Knowledge Representation in a Text-to-Scene Conversion System

Richard Johansson (d99rj@efd.lth.se)

September 2003

Abstract

This report describes the design and implementation of a knowledge representation in the CarSim system, which is a computer program which produces animated computer graphics from car accident reports. The report then goes on to describe the integration of the knowledge representation with lexical resources such as WordNet, as well as a simple algorithm for detection of road objects and coreference resolution built on the new knowledge representation. An overview of the topic of knowledge representation in general is also provided.

Contents

1 Introduction 3
1.1 Overview of This Report 3
1.2 A Short History of the CarSim System 4
1.3 Project Organization 4

2 Introduction to Semantics and Knowledge Representation 4
2.1 Information Extraction Systems 5
2.2 The Berkeley FrameNet 5
2.3 Representing an Accident 7
2.4 A Case Study: WordsEye 8

3 A New Knowledge Representation 9
3.1 Lessons from CarSim 2.0 9
3.2 Requests from Users 9
3.3 Shortcomings of the Old Template 10
3.4 The Scene Object 10
3.5 Road Object Ontology 11
3.6 Representation of Dynamic Information 13
3.7 Some Objections 14
3.8 A Dynamic Information Representation Based on Frames 15
3.9 Evaluation of the Descriptive Capacity 16

4 The Implementation 17
4.1 Early Stages 17
4.2 Automatic Generation of DOM Tree Wrapper Classes 17

5 Consequences of the New Knowledge Representation 21
5.1 Architecture of the New System 21
5.2 Extraction of Scene Data 21
5.3 Resolution of Anaphora 21
5.4 The Various WordNets 23
5.5 How to Integrate the WordNets with Our Ontology 24
5.6 A General Approach to Road Object Detection and Anaphora Resolution 26
5.7 A Manual Experiment on the Swedish News Corpus 28

6 Conclusion 29

7 Acknowledgements 29

A Complete Template Description File 30

B Glossary 34

1 Introduction

This is a Master's degree project report for a project in natural language processing at the Department of Computer Science at Lunds Tekniska Högskola, Sweden, under the supervision of Pierre Nugues.

CarSim is a computer tool which produces animated computer graphics representing road accidents. As Nugues [21] writes, it is intended as a support tool for analysts of motor vehicle accidents and to produce animated graphics for the education of drivers. In Figure 1, there is an example of what its output can look like.

Figure 1: An example of an animation produced by CarSim.

In brief, the CarSim system works as follows: an accident report is read from a file. The information contained in the accident description is then output in some format which can later be of use when producing the animated graphics. As we can see, the system consists of two distinct modules: the *linguistic* module which reads a text in *human language* and produces an intermediate representation in a *formal language*, and a *visualizer* module which produces the graphics from the formal language (see Figure 2). This report solely deals with the linguistic module, and its emphasis is on how the intermediate representation should be designed and implemented.

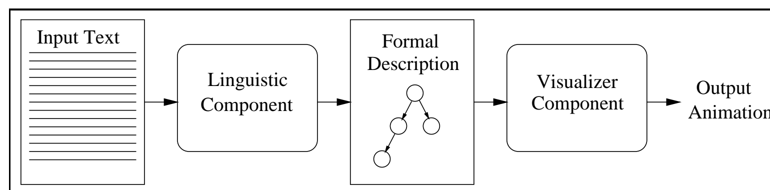

Figure 2: The stages of the CarSim system.

1.1 Overview of This Report

In section 2, I try to give a broad introduction to the topic of semantics and knowledge representation in general. Next, in section 3, I discuss what information should be contained in a knowledge representation for a text-to-scene conversion system for car accidents. The development of the framework used for implementing the new knowledge representation is described in section 4. Some discussions of the consequences of the knowledge representation, and some explorations of new directions to take, are presented in section 5. Finally, section 6 gives a conclusion.

Since this report must be intelligible to any computer science student near graduation even if he or she isn't familiar with all technical terms used, there is a glossary in appendix B. I found it more convenient to collect all explanations there instead of littering them all over the text.

1.2 A Short History of the CarSim System

The first CarSim system, described in [7], was designed for French texts obtained from an insurance company, the MAIF. Its linguistic component was written in Prolog and designed as a standalone program, which produced a data file which was then input to the visualizer component, which was written by Egges [8], and a new graphical user interface was later developed by Schultz [26].

The CarSim 2.0 linguistic component, which was designed for texts written in English, was written in Java by Svensson and Åkerberg [29]. They used texts from the National Transport Security Board in the United States, and this corpus was very different from the French corpus. The French texts, which were written for an insurance company by the people involved in the accidents, typically describe quite small, harmless and uncomplicated incidents. The NTSB texts, on the other hand, were written by officials about serious accidents, often involving fatal injuries. The accidents described in these texts were usually very complicated, sometimes involving over a hundred vehicles. The style of writing was generally more intricate as well. These factors made the analysis (by machines as well as by humans) more difficult for the English texts than the French ones.

Svensson and Åkerberg successfully implemented a prototype information extraction engine. The results were somewhat imperfect, and this work started as a series of improvements on their work.

Andersson [1] has implemented a prototype of a Swedish version of CarSim 2.0, using the same architecture but adapting to the limited availability of language tools for Swedish. For example, the English version used the Link Grammar parser [27] for deep parses. Andersson replaced this parser by shallow parses using a Swedish part-of-speech tagger. The texts used during the development of the Swedish version of the CarSim system were taken from Swedish newspaper sources (the "Nyheter corpus", as we call them). These texts are quite different compared to the NTSB texts, which are very detailed, since they leave a lot of details unspoken.

1.3 Project Organization

This report is the result of a project sponsored by Vinnova, the Swedish Agency for Innovation Systems. The project will during 2003 and 2004 attempt to produce a working text-to-scene conversion system for Swedish road accident reports.

To evalutate our results, there is a reference group consisting of traffic experts from the Department of Traffic and Road at Lunds Tekniska Högskola.

The system will be distributed under a GNU-like license. We will as far as possible avoid dependencies on external components which are proprietary.

2 Introduction to Semantics and Knowledge Representation

Let us set up the following naïve model of communication: a pattern of firing neurons forming a *thought* emerges in a human brain. A reader who is inclined towards less mechanistic world view might instead prefer to say that a divine flash appears in a human soul. By means that are presently only partially understood, the pattern or divine flash takes the shape of layers of structures – sentences, words, morphemes – which can then be represented as sequences of primitive symbols which are suitable for human communication: the phonemes. These are then transmitted into the air via vibrations of the vocal cords and movements of various parts of the mouth and throat. The sounds enter the ear of a receiver who recreates the communication structures and *interprets* them to "recreate" the original thought in the mind of the receiver. This process can of course be generalized to the written word as well.

This is a classic model underlying the design of many natural language processing systems: the transmitter creates a "thought", shapes it into sentences, codes them as a sequence of symbols and transmits it through some medium. The other end of the communication receives the sequence of symbols, recreates the sentences and is finally able to "understand" the thought.

What layers should be present in the receiving end is a matter of endless dispute. But one step which clearly needs to be taken by every system which claims to "understand" a message in human language is the leap from the *realization* of the message to the abstract *representation* of the idea the message is intended to convey. This is the task of the *semantic* part of the system, as opposed to syntax (sentence structure), morphology (word structure), phonology (sound structure) and other parts which deals with the actual realization into language of the message. The output from the semantic part of the system should be *independent of language.*

What should this semantic information look like? The classical answer to this question has been first-order logic, which is a very powerful way to express knowledge. First-order logic can be traced back to the ancient Greeks but emerged fully fledged first in the works of the German mathematician Gottlob Frege, who used a notation which has become notorious for its awkwardness. It has then been endlessly described and first appeared into the world of computer programs in John McCarthy's famous paper from 1958, *Programs with Common Sense* [18]. The sheer power of unrestricted first-order logic, as well as its generality, however leads to a certain brittleness and systems which employ it often do not perform very well.

The problem of knowledge and information representation is not restricted to natural language processing alone, but applies to artificial intelligence in general. To get a useful system, you must compromise the format of the knowledge. Many representations start from first-order logic and impose various restrictions, for example the classical STRIPS logic and its descendants used to make planners feasible, see for example [23].

2.1 Information Extraction Systems

The reason for applying a computer system instead of humans is often that the task at hand is enormous, menial, repetitive, and very time-consuming for a human. A typical example is the task of collecting stories about a certain topic from newswires, recording a few specific details and storing the text along with the information in a database. Collecting and processing for example business reports would be a full-time job for a human. If a computer could do this task instead, large amounts of time and money could be saved. This is a typical task of *information extraction* systems.

More precisely, an information extraction system is a computer system which reads a text and tries to answer a *finite number of questions* about its content, like "what company was taken over", "how many people were injured in the bombing incident", and so on. They are typically applied in a very restricted domain. The problem of representing knowledge has thus been reduced to the filling of a *template* with a fixed number of slots. Together with the restriction of domain (which makes it possible to fine-tune the system to a high degree), this simplicity of the knowledge representation is probably a significant reason for the success of information extraction systems.

But on the other hand, this restriction of what information is expressible is also a restriction to the range of application of the information extraction system. The canonical example of an information extraction system is Fastus [15], which used domain-specific templates for its knowledge representation in the different domains it was applied. Fastus thus had to be partially rewritten for each new domain where it was applied, which clearly must have been an annoyance.

The question then is how general, or how specific, the knowledge representation should be.

2.2 The Berkeley FrameNet

We will here have a look at an attempt to strike a middle ground between the two ends of the spectrum: the powerful, general and intractable first-order logic and the rigid, specific and simple templates of information extraction. This is the Berkeley FrameNet project [3]. FrameNet is organized around the concept of the *frame.* A frame can more or less be described as an object representing a situation. As an example of a frame which is relevant for our project, we will discuss the Impact frame which represents the situation that physical objects (for example cars) collide. Each frame has a number of *frame elements*, which are slots where the relevant

information about the situation can be stored, where each slot represent a role in the situation. The Impact frame has nine frame elements which are listed in Figure 3. In addition, there are other elements which the Impact frame shares with other frames, such as Place, Time, and others.

Cause	The reason for which an Impact occurs.
Force	The amount of force in the course of the impact.
Impactee	The entity which is hit by the Impactor.
Impactor	The entity that hits the Impactee.
Impactors	The multiple entities that collide.
Period of Iterations	The time throughout which the impact repeatedly takes place.
Result	Result of an event.
Speed	The speed at impact.

Figure 3: The frame elements of the Impact frame in FrameNet.

To connect this idea into actual language use, the FrameNet team works with a large corpus to obtain example sentences and recorded the *headwords*, which are the words that realize the frame into language. For example, in the sentence "The car hit the tree", the word "hit" is the headword. In Figure 4, the recorded headwords of the Impact frame are listed, and in Figure 5 two examples of the Impact frame in road accidents taken from the FrameNet corpus. The headword is in bold font.

Verbs	bang, bump, clang, clunk, collide, crash, crunch, graze, hit
Nouns	collision, crash, hit, impact
Adjective	glancing

Figure 4: Recorded headwords of the Impact frame in FrameNet.

Then [*the police car*]$_{Impactor}$ *had* **crashed** [*into a fence*]$_{Impactee}$.
Two people were injured in a [*three car*]$_{Impactors}$ **crash** [*on a back road near Woodham*]$_{Place}$.

Figure 5: Two tagged accident sentences from the FrameNet corpus.

FrameNet has evolved out of a classical way of representing semantic information: semantic cases. In modern linguistics, this approach has been most famously described by Fillmore in [10]. Fillmore is today the main participant of the FrameNet project. The idea of semantic cases is similar to FrameNet, but instead of the frame-specific roles of FrameNet, ten to twenty all-round roles are used, for example Actor, Cause, Instrument and so on. This idea comes from the syntactic cases of classical languages like Latin, which can be said to be the mapping into syntax of the semantic roles. In fact, this idea is very old and seems to have occurred first in ancient India about 2500 years ago, see for example [16]. There is one serious drawback with this approach: that each role associated with a situation needs to be squeezed into one of the all-round roles. FrameNet is an attempt to sidestep this annoyance.

Although FrameNet is very generally designed, the concept of the frame is quite similar to information extraction templates and it is clear that it has important implications for information extraction technology. In [11], an introduction to knowledge representation using frame semantics in an information extraction context is given. A recent work done by Gildea and Jurafsky [12], which is very important from an information extraction point of view, is an experiment where they try to automatically identify the semantic roles of the constituents in a given input sentence. The semantic roles can be either the all-round rules of the classical case approach, or the frame-specific roles of FrameNet.

2.3 Representing an Accident

What information about a road accident is relevant to present it graphically? This is one of the questions that this report tries to answer, but we can already give some preliminary hints. Clearly, the participating vehicles must be represented. We would also like to know their movements and actions. All non-moving entities significant to the accident, such as trees or traffic lights, will also need to be recorded. Finally, some information about the environment, for example what kind of road the accident takes place on, might be essential for correct understanding of the accident. The tragic details about human casualties are not handled by the CarSim system, just the sequence of events constituting the accident.

This information might be represented by means of any of the abovementioned methods of knowledge representation. The implementers of the original CarSim system [7] initially used a quite complicated knowledge representation, but they failed to obtain result until they lowered their aim and switched to information extraction techniques using a relatively simple template format, which was intended to be able to express exactly the information needed for a graphic presentation.

The template contained the following pieces of information:

- A list of *dynamic objects*, which represented the participants in the accident. Each dynamic object contained a list of movements (so called *events*).
- a list of *static objects*, which included every object present on the scene of the accident which was not a dynamic object.
- a list of *collisions* between a dynamic object and a static or a dynamic object.

In Figure 6, we see an example of a template produced by the CarSim 2.0 linguistic component for the following text:

> "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."

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE accident SYSTEM "file:accident.dtd">
<accident>
    <staticObjects>
        <road kind="straightroad"/>
    </staticObjects>
    <dynamicObjects>
        <vehicle id="bus1" initDirection="east" kind="truck">
            <startSign>State Route 79</startSign>
            <eventChain>
                <event kind="leave_road"/>
                <event kind="overturn"/>
            </eventChain>
        </vehicle>
    </dynamicObjects>
</accident>
```

Figure 6: An example template produced by the CarSim linguistic module.

This version of CarSim used XML as the output format. We see that one dynamic object is present: the Greyhound bus, which is represented as a truck since the only vehicles in CarSim 2.0 are cars and trucks. One static object, the road, is recorded. The movements of the bus are recorded in its event chain. No collisions took place during the accident, so we find no collision list in this template.

This template format more or less served its purpose in the original CarSim system, which was intended to visualize only very simple accident reports. But the simple format proved to be too limiting when more complicated reports were to be processed. In section 3, we will try to overcome these limitations.

2.4 A Case Study: WordsEye

What you invariably find if you do a Google search for "text-to-scene conversion" is the ambitious WordsEye project [6], which is a system which converts small texts into images. We will here give a short description of this system and briefly discuss its knowledge representation and contrast it with the representation used in CarSim.

The goal of WordsEye is, in the words of its authors, "to provide a blank slate where the user can literally paint the picture with words", that is to create a natural-language alternative to the usual 3D software tools which are typically quite complex. Although the authors give some examples of application, such as e-postcards, cookbook instruction and product assembly, the WordsEye system above all seems to be intended as a research system to investigate the relation between text and images. They don't seem to have tested the system on real-world texts.

The authors of WordsEye have a very different aim compared to CarSim: they want their system to depict almost any kind of participant in any kind of situation, unlike CarSim which invariably depicts vehicles in road accidents. CarSim is intended to reproduce the accident as it occurred in reality, as far as this is possible, but WordsEye has no such aim. Another fundamental difference is that in CarSim there is a *timeline*, that is the sequence of events constituting the accident. Since WordsEye and CarSim have very different needs, we would expect their knowledge representations to be quite different from each other.

The knowledge representation of WordsEye consists of a list of *semantic representation fragments* which are derived from the output of the parsing engine. The main types of semantic fragments are *Entity*, *Action*, *Attribute* and *Relation*, which roughly correspond to nouns, verbs, adjectives and prepositions. During recent research on WordsEye, its authors have been incorporating semantic frames using FrameNet instead of their own indigenous verb frames playing a roughly equivalent role.

```
(("node2" (:ENTITY :3D-OBJECTS ("mr_happy")
           :LEXICAL-SOURCE "John" :SOURCE SELF))
 ("node1" (:ACTION "say" :SUBJECT "node2"
           :DIRECT-OBJECT ("node5" "node4" "node2") ...))
 ("node5" (:ENTITY :3D-OBJECTS ("cat-vp2842")))
 ("node4" (:STATIVE-RELATION "on" :FIGURE "node5" :GROUND "node7"))
 ("node7" (:ENTITY :3D-OBJECTS
           ("table-vp14364" "nightstand-vp21374"
            "table-vp4098" "pool_table-vp8359" ...))))
```

Figure 7: WordsEye's representation of the sentence "John said that the cat was on the table".

In Figure 7 we see a slightly simplified example of the semantic representation (in the form of a Lisp s-expression since the implementation language of WordsEye is Common Lisp) which the WordsEye linguistic component would produce if given the sentence *"John said that the cat was on the table"*. We can see that it consists of five semantic representation fragments: John, the cat and the table (three entities), an action "say" and a relation "on".

We see that WordsEye has a fairly general knowledge representation, which is designed to be able to express quite complex relationships between objects, unlike the relatively rigid template format of CarSim 2.0. The involved objects can be described with an arbitrary number of attributes, and they can be grouped together using spatial relations. But there are similarities to CarSim as well: all the involved objects and their actions must of course be represented in some way.

3 A New Knowledge Representation

This section will discuss what information a knowledge representation for an ambitious and large-scale text-to-scene conversion system should be able to express. The first subsections describe some reasons why a new representation is needed. Then, the design is described.

As the data structure holding the knowledge information grows increasingly complex, it might be argued that it should not be called a "template" since it no longer resembles simple forms to fill, but for convenience we will still stick to this term.

3.1 Lessons from CarSim 2.0

The feeling I had after working on CarSim 2.0 for a while was that although we obtained results, we were rapidly approaching the limits of manageability. Each new addition to the system added more to the complexity. In fact, the most ambitious project I had taken on – the coreference resolution module (see subsection 5.3) – was implemented as a purely textual preprocessor, which was placed *before* the text was input to the information extraction module. The reason for this was simply that the intricacies of the inner workings of the information extraction module made it impossible to make substantial changes without making everything fall apart. Andersson [1] expresses similar sentiments after building a Swedish version of the CarSim system using the same architecture. Although the the CarSim 2.0 system had worked well given the small ambitions of that project, it was clear that it had not been properly designed to grow into a large system.

A significant reason for this was the absence of an explicit knowledge representation. There was one component called the Template, but it was not designed as a pure storage of information but was intimately tied to the Link Grammar parsing, and adding and using information was extremely inconvenient. This was clearly something which had to be seriously addressed in order to build a system which could cope with the increasing complexity we were expecting to face.

Another weakness was that the design was not explicitly modular. The different stages in the information extraction process were implemented as some obscure methods in the Template class. The responsibilities were also not strictly defined, which made it very difficult to predict what effects a small change somewhere would cause "downstream". Although this issue is not the main topic of this report, it will be addressed briefly in subsection 5.1.

3.2 Requests from Users

Since we wanted some feedback from our potential users, we had a meeting with our reference group consisting of some members of the Department of Traffic and Road. They were shown a short demonstration of the CarSim system and came up with a lot of suggestions for improvements, for instance:

- A lot of road configuration details which can be seen in the texts are not visualized by the system. This includes the number of lanes, side guardrails and central separation, road markings and signposts.
- A symbolic representation of the environment is needed. The current system always displays something which evokes a rural environment.
- We should be able to describe the weather and the state of the road (slippery, dry, wet, ...).
- Human beings, not just vehicles, could be present on the scene of the accident.
- The system needs to be neutral about details which are not specified in the texts. For example, the road markings which are presently displayed conveys too much meaning to the experts.
- The presentation should be improved. This includes a facility for changing the viewpoint, and possibly to incorporate a soundtrack.

Their suggestions were very valuable to us, since they came from people who have an opposite viewpoint compared to us: they are experts in the field of traffic, but have no expertise in the field of computer science. To satisfy them, the system needs to be precise about details regarding the traffic situation and easy to use in practice.

What is evident from their suggestions is that we need a more descriptive way to represent details about the environment and the road, and more kinds of objects, for example human beings, will need to be included in a new knowledge representation.

3.3 Shortcomings of the Old Template

As previously mentioned, the original template consisted of the following components:

- a list of static objects,
- a list of dynamic objects,
- a list of collisions.

The road, some indication signs, traffic lights and obstacles such as trees were counted as static objects. Each dynamic object – a car or a truck – kept a list about the events (except collisions) it was involved in. Some road name signs were kept in the road object, some other road name signs were kept in the dynamic objects.

We will now have a look at some of the shortcomings of this design.

First of all, the most obvious drawback was the very small vocabulary of objects which could occur on the road. For example, cars and trucks were the only vehicles which could appear, and trees and traffic lights were the only obstacles. This is however not a fundamental problem, since the remedy is just to add more objects.

Moreover, there was significant conceptual confusion about the notion of a "static object", which seemed to be a catch-all notion which captured all information which could not reasonably be related to dynamic objects. There was no clear idea of what signified a static object, except that it was not a dynamic object. For example, some static objects (obstacles and signs) had a specific location but others had not (roads), and some could be involved in accidents (obstacles) but others could not (signs and roads). Digging deep in the code, one could find a notion of "static crashable object" beginning to emerge, which is clearly a sign that an insight was coming to the authors that the description language was lacking something. That there was no way of knowing what to expect of a static object, and the unstated rules about which static objects could do what, made the information extraction more complex.

Another problem is where the details requested by the reference committee should be put. Sticking to the tradition of the designers of the old template, the obvious way would be to label them as static objects, but as we have just discussed, this is not the way to go.

3.4 The Scene Object

A way towards some conceptual clarity could be to envision the environment where the accident described in the text takes place as a *scene*, resembling the stage of a theatre. The scene should be seen as an empty space waiting for the physical objects to be spread out. The scene itself has some properties, like the light, type of terrain and weather. The road should also be seen as a part of the scene, and so should the "abstract" static objects such as road names, speed limits, and so on.

The scene object should hold all information which is relevant about the accident. The implementation suggested here contains the following information:

- Date, time and location of the accident,
- type of environment (urban, forest, rural or mountainous),

- weather conditions (clear, cloudy, rainy, snowy or foggy),
- light conditions (light or dark),
- state of the road (dry, wet, icy, snowy, muddy),
- a road configuration object, which contains all information about the roads of the scene.

The road configuration object, which is contained by the scene object, either consists of a single road object (a straight road or a bend), or is a compound of more than one road (a crossroad, t-crossing or a roundabout). In each road object, the following pieces of information is kept:

- The name of the road,
- the type of road (highway, street, gravel or a road under construction),
- the kind of separation between the left and right side (nothing, dashed line, straight line, guardrails or embankment),
- the number of lanes on each side,
- the speed limit on each side,
- the type of edge on each side (same types as separation mentioned above).

3.5 Road Object Ontology

A central problem of knowledge representation is the formulation of *what entities exist* and *how they are related*. When this is expressed explicitly, we have what is known in artificial intelligence as an *ontology*. In this subsection, the ontology of the physical objects which appear in accidents is discussed.

Since so many projects working on knowledge representation spent large amounts of time developing their ontologies, some attempts have been made to develop standard ontologies which can be used in many different situations and which should not be restricted to any particular problem domain. Such an ontology is called an *upper ontology* or *foundation ontology*. The most well-known ontology is Cyc [17], which consists of an upper ontology along with several domain-specific ontologies. Cyc is proprietary, but a subset of it has been released under the LGPL. Among other attempts to formulate ontologies, which are also nonproprietary, are the Suggested Upper Merged Ontology or SUMO which is being created by an IEEE working group, and the ThoughtTreasure ontology [20]. The design of general ontologies is extensively discussed in [23].

But, as we have already touched upon in other parts of this report, if one strays too far from the hard realities of domain-specificness, one can expect a decline in usability. This certainly applies to ontologies as well, and such a large-scale ontology is typically not very useful in itself but needs to be adapted to the particular problem domain to be valuable in the real world. A very interesting attempt in an information extraction context to attack the generality problem from the other direction, that is to construct a system which is *easy to rebuild*, is described by Poibeau in [22]. His system prototype uses machine-learning techniques in assisting the user to define domain-specific ontologies for information extraction systems.

CarSim currently does not have the ambition or scope to motivate the use of an upper ontology, so we have instead opted for a small hand-developed ontology consisting of about fifty different kinds of road objects. We do however make use of the well-known WordNet hierarchical dictionary (see subsection 5.4). WordNet is sometimes known as an upper ontology, but this is not strictly correct since WordNet is specific to a certain language (typically English) and the relations and distinctions between concepts are different in every language. For example, the verb "to put", which is represented by a single sense in WordNet, has three counterparts with distinct meanings in Swedish: *sätta*, *ställa* and *lägga*. An ontology, on the other hand, should be designed independently of any language. Hence a mapping from the WordNet synsets to the concepts of the ontology we use will be needed, and this topic will be briefly addressed in subsection 5.5.

A minimal ontology would resemble the (implicit) ontology of the original CarSim system: just a small list of objects which may be present in an accident. In addition to just a list of possible objects, we might add relations between the objects to our ontology. One relation is present in almost every ontology: the *is-a*

relation, or as it is also known: *inheritance*. This is the idea that sets of objects have properties in common with other sets, and that it is useful to refer to these sets as a whole. For example, we might say that the set of cats inherits from the set of mammals, because cats breast-feed their young. Computationally it is most convenient to restrict the inheritance to *single inheritance*, which means that each set inherits from at most one superset and the inheritance graph becomes a tree. This is of course sometimes an unacceptable restriction, for example if we want to refer to cats *both* as mammals and as carnivores. The idea of inheritance is most useful but is more problematic than it might seem at first sight: one paper [5] detected 29 different interpretations of this relation. In [30], a thorough mathematical analysis of the concept of inheritance is performed.

The ontology of CarSim 2.0 (which was never actually expressed as an ontology) did not explicitly make use of inheritance. There was, as previously mentioned, a grouping of objects (achieved by means of two separate lists) into static (nonmoving) objects and dynamic (moving) objects. Although there of course is a difference between static and dynamic objects, this separation and lack of a common way to refer to them sometimes makes programming cumbersome. For example, if we would like to represent that an vehicle hit something in an accident, there should be no fundamental between hitting a static object and hitting a dynamic object. Therefore, we need to have a common way to refer to static and dynamic objects. The way to do this is to say that both static objects and dynamic objects belong to the general category of *road objects*. Since arbitrary subclassing (although only single inheritance) was supported in the language which had been defined to facilitate the implementation of a knowledge representation (see subsection 4.2), this could easily be solved by making `StaticObject` and `DynamicObject` inherit from a general `RoadObject` class.

When the idea of inheritance thus has been brought into the road object ontology to unify the static and dynamic objects, we might wonder if it can be used for other purposes. Since we want to bring more classes of objects into our knowledge representation, maybe it could be a good idea to group them into further subgroups, not just static or dynamic objects. This could be advantageous, since there could be properties shared by certain types of object but not by others. It could also be helpful in a less ad-hoc coreference resolution algorithm, see subsection 5.6.

A couple of questions then occur: when is it useful to group classes of objects into subgroups? Could extensive use of inheritance be harmful? It might be useful to specify some criteria for when the inheritance should be used. In general, inheritance was used when one or more of the following held:

- The objects shared some attributes, for example the horizontal obstacles which shared a direction attribute.
- The system needed a common way to refer to the objects, like the `RoadObject` superclass used to capture both static objects and dynamic objects.
- It was likely that the superclass could be used in coreference situations. It would for example be likely that an entity introduced as “a large bus” would later be referenced as “the other vehicle”, and therefore `Bus` should be a subclass of `Vehicle`.

The road object hierarchy was now worked out. No systematic method (except just reading the NTSB corpus) for finding concepts was used. Most concepts and their arrangement into subgroups emerged during a number of discussions with Pierre Nugues and Per Andersson, but some were later added during evaluation (see subsection 3.9).

The following subgroups of static objects can be found in the proposed road object hierarchy:

- *Obstacles*, which are those objects which could be crashed into. The obstacles are further divided into vertical obstacles such as trees, poles and rocks, and horizontal obstacles such as walls, fences and buildings.
- *Surface objects* such as patches of water and ice.
- *Hollow objects* which dynamic objects can fall into, such as pits, ditches and pools.
- *Indication objects* for traffic, such as pedestrian crossings and traffic lights.

The dynamic objects are grouped into the following subgroups:

- *Vehicles* such as cars, buses and trucks.
- *Trailers* and caravans.
- *Animates* such as pedestrians and large animals.
- *Moving objects* such as mechanical bridges.

Some nonphysical properties of the scene, such as road names, speed limits, were earlier regarded as static objects which would be realized as signs along the road. I believe that it is healthy to have a clear idea about what the notion of static object means, and that the obvious definition is that a static object is a nonmoving entity which exists at a certain place in the physical universe. Hence the abstract properties should be represented in some other way, and as previously described they have been incorporated in a natural way into the scene object. Another reason why the abolishment of the abstract properties as static objects makes sense is that the method of displaying these properties is completely arbitrary and could be changed according to the user's preferences, unlike the real static objects which should of course resemble their physical counterparts.

3.6 Representation of Dynamic Information

Now that we have described how the scene information as well as the participating objects can be captured by the knowledge representation, we now turn to the representation of the events which which constitute the accident. This information will be called the *dynamic information*. This subsection will discuss how to represent this dynamic information.

The CarSim 1.0 and 2.0 systems made a distinction between *events* and *collisions*. The events represented the movements of the participants, for example turning, driving forward or stopping, and were thought of as *intervals* of the timeline of the accident. The collisions on the other hand were thought of as *points* on the timeline. This distinction lies at the core of the planner module of the graphical component. Therefore, we will here investigate a template based on this approach.

We define a template where this information information is kept similarly to how it was done in the old CarSim system: collisions are kept in a separate list, but all other kinds of events are kept by the dynamic objects in their own lists. In Figure 8, we see the relevant parts of the code that defines a template using this model. (For an explanation of the format of the description file, see subsection 4.2).

To elucidate, I present an example. Imagine that we have a car and a truck, and that there is a collision between them while the car turns left and the truck turns right. Then the template will contain the following information:

- The car will have a turn-left in its event list.
- The truck will have a turn-right in its event list.
- The collision list will contain a collision between the car and the truck.

CarSim 1.0 and 2.0 would mark the turn-left and turn-right events as "critical", which meant that these were the events during which the collision took place. This might seem like a strange idea since if there are more than one collision described in the report, we cannot say which events were related to which collision. It however makes sense if you remember that the original CarSim system was designed for insurance company reports describing very small and harmless accidents. With the NTSB texts, when there are sometimes very complicated serial collisions involving over a hundred vehicles, we have a completely different situation. Hence, if a collision happens during an event, the event object needs to contain a reference to the collision object in question.

The chronology of events and collisions is represented by means of the following rules:

- Events in the event chains are ordered in time.
- Collisions in the collision list are ordered in time.
- An event and a collision which are linked are simultaneous.

```
<?xml version="1.0" ?>
<!DOCTYPE package SYSTEM "../autogenerator/autogen.dtd">
<package name="se.lth.cs.carsimcore.template"
         dtd="file:template.dtd"
         root="Template">

  <class name="Template" comment="This is the main wrapper class.">
    <attribute name="scene" type="Scene">
    <attribute name="object" type="RoadObject" collection="yes">
    <attribute name="collision" type="Collision" collection="yes">
  </class>

  <class name="RoadObject">
    <class name="StaticObject">
      <class name="Tree"/>
    </class>
    <class name="DynamicObject">
      <attribute name="event" type="Event" collection="yes"/>
      <class name="Car"/>
    </class>
  </class>

  <class name="Event">
    <attribute name="collision" type="Collision" reference="yes"/>

    <class name="DrivingForward"/>
    <class name="Turn">
       <attribute name="direction" type="Direction"/>
    </class>
    <class name="Overtake">
       <attribute name="otherObject" type="DynamicObject"/>
    </class>
  </class>

  <class name="Collision">
    <attribute name="actor" type="DynamicObject" reference="yes"/>
    <attribute name="actorSide" type="Side" default="unknown"/>
    <attribute name="victim" type="RoadObject" reference="yes"/>
    <attribute name="victimSide" type="Side" default="unknown"/>
  </class>

</package>
```

Figure 8: Code fragment of the first template definition.

3.7 Some Objections

This template is very similar in spirit to the template used in the old CarSim system, and this has the very significant advantage that the modifications in the planner used in the visualizer would be relatively small. Therefore, this is the template which has been used during the evaluations described in later sections. But this approach still has some drawbacks and these will be described here, along with an example of how another template format could look.

First, one sore thumb which sticks out is the separation between collisions and other kinds of events. It is above all aesthetically unappealing, but it has some negative consequences in practice as well: if an object is involved in collision while there are no particular events, dummy events (for example driving-forward) must be added to the event list to allow storage of the links to collisions. We could of course allow multiple collision links from an event, but at least one event is always needed. Schultz [26] is also critical of this separation.

Another complication, which is a case of lack of descriptive power, is that there is no way (except implicitly through collisions) to relate events to each other in time. For example, imagine a situation where an unexpected turn of one vehicle forces another vehicle off the road (without collision). Then, we would like that the turn occurs *before* the leave-road event. We would also like to describe that two events occur at the same time. A remedy to this problem could be to add *constraints*, for example in the form of a list of references to the preceding events. Another approach to this problem could be to make the timeline of the accident more visible to the linguistic component (presently this is exclusively the domain of the planner in the graphical component), and allow time information about the events or collisions to be expressed whenever this is necessary.

3.8 A Dynamic Information Representation Based on Frames

In this subsection, another approach to the problem of representing the dynamic information will be outlined. This approach is somewhat influenced by semantic cases and the FrameNet, which have previously been described in subsection 2.2. The relevant code of the definition of the template is shown in Figure 9.

```
<?xml version="1.0" ?>
<!DOCTYPE package SYSTEM "../autogenerator/autogen.dtd">
<package name="se.lth.cs.carsimcore.template"
         dtd="file:template.dtd"
         root="Template">

  <class name="Template">
    <attribute name="scene"  type="Scene"/>
    <attribute name="object" type="RoadObject" collection="yes"/>
    <attribute name="frame"  type="Frame" collection="yes"/>
  </class>

  <class name="RoadObject">
    <class name="StaticObject">
      <class name="Tree"/>
    </class>
    <class name="DynamicObject">
      <class name="Car"/>
    </class>
  </class>

  <class name="Frame">
    <attribute name="actor" type="DynamicObject" reference="yes"/>
    <attribute name="predecessor" type="Frame" reference="yes"
                                             collection="yes"/>
    <class name="IntransitiveFrame">
      <class name="DrivingForward"/>
      <class name="Turn">
        <attribute name="direction" type="Direction"/>
      </class>
    </class>

    <class name="TransitiveFrame">
      <attribute name="victim" type="RoadObject" reference="yes"/>

      <class name="Impact">
        <attribute name="subjectSide" type="Side"/>
        <attribute name="objectSide" type="Side"/>
      </class>

      <class name="Overtake"/>
    </class>
  </class>

</package>
```

Figure 9: Code fragment of the second template definition.

As previously, we need to define different types of events which can then be collected in a data structure. This time we drop the distinction between collisions and other kinds of events. We also drop the separate event chains and introduce a more general mechanism: each event contains a list of preceding events. This allows us to express that an event of one participant happens before the event of another participant. We could express simultaneousness if we stipulate that if an event happens a the same time as the other, then the first event will be marked as a predecessor of the second and the second will be marked as a predecessor of the first.

It is clear that this representation of the dynamic information is *strictly more expressive* than the other, since the separate event chains could be reconstructed given a collection of frames with ordering constraints. The ordering of the frames could on the other hand not be reconstructed, since the event chains only represent chronology internally and through collisions.

The definition of events uses inheritance, but not in the same sense as FrameNet where inheritance between frames is used to group frames with similar *meaning* together, for example when the Driving and Walking frame inherit the Transportation frame. Here the inheritance is used to group situations with similar *structure* together, which is meant to facilitate for the linguistic analysis. For example, the `DrivingForward` and `Turn`

will both inherit from `IntransitiveFrame`, which are the class of frames with an actor but no victim, while `Impact` and `Overtake` inherit from `TransitiveFrame`, which denotes the class of frames with both actor and victim. When we add more frames with a direction attribute, for example `LeaveRoad` and `Overturn`, we could group these together with `Turn` under a common `DirectedIntransitive` class.

3.9 Evaluation of the Descriptive Capacity

When a tentative new knowledge representation had been developed, we had to check if it could describe all relevant information in the accident reports which were available. Since there was no automatic tool ready which could fill the templates – this is something that will be implemented in the future – I assumed the role of a perfect information extraction engine and tried to describe some texts using the new knowledge representation.

There is of course a problem about the implementer doing the evaluation. Ideally, independent analysts (for example the reference committee) would have done this analysis. But this task is very time-consuming and the analysts need to understand properly how the knowledge representation works. Therefore, inappropriate as it may be, I was the only one who had the opportunity to do the analysis.

The strategy which was adopted was to assume that an ideal information extraction mechanism was available. All worries about "unimplementable" features were ignored. The texts were then read and the extractable information put into the templates. They were then subjectively classified as follows:

- *Good* if all relevant information can be captured by the template,
- *Acceptable* if most important details can be represented,
- *Insufficient* if there is any detail which is essential to the understanding of the accident which can not be represented.

In general, we evaluate if all information can be represented at all, not if it is conveniently expressed. In some cases, results have been downgraded because the description was very clumsy.

The evaluation strategy was as follows: the performance on the NTSB texts was measured while making changes to the knowledge representation until as many texts as possible can be represented. Then the performance on some of the Swedish news texts was evaluated and compared to the performance on the NTSB texts. This might of course be questionable since the two corpora are very different.

The performance figures on the 22 NTSB texts can be seen in Figure 10. As we can see, the results are not perfect, which might seem strange since the knowledge representation was continuously updated to make the relevant information from these texts fit into the templates. This is due to the fact that a limit had to be set somewhere, and some texts contain significant details which would need a completely different knowledge representation. For example, the texts where the vehicle get out of control due to steep slopes cannot be described, since we have intentionally omitted all information about the shape of the landscape. Some other texts are extremely complex.

Good	8
Acceptable	11
Insufficient	3

Figure 10: Performance figures on the NTSB corpus.

Then, the performance on the Swedish news corpus was measured, and the results from this experiment can be seen in Figure 11. The performance on these texts is good, with only one text marked as insufficient. This text described an accident which was caused by a truck which dropped its load. This scenario is a bit exotic, and it is unlikely that it will be covered by the knowledge representation in the future.

Good	5
Acceptable	10
Insufficient	1

Figure 11: Performance figures on the Nyheter corpus.

The manually filled templates can be viewed on the web. Each text is displayed along with its resulting template. The address is `http://www.df.lth.se/~richardj/carsim/eval`, but for copyright reasons we have restricted the access to this page to computers inside the LTH network.

4 The Implementation

In this section, the incremental process of implementing a new knowledge representation and the development framework which evolved are described.

4.1 Early Stages

The first attempt at a new knowledge representation was to make the implicit knowledge representation of CarSim 2.0 explicit, that is to implement a *pure store of information* with no intricate dependencies, just a simple interface to the XML document which eventually should be output from the linguistic component.

An interface to XML documents as node trees is provided by the Java libraries and is called *Document Object Model* or DOM. The definition of DOM is not tied to or particularly useful for the Java language, and working directly with the DOM trees would not be a convenient way of handling the knowledge representation, and would surely not be the way to go for a growing system. Instead, the DOM tree was hidden behind wrapper classes. The first version of the new knowledge representation was just an implementation of wrapper classes for the DOM trees representing XML documents conforming to the DTD of CarSim 2.0.

The wrapper classes were meant to provide a convenient programming interface to the XML document. For example, instead of the laborious and error-prone work of inserting a subtree representing a vehicle, the user could just tell the `Template` object to insert a new `Vehicle` object into its list of dynamic objects.

The focus of the first implementation was to improve *usability*, not *expressivity*. Thus the DTD, which describes which XML documents are legal, was left untouched.

The first implementation was completed but never used in practice. The focus soon shifted towards giving a more *expressive* knowledge representation, partly in response to the requests which were given by the reference committee. The work was now focused on refining and expanding the DTD, and after each change the wrapper classes were modified to reflect the changes.

4.2 Automatic Generation of DOM Tree Wrapper Classes

The task of writing wrapper classes was easy and soon converged into repeating well-known patterns, but it was boring, error-prone and time-consuming and was an obstacle to our creativity during the improvement of the expressive power of our knowledge representation. It seemed that we were again hitting the walls raised by a bad design strategy, and that these limitations had to be overcome in order for the system to grow large while still keeping its health.

If the task of writing these wrapper classes by hand could be eliminated, it would clearly be a benefit to the project. This could be achieved either if the wrappers were general and powerful enough to be able to handle each new DTD without having to be rewritten, or if the code needed for the wrapper classes (which was very predictable and repetitive) could be provided automatically. In the framework of Java/XML, the first alternative was considered impossible, so the focus had to be on the second alternative.

The idea of making repetitive chores automatic is an old one and it is strange that the present Java language provides absolutely no support for this. It has indeed been incorporated in a number of programming languages. The purely textual macro system of the C programming language, which is notoriously error-prone, is a first step. The templates of C++ are more powerful and expressive, but the idea is full-blown only in the different macro systems of the Lisp family of languages, where the notion of *programs which transform programs* is clearly articulated, and which makes these languages so easy to extend with new language constructs. Similar ideas have however finally begun trickling into Java too, and in the 1.5 release (Tiger) there will be a template system similar to the one in C++, in addition to "metadata" tags to make interaction with tools which automatically generate the code easier.

In present-day Java, this task has to be accomplished through automatic generation of Java files. As I had previously experienced the benefits of automatic code generation while working with compiler frontends, this idea came naturally. The question was then, could an existing tool or library be found, or should a new be developed?

The idea of automatic code generation came from the JastAdd tool written by Görel Hedin and Eva Magnusson [14], which generates Java classes for nodes in the abstract syntax trees found in compiler frontends. With this tool you build a class hierarchy using a description file with a very concise and readable syntax. Although this tool is nice, it was soon discarded since the implementation had to provide textual (preferably XML) data input and output, which JastAdd did not.

What was needed was a tool which could save us from the tedium of writing the Java classes, but this was not the only concern. The unexpressive DTD language was also an annoyance, since the simple constructs it provides did not suffice to express the kinds of structures we wanted, for example subclass relations. The DTD replacement, XML Schema, provides more expressiveness but since Schema is not yet widely supported we still wanted to stick to DTD, although we may switch to Schema in the future. Hence, a new means to express the data structures was needed, and given this a generator should produce both a DTD and the necessary Java wrapper classes.

Although XML is ugly, the obvious way of creating a task-specific programming language, especially in a Java framework, is to use XML as the format. Then the parsing and syntax-checking come for free, and the programs which would use the described information could be built using the large support libraries for XML in Java, or use the powerful XSLT transformation language. Although XSLT is the usual way to generate new data given an XML document, it was decided to opt for a generator written in Java, partly due to the data structures needed (which more or less resemble those of a compiler) and partly due to my complete inexperience with XSLT. Some examples of code generation using XSLT can be found in [24].

The description language that was settled for consists of the following constructs:

- A package declaration, which is the top element of the XML document.
- Class declarations, which will be output as DTD elements and Java classes. The subclasses of a class are nested within its class declaration.
- Enumerated type declarations, that is a type consisting of a small number of values (for example the type `Direction` having the values `Left` and `Right`). The enumerated types which will appear in the Tiger release of Java are not employed here, but might be in the future.
- Attribute declarations of a class. Each attribute has a type, which is either a class, an enumerated type or a primitive type (string, integer or boolean). They can also be marked as collections (which denotes a container of zero or more element) or references (which means that the actual data are stored somewhere else in the XML data structure). A default value for the attribute may also be given, and attributes may be marked as mandatory (which means that a value must be provided for the document to become legal).

An example of how a description file looks can be seen in appendix A and Figures 8 and 9. The contents of these files have been thoroughly discussed in section 3

In Figure 12, we see an example of how the data are represented in XML. (The data were produced during the experiment described in subsection 5.6.) What is important to note is that subclassing, although not

supported in XML, can be mimicked by nesting the subclass information inside the superclass information. Here, for example, the two objects both have the `introducedAs` string attribute which was declared in the `RoadObject` class. The bus keeps XML elements to tell that it is a dynamic object, a vehicle and a bus, and the embankment has elements organized in a similar way.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Template SYSTEM "file:template.dtd">
<Template>
    <Template_object>
        <RoadObject id="RoadObject1"
                    introducedAs="A 1979 Motor Coach Industries MC-9,
                                  47-passenger charter motorcoach">
            <DynamicObject>
                <Vehicle>
                    <Bus/>
                </Vehicle>
                <DynamicObject_event/>
            </DynamicObject>
            <Position/>
        </RoadObject>
        <RoadObject id="RoadObject2" introducedAs="a rock embankment">
            <StaticObject>
                <Obstacle>
                    <HorizontalObstacle>
                        <Embankment/>
                    </HorizontalObstacle>
                </Obstacle>
            </StaticObject>
            <Position/>
        </RoadObject>
    </Template_object>
    <Scene date="On March 2, 1999, ">
        <RoadConfig>
            <StraightRoad>
                <SideAttrs/>
                <SideAttrs/>
                <RoadAttrs/>
            </StraightRoad>
        </RoadConfig>
    </Scene>
    <Template_collision/>
</Template>
```

Figure 12: Example of XML code used to represent the data.

The first version of the DTD and wrapper class generator was developed in a few days. Its work is performed in the following steps:

- The XML document is read, parsed and validated against its DTD, and a DOM tree is constructed, all with the help of the Java XML libraries.
- A syntax tree is built from the DOM tree.
- The adherence to various semantic rules is checked.
- A DTD is written.
- Java files are written.

Only small extensions, additional rule-checking and bug fixes have been added to the generator since its conception. The generator is of course usable in other contexts than CarSim, so it has been published under the LGPL license and set up for free download on its own web page (presently it resides at `http://www.df.lth.se/~richardj/carsim/autogen`).

As the creative limits set by the laborious work of hand-writing wrapper classes were eliminated, our knowledge representation suddenly began to expand very rapidly, and the number of wrapper classes grew to 79 from the few we had had just before. Presently, there are about 100 wrapper classes in our knowledge representation.

As the complexity of the XML code output from the linguistic component increased, it became desirable to present the code in a more legible way. Therefore, an extensible printout mechanism was added in the library which comes with the generator. Currently, it supports text and HTML printout, but it can be extended to produce output of any format. The HTML, which presents the XML data in nice nested tables, is used to display the accident data in the graphical user interface. An example, which actually is the HTML representation of the data from Figure 12, can be seen in Figure 13.

Template

<table>
<tr><th colspan="2">Template</th></tr>
<tr><td>Scene</td><td><table><tr><td>Scene</td><td><table>
<tr><td>Date</td><td>On March 2, 1999,</td></tr>
<tr><td>Road condition</td><td>dry</td></tr>
<tr><td>Environment</td><td>rural</td></tr>
<tr><td>Light</td><td>light</td></tr>
<tr><td>Road configuration</td><td><table><tr><td>Straight road</td><td><table>
<tr><td>Left attrs</td><td><table><tr><td>Side attrs</td><td><table><tr><td>N lanes</td><td>1</td></tr></table></td></tr></table></td></tr>
<tr><td>Right attrs</td><td><table><tr><td>Side attrs</td><td><table><tr><td>N lanes</td><td>1</td></tr></table></td></tr></table></td></tr>
<tr><td>Road attrs</td><td><table><tr><td>Road attrs</td><td><table><tr><td>Separation</td><td>dashed line</td></tr><tr><td>Road type</td><td>highway</td></tr></table></td></tr></table></td></tr>
</table></td></tr></table></td></tr>
<tr><td>Weather</td><td>clear</td></tr>
</table></td></tr></table></td></tr>
<tr><td>Objects</td><td><table>
<tr><td>Bus</td><td><table>
<tr><td>Events</td><td></td></tr>
<tr><td>Position</td><td><table><tr><td>Position</td><td></td></tr></table></td></tr>
<tr><td>Introduced as</td><td>A 1979 MotorCoach Industries Mc-9 , 47-passenger charter motorcoach</td></tr>
</table></td></tr>
<tr><td>Embankment</td><td><table>
<tr><td>Position</td><td><table><tr><td>Position</td><td></td></tr></table></td></tr>
<tr><td>Introduced as</td><td>a rock embankment</td></tr>
</table></td></tr>
</table></td></tr>
<tr><td>Collisions</td><td></td></tr>
</table>

Figure 13: Example HTML output.

The implementation of the DTD and wrapper class generator was a good example of a work philosophy that I find healthy: solve the general case, and make the desired result come out as a bonus. This has allowed us to create a *domain-specific* knowledge representation which is *easy to rebuild.*

5 Consequences of the New Knowledge Representation

This section discusses the work which will follow the implementation of the new knowledge representation. Most of it has already been done partly or experimentally.

5.1 Architecture of the New System

When a new knowledge representation has been implemented, the remaining infrastructure of the new CarSim system can be designed.

One clear reason which made CarSim 2.0 difficult to manage was the lack of separation between the different stages of the information extraction. In reaction to that, I here propose an architecture which is explicitly *modular*. As in all modern information extraction systems, the system should be implemented as a *cascade* of *processors* with different tasks. Each processor is responsible for certain data to be put into the template, and the processors should ideally be totally independent of each other, so one of them could be removed or replaced without consequences for the other processors.

The data which pass down the stream of processors of course include the template, which contains the information which should be output from the information extraction engine. But some other data beside the template could also need to be passed along as well. The text of the accident report, for example, obviously needs to be available to some stages. Some other stages might request the data in the form of a dependency structure or tagged with parts of speech. One problem in the implementation of CarSim 2.0 was the lack of separation between the template and these auxiliary data – the code managing the template was very intimately tied to some data structures related to the Link Grammar parsing. It is clear that these data need systematic management as well.

5.2 Extraction of Scene Data

The scene object in new knowledge representation contains some small pieces of information which need to be extracted, for example the time, date, location, weather and light conditions on the occasion of the accident. The extraction of these data is a quite traditional information extraction task. Some of these data, particularly the date and time expressions, often come in certain fixed forms, like "at 9 a.m. on April 14, 1999". Fixed patterns of this kind are easily detected using hand-crafted regular expressions, and for the English version of CarSim this had already been achieved during the text simplification phase. The date and time could thus be stored into the template instead of just pruned from the text. We will in the future investigate if other pieces of information can be detected in this way.

5.3 Resolution of Anaphora

In most texts, for convenience and stylistic reasons, the author uses different words to refer to the same entity. This phenomenon is called *coreference*, and must be taken into account by computer systems which handle texts written by humans. Coreference usually comes in the shape of back reference or *anaphora*, which means that the reference points "backwards" to something which was previously mentioned. Forward reference or *cataphora* is less common. The CarSim 2.0 information extraction module had no coreference resolution mechanism. In order to find ways to improve the accuracy of the information extraction, experiments with a very intuitive and simple algorithm for resolving anaphora regarding vehicles were undertaken. Cataphora was ignored. The resolution strategy was as follows:

- Using carefully hand-crafted regular expressions, extract (in the order they appear) the following kinds of expressions: pronouns (in this case just the word "it"), word groups starting with an indefinite article ("a", "an", "another") and word groups starting with a definite article ("the").
- If the expression found starts with an indefinite article, assume that it refers to a vehicle which has not been mentioned previously. Then replace the expression with an identifier for the new vehicle (like "the first car") which will also be used when the back references appearing later in the text are removed.
- If the expression is a pronoun ("it") or starts with a definite article and refers to a general entity like "vehicle", then assume that it refers to the last vehicle previously mentioned.

- If the expression starts with a definite article and refers to a certain kind of vehicle (for example "the car"), assume that refers to the last vehicle of the same kind previously mentioned.
- If no vehicle which is likely to be the referent of a pronoun or a definite expression is found, just ignore the expression.

This is a conceptually simple strategy which is clearly also very easy to implement. It is worth pointing out the underlying linguistic assumptions:

- Assume that every object is introduced with an indefinite article.
- Assume that the referent is always the closest matching entity.
- Assume that later references to the object is not more specific than the introduction. Typically they are less general. Thus if we see an expression like "the car" it is probably not a reference to something like "a large green vehicle", but more likely some car which was introduced earlier in the text. On the other hand, expressions like "it" or "the vehicle" will most likely refer to the latest vehicle previously mentioned, regardless of its kind.

There are some problems which arise out of this simplistic approach. They are of course very serious considerations, but they were for the moment ignored.

- The language of humans does not always behave according to the assumptions made above.
- The word "it" is often just a slot-filler for the mandatory subject in English. This is common in text describing environment conditions, like "it was dark" or "it rained".
- The regular expressions are hand-crafted and it is difficult to make them accept all texts referring to vehicles without making them too general. Cases like "a car crash", "the bus driver" will almost certainly be erroneously classified if only regular expressions are used. It will be impossible to predict all the kinds of vehicles which can be present in the accident reports. For example, should "church activity bus" be hard-coded into the regular expressions?
- Apart from the kind of vehicle, all information in the definite expressions is discarded. For example, ordinal expressions like "the first truck" which could be very useful to the reference resolution are not taken advantage of. Neither are distinctions like "the red car", "the military vehicle" and so on.

The algorithm was implemented in a preprocessor which was placed before the information extraction engine and worked on a totally textual level. This meant that the vehicles detected here were not regarded later on when the information extraction engine was looking for vehicles, which is of course a pity (especially since the Link Grammar-based collision detection very often fails to properly detect vehicles). But on the other hand, it also meant that no internal changes were needed in the quite complicated mechanisms of dynamic object detection.

When the algorithm had been implemented, an evaluation of its success was necessary. How should then the performance of the method be measured? One obvious measure, which we will call the *precision*, is to divide the number of correct answers by the number of answers produced. If the system produces 10 answers and 9 of them are correct, then the precision will be 0.9. But on the other hand, if we have a very cowardly system which produces answers only when they are very easy, we probably don't have the system we want even though the precision will be high. A complementary measure is also necessary: the *recall* which is the number of correct answers divided by the number of answers which would be produced by an ideal system. The cowardly system would have a high precision but a low recall, but a "promiscuous" system which finds all correct answers by producing a mass of answers would have a low precision but a high recall. For an ideal system, the precision and the recall would both be 1. It is also common to measure their weighted harmonic mean, the *F-measure*, which is defined as follows:

$$F = \frac{(\beta^2 + 1)PR}{\beta^2 P + R}.$$

The constant β is a weighting constant denoting the relative importance of precision and recall. If they are equally important, β should be set to 1. The precision and recall measures are used to evaluate many different tasks in natural language processing.

The NTSB texts were then manually read and compared against the output of the implemented preprocessor. I counted and "resolved" the number of back references which were found by inspection of the texts, and then counted the number of detected references and the number of correctly resolved references. It is of course not an ideal situation that the counting and resolution were done by me, since I was the implementer of the algorithm and thus could not be considered impartial, but independent analysts were simply not available.

The task of locating the references in the texts is not easy, and it is often a question of judgment how to count them. The figure of 105 references in the texts should not be taken too strictly, and another judge might arrive at a different figure. The figure of 77 detected references is of course unquestionable since this is just the number of references which the system reports. The number of correctly resolved references is again a matter of judgment.

Total number of back references	about 105
Number of detected back references	77
Number of correctly resolved references	71
Precision	0.92
Recall	0.68
F-measure ($\beta = 1$)	0.78

Figure 14: Statistics for the coreference resolution on the 22 NTSB texts.

The results of the evaluation can be seen in Figure 14. We can see that the precision is relatively high, but the recall is lower. This signifies a tendency to miss a significant part of the references, but most of those which are detected are correctly resolved. The precision problems are probably partly related to the objections above, for example that the ordinal numbers or characteristics are discarded. The simplicity of the algorithm could also be a reason. The recall problems, which are more significant than the precision problems, are usually due to the insufficientness of regular expressions.

The literature on anaphora resolution is vast. [2] gives a good introduction and describes a less simplistic algorithm than the one used here. [19] gives a general overview.

5.4 The Various WordNets

CarSim is intended to be as generally designed as possible, and to make use of external general-purpose components whenever they are useful. One problem which needs to be solved in natural language systems is how to obtain the information about words and their meanings, the *lexical information*. The most ambitious project trying to solve this problem once and for all is the Princeton WordNet [9] and its descendants. WordNet is most conveniently described as a hierarchical dictionary, where the most general words sit at the top and the most specific at the bottom.

The minimal unit of the WordNet database is not the word, but the *sense* of the word. For example, the word "car" has several senses: it could for example mean an automobile or a railway carriage. Disambiguation between word senses is a complex problem in natural language processing which WordNet does not address (and is unable to address, since it deals with words out of context). The senses are grouped together into *synsets*, sets of synonyms, forming what is roughly equivalent to a concept in an ontology, except that WordNet is language-specific and an ontology should be independent of language. Each synset can be linked to other synsets via links, where the most common link types are the *hypernym links* (links to more general synsets) and the *hyponym links* (links to more specific synsets). Other links are possible as well, for example the *meronym link* or part-of link. The current English WordNet (version 2.0) contains about 200,000 senses of words in its database after many years of work.

In addition to the original Princeton WordNet, which is only available for the English language, various similar projects for many languages are currently being implemented. Many wordnets for European languages reside under the umbrella of EuroWordNet [4], which is an organization which provides a common data format and an interlingual structure which for example can be helpful in machine translation. A Swedish WordNet in the context of EuroWordNet is currently under development at the Department of Linguistics at the university of Lund.

The English version of CarSim makes use of lexical information from the Princeton WordNet. The advantages of using a WordNet instead of hand-crafting the huge amounts of lexical information, which has taken years to acquire, are of course plentiful. Svensson and Åkerberg [29] undertook the laborious task of developing a Java interface to WordNet.

As mentioned before, Per Andersson has built a prototype for a Swedish version of the old CarSim system [1]. He had no access to a Swedish WordNet, and all lexical information used had to be added manually after inspecting the corpus. This is something which should be avoided if possible, since the new version of CarSim is intended to be as generally designed as possible. Fortunately there was a solution available since we obtained some parts of the Swedish WordNet being developed at the Department of Linguistics.

To be able to use these data in our system, a facility for using them had to be developed. This was a simple task. First, the indigenous data format used in the EuroWordNet was converted into XML. Then, the extensive XML libraries available to Java were used to develop Java classes to access these XML files. Although the files we obtained only contained a few hundred words, for scalability hash tables were used to make the dictionary search quick.

5.5 How to Integrate the WordNets with Our Ontology

In this subsection, we will see how we can make use of the WordNets in practice. The linguistic module of CarSim 2.0 contained collections of words corresponding to the objects which could appear in the accidents. For example, the truck word collection contained words like “truck”, “lorry”, “tractor-semitrailer” and so on. The word collections were built from WordNet by finding all hyponyms for a given word. As we have previously discussed, the program dealt with just a few different kinds of objects. Hence, the word collections could simply be hardcoded into the program code. When we have a large ontology and a development framework to make it easy to rebuild, we need to replace the hardcoded word collections with a more general mechanism.

One simple way to do this is to use a dictionary where words from WordNet could be mapped to class names from the ontology. For example, a lookup on the word “lorry” would yield the class name `Truck`. In Figure 15, we can see a schematic figure of this procedure. The question is how the dictionary should be built.

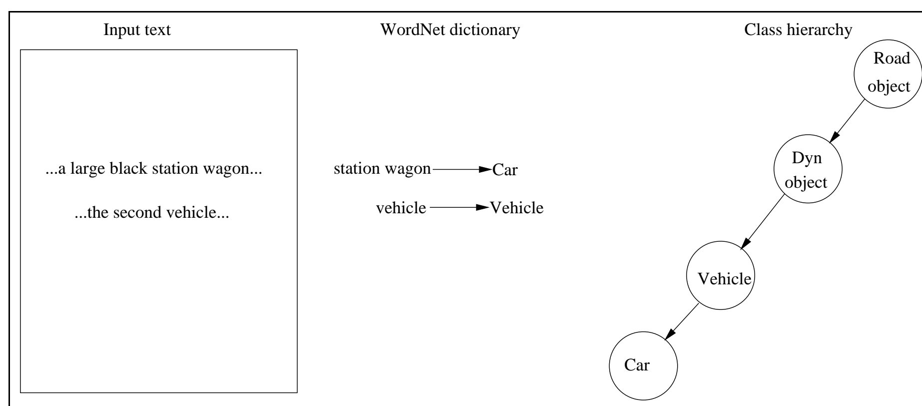

Figure 15: Integration of WordNet data with the class hierarchy.

The solution proposed here is to use a description file similar to the ontology description file, where pointers into the WordNet and literal words describe which data should be included. In Figure 16, we see the file which describes the Swedish dictionary.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE entities SYSTEM "wordnet_entities.dtd">
<entities>
  <entity name="RoadObject">
    <entity name="DynamicObject">
      <entity name="Vehicle">
        <include entry="fordon/1"/>
        <entity name="Car">
          <include entry="personbil/1"/>
        </entity>
        <entity name="Truck">
          <include entry="lastbil/1"/>
        </entity>
        <entity name="Bus">
          <include entry="buss/1"/>
        </entity>
        <entity name="Tractor">
          <include entry="traktor/1"/>
        </entity>
        <entity name="Bicycle">
          <include entry="cykel/1"/>
        </entity>
        <entity name="Motorcycle">
          <include entry="motorcykel/1"/>
        </entity>
      </entity>
    </entity>
  </entity>
</entities>
```

Figure 16: Description file for the Swedish dictionary

This file for example tells us that the word *personbil*, and all the synonyms and hyponyms of its first sense, should be mapped to the class name `Car` in the dictionary, and similarly should the synonyms and hyponyms of *fordon* be mapped to `Vehicle`. Now we arrive at a slight complication: conflicts about words. The hyponyms of the word *fordon* contain *personbil*, so which class name should appear in the dictionary? We of course still want `Car` to appear, since this was what we wrote in the file. But the word *limousin* is a hyponym of both *personbil* and *fordon*. To what should this word be mapped? We want this to be mapped to `Car`, so we introduce the following rule: subclasses have precedence over superclasses. If we encounter a conflict where none of the classes claiming the word has declared it explicitly, and where none is a subclass of the other, we have a nonresolvable conflict and signal an error and exit. To overcome this problem, we also provide a mechanism to exclude words. In Figure 17, we see the description file for the English dictionary, which is much more complicated than the Swedish one since the Swedish WordNet data are very fragmentary and does not generate any non-resolvable conflicts.

The dictionary generation is intended to be done statically, just like the compilation of source code. Its output is a plain dictionary file in XML form.

The focus of this subsection has been on the construction of a mapping from WordNet words to names of road objects (and this will be used in the next two subsections), but we might imagine a further step in this generalization: to handle event frames as well. To do this, we could proceed just as we have done with the road objects: set up a dictionary that would for example map the verb "crash" to the class name `Impact`. But then we are already approaching the domains of FrameNet (see subsection 2.2), which unlike WordNet includes information about how the associated semantic roles are realized in language, and would therefore probably be much more useful in the linguistic analysis.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE entities SYSTEM "wordnet_entities.dtd">
<entities>
  <entity name="RoadObject">

    <entity name="StaticObject">
      <entity name="Obstacle">
        <include entry="obstruction/1"/>
        <include entry="structure/1"/>
        <exclude entry="pit/6"/>
        <exclude entry="floor/2"/>
        <exclude entry="area/4"/>
        ....
      </entity>
    </entity>

    <entity name="DynamicObject">
      <entity name="Vehicle">
        <include entry="vehicle/1"/>
        <exclude entry="police van/1"/>
        <exclude entry="craft/2"/>
        <exclude entry="rocket/1"/>
        <exclude entry="carriage/2"/>
        <exclude entry="mobile home/1"/>
        <exclude entry="diner/2"/>
        <words>tram</words>

        <entity name="Car">
          <include entry="car/1"/>
          <words>brougham, roadster, cab, passenger car</words>
        </entity>

        <entity name="Truck">
          <include entry="truck/1"/>
          <words>van, camion, lorry</words>
          <exclude entry="minivan/1"/>
          <exclude entry="wagon/2"/>
        </entity>
      </entity>
      ....
    </entity>
  </entity>
</entities>
```

Figure 17: Description file for the English dictionary.

5.6 A General Approach to Road Object Detection and Anaphora Resolution

In the light of the development of a new ontology and its integration with the WordNets, the simple coreference resolution strategy described in subsection 5.3 can be generalized. The detected objects, which the old system did not take advantage of outside the coreference resolution, could also be stored into the template. In this subsection, a simple experiment on the NTSB corpus with a general approach to entity detection and coreference resolution is described.

If you reread the text about the coreference resolution algorithm, you might notice that the strategy taken is very specific to the old CarSim system. Three types of objects are detected: cars, buses and trucks, and they can also be referred via more general expressions such as “the vehicle” or “it”. The objects were found in the text with the help of hand-crafted regular expressions.

One assumption which is at the core of the algorithm is that the most specific reference to an object is the introduction of the object. This algorithm can now be generalized through the use of the road object inheritance hierarchy. For example, if a car has been introduced earlier in the text, it can be referenced as a car, a vehicle, a dynamic object or a road object. So, if the expression “the vehicle” pops up in the text, the class name `Vehicle` is obtained from the dictionary. Via the Java wrapper classes, the car object can be tested for membership of the `Vehicle` class by means of a class instance test.

One problem is *how to obtain* the text fragments to handle. The simple version of the algorithm used regular expressions which were hand-crafted for cars, buses and trucks, but they must now be abandoned in favour of something more general. This can be done in many ways, for example could parse tree constituents

be used. The strategy used in this experiment was to use a part-of-speech tagger to find *noun groups*. The noun groups used here consist of a determiner ("a", "the" or similar) followed by a series of modifying words (adjectives, nouns) and end with a noun, for example "my little green cactus". The tagger used was the Tree-Tagger [25], which is very easy to install and use but which unfortunately has a proprietary license. The noun groups were detected using a finite-state automaton. When a noun group was detected in the text, it was searched for in the dictionary. If no match was found in the dictionary, the first word was removed and a new search was made, and so on. For example, if the noun group "a large black station wagon" were detected in the text, first "a large" would be removed, then "black". The compound "station wagon" would be found in the dictionary, yielding a `Car` object. Since for example a station wagon is something else than just a wagon, the longest possible part of the noun group should be used.

In the previous coreference resolution algorithm, only dynamic objects were taken into account. In this experiment, all kinds of objects except for indication objects such as road signs were handled. This will of course be a challenge, since the number of objects which could be referents will increase.

Again, the NTSB texts were manually read and compared to the output of the experimental implementation. This time, the number of objects present in the texts were noted as well. (The objects in the texts had already been recorded during the knowledge representation evaluation.) In Figure 18, we see the performance figures.

Total number of objects in texts	about 86
Number of detected objects	74
Number of correctly detected objects	64
Precision	0.86
Recall	0.74
F-measure ($\beta = 1$)	0.80
Total number of back references	about 105
Number of detected back references	101
Number of correctly resolved references	77
Precision	0.76
Recall	0.73
F-measure ($\beta = 1$)	0.75

Figure 18: Statistics for the experiment on the NTSB corpus.

We see that for coreference resolution the precision is lower than in the earlier experiment, but that the recall is higher. The lower precision has a number of reasons, mainly that the dictionary was not perfectly crafted (there were several misclassifications due to confusion between "trailer vehicles" and "trucks"). The reason for the slightly higher recall might be that the method of detecting noun groups spots more references than the method using hand-crafted regular expressions. The most significant reason that the figures for object detection aren't higher is that there is no support for cardinal numbers (for example "three cars"), which means that several objects are just missed. The method of noun group detection, which is very primitive and does not take context into account, is also responsible for a couple of objects which should not have been detected, for example a building which was detected when it was stated in the text that the victims were taken to the hospital.

The data which have earlier been presented in Figures 12 and 13 were produced during this experiment for the following text:

> "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."

5.7 A Manual Experiment on the Swedish News Corpus

Since an interface to the Swedish WordNet data had been developed, it would be interesting to see if we could do a similar experiment on our Swedish news corpus. A road object dictionary similar to the English one was easily compiled with the help of the WordNet interface. There are however a couple of complicating factors about the nature of the Swedish noun morphology (internal word structure) which made it impossible to duplicate the previous experiment within the short time span that was availbable. For example, attributeless definite expressions like "the car" are not expressed via a separate article like the english "the", but by means of a suffix: *bilen*, where *-en* is a suffix denoting definiteness. This means that the suffix must be separated from the stem before the dictionary lookup can be done. Other complicating factors are the compounding of nouns, for example the compound *olycksbil* (accident car) which was found in one text in the news corpus, and the fact that pronouns and definite expressions must agree in gender with their referents. We have access to a Swedish part-of-speech tagger, which might be helpful in solving some of these problems in the future.

So instead of an experiment using automatic tools, I did a simple experiment where I assumed the role of a noun group detector with perfect knowledge of Swedish noun morphology. Since we only had obtained WordNet data for vehicles, only these were considered. The results can be seen in Figure 19.

Total number of vehicles in texts	about 60
Number of detected vehicles	64
Number of correctly detected vehicles	55
Precision	0.86
Recall	0.92
F-measure ($\beta = 1$)	0.89
Total number of back references	about 62
Number of detected back references	53
Number of correctly resolved references	33
Precision	0.62
Recall	0.53
F-measure ($\beta = 1$)	0.57

Figure 19: Statistics for the experiment on the Nyheter corpus.

Most vehicles described in the accidents could be detected using the WordNet information. Four words for vehicles which were present in the corpus were not found in the WordNet: *taxi*, *skoter*, *minibuss* and *långtradare*. I got the impression that the part of the Swedish WordNet which describes vehicles has a slight leaning towards technical language use: informal but very common words like *skoter* and *långtradare* are not included, unlike specific juridical terms like *lätt lastbil* which would hardly ever be present in a car accident report. Another problem of slight annoyance was that vehicles are sometimes referred to by their make. The cases of missing WordNet information and reference by make explained 8 cases of missed references and 5 of the vehicles which were not detected. Many of the erroneously resolved references can be explained by the simplistic algorithm used here, where for example anaphoric indefinite expressions lead to introductions of new vehicles. The nature of the texts, which unlike the NTSB texts contain lots of information not directly related to the sequence of events in the accident, is also somewhat troublesome.

The figures for vehicle detection are very high, which might seem strange. This is because I did not count vehicles which were not explicitly mentioned in the texts. Neither did I count objects which we did not have WordNet data for, which in this example corpus amounts to two pedestrians and one elk.

6 Conclusion

In this project, I have tried to build a foundation for a text-to-scene conversion system which can cope with increasing complexity. In order to do that, a more descriptive knowledge representation has been developed. The separation of the knowledge from other data has been emphasized. The knowledge representation has been manually tested on English and Swedish road accident reports, with good results. A development framework to make the knowledge representation easy to rebuild has been implemented.

When the new knowledge representation had been developed, I was able to integrate it with WordNet and to use this in a simple algorithm to detect road objects and to resolve anaphoric expressions. It performed well on the English corpus. Acceptable results were obtained in a similar (but manual) experiment on the Swedish corpus of news texts.

7 Acknowledgements

This work wouldn't have been possible without the grant from Vinnova, the Swedish Agency for Innovation Systems.

Several people have been involved during the course of this work. The redesign of the knowledge representation was in large part driven by the valuable suggestions by the reference group of traffic experts from the Department of Traffic and Road at Lunds Tekniska Högskola: Karin Brundell-Freij, Åse Svensson, and András Várhelyi.

The experiment on the Swedish news corpus was made possible by the very kind contribution of parts of the yet unpublished Swedish WordNet from the Department of Linguistics. We expect this contribution to save a lot of work for us in the future. The people we have been in touch with are Kerstin Lindmark, Åke Viberg, Ann Lindvall, and Johan Dahl.

Finally, I would like to thank Pierre Nugues and Per Andersson, who during several long discussions have contributed a lot to the knowledge representation which finally emerged. Pierre, my supervisor, provided several suggestions which finally helped me to get this report into a decent shape.

A Complete Template Description File

```
<?xml version="1.0" ?>
<!DOCTYPE package SYSTEM "../autogenerator/autogen.dtd">
<package name="se.lth.cs.carsimcore.template"
         dtd="file:template.dtd"
         root="Template"
         comment="This package holds the DOM tree wrapper classes.">

  <!-- Top class. -->

  <class name="Template" comment="This is the main wrapper class.">
    <attribute name="scene" type="Scene"
               comment="The scene information."/>
    <attribute name="object" type="RoadObject" collection="yes"
               comment="The involved objects."/>
    <attribute name="collision" type="Collision" collection="yes"
               comment="The collisions."/>
  </class>


  <!-- A very frequently needed enum. -->

  <enum name="Direction" values="left, right"
        comment="Direction, used as attribute for various objects."/>


  <!-- Scene information. -->

  <class name="Scene" comment="Scene information for one accident.">
    <attribute name="date" type="string" comment="Date."/>
    <attribute name="time" type="string" comment="Time."/>
    <attribute name="location" type="string" comment="Location."/>
    <attribute name="environment" type="Environment" default="rural"
               comment="Environment."/>
    <attribute name="weather" type="Weather" default="clear"
               comment="Weather."/>
    <attribute name="light" type="Light" default="light"
               comment="Light condition."/>
    <attribute name="roadCondition" type="RoadCondition" default="dry"
               comment="Road condition."/>
    <attribute name="roadConfiguration" type="RoadConfig"
               comment="Road configuration." default="StraightRoad"/>
  </class>

  <enum name="Environment" values="urban, forest, rural, mountainous"/>
  <enum name="Weather" values="clear, cloudy, rainy, snowy, foggy"/>
  <enum name="RoadCondition" values="dry, wet, icy, snowy, muddy"/>
  <enum name="Light" values="light, dark"/>


  <!-- Road configuration. -->

  <class name="RoadConfig" comment="Road configuration.">
    <class name="StraightRoad" comment="Straight road.">
      <attribute name="roadAttrs" type="RoadAttrs"/>
      <attribute name="leftAttrs" type="SideAttrs"/>
      <attribute name="rightAttrs" type="SideAttrs"/>
    </class>
    <class name="Bend" comment="Bending road.">
      <attribute name="roadAttrs" type="RoadAttrs"/>
      <attribute name="leftAttrs" type="SideAttrs"/>
      <attribute name="rightAttrs" type="SideAttrs"/>
      <attribute name="direction" type="Direction"/>
    </class>
    <class name="Crossroad" comment="Crossroad.">
      <attribute name="road" type="StraightRoad" collection="yes"/>
    </class>
    <class name="Roundabout" comment="Roundabout.">
      <attribute name="road" type="StraightRoad" collection="yes"/>
    </class>
    <class name="TCrossing" comment="T-crossing.">
      <attribute name="main" type="StraightRoad"/>
      <attribute name="incoming" type="StraightRoad"/>
    </class>
  </class>


  <!-- Road and side attributes. -->

  <class name="RoadAttrs" comment="Road attributes.">
    <attribute name="roadName" type="string"/>
    <attribute name="roadType" type="RoadType" default="highway"/>
    <attribute name="separation" type="Edge" default="dashedLine"/>
  </class>
```

```
<class name="SideAttrs" comment="Road side attributes.">
  <attribute name="nLanes" type="int" default="1"/>
  <attribute name="speedLimit" type="int"/>
  <attribute name="edge" type="Edge"/>
</class>

<enum name="RoadType" values="highway, street, gravel, underConstruction"/>
<enum name="Edge" values="dashedLine, straightLine, guardRail, embankment"/>


<!-- Road objects. -->

<class name="RoadObject">
  <attribute name="introducedAs" type="string"/>
  <attribute name="refName" type="string"/>
  <attribute name="position" type="Position"/>

  <!-- Static objects. -->

  <class name="StaticObject" comment="A non-moving object.">
    <class name="Obstacle" comment="An object which can be crashed into.">
      <class name="HorizontalObstacle">
        <attribute name="Direction" type="HorizontalDirection"/>

        <class name="FallenTree"/>
        <class name="Fence"/>
        <class name="Hedge"/>
        <class name="Wall"/>
        <class name="Embankment"/>
        <class name="Building"/>
      </class>
      <class name="VerticalObstacle">
        <class name="Pole"/>
        <class name="StreetLight"/>
        <class name="Tree"/>
        <class name="Rock"/>
      </class>
    </class>

    <class name="SurfaceObject" comment="Object on the road surface.">
      <class name="Water"/>
      <class name="Snow"/>
      <class name="Ice"/>
      <class name="Mud"/>
      <class name="Leaves"/>
    </class>

    <class name="HollowObject" comment="An object which one can fall into.">
      <class name="Pit"/>
      <class name="Ditch"/>
      <class name="Pool"/>
    </class>

    <class name="IndicationObject" comment="Traffic indication object.">
      <class name="PedestrianCrossing"/>
      <class name="Sign">
        <class name="StopSign"/>
      </class>
      <class name="TrafficLight">
        <attribute name="color" type="LightColor" default="inactive"/>
      </class>
    </class>
  </class>


  <!-- Dynamic objects. -->

  <class name="DynamicObject">
    <attribute name="event" type="Event" collection="yes"/>
    <attribute name="speed" type="int"/>
    <attribute name="initDirection" type="CompassDirection"/>
    <attribute name="origin" type="string"/>
    <attribute name="destination" type="string"/>

    <class name="Vehicle" comment="Road vehicle.">
      <attribute name="make" type="string"/>

      <class name="Car"/>
      <class name="Truck"/>
      <class name="Bus"/>
      <class name="Tractor"/>
      <class name="Motorcycle"/>
      <class name="Bicycle"/>

      <class name="TrailerVehicle" comment="A tractor with trailers.">
        <attribute name="trailer" type="Trailer" collection="yes"/>
```

```
      </class>
    </class>

    <class name="Trailer" comment="Trailer.">
      <class name="Caravan" comment="Caravan."/>
    </class>

    <class name="Train" comment="Train."/>

    <class name="Animate" comment="Human or animal.">
      <class name="Pedestrian"/>
      <class name="Animal">
        <class name="Deer"/>
        <class name="Elk"/>
      </class>
    </class>

    <class name="MovingObject" comment="Moving mechanical objects.">
      <class name="LevelCrossing"/>
      <class name="LiftableBridge"/>
    </class>

  </class>

</class>


<!-- Some classes needed by the road objects. -->

<class name="Position">
  <attribute name="x" type="int"/>
  <attribute name="y" type="int"/>
</class>

<enum name="HorizontalDirection" values="parallel, across"
      comment="Direction compared to the road."/>

<enum name="LightColor" values="red, amber, green, inactive"
      comment="The color of a traffic light."/>

<enum name="CompassDirection" values="north, south, east, west"
      comment="Compass direction."/>

<!-- Events for the dynamic objects. -->

<class name="Event">
  <attribute name="collision" type="Collision" reference="yes"/>
  <attribute name="position" type="Position"/>

  <class name="DrivingForward"/>
  <class name="Turn">
     <attribute name="direction" type="Direction"/>
  </class>
  <class name="UTurn"/>
  <class name="Stop"/>
  <class name="Overtake">
     <attribute name="vehicle" type="Vehicle" reference="yes"/>
  </class>
  <class name="ChangeLane">
     <attribute name="direction" type="Direction"/>
  </class>
  <class name="LeaveRoad">
     <attribute name="direction" type="Direction"/>
  </class>
  <class name="Overturn">
     <attribute name="direction" type="Direction"/>
  </class>
  <class name="Override">
     <attribute name="vehicle" type="Vehicle" reference="yes"/>
  </class>
  <class name="CatchFire"/>
  <class name="BackOff"/>
  <class name="Bounce"/>
  <class name="Slide"/>
  <class name="Separate">
     <attribute name="separatedFrom" type="DynamicObject" reference="yes"/>
  </class>
  <class name="Rotate"/>
  <class name="Accelerate">
     <attribute name="speed" type="int"/>
  </class>
  <class name="Decelerate">
     <attribute name="speed" type="int"/>
  </class>
</class>
```

```
  <!-- Collisions. -->

  <class name="Collision">
    <attribute name="actor" type="DynamicObject" reference="yes"/>
    <attribute name="actorSide" type="Side" default="unknown"/>
    <attribute name="victim" type="RoadObject" reference="yes"/>
    <attribute name="victimSide" type="Side" default="unknown"/>
  </class>

  <enum name="Side" values="front, rear, leftside, rightside, unknown"
        comment="The side of an object in a collision."/>

</package>
```

B Glossary

Corpus	A collection of real-world texts. A corpus (which should be as large as possible) is needed when any natural language processing system which aims to be usable in the real world is developed.
DTD	Document Type Definition – a document describing the structure of XML documents. Its semantics (but not its ugly syntax) roughly resembles the Backus-Naur Form often used to describe the grammars of programming languages. In each XML document there is usually a reference to a DTD, and before using the XML document one should *validate* the document against the DTD to see that it is structurally well-formed. The DTD language is a historical remnant of the SGML DTDs and considered clumsy and unexpressive by many. Superseded by *XML Schema*, which however is not available in many implementations yet.
First-order logic	Also called predicate logic. A mathematical language to represent facts about the objects in the world. It constists of *atoms*, which are names referring to objects, *predicates* describing the objects, *logical connectives* such as And (&), Or (v) and Not (∼), and *quantifiers* for-all (∀) and exists (∃). Example: "Everyone loves the ape, but the ape loves noone" can be expressed in first-order logic as $\forall x\ Loves(x, Ape)\ \&\ \sim \exists x\ Loves(Ape, x)$.
Parsing	The process of determining the syntactic structure of a text. In computer science this usually involves building a symbolic tree representation, called the *syntax tree*. For natural language, unlike for example programming languages, the mapping from sentence to syntax tree is often not unique due to the highly ambiguous nature of human language. Some parsers produce a *dependency structure*, which is a less rigid structure than a syntax tree.
Part-of-speech tagger	A computer tool which reads a text and assigns a part of speech (e.g. noun, verb and so on) to each word. It takes context into account. For example, in "to have a drink" the word *drink* should be regarded as a noun, but in "to drink" as a verb.
Regular expressions	A concise and powerful way to describe sets of strings. Formally equivalent to finite-state automata. Constructed by concatenating, alternating or repeating substrings. Example: `a(b|c)*d`, which denotes the string set $\{ad, abd, acd, abbd, abcd, acbd, accd, \ldots\}$.
Semantics	The processing of the *meaning* of texts, as opposed to the *structure* (syntax). A classical example by Chomsky is *"Colorless green ideas sleep furiously"*, which is a syntactically correct sentence which makes no sense semantically.
Syntax	The processing of the *structure* of texts, as opposed to the *meaning* (semantics).
XML	Extensible Markup Language – a language for describing structured data as a text document. Even though it is verbose and aesthetically unappealing, the availability of tools and libraries, and its gradual acceptance as *the* data format, makes this a primary candidate for text representation of data.
XSLT	XML Stylesheet Language, Transformations – an ugly XML-based programming language used for describing transformations of XML documents.

References

[1] Andersson, Per. *A Prototype to Extract and Visualize Information from Car Accident Reports in Swedish.* M. Sc. Project Report. Department of Computer Science, Lunds Tekniska Högskola. 2003.

[2] Appelt, Douglas E. and David Israel. *Introduction to Information Extraction Technology.* Tutorial Prepared for IJCAI-99. Artificial Intelligence Center, SRI International. 1999.

[3] Baker, Collin F., Charles J. Fillmore and John B. Lowe. *The Berkeley FrameNet Project.* In COLING-ACL '98: *Proceedings of the Conference, held at the University of Montréal*, pp 86 – 90. 1998.

[4] Bloksma, L., P. Díez-Orzas, P. Vossen. *User requirements and functional specification of the EuroWordNet project.* EuroWordNet (LE-4003) deliverable D001, University of Amsterdam. Downloadable from `http://www.illc.uva.nl/EuroWordNet/docs.html`. 1996.

[5] Brachman, Ronald J. *What IS-A is and isn't.* In IEEE Computer, special issue on knowledge representation, 1983, 16(10), pp. 30 – 36. 1983.

[6] Coyne, Bob and Richard Sproat. *WordsEye: An Automatic Text to Scene Conversion System.* In *Proceedings of SigGraph.* 2001.

[7] 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.

[8] Egges, Arjan. *Generating a 3D Simulation of a Car Accident from a Formal Description: the CarSim System.* M.Sc. Project Report, Université de la Twente, Netherlands. July 2000.

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

[10] Fillmore, Charles J. *The Case for Case.* In Bach, E. and R. T. Harms (ed.) *Universals in Linguistic Theory*, pp. 1 – 88. Holt, Rinehart and Winston, Inc. 1968.

[11] Fillmore, Charles J. and Collin F. Baker. *Frame Semantics for Text Understanding.* In *Proceedings of WordNet and Other Lexical Resources Workshop*, NAACL, Pittsburgh. 2001.

[12] Gildea, Daniel and Daniel Jurafsky. *Automatic Labeling of Semantic Roles.* In *Computational Linguistics* 28(3): pp 245 – 288. 2002.

[13] Hayes, Patrick J. *The Logic of Frames.* In Metzing, D. (ed.) *Frame Conceptions and Text Understanding*, pp. 46 – 61. de Gruyter. 1979.

[14] Hedin, Görel and Eva Magnusson. *JastAdd – a Java-based system for implementing frontends.* Draft. LUCAS, Department of Computer Science, Lunds Tekniska Högskola. 2000.

[15] 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.

[16] Kiparsky, Paul. *On the Architecture of Pāṇini's Grammar.* Three lectures delivered at the Hyderabad Conference on the Architecture of Grammar, Jan. 2002, and at UCLA, March 2002. Can be downloaded at `http://www.stanford.edu/~kiparsky`. 2002.

[17] Lenat, D. B. *Cyc: A Large-Scale Investment in Knowledge Infrastructure.* In *Communications of the ACM*, 38(11), pp. 33–48. 1995.

[18] McCarthy, John. *Programs with Common Sense.* In *Proceedings of the Symposium on Mechanisation of Thought Processes*, vol. 1, pp. 77 – 84. Her Majesty's Stationery Office. 1958.

[19] Mitkov, R. B., B. Boguraev and S. Lappin. *Introduction to the Special Issue on Computational Anaphora Resolution.* In *Computational Linguistics* 27(4), pp. 473 – 477. 2001.

[20] Mueller, Erik T. *Natural language processing with ThoughtTreasure.* New York: Signiform. Available online at `http:// www.signiform.com/tt/book/`. 1998.

[21] Nugues, Pierre. *Development of a Text-to-Scene Converter for Vehicle Accident Reports.* Project application to the Vinnova Program. 2002.

[22] Poibeau, Thierry. *Deriving a Multi-Domain Information Extraction System from a Rough Ontology.* In *Proceedings of the 17th International Conference on Artificial Intelligence*, Seattle, pp. 1264 – 1270. 2001.

[23] Russell, Stuart and Peter Norvig. *Artificial Intelligence – A Modern Approach.* Prentice Hall. 1995.

[24] Sarkar, Soumen. *Model-Driven Programming Using XSLT.* In *XML Journal*, 3(8). 2002.

[25] Scmid, Helmut. *Probabilistic Part-of-Speech Tagging Using Decision Trees.* In *International Conference on New Methods in Language Processing*, Manchester, England. pp. 44 – 49. 1994.

[26] Schulz, Bastian. *Development of an Interface and Visualization Components for a Text-to-scene Converter.* M.Sc. Project Report. Department of Computer Science, Lunds Tekniska Högskola. 2002.

[27] Sleator, Daniel and David Temperley. *Parsing English with a Link Grammar.* In *Proceedings of the Third International Workshop on Parsing Technologies.* 1993.

[28] Sproat, Richard. *Inferring the Environment in a Text-to-Scene Conversion System.* In *Proceedings of the K-CAP'01.* 2001.

[29] Svensson, Hans and Ola Åkerberg. *Development and Integration of Linguistic Components for an Automatic Text-to-Scene Conversion System.* M.Sc. Project Report. Department of Computer Science, Lunds Tekniska Högskola. 2002.

[30] Touretzky, David S. *The Mathematics of Inheritance Systems.* Pitman and Morgan Kaufmann. 1986.