CarSim information extraction module was however developed using this scheme.

4.2.2 Linguistic analysis tools

As mentioned earlier in this master thesis document, CarSim takes advantage of available and powerful tools in order to parse texts and carry out lexical semantics interpretation.

- **Link Grammar** is syntactical parser that produces information about the relationship between the words in a sentence. It also determines the part-of-speech of the words. It is important to know that Link Grammar is not failsafe. It will not give meaningful data in all cases.
- **WordNet** is lexical database supplying semantic interpretations of words. Especially interesting is the hyponym and hypernym function. A hypernym of a word is a more general category of the word. An example is “horse” which has the hypernym “animal”. A hyponym on the other hand is a more specific possibility to describe the word. The word “plant” has the hyponyms “flower”, “tree”, and “orchids”. Consider the words car and vehicle that are both linked by the relations hypernym and hyponym. All cars are vehicles, but not all vehicles are cars: there are also trucks, motorcycles, bicycles etc.

Both Link Grammar and WordNet are written in the C programming language but the CarSim 2.0 project was developed using the Java programming language. There are major differences between the two languages and reusing a library written in C in an application written in Java is not straightforward. One of the foundations of Java is the ability referred to as “write once, run everywhere” principle. This means that the same Java program should be able to run on any platform may it be UNIX, Windows or Mac OS. To afford this feature Java is not compiled as a C program would be but it is transformed into an intermediate file conforming to the byte code format. This file is interpreted by the Java Virtual Machine (JVM). The JVM is a virtual computer that provides the abstraction between the pre-processed Java program and the underlying hardware platform and operating system. The byte codes can be thought of as the machine language of the JVM. Unlike a Java program, the JVM is not portable but must be implemented on each platform per se.

Luckily the people at SUN who invented the Java programming language have foreseen this problem and have therefore included support to reuse the old libraries written in native code. The middleware is called JNI which stands for Java Native Interface.

4.2.2.1 WordNet Java Native Interface

The procedures to reuse Link Grammar and WordNet are quite different. In the case of WordNet, Bou (2002) constructed a fairly general JNI that renders use of WordNet in a Java environment possible. The project is called the WordNet JNI and Bou uses the acronym WNJN.

WordNet was not to our knowledge implemented with the ability to be imported as a library package in mind therefore Bou has implemented an interface to WordNet using C and C++. The components of WNJN are one static library supporting either WordNet version 1.6 or 1.7. They are called wn16 and wn17 respectively. WordNet version 1.7 is not yet supported on the Win32 platform. These two projects are written in C and you can only use one at the time.

The "wnjn" dynamic shared library is also included in WNJN project. An API with the same name that hides the more difficult JNI and low level type of instructions on the Java side is also available.

Figure 4.1 shows a schematic picture of WNJN. The "WN C Library" can be considered to be a delimited API or a protocol to access the functions in WordNet.

The "WNJN C++ Shared Library" functions as a broker. It can communicate with WordNet, request and receive information about a word. The information is then transmitted through JNI to the Java side. On the Java side, Bou has implemented an API that masquerades the complex low level instructions dealing with the communication over the JNI interface, denoted as "WNJN Java glue files" in Figure 4.1. This means that once you have set up the WNJN correctly you should not have to bother with the intricacies of JNI functions and WordNet can be called just like ordinary Java functions.

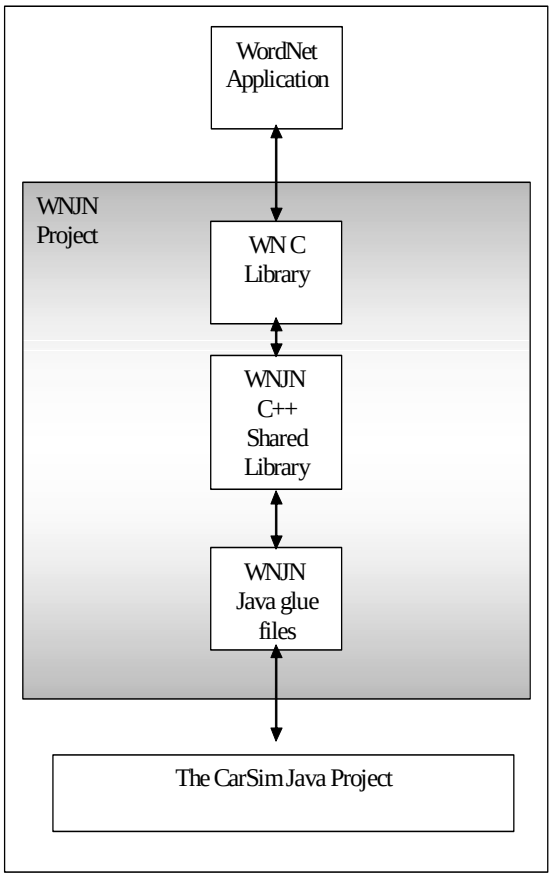

Figure 4.1 The principles of WordNet JNI, the WNJN

The WNJN was created with the Windows platform as a target. The CarSim project was however developed on the UNIX platform, more specifically the SOLARIS platform. The challenge at hand was to find a way of compiling the WNJN project on this platform. We chose gcc, the GNU C Compiler, and GNU make because they are free to use according to the GNU general public license.

A first attempt was made to translate the nmake build files that were shipped with the WNJN project. These files are created automatically and not intended to be read by users. The attempt failed and we decided to start from scratch and construct new makefiles. We tried to use WNJN as a dynamic shared library that statically links the WN library when trying to port the system to the UNIX platform. This conformed to WNJN original architecture on Windows. It did not work. A link error occurred when trying to run WNJN test program. After a lot of tedious work of researching this failure, we found that this was a common problem. Many reports on the Internet address this issue; none however had any solutions to the problem.

To solve the problem, we abandoned the WNJNI architecture and we replaced it with a solution based on two dynamically linked libraries. This was a complete success. The test program could be run and we could use WordNet in the CarSim application.

A lot could be learned from the test program but it did not fully meet the requirements in the CarSim system. A negative point to the WNJN project is that the low level JNI instructions are intermixed with display formatting instructions of the information received. They are not easily separated and it presents a problem because in CarSim it is not valuable to get all the data from a look up in WordNet compressed in one `String`. A list of the words is preferred because it is easier to manipulate. This was discovered very late in the progress of the integration of WordNet. We thought that the formatting was done by WordNet itself. That led to a lot of String manipulation to gather the information that indeed was important. This procedure was both difficult and time-consuming. When a lot of calls are made to WordNet this also leads to a negative impact on the response time as perceived by the user. A better way would be to rewrite partially the WNJN Java glue files and it should be taken into consideration to improve this feature in future releases of CarSim.

4.2.2.2 Link Grammar

Link Grammar is a syntactic parser that was written in C. The problem of getting access to Link Grammar features in CarSim is quite similar to the ones faced when integrating WordNet. In the case of Link Grammar however there were no projects available that solved the problem. The developers of Link Grammar had anticipated a need for integrating Link Grammar in to other products and hence they have developed an API called the Link Parser Application Program Interface. The Interface enables other applications to control the way Link Grammar works. The user can set things like parse time limit, the maximum links allowed and if words can be left unconnected. One can also pass a sentence to Link Grammar and receive the result, either formatted for display or packed into containers for further computer manipulations. We took advantage of it and we implemented a JNI bridge to be able to access the Link parser API from the Java side.

Figure 4.2 shows a schematic picture on how the JNI interface to Link Grammar works. On the Java side, we wrote a class called `LinkGrammarAPI` and its peer on the C side called `carsim_wrapper`. The names can be disputable but the idea is that from

CarSim application the class works as an agent for Link Grammar. In fact, from CarSim it appears as Link Grammar hence the name `LinkGrammarAPI`. The same is true on the C side were `carsim_wrapper` is viewed as any other C program using the Link Parser API to communicate with Link Grammar. To save time, the JNI solution did not implement the complete Link Parser API but only functions that bundled the appropriate Link Grammar instructions together and produced a result needed by CarSim. All the parse options are set as constants and cannot be altered during run-time. The options are optimized to get the best results from the type of texts CarSim deals with.

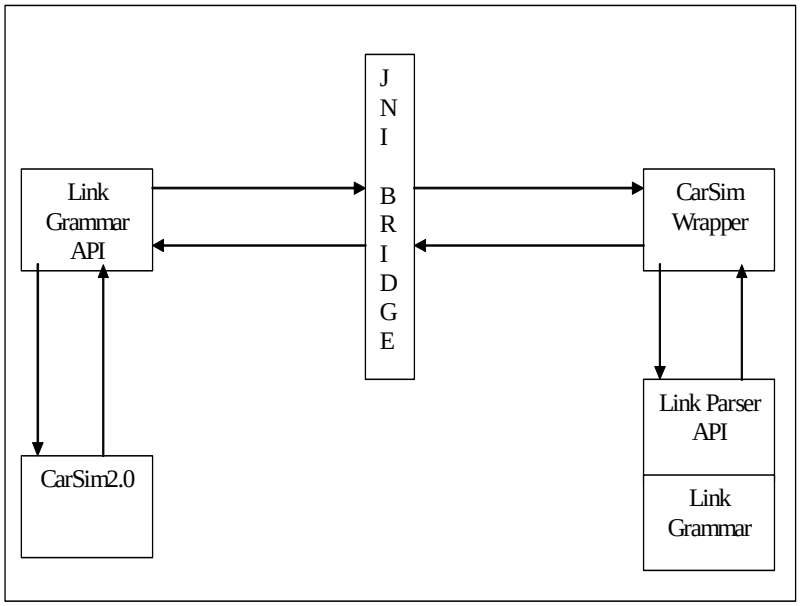

Figure 4.2 The JNI used to gain access to Link Grammar

In CarSim, the sentence is parsed in three passes. First, Link Grammar uses intricate rules and then if it fails to produce an output the rules are weakened. This increases the likelihood of an accurate parsing. Table 4.1 shows the options used in the different sets. `short_length` determines how long the links are allowed to be. If the `all_short_connectors` variable is set to true, then all connectors have length restrictions imposed on them. They can be no farther apart than `short_length`. `islands_ok` controls whether or not unconnected words are allowed. To allow the parser to assign some structure to a sentence even when it cannot fully interpret it, `allow_null` can be set to true. Basically, if the parser cannot parse a sentence normally, it tries to ignore one word in the sentence. It finds all the linkages it can, ignoring some words. If `panic_mode` is set to true then it enables the parser to parse even very long sentences quickly, but with considerably reduced accuracy. Link Grammar utilizes a system for assigning a cost to a linkage. This allows the parser to express preferences among the linkages it finds. It is controlled by the parameter `disjunct_cost`. `MAX_SENTENCE` is the largest number of words in one sentence that Link Grammar can parse. Table 4.1 shows the option sets which Link Grammar utilizes.

Name	Set 1 Value	Set 2 Value	Set 3 Value
verbosity	False	False	False
islands_ok	True	True	True
allow_null	True	True	True
panic_mode	False	False	False
max_sentence_length	1000	1000	MAX_SENTENCE
max_parse_time	30 s	30 s	60 s
linkage_limit	10 000	10 000	100
short_length	10	10	6
disjunct_cost	2	2	3
min_null_count	0	1	1
max_null_count	0	Length of sentence	MAX_SENTENCE
all_short_connectors	False	False	True

Table 4.1 Link Grammar options

The bundled methods supported in the `carsim_wrapper` class are:

```
Java_se_lth_cs_carsim_LinkGrammarAPI_allocStaticLinkGrammarStructs
Java_se_lth_cs_carsim_LinkGrammarAPI_freeStaticLinkGrammarStructs
Java_se_lth_cs_carsim_LinkGrammarAPI_parse
```

The peers in `LinkGrammarAPI` are declared:

```
private static native void allocStaticLinkGrammarStructs();
private native void freeStaticLinkGrammarStructs();
private native void parse(String input);
```

The `LinkGrammarAPI` class is implemented following the Singleton programming pattern. This means that measures are taken to assure that only one instance of the class is created and used. The reason for this is to limit resources used by the CarSim system. The library is loaded in a static initializer. Right after the system is loaded, the `allocStaticLinkGrammarStructs` method is called. It initializes some resources used by Link Grammar. It notably reads the dictionaries from files and creates the parse option sets. This is a time-consuming task but it is only done once during a run of CarSim, when `LinkGrammarAPI` is created. These resources are freed when the application shuts down. This is done in the destructor `finalize` statement.

To parse a text, the method `parse` is called and the sentence is passed as a parameter. Observe that Link Grammar only parses one sentence at a time. This means that the text has to be split up into sentences. It is not as easy as it sounds. Callback methods are used to pass information from the C side to the Java side. They are declared as follows.

```
  private void callBackSaveParsedWord(String saveString)
  private void callBackSaveLink
     (int LeftWordIndex, int RightWordIndex, String LinkLabel)
```

In order to make it possible to call these functions from the `carsim_wrapper` a certain procedure has to be followed. First a class look up is made to determine what type of Java object that should be called:

```
jclass classInstance;
classInstance = (*env)->GetObjectClass(env, obj);
```

The `env` and the `obj` resources are provided as parameters in the call of the C functions. `Env` which is of type `JNIEnv` accesses the virtual machine functionality by calling various functions exported through the `JNIEnv` interface pointer. `Obj` is of type jobject and is a reference to the object calling the C code.

The next step is to get a method ID. The native method calls the JNI function `GetMethodID`, which performs a lookup for the Java method in a given class. The lookup is based on the name of the method as well as the method signature. The method signature can be generated with the Java class file disassembler tool `javap`. We use `javap` with the flags *–s* and *–p* to create a file with the signatures of the target class. Because a class look up is a rather an expensive operation, a technique called caching of method IDs is employed. The effect of this is that the class look up is only made once during an execution and the time spent in this code is reduced.

```
mid1_s = (*env)->GetMethodID
 (env, cls,"callBackSaveParsedWord", (Ljava/lang/String;)V" );

mid2_s = (*env)->GetMethodID
  (env, cls, "callBackSaveLink", "(IILjava/lang/String;)V" );
```

Java objects must be prevented from being unloaded from the Java Virtual Machine. This is accomplished by keeping a reference to the Java object in a static variable. The reference must be kept in a global reference to make it work. The global reference must be freed when it is not used anymore.

```
jclass cls1 = (*env)->GetObjectClass(env, obj);
cls = (*env)->NewGlobalRef(env, cls1);
```

The variable referring to the looked up method ID is of type `jmethodID`. Lastly, the native method calls the JNI function `CallVoidMethod`. The `CallVoidMethod` function invokes an instance method that has a `void` return type. You pass the object, method ID, and the actual arguments to `CallVoidMethod`.

```
(*env)->CallVoidMethod( env, obj, mid1_s,
                        (*env)->NewStringUTF(env, a_parsed_word) );
```

And now the information about the structure of the sentence is successfully passed to the Java side and can be further manipulated.

4.2.3 Splitting texts up into sentences

The Link Grammar parser only works on a sentence at a time. This means that a whole text has to be split up into sentences before it is submitted to Link Grammar. It may seem a simple task to split a text up into sentences however it is not. The punctuations

of for example dates and abbreviations as well as the capital letters in names make it more difficult to determine the beginning and end of a sentence.

We searched resources and we decided to use the `BreakIterator` class. We choose it because it is a part of the Java Development Kit (JDK). `BreakIterator` is a convenient class to use when manipulating texts in natural language. It is very versatile and implements methods for finding the location of boundaries in text. Instances of `BreakIterator` maintain a current position and scan over text returning the index of characters where boundaries occur. Internally, `BreakIterator` scans text using a `CharacterIterator`, and is thus able to scan text held by any object implementing that protocol. Examples on the usage of `BreakIterator` are splitting up texts into characters, words and lines. In CarSim, the interesting feature was to split a text into sentences. When this was implemented, we detected a flaw in the `BreakIterator` algorithm. In abbreviations such as “U. S. State Route 7” and “Mr. Johnson” there is a period and `BreakIterator` wrongly interprets this as a full stop. The texts scanned by CarSim contain a lot of abbreviations and `BreakIterator` fails miserably. However, we found the rest of the functionality of `BreakIterator` useful so we made the decision to add a post-processing stage in the split. Example 4.1 and Example 4.2 illustrate the error.

Example 4.1 A first example of incorrect tokenization made by `BreakIterator`

Sentence: “I was just talking to Mr. Johnson about the problem”
After split: “I was just talking to Mr.”
“Johnson about the problem”

Example 4.2 A second example of incorrect tokenization made by `BreakIterator`

Sentence: “I was driving on U. S. 13 when it started to rain”
After split: “I was driving on U. S.”
“13 when it started to rain”

The functionality of the post processing stage includes the detection of abbreviations at the end of a sentence previously split up using `BreakIterator`. When such a sentence is found the next sentence, it is concatenated to the next sentence. This newly constructed sentence is then set as active sentence; the next one to be examined.

The detection of abbreviations is done by isolating the last word of a sentence and checking it with regular expressions to see whether or not it matches the desired patterns. The patterns used in this scheme are gathered from Greffenstette and Tapanainen (1994), adapted to our specific needs. In some cases, we constructed completely new patterns. Table 4.2 shows regular expressions useful in determining an abbreviation.

Fractions, dates, decimal numbers and statements with the “%” sign also pose problems. We did not take them into consideration because they are correctly interpreted by `BreakIterator`. Using this method, CarSim achieved a 100 percent success on the corpus. All the twenty-three texts in the corpus were split up into sentences correctly.

Rule #	Regular Expression	Description
1	`([A-H,J-Z,a-z]\\.([A-Z,a-z,0-9]\\.)+)`	This pattern tries to match a sequence of letter-period-letter-period. The letter "I" is not handled because it could refer to a non abbreviation construct. Examples: "U. S." or "m.p.h."
2	`([A-Z][bcdfghj-np-tvxz]+)`	The regular expression describes a pattern of capital letter followed by a sequence of consonants followed by a period. Examples: "Mr." and "Mrs."

Table 4.2 Regular expressions used to find abbreviations

The newly released version of the Java SDK 1.4.0 has modified the `BreakIterator` class. This renders the workaround solution that cleans up abbreviations obsolete. The `BreakIterator` can now handle some abbreviations such as "U.S. 13" and "m.p.h.". It still cannot correctly interpret abbreviations as "Mr. Jones" and "U. S. 13". In the last example, there is a space character between "U." and "S.". We designed a new regular expression to adapt to the new behavior of `BreakIterator` (Table 4.3).

Rule #	Regular Expression	Description
1	`([A-Z,a-z]\\.(\\s[A-Z,a-z,0-9]\\.)*)`	The only difference is that a space character "`\\s`" is added. The group `([A-Z,a-z,0-9]\\.)*)` has been converted into `(\\s[A-Z,a-z,0-9]\\.)*)` Examples: "U. S. 13"

Table 4.3 Updated regular expressions used to find abbreviations

4.2.4 Building a dictionary of words using WordNet

The class `WNDictionary` contains a dictionary of the words that are used in this project. Most of the words are looked up in WordNet. The program calls WordNet to find the hyponym trees of different selected super words. They are saved as `String` objects. The strings that are created in `WNDictionary` are configured in a way that makes it possible to do pattern matching with regular expressions without changing anything. The structure conforms to a standard: left parenthesis, word, vertical bar, word, vertical bar, and so on. The string ends with a right parenthesis. Table 4.4 shows the words used by the `WNDictionary` class.

The result from WordNet is not always the best in terms of extraction success and we had to add some words manually to `WNDictionary`. When looking up words in WordNet there are some parameters to ponder. As mentioned in section 3.4.1, a WordNet entry comprises a part of speech, a sense and a super word. The super word is the word that WordNet uses as a starting point when searching for hyponym trees. One other operation that WordNet does not do is to conjugate verbs. The solution to bypass

this problem is to apply a morphological look up on the list of verbs returned from WordNet. Each verb in the list is used as a key in a `HashSet` and the value contains the verbs conjugated forms. The values in the `HashSet` are hard coded. Both the conjugated forms and the verb looked up in the WordNet database is saved in a String in `WNDictionary`.

Some words are also removed from the hyponym trees. This is done because they are ambiguous (have two or several meanings). The words we removed are shown in Table 4.4. For example, "*obstruction*" is a super word and the word "*stop*" is included in the hyponym tree resulting from a look up. If "*stop*" is not removed it will be detected twice in the CarSim system. Once when extracting obstructions and once when detecting movements of vehicles.

The next task is to find words describing the 3D object BLOCK which symbolizes heavy vehicles. The approach is a little different. As can be seen in Table 4.5, truck, bus and trailer are looked up in WordNet and the resulting hyponym trees are put together in the same string.

An explanation of the heads in the table follows.

Super word	The top word that is looked up in WordNet.
Sense	Which sense that is looked at for the specific word
POS	Part of speech.
Hard code	Yes, if the word is hard coded.
Remove	Which word that was removed.
Add	Which word that WordNet did not have in its word bank.

Super Word	Sense	POS	Hard coded	Remove	Add
Road	1	Noun	No		Interstate
Initial direction			Yes		
Bend	1, 3	Noun	No		
Intersection	2	Noun	No		
Sign	2, 4	Noun	No		Stop sign
Light	13	Noun	No		
Obstruction	1	Noun	No	Turnpike, stop,	Tree, curb, kerb, ditch, embankment
Hit	2, 3	Verb	No		Drive into
Car	1	Noun	No	Bus	Patrol wagon, paddy wagon, Police car
Bus	1	Noun	No		
Trailer	3	Noun	No		
Truck	1	Noun	No		
Traveling	1	Noun	No	Turn, way	En route, traveled
Move, veered			Yes		
Turn			Yes		
Overtake			Yes		
Stop			Yes		

Table 4.4 Words that the system uses

3D Object	Remove	Add
Block	Patrol wagon, paddy wagon, police wagon	Tractor-semi trailer

Table 4.5 The special modifications for heavy vehicles

4.2.5 Detecting the road configuration

The road configuration is the physic shape of the road where the accident occurs. CarSim can handle four types of roads: `straightroad`, `left_turn`, `right_turn` and `crossroad`. There are some restraints on these road types. A `straightroad` has an east to west orientation. A `left_turn` has an east to north orientation and a `right_turn` has an east to south orientation. The actual orientation of the road and hence the initial directions of vehicles have to be translated into values, which the graphical module can display. This is explained further later on.

The extraction of road configuration uses regular expression and WordNet. The principle is fairly simple. Two categories of words are used: intersections and bends. The hyponym trees of these words are looked up in WordNet and used in regular expressions. The regular expression then scans the text and reports any findings. First, the text is searched for words of intersection type. If that fails, the scan moves on to bend types. If that fails a straight road is assumed. Observe if the first two pattern matching techniques fail, the information extraction is done No further work will be performed. As CarSim makes use of directed information extraction, any text, may it be a poem, should produce a accident description. To make this feasible, if no intersections or bends are found, a default straight road is added.

The information extracted about road configuration is added to the template as a child node to the static objects structure.

4.2.6 Extracting the names of the roads

The task of extracting the names of roads makes heavy use of regular expressions. The method is fairly similar to that used when extracting road configuration but is more complex.

Names of roads in the United States follow a certain pattern that can be used when trying to locate them in a text. Examples are “Pennsylvania Turnpike”, “State Route 30A”, “U.S. 160” and “Farm to Market Road”. Each name begins with a capital letter. In each name phrase, a word depicting some type of road word like “Turnpike”, “Route” and “Road” is usually present in the examples of our corpus. Another family of road names is the abbreviations. They comprise a number of capital letters and an integer. “U.S. 160” and “I-22” are typical members of this family.

The trick to extract these involves the usage of WordNet and regular expressions. WordNet is used to look up words to target and match road words as discussed above. Hyponyms of word “road” makes up a lexicon that can be used to build regular expression patterns. The regular expression pattern used to find names of roads from the first family are constructed to take in to consideration different types of combinations.

Rule #	Regular expression	Description
1	`([A-Z][A-Za-z0-9\\.]*\\s?(to\\s)?)*`	This regular expression is designed to find constructs with a word starting with a capital letter and then a space character followed by the word “to” then again a space character. Example: “Farm to Market Road”
2	`(\\([A-Z]\\w+\\)\\s)?`	This pattern matches a construct enclosed in parenthesis and starting with a capital letter. After the parenthesis expression there must be a space character. Example: “(Florida)” in “Levy County (Florida) Road C-32”
3	`\\s([A-Z]\\w+\\s?)?`	A word with a capital letter in the beginning and with a starting space character and an optional trailing space character. Example: “State” in “State Route 7”
4	`(\\d+\\w?)?`	A sequence of integers, ending with optional letters. Example: “30A” in “State Route 30A”
5	`(\\w-\\d+)?`	The pattern matches a group of integers followed by a dash and ending group of letters. Example: “I-95”

Table 4.6 Regular Expression used to find road named from the first family

Rule #	Regular expression	Description
1	`[A-Z]\\.?`	This part of the pattern matches one capital letter followed by a optional dot. Example: “U.” as in “U. S. 13”
2	`(\\s?[A-Z]\\.?)*`	One optional space character followed by one character in upper case and one optional ending dot. Observe that the whole group must exist zero or more times. Example: “S.” as in “U. S. 13”
3	`(\\s|-)\\d+`	This group starts either with one space character or a dash. Must be followed by one or more integers. Example: “-2” as in “IS-2”
4	`(\\s|,|\\.|\\?|\\!)`	Here just one character is sought but it has to be a space character or a comma sign or a dot or a question mark or a exclamation mark. Example: “?” as in “U. S. 13?”

Table 4.7 Regular Expression used to find road named from the second family

Table 4.6 shows the regular expressions used to find road names from the first family. These make up a pattern that finds most of the road names that are constructed with a phrase containing some type of road word. The subparts are concatenated as

follows 1 + 2 + “a list of road words looked up using WordNet” + 3 + 4 + 5. This pattern is used in the first pass of the text. In the second pass, another pattern is used. Table 4.7 lists the subcomponents.

This pattern is created by concatenating the subpatterns in the order they are presented in Table 4.7. With this pattern, CarSim can detect constructs such as “U.S. 13”, “U. S. 13”, and “II-2” which are members of the second family of road names as described above. In most cases, successful matches are found but the second pattern introduces an error, which cannot easily be avoided. Consider the sentence in Example 4.3.

Example 4.3

“On March 2, 1999, a 1979 Motor Coach Industries MC-9, 47-passenger charter motorcoach, owned and operated by Shuttle Jack, Inc., (Shuttle Jack) of Santa Fe, New Mexico departed the Santa Fe Ski Basin, carrying the driver, 2 adult chaperons, and 34 middle school-age children.”

The phrase “MC-9,” in this sentence would be detected by the second regular expression pattern as a valid name of a road. This is wrong but only one occurrence has been observed in the corpus used by the CarSim project. Many road names belonging to the second family are however correctly detected and the conclusion is that this pattern does more good than harm.

In some other cases, the system does not detect the road names properly. One reason is that some summaries do not mention the road names. One other reason is the inadequacy of our regular expressions.

There is also the problem of connecting a dynamic vehicle to the road on which it is driving. The name of the road is an attribute to the Dynamic vehicle construct in the template and is called `startSign`. No co-referencing techniques are employed and more vehicles are detected than actually present. A personal pronoun such as “it” that co-refers to a noun is interpreted as a unique vehicle. Hence, these references are not connected to a road name.

4.2.7 Extracting the static objects

Static objects represent a category containing the road configuration and the non-moving entities that interact with vehicles in the accident. Examples of the static objects are trees and signs. Any type of static obstacles belongs to this category. The objects that CarSim includes are `road`, `tree`, `sign`, `trafficLight`, and `levelCrossing`.

One can argue about the logic of this selection. The subject of discussion is whether this set is consistent. The road configuration is static but cannot participate directly in an accident situation. It is essential to the animation though as it is the base of the view upon every other object is placed. The issues of `levelCrossing` are very similar to that of `road`. Maybe it would have been a better solution to break out these objects and divide them in two or more categories to minimize confusion. However one of the goals of CarSim is to keep the template representation as short and simple as possible and CarSim was designed to be compatible to the old template structure.

In addition to these objects, the new CarSim information extraction module can detect a group called obstacles. The obstacles cover a wider range of possible entities that are capable of direct interaction in an accident situation. The vision is that as an

extension to the existent static objects two new categories are added: vertical obstacles and horizontal obstacles. Examples of the first groups are different kinds of signs, streetlights. Fences, embankments and rails would fit into the horizontal obstacle category. The implementation of the information extraction of these is done however; the template structure cannot represent these objects.

Streetlights are represented as `trafficLight`. All obstacles will appear as `tree`. Different kind of signposts are entered in the `sign` category. Note that the `sign` category can currently only represent stop signs, although it may seem as a group that can reflect a wider variety of signs.

The technique used to extract static objects is to combine the pattern matching strength of regular expressions and the versatility of WordNet in the respect of extracting groups of words that are similar and describes the same meaning. This technique was applied when searching for extraction of road configuration and the names of roads. To get a list of words that fit each category a survey of WordNet was made and the proper hypernyms was selected.

The base algorithm introduces errors because some words are missing and some words that do not fit the category satisfyingly are found using WordNet. The hyponym trees delivered from WordNet are inspected and some words will be removed while some other words will be added. The reason for this is achieving the highest throughput possible in regards of correctly interpreted texts.

The slightly altered hyponym tree for each group is then combined with some regular expression constructs. These are added to make sure that only a whole word is matched, not a part of a word. This is done by matching a white space character in the beginning and any of “,”, “.”, “?”, “!” or a white space character at the end. The whole group is extracted so these characters have to be trimmed away. A static method called `rightTrim` in the class `Util` is provided to perform this operation.

We also implemented a `RegExWordFinder` class that provides tools for finding information in the vicinity of a matched phrase. It works in conjunction with a class called `RegularExpression`, which actually does nothing but wraps the Java standard classes `Pattern` and `Matcher`. Consider the following example:

“The driver veered to the left of the road and crashed into a tree.”

The thought is that when the word “tree” is found in the sentence there may be additional clues to extract about the location of the tree. Here it is obviously stated that the tree is located on the left hand side of the road. In CarSim, this is handled by the class `RegExWordFinder`, from the index of the matched word, in this case “tree”. `RegExWordFinder` makes it possible to inspect the words on either side of the match. In the example, there are no words to the right of the match but many to the left. The interesting word that is desired to be extracted is “left”. The `RegExWordFinder` is setup to search for the words “left” or “right”. It tries to match these words against the words found directly adjacent to the match on either side. If it finds a match this is reported and the search is over. In case of failure, the second nearest two words are tried and so on. If no matches are found when the whole sentence is checked, a default location will be added. Furthermore, `RegExWordFinder` does not read past a comma sign but ends the search. This is done to prevent a case where a false location is added to the information extracted about the “tree”. This idea is based on the locality of information. This means that the information about the “tree” should be located close to the words location in the sentence hence the search from the word outwards towards its boundaries.

Internally the `RegExWordFinder` uses regular expressions and the class `BreakIterator`. The `BreakIterator` is used to step through the sentence one word at a time, beginning from the match and outwards towards the sentence boundaries. Regular expressions are constructed using the words specified in the search criteria, for example “left” and “right”. The regular expressions in `RegExWordFinder` are then applied to match the words provided from the `BreakIterator`.

Example 4.4

```
RegularExpression myRegEx = new RegularExpression("tree", sentence);
while (myRegEx.hasNextMatch()) {
    Tree myTree = new Tree(RegExParser.TREE, myRegEx.nextMatch());
    if (myRegEx.hasAdjacentMatchingWord("left"))
           put tree on the left hand side of the road
    else
           put tree on the right hand side of the road}
```

Example 4.4 is somewhat simplified but shows how it is intended to be used. The use of `RegExWordFinder` is hidden inside the class `RegularExpression` but the principle of how this method is used should be apparent.

In the case of obstacles, it is used to get a location of the object. The color of a traffic light is also extracted using this method. To save the information extracted, three container classes were implemented: `Road`, `StaticCrashableObject` and Light. As discussed above a road configuration differs a lot from the other types of static objects and is therefore represented by a separate class. Tree, sign and level Crossing share the same target attributes extracted from the text and are saved in the class `StaticCrashableObject`. Traffic lights have very similar attributes to tree, sign and level Crossing but have the additional parameter of color. A container class called `Light` was created to hold data about this object. `Light` inherits from `StaticCrashableObject` and adds only constructs for handling of operations tied to the color attribute. The type of the object is indicated by a member variable in the `StaticCrashableObject` class.

4.2.8 Integrating the detection of collisions

As described above, the CarSim project includes the implementation of a JNI interface to Link Grammar. It does not however include the work of collision detection. This work was carried out by Torbjörn Ekman and Anders Nilsson (2002), two PhD employees of the Computer Science department at the Lund Institute of Technology. The collision detection makes use of the Link Grammar JNI interface. It first tries to locate the interesting sentences by searching crash verbs with regular expressions. Crash verbs are commonly found in language constructs to describe an accident situation.

When these sentences are singled out, they are passed to Link Grammar. With the information provided about the sentence, a technique is applied which follows certain types of links. This enables to us to connect the actor and the victim to the crash verb. Observe that this solution does not cover a sentence using the passive form,

We used an expansion of the technique to connect a verb to the actor and the victim to determine the movement of the vehicles. It involves the vehicle directions so the model has to be adapted to extract additional information. New link combinations were added to achieve this. We explain this later in the report.

4.2.9 Mapping information into a predetermined accident structure

The collision detection enables the system to extract information about accidents. The collision detection system gathers the data about an accident situation and stores this data in a structure called `VerbPhrase. It` holds information about a subject-verb-object chain. It is in some ways a wrapper to the `Collision` class and in fact, `VerbPhrase` inherits from it. Anders Nilsson and Torbjörn Ekman implemented the `Collision` class but the interface provided did not suite the needs in the CarSim project.

The `VerbPhrase` class manipulates the information provided from the collision detection and presents it in way that makes it simpler to use. It provides the means for extracting what parts of a vehicle are involved in the accident. It determines whether it is a frontal, a rear or a side collision. `VerbPhrase` also connects actors to dynamic objects and victims to either dynamic or static objects. `VerbPhrase` has additional functionalities that will be further explained in section 4.2.10.

The extraction of the vehicle parts that participates in the accident uses subject and object modifiers. Modifiers are dependents of the head nouns such as adjectives as for instance “old” and “blue” in the sentence “The old blue car struck a tree”. In addition to the subject and object, the collision detection extracts some of their modifiers. `VerbPhrase` puts the modifiers in a list for future reference. CarSim searches the words *left*, *right*, *front* and *back* as well as variations of them.

The `VerbPhrase` class itself is used as a container for the accident information and implements means to output the data in the old template format.

4.2.10 Detecting movement

The strategy to determine the vehicle movements described in the text is similar to the collision detection. Instead of crash verbs, any verb that could initiate a movement is used to locate sentences of interest. Examples on such verbs are driving and traveling. In CarSim, the movements have been divided into five subcategories: travel, overtake, stop, turn, and sideways movements. These groups are filled with words using WordNet. The words in the travel group use “travel” as a hypernym. The other groups did not fit well in the WordNet hierarchy so we determine them specifically for the program. The sideways movement group contains words such as *move* and *veer*. These words are usually used in conjunction with a direction. The same is true with the words in the turn category. As a contrast to these, the words in the groups stop and overtake almost never take a direction as a modifier.

The collision detection system uses the class `VerbPhrase` again to represent the information it gathers. Compared with the collision detection, the rules are weakened. The collision detection extracts the subjects and the objects. The direction detection only considers the subjects and if possible the objects. The examination starts with the location of the subject in the sentence. It inspects each word to the right of the subject and it tries to match words to determine whether it describes a movement to the left or the right. The detection is rather aimed at change in the movement than in the usual sense of movement. Compared with the collision detection, we added some new link chains to interpret more sentences correctly and return a result in more cases.

The data produced in this step is then used to figure out the events that a vehicle experience according to the text.

4.2.11 Linking dynamic and static objects to accident frames

When all information is extracted, CarSim fills the accident frames with the dynamic and static objects. It applies a resolution algorithm to the information gathered from the collision detection system. The algorithm identifies dynamic objects from subjects and objects. It maps each entity it identifies to an instance of the class `DynamicObject` that holds data about the vehicle. Some of the `DynamicObjects` have already been created in the direction detection phase. The algorithm checks whether the object already exists if not creates an instance of it. This is notably the case for pronouns.

When a new dynamic object is found it is significant to know its membership. As said before there are two different dynamic types: cars and trucks. The algorithm tries to match the name of the new dynamic object to CAR elements contained in `WNDictionary`. If the match succeeds, the dynamic object type will be set to `car` otherwise to `truck`. Figure 4.3 shows how a car and a truck look like in the system.

The object in a subject-verb-object chain can be a static object for instance when a car hits a tree. The `DynamicObject` class binds the actors and victims in an accident to the appropriate dynamic or static objects.

In addition to resolving dynamic and static objects, an event list is build for each dynamic object. Events that can be detected are `driving_forward`, `stop`, `overtake`, `turn_left`, `turn_right`, `change_lane_left` and `change_lane_right`. A separate class, called `Event`, has been implemented that manages all operations necessary when dealing with events.

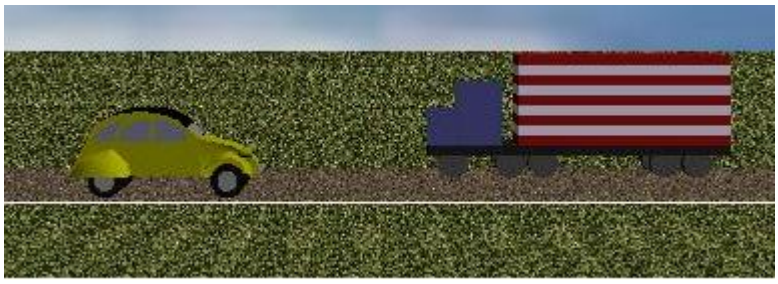

Figure 4.3 How the car and truck looks in the CarSim system

An example will be discussed in the text below to illustrate the procedure. First, the collision detection system analyzes the sentence in the collision detection mode. It finds the verb “crashed” and identifies the subject and object of this verb: “car” and “tree”. The `VerbPhrase` object saves this information and is equivalent to the accident object. Then, the collision detection system is run in the movement detection mode and detects the verb “departed”. Again, a search for subject and object is carried out and it detects the subject “car”. Remember that when in the movement detection mode, the subject suffices for a successful extraction. This results in a new `VerbPhrase` object. In the next steps of this example, we assume that no `DynamicObject` have been created earlier.

The two `VerbPhrase's` are scanned. It investigates the first `VerbPhrase` and creates a new `DynamicObject` from the subject. It finds that the grammatical object is a static object that already exists as an object in the static object list. The `VerbPhrase` describes a collision and it is saved. The `VerbPhrase` (which represents an accident) are assigned with one reference to the new `DynamicObject` and one reference to the

static object. Now the system has a description of an accident and references to the participants. The algorithm investigates the second `VerbPhrase`. Since the subject nouns of both sentences are exactly equal, the system will find the dynamic object extracted earlier and determine that they are referring to the same entity.

The system then tries to find out if any valid events can be found in the `VerbPhrase`. The verb *depart* is in a category of words that is likely to take a direction as a modifier. The sentence is searched to the right of the verb for a clue indicating in which direction the movement takes place. The word “left” is found and this generates an event of type `change_lane_left`. This is not completely correct as `change_lane_left` event is intended to be used in an overtake situation but it is the closest event that the graphical module can interpret. However, the accident planner is intelligent enough to render the scene of the collision in a realistic way. Note that the word “left” will also be used when extracting the static object “tree”. This will assure valid coordinates of the location of the “tree” in the animation.

Finally, since the second `VerbPhrase` does not describe a collision and we delete it. Example 4.5 and Figure 4.4 show an overview of the solution to the problem.

Example 4.5

Sentence: “The car departed the left side of the roadway and then crashed into a tree.”

Subject-verb-object chain: car-crashed-tree

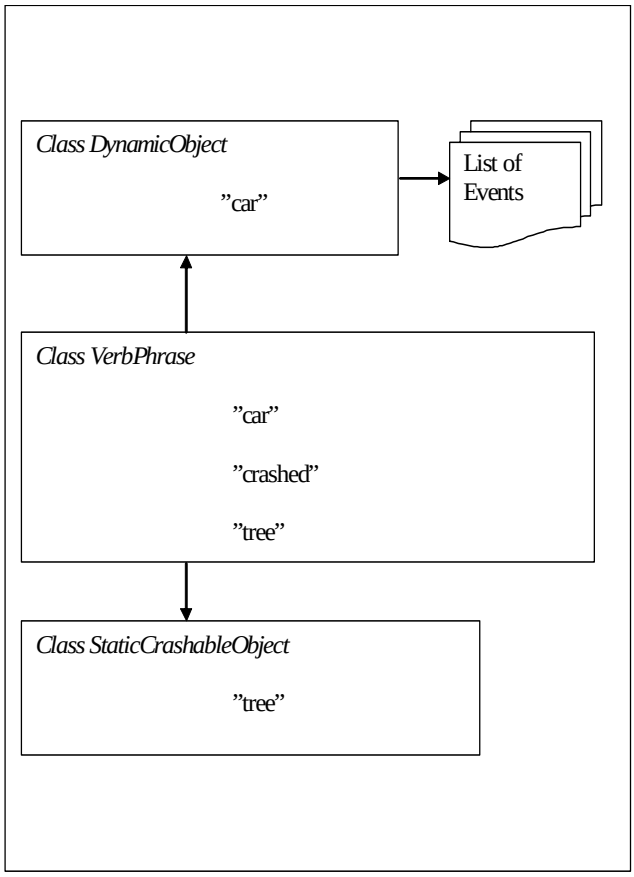

Figure 4.4 Generated objects

4.2.12 Initial directions

With the knowledge of the subject-verb-object chain and their individual locations in the sentence, this task investigates in what direction the vehicles (dynamic objects) are moving.

All the dynamic objects, which are detected and created, need an initial direction. This is important because the simulator requires an initial direction to know where the dynamic objects start. If the direction is not set, the system will throw an exception. At this point in the program, the accidents are detected and the involved dynamic objects are set. The detection algorithm has the following steps:

- Try to detect traveling words.
- Find a subject and a direction.
- If only a subject and a traveling word are detected, try to find direction with pattern matching.
- If a dynamic object is detected, check if it is already in the list of existing objects, otherwise create a new one.
- Check if all the dynamic objects have an initial direction, otherwise try to set one using rules. If that fails, set a default direction.

The detection of directions is similar to the detection of the subject, crash verb, and object set. The class `Template` contains the direction settings. The first task consists of the detection of traveling words in the accident description. It uses pattern-matching techniques. The system obtained the traveling words from WordNet and stored them in the class `WNDictionary`.

The second task involves the detection of the subject and the direction. It uses the links produced by Link Grammar. Sometimes, the links lead to a word that is not a direction. These are singled out using pattern-matching techniques. If the word is not a direction, it will be replaced by an empty `String`. Later in the program, that string will be set to a default initial direction.

Sometimes, the system only detects a subject and the traveling verb. The system then tries to find a direction with regular expressions. The system looks at the words following the traveling verb in the sentence. It scans the words until it matches a direction or a comma. This is the third task.

In the fourth task, the program searches the already detected dynamic objects and checks if the subject found in task two refers to the same entity. If a match succeeds, then the initial direction found in task two is set to this dynamic object. The entity represented by the subject does not have to be involved in an accident. It can simply be a vehicle driving on the road. In that case, a new dynamic object is created and the initial direction is set.

The last task checks all the dynamic objects to ensure that an initial direction has been set. At this point, there can only be two kinds of directions: the first one is a valid direction or the second one is an empty `String`. The system scans the objects to replace the empty `Strings`. It sets initial direction values using heuristic rules contained in the `Rulemap` class. The rules consider the extracted information for the other vehicle in the dynamic object list:

- The part of the first vehicle directly involved in the accident: `rear, left_side, right_side, front` or `unknown`.
- The part of the second vehicle directly involved in the accident, the same as above.
- Road configurations: `straightroad, turn_right, turn_left` or `crossing`.
- The initial direction of the second vehicle: the four cardinal points.

The second dynamic object may have no initial direction either (an empty `String`). This gives a fifth potential value for the initial direction.

The algorithm maps each combination of values to an initial direction. The number of combinations of these different variables is large and we only give an example to show the principles of the decision algorithm. Consider the following situation:

- The first vehicle's initial direction string is empty.
- The second vehicle's initial direction is `west`.
- The part of the first vehicle directly involved in the accident is `front`.
- The part of the first vehicle directly involved in the accident is `front`.
- The road configuration is `straightroad`.

In this case, the system will set the initial direction to `east` for the first vehicle. When looking at the corpus, the most common accident on a straight road is a front-rear collision. We took this into consideration when creating the rules. The conclusion is that if the system has not been able to detect what kind of vehicle parts have been directly involved in the accident, it tries to set possible directions matching a front-rear collision. The system cannot always decide the direction using the `RuleMap`. In this case, the system will set the initial direction to the default value of east.

4.2.13 Direction wrapper

The CarSim visualization module can only simulate straight roads from east to west and not from south to north. In the same way, it can only visualize left turns from east to north and right turns from east to south. To avoid situations where a road cannot be correctly displayed; the class `DirectionWrapper` translates the orientation to a legitimate case. Of course, it would work without the translation, but vehicles that have an initial direction with the north value on straight road will start orthogonal to the road. `DirectionWrapper` adjusts the template before it is passed to the simulator. The system looks at the road type parameters and the vehicles initial direction. It checks all the dynamic objects. Table 4.8 below shows which parameters the class uses and how they are changed.

Road type	Which road type for this simulation.
Initial direction	The initial direction on the dynamic object.
Change to	The new initial direction on the dynamic object.

Road type	Initial direction	Change to
Straight road	North	East
Straight road	South	West
Turn left	North	West
Turn left	South	West
Turn right	North	East
Turn right	South	East

Table 4.8 Describing how the system changes initial directions.

4.2.14 Output data according to template format

When we started our project, we developed the output layer with the CarSim 1.0 template format to be compatible with the visualization module. We rewrote it in XML when Bastian Schulz (2002) started the development of the new graphical module.

The first output layer was integrated in each object: The objects could present the data they contained. The XML output layer uses a separate class. This enables us to use the two output formats in parallel.

We first tried to use SUNs standard development classes to write the XML output layer. These classes output XML tags without blank separators. This was not acceptable because we wanted to give the user possibility to proofread the XML output. A file with no blank separators is extremely difficult to read. We decided to use the Xerces package instead, which is part of the XML manipulating tools from Apache. Xerces provides tools for easy formatting.

We wrote the class `XMLPrint` that constructs the XML output of CarSim project. It uses `xercesImpl.jar` and `xmlParserAPIs.jar` from the Xerces package. This class reads the object structure that represents the information gathered from the texts, format it, and write it to a character stream. The new CarSim graphical user interface handles the file input/output operations. The GUI does not have any options to receive the data in the old template format but the information extraction module can deliver it if desired.

Example 4.6 shows an example of a minimal template in XML format. The first line is just a description of the format in the file. The second line indicates a DTD Document Type Definition containing rules of the format of the XML file. The DTD is presented in appendix C. It is followed by the data gathered from texts.

Example 4.6 Minimal template

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE accident SYSTEM "accident.dtd">
<accident>
    <staticObjects>
                <road kind="straightroad"/>
                <tree id=”tree1”>
                      <coords x="-5.0" y="5.0"/>
                </tree>
    </staticObjects>
    <dynamicObjects>
                <vehicle id="truck1" initDirection="south"
          kind="truck">
                <eventChain>
                      <event kind="driving_forward"/>
                </eventChain>
                </vehicle>
    </dynamicObjects>
    <collisions>
          <collision>
                <actor id="truck1" side="unknown"/>
                <victim id="tree1" side="unknown"/>
          </collision>
    </collisions>
</accident>
```

4.2.15 Development GUI

The information extraction task of the first version of CarSim was written in Prolog with a command-line interface. The new information extraction engine is written in Java. With Java, a whole new range of possibilities are presented, one of them is to create a graphical user interface GUI, which is preferable, compared to a command-line interface. It is easier to interact with a descriptive GUI. In CarSim, it was decided to create a GUI. As a GUI can take a lot of time for the inexperienced programmer to create, the graphical module of the old system was taken as a model. This was however written with the Java AWT components, in the new CarSim Java Swing components were used but the basics structure were inherited from the old one. This GUI was rendered obsolete when the information extraction engine and the graphical module was combined and controlled from one new user interface described in section 4.2.16 but the original GUI was very helpful developing the project.

4.2.16 Graphical User Interface

Bastian Schulz (2002) created a new GUI (Graphical User Interface) for CarSim. This GUI is based on windows, icons, menus, and pointers (WIMP). The new GUI gives a sharper and more professional look to CarSim. It integrates the information extraction and the visualization modules.

This new version has four drop-down menus. The first one gives the user the ability to choose language. The languages available are English, German, Spanish, French and Swedish. It also contains an exit item. The second menu gives the user the ability to create and delete files. The user can write a new text, save it to file and parse it with the information extraction module. The third drop-down menu enables the user to create XML files, save or delete them. It also enables validation of the XML code, check if it conforms to the DTD (Document Type Definition). The last drop-down menu is called 3D. It starts the simulation. Demonstrations and configuration can be accessed from this menu as well.

Figure 4.5 below shows a screenshot of the Graphic User Interface.

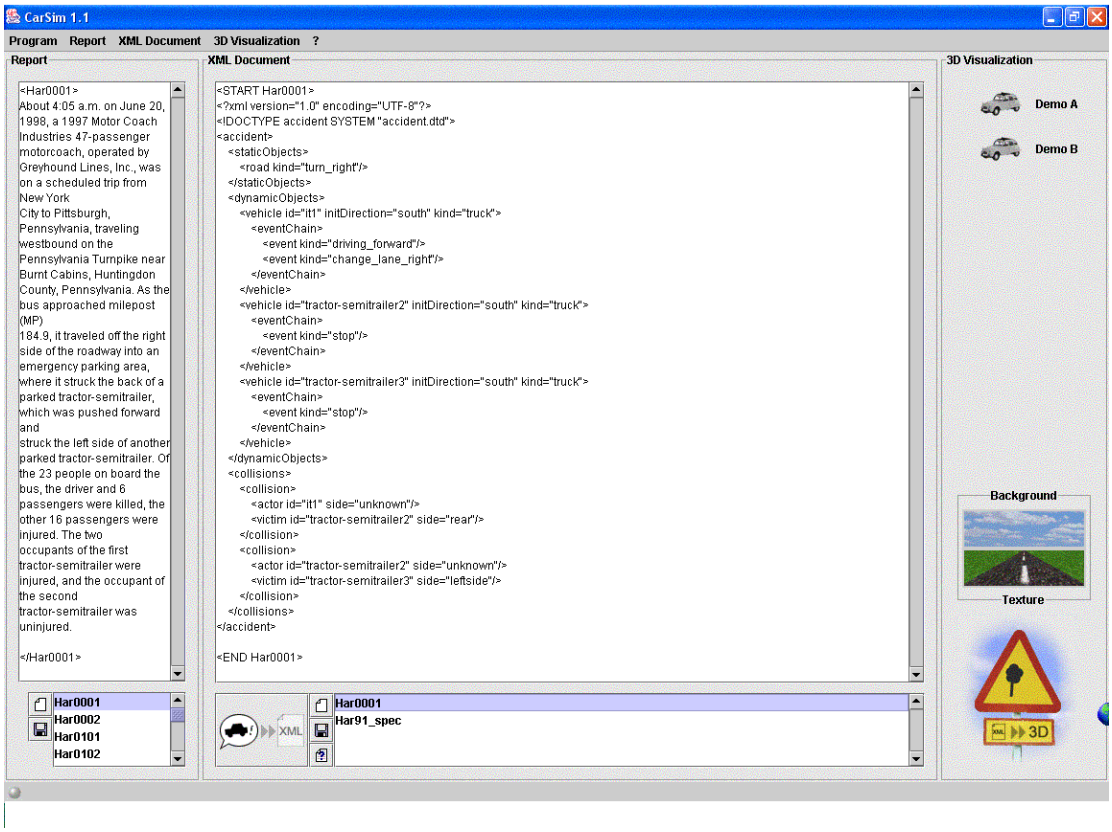

Figure 4.5 A picture of the new GUI

4.2.17 Snapshots of the visualization

The final objective of the system is to visualize the accident in a 3D environment. Figure 4.6 shows a snapshot from the visualization of the text Har0001. The text describes a bus that first changes lane to the right into an emergency parking area and there hits a parked tractor semi-trailer and that parked semi-trailer hits another parked semi-trailer in the rear. The information extraction module correctly extracts all the data from text. The problem is that the parked tractor semi-trailers start from the same coordinates as the bus. This leads to an incorrect visualization. We discuss this kind of problem in section 5.2.

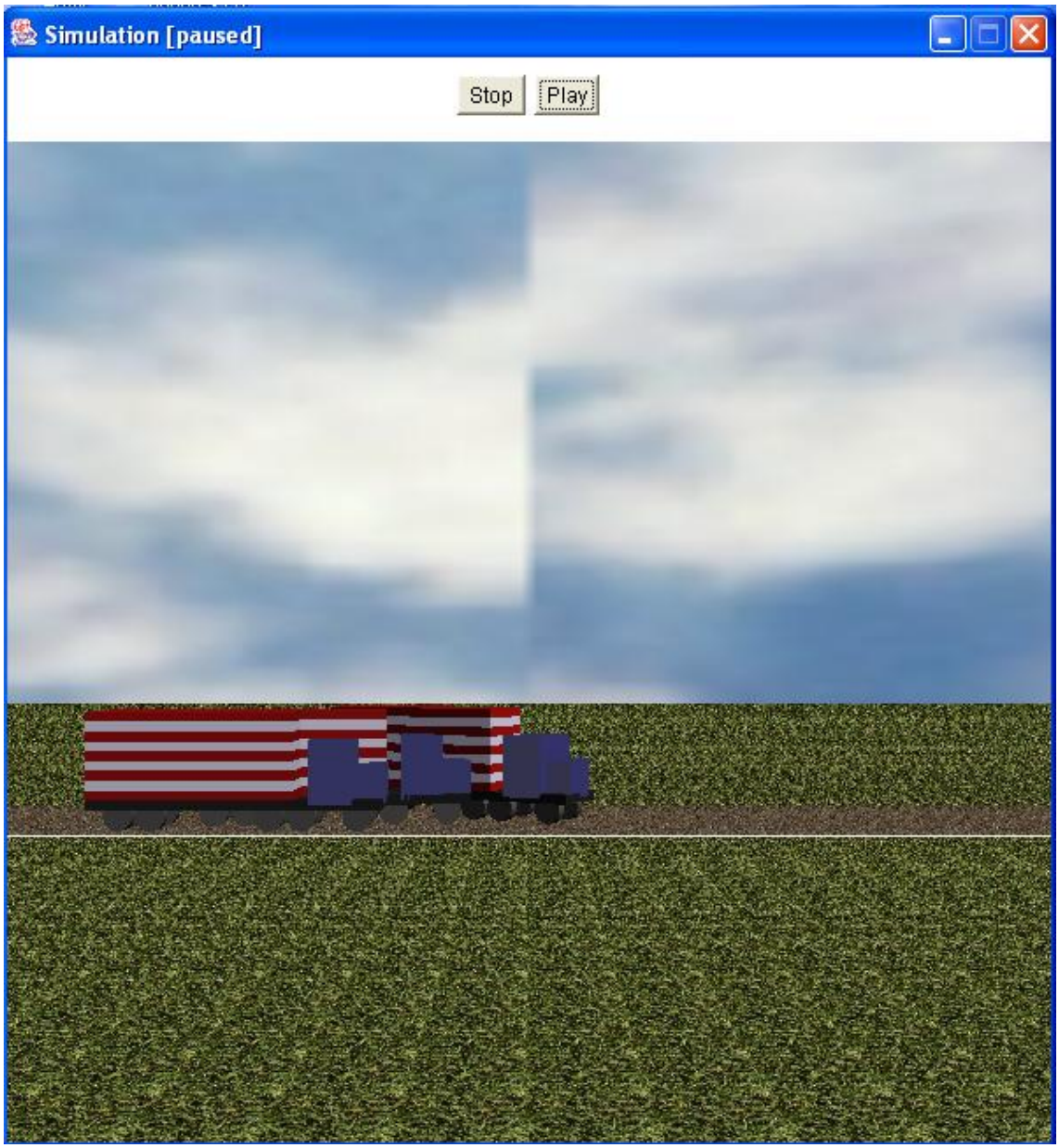

Figure 4.6 A snapshot from the visualization of text Har0001

Figure 4.7 shows a snapshot from the text Har0002. The text describes a collision between a bus and a truck at an intersection. The bus is driving on State Route 30A as can be seen in the picture. This is how it looks when the system makes a correct extraction of the text and a flawless visualization.

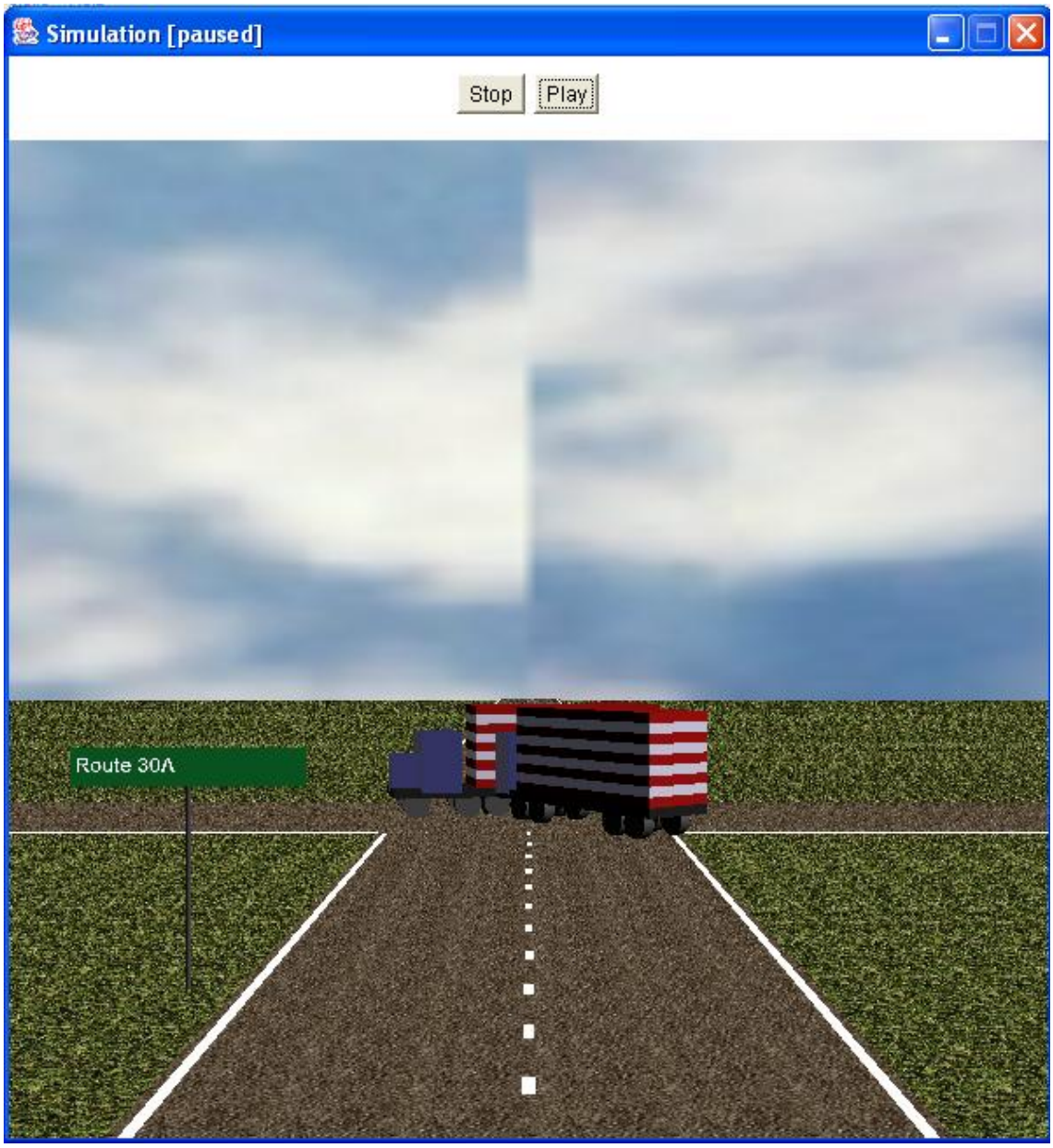

Figure 4.7 A snapshot from visualization of the text Har0002

Chapter 5 Conclusion

5.1 Results

Table 5.1 shows the complete results of the CarSim project. We ran the system on 23 different texts from NTSB. The extracted information is shown below here. Table 5.2 includes comments about texts, telling what went wrong or if it works.

Name	Explanation
1	correct
0	false
?	Can't be extracted from the text. It means that information cannot be found in the text.
The text	Which text that has been parsed.
Actors	If the number of actors are correct extracted.
Crash	If the crash/crashes was/were correct detected.
Events	Is a control over the dynamic objects´ event lists.
Initial direction	Checks if the initial directions were right
Coordinates	A control over the static objects placements.
Static Objects	Controls if the static objects are correct.
Number of elements found	Gives the number of how many objects that were detected in the text.
Realistic	True, if all the facts from the text are correct extracted and the visualization is correct
Correct IE	True, if the information extraction of the text is correct
Correct without the events	True, if the extracted text is correct disregarding the event lists.

The text	Actors	Crash	Events	Initial direction	Coordinates	Static objects	Number of elements found	Realistic	Correct IE	Correct without the events
Har91_01	0	1	0	0	0	?	6	False	False	False
Har91_sp	1	0	1	?	?	?	1	False	False	False
Har0101	0	0	0	1	1	1	5	False	False	False
Har0102	0	0	0	?	1	1	1	False	False	False
Har8506	1	1	0	1	?	?	2	False	False	True
Har8602	0	1	0	0	?	?	6	False	False	False
Har8603	0	1	0	0	1	1	6	False	False	False
Har8702	0	0	0	1	1	1	6	False	False	False
Har8703	0	1	0	1	?	?	3	False	False	True
Har8705	0	1	0	1	?	?	3	False	False	True
Har8706	0	0	0	0	?	?	1	False	False	False
Har8901	0	1	0	1	?	?	4	False	False	True
Har8902	1	0	1	1	?	?	2	False	False	False
Har8801	0	0	0	0	?	?	1	False	False	False
Har8903	1	0	1	0	1	1	2	False	False	False
Har9002	1	1	0	0	1	1	4	False	False	False
Har9101	1	1	1	?	1	0	6	True	True	True
Har9201	1	0	1	?	?	?	1	False	False	False
Har9202	0	1	0	0	?	?	5	False	False	False
Har9503	0	0	0	0	?	?	2	False	False	False
Har0001	1	1	1	0	?	?	3	False	True	True
Har0002	1	1	1	1	?	?	2	True	True	True
Har87_02	0	0	0	0	0	1	2	False	False	False

Table 5.1The results of the CarSim project over 23 texts

Har91_sp	The vehicle in this text overturned. “Overturned” is a crash verb that the system does not detect.
Har91_01	This text does many co-referencing mistakes. The system detects the crash verb but the links from Link Grammar are incorrect.
Har0101	This is a single driver accident where the bus crashes with static objects. The system does not properly extract this kind of accidents.
Har0102	Here is one more single driver accident. The system is also unable to detect the bus.
Har8506	This text almost passes through the system correct. The error is in the event list of one of the vehicles. Another mistake is that Link Grammar does not link to the right names of the vehicles but in this case it will be correct anyway.
Har8602	The system detected properly the accidents but created too many dynamic objects. A co-referencing module would solve this problem.
Har8603	This text is hard and CarSim totally failed.
Har8702	The accident verb in this text is overturned that why no accident was detected. The system detects too many dynamic objects as well.
Har8703	This accident describes a bus which overtakes 3 trucks and went back to the right lane and then hit a fourth truck in the rear. The system cannot detect the three trucks, which are overtaken and it creates an extra dynamic object. The crash though is correctly detected but the event list of the truck that is hit is wrong.
Har8705	Here is another co-referencing problem and Link Grammar does not link to the right word in the accident, so both the actor and victim in the crash have wrong names. The system also creates an extra dynamic object.
Har8706	This text is hard and CarSim totally failed. No detection of an accident was made.
Har8801	This text is hard and CarSim totally failed. No detection of an accident was made. The system did detect the bus and gave it a correct initial direction.
Har8901	Link Grammar links the wrong words in the accident. The accident are correctly detected. There is also a co-referencing problem.
Har8902	The system finds two vehicles in this text and gives them a correct initial direction. The accident is not detected.
Har8903	This text describes a single driver accident and the system is not able to detect them. That is why the accident was not detected.
Har9002	The system detects the crash verb but the links from Link Grammar to the subject and object are wrong.
Har9101	Here is another single driver accident where the truck crashes into an embankment ditch. Here is also a co-referencing problem but because there is only one dynamic object it becomes right anyway. The system creates too many static objects in this text.
Har9201	Another single driver accident with crash verb *overturned*. That is as said before, the system unable to detect such verbs.
Har9202	This text describes a chain collision with many vehicles. The system cannot separate all the vehicles.
Har9503	This text is hard and CarSim cannot extract the information.
Har0001	The system is unable to detect the initial direction, which is west and one of the dynamic objects is called *it* where *it* relates to the bus. Luckily, it will be correct here anyway but with a working co-reference module, it would have been perfect. Another thing is that two of the dynamic objects are

	parked. That means they will start from their initial direction corner and they will be standing there. They should have started from the center of the screen.
Har0002	Works!
Har87_02	This report describes a single car accident and it does not work. Link Grammar was unable to set links for this sentence.

Table 5.2 Comments about the texts that the system has extracted

As we can see, many failures are due to the lack of a co-reference module and to the incorrect links produced by Link Grammar. Section 5.2 discusses the future development of a co-reference module. Another issue is how to improve Link Grammar to raise the correct throughput. It also discusses the insertion of some verbs the system cannot currently detect, such as *overturn*.

According to the results in Table 5.1, the CarSim-project is able to extract information from approximately 10-15% of the texts. This is a promising start although much work remains to be done. We believe that the platform we have created and the tools we have developed will give a good fundamental ground to start with.

5.2 Future Development

There are crucial modules to develop to improve the current version of CarSim. Features that should be implemented include co-reference resolution and the detection of accidents in the passive form. These are known issues and our initial idea was to implement them. However, the time dedicated to this project did not permit it.

A co-reference module would connect a pronoun to its antecedent. Consider the following sentence:

Example 5.1

"*As **the pickup truck** rotated during impact, **it** struck a passenger car travelling southbound...*"

The pronoun in Example 5.1 is "it" and refers to the head noun "truck". CarSim will detect them as two separate vehicles although they form one single entity. This leads to a long list of vehicles and the graphical module will not display the accident in the text correctly. Co-reference resolution techniques are well known and some are quite simple although not with the highest success rate. We studied anaphora resolution techniques but we had no time to integrate them. The co-reference resolution would solve many problems with the current version of CarSim.

The grammatical function module (Ekman and Nilsson 2002) does not consider sentences in the passive form. Example 5.2 illustrates it.

Example 5.2

"*The bus was struck by the dump truck.*"

The grammatical function module inverts the actor and the victim when a sentence is in the passive form. It leads to the incorrect animation of Example 5.2 where the “bus” will hit the “dump truck”. A future work to extract the accidents should extend CarSim to cover this case as well. This should not be particularly hard to implement from the Link Grammar output.

We ran CarSim on a corpus written in technical English. The NTSB reports use a somewhat unwieldy language and Link Grammar is not always able to produce correct results. We observed that Link Grammar had particularly problems with long noun phrases. Example 5.3 displays an example of such a phrase from the corpus.

Example 5.3

> “*About 3:14 p.m. mountain standard time on April 29, 1985, a Bell Creek, Inc. tractor-semitrailer transporting 99 head of cattle and traveling about 59 mph struck the rear of a 1977 Tuba City Unified School District schoolbus on eastbound U.S. 160 about 16 miles north of Tuba City Arizona*.”

The long expression “*1977 Tuba City Unified School District schoolbus”* with many modifiers to the head noun creates huge problems for Link Grammar. Not only in the sense of time spent analyzing such a sentence but also in the failure to produce a valid linkage. The suggestion to a solution involves a pre-processing step were these phrases are identified and replaced. The modifiers very often contain names, words starting with one capital letter. These could easily be found using regular expressions. The noun phrases could be replaced by an arbitrary but unique identifier for example the word “*vehicle*” in conjunction with a sequence number. This would increase the correct throughput of Link Grammar and as the extraction relies heavily on the performance of Link Grammar the overall result would be significantly improved.

The template format shared between the information extraction module and the graphical module is not by any means perfect. In conjunction with more 3D objects of vehicles, more values could be added to the type of a dynamic object. This would enhance the animation and convey more information. For example, all busses, trucks, and tractors are displayed using the same 3D object and information so painstakingly extracted is lost during animation. The actions “*parked*” and “*overturned*” were frequently found in the corpus but there is no way to animate these events. The event “*overturned”* has no close relationship with the events that are specified in the template format. “*Parked”* is assimilated as a `stop` event. However, the animation will not resemble the reality. Consider the accident sequence of Example 5.4.

Example 5.4

> “*As the bus approached milepost (MP) 184.9, it traveled off the right side of the roadway into an emergency parking area, where it struck the back of a parked tractor-semitrailer, which was pushed forward and struck the left side of another parked tractor-semitrailer*.”

The two parked “*tractor-semitrailer*” entities will both only get a `stop` event and be placed at the beginning of a road. The animation will not bear a resemblance to the story of the text.

The corpus the CarSim system was tested on did not include any level crossings, nor any traffic light so these are not thoroughly tested.

The time spent during parsing can be perceived as annoyingly drawn out. The largest amount of time during an execution is spent in the Link Grammar parse phrase. The way Link Grammar is set up it can do three passes on one sentence using different options. The current system does not take into consideration that a sentence may be parsed twice. For example, it parses the text for the accident extraction and then again during the movement extraction. The same information is returned from Link Grammar in both cases. To minimize time spent parsing with Link Grammar the system should cache information returned from Link Grammar. The next time information is requested about this sentence the cached version of the result should be returned. In conjunction with this problem, the splitting of sentences is done each time Link Grammar is used. This is not an expensive task although the work performed is redundant. A system that keeps track of what sentences have been parsed should be considered. Both the sentence split and the parsing of each sentence should only be done once.

Another time consumer is WordNet look-ups. This action is only performed once when the first sentence is parsed and is therefore not the biggest time consumer in the CarSim system. We used the WNJN as is. We did not want too analyze the lower levels of the WordNet API. However, this means that the result from a hyponym tree look-up is formatted and cannot be directly manipulated. We could filter out the characters not conveying any information. They exist solely to produce a pleasant output.

Many string operations have to be done to ensure that the words are separated from the garbage characters and put in a container to enable easy manipulation of the data. This extra work can be made obsolete if the class `Synset` is rewritten or if one desires to retain the structure of the WNJN. A new class can be constructed that inherits from `Synset`. This should further improve the perceived response time of the program. An attempt was made but had to be abandoned due to lack of time.

To lower the perceived response time of the program the extraction can be run in a different thread. A solution employing threads was tried out but rejected due to the lack of time. A threaded architecture would remove the GUI lock up when no redrawing is performed. This is would improve the behavior in terms of user friendliness.

A feature that easily could be added is an interface to Link Grammar and WordNet. The tools are already hooked up and available from within the code of CarSim. Why not make a simple interface so the user has the advantage of the full set of powerful features of these linguistic analyzing resources. Another feature that would be interesting is the possibility to alter the options, which controls the way Link Grammar handles a sentence.

Chapter 6 References

Bou Bernard. “WordNet JNI Java Native Support”, Technical report of Lycée Champollion, Figeac, France. 6 April 2002, ,http://wnjn.sourceforge.net/

Church Ken “A Stochastic Parts Program and Noun Phrase Parser for Unrestricted Text”. *In Proceedings of the Second Conference on Applied Natural Language Processing* Morristown, NJ, 1988

Collins Michael. “Head_Driven Statical Models for Natural Language Parsing” PhD thesis, University of Pennsylvania, Philadelpiha, PA, 1999

Coyne Bob and Richard Sproat. “Wordseye: An Automatic Text to Scene Conversion System”, In *Proceedings of SigGraph*, 2001.

Dupuy, Sylvain Arjan Egges, Vincent Legendre and Pierre Nugues. “Generating a 3D Simulation of a Car Accident from a Written Description in Natural Language: the CarSim System”. In *The Proceedings of the ACL Workshop on Temporal and Spatial Information Processing*, Toulouse, July 2001.

Ekman Torbjörn and Anders Nilsson. “Identifying Collisions in NTSB Accident Summary Reports” Technical Report. Computer Science Department LTH 2002.

Fellbaum, Christiane (Ed.), *WordNet: An Electronic Lexical Database*. MIT Press, 1998.

Greffenstette Gregory Pasi Tapanainen, “What is a word, What is a sentence? Problems of Tokenization”. Rank Xerox Research Centre Grenoble Laboratory Meylan, France 7 July 1994.

Hobbs Jerry R., Douglas Appelt, John Bear, David Israel, Megumi Kameyama, Mark Stickel, and Mabry Tyson. “FASTUS: A Cascaded Finite-State Transducer for Extracting Information from Natural-Language Text”, In *Finite-State Language Processing*, Emmanuel Roche and Yves Schabes (Eds), Chapter 13, MIT Press, 1997.

National Transportation and Safety Board.”Highway Accident Reports Publications” http://www.ntsb.gov/Publictn/H_Acc.htm

Schulz Bastian “Development of an Interface and Visualization Components for a Text-to-Scene Converter” Technical Report. Computer Science Department LTH 2002.

Sleator Daniel and Davy Temperley. “Parsing English with a Link Grammar”, *Carnegie Mellon University Computer Science technical report CMU-CS-91-196*, October 1991

Appendix

Appendix A

The texts, on which CarSim was tested, were downloaded from National Transportation Safety Board, NTSB homepage. NTSB is an independent federal agency working in the United States. Their investigators examine major civil transportation accident on air, see, and land. They try to clear out how and why the accident occurred and to give suggestions so accidents of that kind will not occur again.

When a major accident happens, NTSB investigators investigate it. They write down what happened in a thorough report and a short abstract of it. This appendix contains all these abstracts. They are written in technical English and some of them include typographical errors. We have corrected these errors and the changes are in Appendix B.

Har0001

About 4:05 a.m. on June 20, 1998, a 1997 Motor Coach Industries 47-passenger motorcoach, operated by Greyhound Lines, Inc., was on a scheduled trip from New York City to Pittsburgh, Pennsylvania, traveling westbound on the Pennsylvania Turnpike near Burnt Cabins, Huntingdon County, Pennsylvania. As the bus approached milepost (MP) 184.9, it traveled off the right side of the roadway into an emergency parking area, where it struck the back of a parked tractor-semitrailer, which was pushed forward and struck the left side of another parked tractor-semitrailer. Of the 23 people on board the bus, the driver and 6 passengers were killed, the other 16 passengers were injured. The two occupants of the first tractor-semitrailer were injured, and the occupant of the second tractor-semitrailer was uninjured.

Har0002

About 10:30 a.m. on October 21, 1999, in Schoharie County, New York, a Kinnicutt Bus Company school bus was transporting 44 students, 5 to 9 years old, and 8 adults on an Albany City School No. 18 field trip. The bus was traveling north on State Route 30A as it approached the intersection with State Route 7, which is about 1.5 miles east of Central Bridge, New York. Concurrently, an MVF Construction Company dump truck, towing a utility trailer, was traveling west on State Route 7. The dump truck was occupied by the driver and a passenger. As the bus approached the intersection, it failed to stop as required and was struck by the dump truck. Seven bus passengers sustained serious injuries; 28 bus passengers and the truckdriver received minor injuries. Thirteen bus passengers, the busdriver, and the truck passenger were uninjured

Har87_02

About 2:43 p.m. central standard time on November 11, 1985, a schoolbus owned by R. W. Harmon and Sons, Inc. was eastbound on II-70 transporting 13 high school students to their homes in St. Louis, Missouri, from the Parkway North Senior High School. As the schoolbus was approaching the Lucas and Hunt Road exit it went out of control, swerved to the right, and the right front of the schoolbus struck a guard rail, a concrete pedestal and a sign support pillar located adjacent to the right eastbound roadway. The schoolbus body and the steering axle separated from the chassis during the collision. The weather was cloudy and the pavement was dry. The schoolbus did not catch fire. Two students were killed; the schoolbus driver and one student sustained serious injuries and the remaining 10 students sustained minor to moderate injuries.

Har91_01
About 5:40 p.m. on July 26, 1990, a truck operated by Double B Auto Sales, Inc., transporting eight automobiles entered a highway work zone near Sutton, West Virginia, on northbound Interstate Highway 79 and struck the rear of a utility trailer being towed by a Dodge Aspen. The Aspen then struck the rear of a Plymouth Colt, and the Double B truck and the two automobiles traveled into the closed right lane and collided with three West Virginia Department of Transportation (WVDOT) maintenance vehicles. Fire ensued, and the eight occupants in the Aspen and the Colt died. The Aspen, Colt, Double B truck, and two of the three WVDOT vehicles were either destroyed or severely damaged. The Double B truckdriver and one firefighter sustained minor injuries.
Har91_spec
On August 3, 1991, about 6:45 a.m., a Greyhound bus traveling from New York City to Buffalo, New York, ran off the right side of the roadway, and overturned on State Route 79 near Caroline, New York. The driver and 33 passengers were injured, and 5 passengers were uninjured.
Har0101
On May 9, 1999, about 9:00 a.m., a 1997 Motor Coach Industries 55-passenger motorcoach, operated by Custom Bus Charters, Incorporated, was traveling eastbound on Interstate 610 in New Orleans, Louisiana. The bus, carrying 43 passengers, was en route from La Place, Louisiana, to a casino approximately 80 miles away in Bay St. Louis, Mississippi. As the bus approached milepost 1.6, it departed the right side of the highway, crossed the shoulder, and went onto the grassy side slope alongside the shoulder. The bus continued on the side slope, struck the terminal end of a guardrail, traveled through a chain-link fence, vaulted over a paved golf cart path, collided with the far side of a dirt embankment, and then bounced and slid forward upright to its final resting position. Twenty-two passengers were killed, the busdriver and 15 passengers received serious injuries, and 6 passengers received minor injuries.
Har0102
On March 2, 1999, a 1979 Motor Coach Industries MC-9, 47-passenger charter motorcoach, owned and operated by Shuttle Jack, Inc., (Shuttle Jack) of Santa Fe, New Mexico, departed the Santa Fe Ski Basin, carrying the driver, 2 adult chaperons, and 34 middle school-age children. The bus began to descend a 14-mile mountainous roadway. About halfway down the grade, the driver discovered that the vehicle's air brakes were no longer capable of slowing or stopping the bus. He noted that the brake air-pressure-gauge reading was between 90 and 120 pounds per square inch, which was the normal system operating pressure for this vehicle. During the next 3.5 miles, the driver made several unsuccessful attempts to bring the bus under control by pumping the air brakes, downshifting the automatic transmission, pulling on the emergency/parking brake valve, and shutting off the engine. Eventually, the driver lost control of the bus while rounding a left-hand curve. The bus departed the right side of the roadway, crashed into a rock embankment, and then rolled onto its left side back onto the roadway. (See figure 1.) The calculated speed of the bus was 60 to 65 mph at the time of the collision. Two passengers were fatally injured, and the 35 other occupants received varying degrees of injuries
Har8506
About 3:14 p.m. mountain standard time on April 29, 1985, a Bell Creek, Inc. tractor-semitrailer transporting 99 head of cattle and traveling about 59 mph struck the rear of a 1977 Tuba City Unified School District schoolbus on eastbound U.S. 160 about 16 miles north of Tuba City Arizona. The schoolbus was stopped with its warning lights

flashing in the eastbound lane of the two-lane highway to discharge passengers. The weather was clear, the pavement was dry, and there were no visibility obstructions for about 1.4 miles to the rear of the schoolbus. Of the 32 schoolbus passengers (ages 5 to 21 years), 2 were fatally injured, 4 sustained serious injuries 4 received moderate injuries, 18 sustained minor injuries, and 4 were not injured. The truckdriver and the schoolbus driver received minor injuries.

Har8602

About 3:20 p.m. on May 31, 1985, a northbound Military Distributors of Virginia, Ine.9 tractor-semitrailer collided with two southbound vehicles on a curve on U.S. 13, about 2.3 miles south of Snow Hill, North Carolina. The first collision on the two-lane, undivided highway was with a 1982 schoolbus operated by the Greene County (North Carolina) Board of Education. After this collision, the Military Distributors vehicle continued northbound and struck a tractor-semitrailer loaded with grain, which had been following the schoolbus on the two-lane highway. During the collision with the grain truck, the Military Distributors semitrailer separated from its tractor, continued northbound and overturned onto its right side in the northbound lane. The rear of the grain trucks semitrailer remained on the highway and was struck by a passenger automobile. After the collisions, the Military Distributors tractor, and the front of the grain truck's semitrailer caught fire. The weather was clear and the pavement was dry. The Military Distributors truckdriver sustained fatal injuries. Of the 27 schoolbus passengers (ages 5 to 13)15 sustained minor or moderate injuries, 10 sustained serious or severe Injuries, and 2 received critical Injuries. Six of the passengers died. The schoolbus driver, the grain truck driver, and the automobile driver and passenger sustained minor injuries.

Har8603

About 7:51 p.m. on June 21, 1985 a privately-owned, 70,000-pound tractor-semitrailer operating in interstate commerce under a trip-lease agreement with C. Maxwell Trucking Company, Inc., lost control while descending a steep 3, 439-foot-grade on southbound State Route 59 in downtown Van Buren, Arkansas. The truck collided with the rear of and overrode a station wagon which was stopped at the bottom of the hill. The truck and the station wagon continued 84 feet forward, across an intersection, up a curb, and through a guardrail. They then traveled another 22 feet and struck two commercial buildings. A fire ensued and engulfed both vehicles and three buildings. Both occupants in the truck and the seven occupants in the station wagon were fatally injured.

Har8702

About 12:40 p.m., e.d.t on September 6, 1985, a 1902 GMC 2-axle truck fitted with a 1973 MC-331 cargo tank overturned while traveling southbound on the Capital Beltway, I-95, near Largo, Maryland. The 2,500-gallon capacity cargo tank contained about 1,375 gallons of propane. The Poist Gas Company truck was traveling between 50 and 55 mph when, according to the drivers the steering wheel started shaking violently and "flew out of my hands." The driver stated that he took his foot off the accelerator but did not brake because be believed that one of the vehicle's tires was experiencing a blow out. The truck veered across the right paved shoulder of the highway and onto a grass shoulder. It then traveled 300 feet down the grass shoulder until the driver steered the truck back to the left to avoid hitting a tree. The truck slipped the north end of a guardrail when it reentered the paved shoulder of the highway as the driver tried to regain control of the truck. The truck then traveled 510 feet down the paved shoulder and the right travel lane of the highway before rotating clockwise about 80" and overturning on Its left side. The vehicle continued to rotate

another 100'* as it slid 400 feet down the highway on Its left side. The truck came to rest facing north (180'* opposite its original direction of. travel) with the top of the cargo tank parallel with and against the guardrail At the time of the accident the roadway was dry and the weather was clear.

Har8703

On September 29, 1986, a Leatherwood Motor Coach Corporation charter bus carrying 38 passengers was traveling northbound on I-295, a four-lane divided highway near Camey's Point, New Jersey en route to Atlantic City, New Jersey. After passing three tractor-semitrailers in the left lane, the bus moved into the right lane and struck the rear of another slower moving tractor-semtrailer. The two vehicles continued forward and traveled northbound about 432 feet before coming to a stop. Two bus passengers were seriously injured, 5 bus passengers were moderately injured and the busdriver and 31 bus passengers received minor injuries. The truckdriver was not injured.

Har8705

About 4:15 a.m. on Monday, July 14, 1986, an intercity bus operated by Trailways Lines, Inc. was traveling eastbound on I-40 near Brinkley, Arkansas when it collided with the left rear of a tractor-semitrailer combination operated by Rising Fast Trucking Company, Inc. (RFT). At the time of the collision the RFT vehicle was making a U-turn which resulted In the RFT semitrailer's blocking both eastbound traffic lanes at a highway crossover from the eastbound to the westbound lanes of I-40 at milepost 210.4 near Brinkley, Arkansas. At the time of the collision the bus was transporting 28 passengers from Little Rock, Arkansas, to Memphis, Tennessee, on a leg of a regularly-scheduled run. One passenger reported that the busdriver screamed, "Hang on" just before the collision. The force of the collision caused the RFT semitrailer to rotate In a counterclockwise direction, and it came to rest In the highway median. The RFT semitrailer did not separate from its truck-tractor and came to rest In the highway crossover facing northwest with the front of the tractor partially blocking the Inside lane of westbound I-40. (See figures I and 2.) After the collision the bus continued in a southeasterly direction, left the pavement of the eastbound roadway, overturned 90o to the left, and came to rest on its left side on a grassy slope facing south with the rear of the bus about 17 feet south of the edge of the eastbound shoulder of the roadway. (See figure 3.) The weather, was clear, it was dark with no artificial highway lighting at the site, and the pavement was dry. The vehicles did not catch fire.

Har8706

About 7:34 a.m. on October 9, 1986, two charter intercity tour buses loaded with European tourists were traveling westbound in the right lane on State Route (SR) 495 m.p.h. in North Bergen, New Jersey, en route to Washington, D.C. As the westbound buses approached the Kennedy Boulevard exit on SR-495, the second bus suddenly veered leftward into the adjacent lane, struck the left rear of a passenger car traveling in that lane, then crossed into the eastbound contraflow lane, and struck a transit bus loaded with commuter passengers en route to New York City. One bus passenger aboard the transit bus was fatally injured and 26 other occupants aboard both buses sustained serious to minor injuries.

Har8801

About 1:45 p.m. on May 4, 1987, while traveling eastbound on Interstate 10 (I-10) in Beaumont, Texas, a tractor-semitrailer (truck) operated by Graebel Van Lines, Inc. (GVL), Jackknifed in the center lane, veered leftward across the left land and median strip, and struck a Trailways bus traveling westbound on I-10 in the left lane. A small fire which started in the bus below the driver's seating area was quickly extinguished by a passerby. The busdriver and 5 bus passengers sustained fatal injuries, 17 bus

passengers sustained serious to minor injuries, and 6 bus passengers were not injured. The truck driver and helper sustained moderate and minor injuries, respectively. It was raining at the time of the accident.

Har8901

About 10:55 p.m. eastern daylight time on May 14, 1988, a pickup truck traveling northbound in the southbound lanes of Interstate 71 struck head-on a church activity bus traveling southbound in the left lane of the highway near Carrollton, Kentucky. As the pickup truck rotated during impact, it struck a passenger car traveling southbound in the right lane near the church bus. The church bus fuel tank was punctured during the collision sequence, and a fire ensued, engulfing the entire bus. The busdriver and 26 bus passengers were fatally injured. Thirty-four bus passengers sustained minor to critical injuries, and six bus passengers were not injured. The pickup truck driver sustained serious injuries, but neither occupant of the passenger car was injured.

Har8902

On August 28, 1987, a 1982 school bus carrying 21 passengers was traveling westbound on Levy County (Florida) Road C-32 when it collided with a two-axle flatbed truck traveling northbound on Levy County Road C-337 near Bronson, Florida. The school bus driver and 5 passengers died; the truckdriver sustained critical injuries and 16 school bus passengers were injured.

Har8903

About 6:45 a.m., central standard time, on November 19, 1988, an intercity bus with 45 occupants, traveling southbound through a construction zone on Interstate Highway 65 in Nashville, Tennessee, suddenly went out of control during a steering maneuver, rotated 190 degrees clockwise in the southbound lanes, overturned on its left side, and came to rest facing northbound on the southbound embankment. Witness reports indicate that the bus was traveling at a high rate of speed in conditions of heavy rain. The unrestrained bus driver and 38 passengers were injured in the accident. Twelve passengers sustained serious injuries, and the bus driver and 26 passengers received minor injuries. Six passengers were not injured. Injured persons were taken to seven area hospitals for treatment.

Har9002

About 7:34 a.m., central daylight time, on Thursday, September 21, 1989, a westbound school bus with 81 students operated by the Mission Consolidated Independent School District, Mission, Texas, and a northbound delivery truck operated by the Valley Coca-Cola Bottling Company, McAllen, Texas, collided at Bryan Road and Farm to Market Road Number 676 (FM 676) in Alton, Texas. After the collision, the truck came to rest facing west on he right shoulder of FM 676. The school bus continued in a northwest direction and dropped approximately 24 feet into a caliche pit (excavation pit) partially filled with water, located in the northwest corner of the intersection. The bus came to rest on its left side facing southeast, totally submerged in approximately 10 feet of water, approximately 35 feet from the nearest shoreline. The bus front boarding door was jammed shut, but the rear emergency exit door was operable. No other emergency exits were on the bus. Nineteen students died at the accident scene, and two died later in the hospital. The 21 fatalities were the result of drowning or complications related to the submersion. Furthermore, 3 students sustained serious injuries, 46 others sustained minor injuries, and 11 students were not injured.

Har9101

About 3 a.m. Pacific standard time on February 13, 1991, a tractor-semitrailer (cargo tank) overturned as the vehicle was traveling on a main urban roadway in Carmichael, California. The tractor and semitrailer were owned and operated by Calzona Tankways,

Inc., of Phoenix, Arizona. At the time of the accident, the truck was being used for the intrastate delivery of gasoline to service stations; the cargo tank contained about 8,800 gallons of automotive gasoline. The driver lost control of the vehicle in a curve. The vehicle overturned onto its side and struck the embankment of a drainage ditch located in a dirt field beside the road. The cargo tank bounced and came to rest in the dirt field and adjacent to the drainage ditch. The rear end of the cargo tank landed on an unoccupied car parked in the field. Gasoline from the cargo tank spilled into the drainage ditch, which extended under the roadway and behind private residences nearby. About 15 minutes after the overturn, the gasoline ignited behind a residence. The fire flashed back and engulfed the overturned cargo tank, and the car under the cargo tank. A second unoccupied car parked near the overturned tank truck also caught fire. Gasoline runoff in the drainage ditch entered the underground drainage system and was also ignited. In addition to the total loss of the tank truck, its cargo, and the two parked cars, four homes and their contents were destroyed or heavily damaged by fire, and the residents from a 2-mile-square area were evacuated. Total property damage and cleanup costs were estimated at nearly $1 million. There were three minor injuries.

Har9201

On June 26, 1991, about 1:50 p.m., a Greyhound bus traveling from Cleveland, Ohio, to Washington, D.C., ran off the right side of the roadway and overturned on the Pennsylvania Turnpike near Donegal, Pennsylvania. One passenger was fatally injured, the driver and 14 passengers were injured, and 1 passenger was uninjured.

Har9202

About 9:10 a.m. on December 11, 1990, a tractor-semitrailer in the southbound lanes of I-75 near Calhoun, Tennessee, struck the rear of another tractor-semitrailer that had slowed because of fog. The uninjured truckdrivers exited their vehicles and attempted to check for damage. After the initial collision, an automobile struck the rear of the second truck and was in turn struck in the rear by another tractor-semitrailer. Fire ensued and consumed two trucks and the automobile. Meanwhile, in the northbound lanes of I-75, an automobile struck the rear of another automobile that had slowed because of fog. Then, a pickup truck and two other automobiles became involved in the chain-reaction rear end collision. No fatalities, injuries, or fires occurred. Subsequently, 99 vehicles in the northbound and southbound lanes were in multiple-vehicle chain-reaction collisions that killed 12 people and injured 42 others.

Har9503

About 1:50 a.m. on Monday, January 9, 1995, a multiple-vehicle rear-end collision occurred during localized fog at milepost 118 on Interstate 40 near Menifee, Arkansas. The collision sequence initiated when an uninvolved vehicle and the accident lead vehicle entered dense fog. As the lead vehicle reportedly slowed from 65 miles per hour (mph) to between 35 and 40 mph, it was struck in the rear. Subsequent collisions occurred as vehicles drove into the wreckage area at speeds varying from 15 to 60 mph. The accident eventually involved eight loaded truck tractor semitrailer combinations and one light-duty delivery van. Eight vehicles were occupied by a driver only, and one vehicle had a driver and a codriver. Three truckdrivers, the codriver, and the van driver were killed. One truckdriver received a minor injury, and four truckdrivers were not injured.

Appendix B

NTSB file:	Our file:	Fault in text:	Changed to:
HAR-87/06	Har8706	SR-495, the se6ond	SR-495, the second
		About 7.34 a m.	About 7.34 a.m.
		to Washington,, DC	to Washington, DC
HAR-87/02	Har87_02	Moderate Injuries	Moderate injuries
HAR-87/05	Har8705	1-40, this error occurred twice in text	I-40
		Hang onl	Hang on
		Arkansas when It collided with	Arkansas when it collided with
HAR-86/02	Har8602	U. S. 13, about	U.S. 13, about
		on the highway an was struck	on the highway and was struck
		and struck a traetor semitrailer	and struck a tractor semitrailer
HAR-00/02	Har0001	"?emergency parking area	emergency parking area
HAR-87/02*	Har8702	vehicie’s	vehicle’s
HAR-92/02	Har9202	About 9: 10 a.m.	About 9:10 a.m.
		1-75, this error occurred twice in text	I-75
HAR-90/02	Har9002	About 7.34 a.m .,	About 7.34 a.m.,
HAR-85/06	Har8506	32 s’choolbus passengers	32 schoolbus passengers
		Inc.tractor	Inc. tractor
HAR-87/03	Har8703	minor Injuries	minor injuries

The accident description, NTSB: HAR-92/01, contain two accidents which occurred with a few days between them, and it was the same company that had these accidents. We have split this text into two files. Their names are Har9201.txt and Har 91_spec.txt

Appendix C

```
  <!ELEMENT accident ( staticObjects?, dynamicObjects?, collisions? )>


     <!ELEMENT staticObjects ( road | tree | sign | trafficLight |
levelCrossing )*>
        <!ELEMENT road EMPTY>
        <!ATTLIST road
           kind   ( crossroad | straightroad | turn_left | turn_right
)     #REQUIRED
        >
        <!ELEMENT tree ( coords )>
        <!ATTLIST tree
           id     ID    #REQUIRED
        >
        <!ELEMENT sign ( coords )>
        <!ATTLIST sign
           kind   ( stop )    #REQUIRED
        >
        <!ELEMENT trafficLight ( coords )>
        <!ATTLIST trafficLight
           id           ID                              #REQUIRED
           colorType    ( red | orange | green | inactive)  #REQUIRED
        >
        <!ELEMENT levelCrossing (coords)>


     <!ELEMENT dynamicObjects ( vehicle )*><!-- possible extensions:
train -->
        <!ELEMENT vehicle ( startSign?, endSign?, eventChain? )>
        <!ATTLIST vehicle
           id           ID                      #REQUIRED
           kind         ( car | truck )         "car"
           initDirection     (   north   |   east   |   south   |   west)
      #REQUIRED
        >
           <!ELEMENT startSign ( #PCDATA )>
           <!ELEMENT endSign   ( #PCDATA )>
           <!ELEMENT eventChain ( event )+>
              <!ELEMENT event EMPTY>
              <!ATTLIST event
                 kind   ( driving_forward | turn_left | turn_right |
stop | overtake | change_lane_left | change_lane_right ) #REQUIRED
                 critical ( yes | no ) "no"
              >

     <!ELEMENT collisions ( collision )+>
        <!ELEMENT collision ( actor, victim, coords? )>
           <!ELEMENT actor EMPTY>
           <!ATTLIST actor
              id        IDREF
      #REQUIRED
              side      (  front  |  rear  |  leftside  |  rightside  |
unknown )   #REQUIRED
           >
           <!ELEMENT victim EMPTY>
           <!ATTLIST victim
              id        IDREF
      #REQUIRED
```

```
              side      (  front  |  rear  |  leftside  |  rightside  |
unknown )   #REQUIRED
            >


     <!ELEMENT coords   EMPTY>
     <!ATTLIST coords
        x CDATA "0"
        y CDATA "0"
     >
```

Appendix D

In our project, we have used auxiliary programs. They improved the software engineering part and made the development easier to manage. Most of these programs are free and can be found on the Internet. The programs are easy to install and by using the work and experience of others, the development time can be shortened.

Log4j
Inserting log statements into your code is a low-tech method for debugging it. It may also be the only way because debuggers are not always available or usable. On the other hand, some people argue that log statements pollute source code and decrease legibility. In the Java language for which a preprocessor is not available, log statements increase the size of the code and reduce its speed, even when logging is turned off. Given that a reasonably sized application may contain thousands of log statements, speed is of particular importance.

With log4j, it is possible to enable logging at runtime without modifying the application binary. The log4j package is designed so that these statements can remain in shipped code without incurring a heavy performance cost. The programmer can control logging behavior by editing a configuration file without touching the application binary. When logging is wisely used, it can prove to be an essential tool. One of the distinctive features of log4j is the notion of inheritance in loggers. Using a logger hierarchy it is possible to control which log statements are output at arbitrarily fine granularity but also great ease. This helps reduce the volume of logged output and minimize the cost of logging.

The target of the log output can be a file, an `OutputStream`, a `java.io.Writer`, a remote log4j server, a remote Unix Syslog daemon, or even a NT Event logger among many other output targets. On an AMD Duron clocked at 800Mhz running JDK 1.3.1, it costs about 5 nanoseconds to determine if a logging statement should be logged or not. Actual logging is also quite fast.

Ant
Apache Ant is a part of Jakarta. It is free to use under the Apache Software License, Version 1.1. and makes management of project files easier. Apache Ant is a very powerful tool. It can handle tasks such as Java, Javac, JavaDoc, ftp and telnet. In our project, we have used Ant for Java, Javac, and to build JavaDoc. An Ant buildfile also managed the ftp connection to the web server where our homepage was hosted. The Ant buildfile took care of uploading new versions of our homepage including the JavaDoc HTML pages.

Ant was used to compile the wnjn project. This includes calling UNIX makefiles, producing JNI header files, and compiling the Java part of the project. Ant was also used to build and run The CarSim project in both release and debug versions.

Xerces
Xerces is a program suite handling XML. Parsing, validation, and different kinds of manipulations of XML are supported. Xerces also provides a partial implementation of Document Object Model Level 3. The components used are distributed in two jar archives: xercesImpl.jar and xmlParserAPIs.jar. Xerces was created by the Apache organization and are licensed with Apache Software License, Version 1.1. In CarSim, Xerces was used to output well-formatted XML templates.

CVS

CVS stands for Concurrent Version System. The CVS system was created by GNU. It is an open source project and is free to use under the Gnu general public license. CVS handles the problem of several programmers working on the same part of a project. CVS also keeps a complete version history of the project files. This means that a project can be reverted to an earlier state with minimal efforts. The feature of concurrent editing support was not as valuable as the version history in this situation but the project grew and it proved wise to use CVS.

JxBeauty and HTML Tidy

The JxBeauty is a tool developed by Johann Langhofer. Its purpose is to parse Java source files, format, and indent them neatly. The tool was original designed to run as a plug-in to the JBuilder Integrated Development Environment. In CarSim, we used emacs to create the Java files and we ran JxBeauty from the command line. JxBeauty is free to use.

HTML Tidy is similar to JXBeauty but it works on HTML and XML files instead of Java source files. The ant build files and the homepage of the CarSim project were formatted using HTML Tidy. HTML Tidy is available under an open source license.

The homepage of the CarSim project was developed using MS FrontPage and Macromedia Dreamweaver (with Homesite). The reason for using both was that MS FrontPage was installed under a license at the computer science institution of LTH. Dreamweaver was used on a private computer.

Debugging

To be able to correct particularly nasty bugs, we used the JSwat debugger. The program is licensed under the GNU General Public License. It provides an easy-to-learn environment.