Development of an Interface and Visualization Components for a Text-to-Scene Converter

Bastian Schulz

Report of a study project during an exchange term at the
Lunds Tekniska Högskolan (Sweden)
written at the Department for Computer Science under the supervision of
Dr. Pierre Nugues (LTH) and **Prof. Dr. Helmut Weberpals** (TUHH)
- August 2002 -

Table of Contents

1 Introduction

This report describes the results of my student research project. I did it under a period of three months during an exchange term at the Lunds Tekniska Högskola[1] in Lund, Sweden. My assignment was to improve and extend the visualization module of the CarSim system. This module was initially developed in 2000 by Arjan Egges (2000) during an internship at the ISMRA[2] Caen, France.

1.1 General Information about CarSim

CarSim is a text-to-scene converter (Dupuy et al. 2001). It takes real car accident reports written in French or English and generates 3D animations of them, which try to visualize the described scenes. Internally, the system uses a formalism to model the information that is extracted from the accident report in a standardized form. The 3D simulation is based on the data contained in this generated document. So, CarSim can start the simulation even if the underlying accident report is unknown or if the document is self-written and there has never been an underlying report at all.

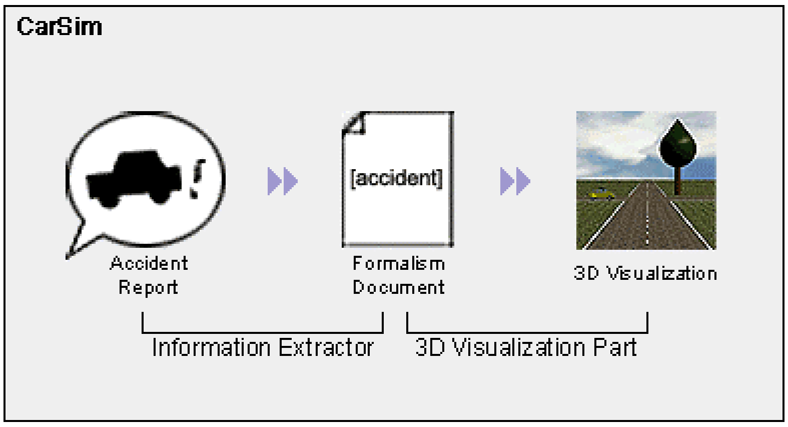


Figure 1:The architecture of CarSim

CarSim consists of two main parts (see Figure 1). The first part is the information extractor. It uses linguistic and semantic analyzers to retrieve the useful data out of the accident report and converts these data to the standardized format. In the first version of CarSim, the information extractor processes only texts in French. At the same time of my study project, two Swedish students implemented an English version of this system during their MSc project[3]. The current status of this part gives no reason for euphoria; the success rate of satisfactory results lies around ten percent.

The second part of CarSim is the 3D visualization module on which my project was focused. The program is completely written in Java. It parses the template and generates an internal representation of the data (an AccidentUniverse object). Then, it uses several planning algorithms, which infer all the missing data that are necessary to show an animation. The program uses classes from the Java3D library for the simulation. Further details of this part of CarSim are given in the chapters 2, 3 and 4.

[1] website: www.lth.se
[2] Institut des Sciences de la Matière et du Rayonment, website: www.ismra.fr
[3] see Svensson and Åkerberg (2002).

1.2 My Task

My study project consists of three parts. Because of the increasing importance of XML[4], my first task was to convert the CarSim proprietary formalism to a new one adopting this new standard. One advantage is that a `java.xml` package already exists. This package provides notably XML parsers so that the previous complicated `FormalismParser` class could be simplified a lot. Chapter 2 deals with this subject.

The graphical user interface (GUI) of the first version of CarSim covered only the visualization part. The information extraction module was command line driven and not integrated in this interface. My second task was to design a new user interface. It should cover both system parts. Further on, the handling of the program should be more user-friendly. See Chapter 3 for the considerations and results of this task.

The last task was the most complex one. I had to implement a new movement. That means on the one hand to extend the formalism by a new possible key word and on the other hand to extend the planning algorithms which compute the necessary data for the animation. I decided to take two movements: a vehicle should be able to leave the road and to overturn (see Chapter 4).

[4] eXtensible Markup Language

2 XML Parser

2.1 Introduction into XML

XML is the abbreviation for eXtensible Markup Language. Although it is a relatively recent language, the whole (IT) world is already talking about it. The World Wide Web Consortium (W3C) defined its specification in February 1998 as a proposal for an official standard for the storage of data. It is a subset of the Standard Generalized Markup Language (SGML), which also defines HTML (Hypertext Markup Language), the language of the World Wide Web. XML comes along with a large amount of associated technologies: DTD (Document Type Definition), SAX (Simple API for XML), DOM (Document Object Model), XSL (XML Stylesheet Language) and XSLT (XSL Transformations), Xpath (XML Path Language), Xlink (XML Linking Language), XML Pointer, XML Base, XML Schema, and others. This chapter gives a short introduction to write XML documents and DTDs and explains the differences between the two parsing methods SAX and DOM. For further information, see the following links:
http://www.xml.com/
http://www.w3schools.com/xml/
http://selfhtml.teamone.de/xml/ (in German)

2.1.1 XML documents

XML is designed for describing data. The kind of the data does not matter. A XML document can describe the content of a text document, an e-mail, measured values of a physical experiment, the accountancy of a department store, the data of a 3D accident visualization software or even a zoo (see the example below). XML uses tags similar to HTML to mark up the data. But in contrast to HTML, the author of a XML document is completely free to create his own tags. Example:

```
<?xml version="1.0"?>
<exampel>
   <my_own_tag>
      That's great!
   </my_own_tag>
   Here follows some simple text.
</exampel>
```

The excerpt above is a well-formed XML document. The first tag is a XML declaration, which indicates that this document is a XML document of the version 1.0[5]. It is not obligatory but is a good programming style. Thereafter, some arbitrary tags follow. XML does not care about the sense of the content or the right spelling of English words. But XML specifies some rules on how the tags have to be set. For example, the tags must not overlap: `<my_own_tag>` has first to be closed with its end tag `</my_own_tag>` before the surrounding `<exampel>` tag can be closed.

2.1.2 Document Type Definitions (DTDs)

Data that should be processed by a program has to fulfill structural requirements in general. A mechanism is needed that checks if a document fulfills these requirements and can be processed by the program or not. This is the purpose of Document Type Definitions (DTDs). They contain the constraints on the structure of a data document. A program that uses XML

[5] Version 1.0 is at the moment the only version. Version 1.1 is in progress.

documents should first parse it and check if its structure fits to the rules given by the associated DTD. Here is an example code for a DTD describing a zoo:

```
<!ELEMENT zoo ((cage)+,(visitor)*)>

<!ELEMENT cage (animals)*>
<!ELEMENT animals (description)?>
<!ATTLIST animals
   species     CDATA #REQUIRED
   number      CDATA #REQUIRED
>
<!ELEMENT description (#PCDATA)>

<!ELEMENT visitor EMPTY>
<!ATTLIST visitor
   kind (child|student|adult|pensioner) "adult"
>
```

The root element is `zoo`. The DTD's file name must then be "zoo.dtd". A zoo consists of one or more cages (indicated by the +) and an arbitrary number (0..n) of visitors (indicated by the *). Each cage contains 0..n `animals` elements. `animals` must have (`#REQUIRED`) the parameters `species` and `number` and can have the subelement `description` (`?` means "optional"). A visitor cannot contain any subelements (`EMPTY`). It has the attribute `kind` with the options "child", "student", "adult", and "pensioner". If this attribute is not assigned, the default value "adult" is assumed.

The following XML document conforms to this DTD.

```
<?xml version="1.0"?>
<!DOCTYPE zoo SYSTEM "zoo.dtd">
<zoo>
   <cage>
      <animals species="elephant" number="3">
         <description>
            Indian elephants. They have smaller ears than the Africans.
         </description>
      </animals>
   </cage>
   <cage>
      <animals species="rabbit" number="20"/>
      <animals species="python" number="1"/>
   </cage>
   <visitor kind="child"/>
   <visitor kind="adult"/>
   <visitor/>
</zoo>
```

This document describes a zoo with two cages and three visitors. In the first cage there are three elephants; in the other one are 20 rabbits and one python. The visitors are one child and two adults. The `DOCTYPE` in the second row declares which kind of document it is (`zoo`) and which DTD the parser should use to validate it (`zoo.dtd`). The `visitor` tags have no start and end tags but are stand-alone tags because they do not contain subelements or character data inside.

The following XML document does not conform to the definition:

```
<?xml version="1.0"?>
<!DOCTYPE zoo SYSTEM "zoo.dtd">
<zoo>
   <cage/>
   <animals species="lion" number="1"/>
   <visitor kind="pensioner"/>
</zoo>
```

The element `zoo` cannot have `animals` as a direct subelement. The lion has to be in a cage.

2.1.3 DOM and SAX Parser

To retrieve the data of a XML document, the program has to parse it. Fortunately, Java already provides XML parser classes, which are described later on. Generally, there are two main technologies: DOM and SAX.

The Document Object Model (DOM) was developed by the W3C. Parsers using this technology build up internally a document tree and create nodes for all the elements of a document. The DOM interface provides methods to navigate in this tree. So the program can read the structure and values of the document, and easily add new entries, edit or remove them as well. The disadvantage of this approach is that the whole tree has to be stored in the main memory of the computer. This could lead to problems with very large documents on limited systems.

The other approach was developed by members of the XML-Dev mailing list to solve this problem. A SAX (Simple Application Programming Interface for XML) parser is event-based. It considers the document as a sequence of events. Each time a new element, character string, comment or processing instruction occurs, the parser calls an associated method. These methods can initiate further actions that can depend on the current element. The model is optimized to read documents and process them, not to edit their content. Because of this voluntary limitation, processing XML documents with SAX parsers needs much less memory resources than DOM parsers.

CarSim reads the XML documents, which contain the accident data, and converts them to an internally data representation to start the simulation. Therefore, the program uses a SAX parser (see Chapter 0).

2.2 Formalism

The old formalism of the first version of CarSim was written in an proprietary, not standardized format. The belonging parser class `FormalismParser` subdivided the incoming String into tokens. If a token or character did not fit to the structure of the formalism, the parser aborts and shows an error message. So far, the parser worked well and the formalism was easy to understand. To convert the formalism (and to adapt the parser according to this) just to follow the current XML hype would have been waste of time. So, why was this conversion necessary?

The answer is extendability. In case that the system has to be extended with e.g. a new movement, the first formalism is poorly flexible because it is hard-coded into the parser. Adaptations can only be made by someone who completely understands the parser (about 600 lines of code) and is able to program in Java[6].

[6] Of course, if the simulation of CarSim should be extended with a new movement this person has to be able to program in Java anyway.

The new approach that uses XML solves this problem a lot easier: simply adjust the DTD (60 LOC). See Appendix A for an example of an accident description in the old and in the new format.

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

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


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

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

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

Figure 2: accident.dtd

The exact formalism is defined by the accident.dtd (see Figure 2). The XML formalism is not completely new. It is an conversion of the old one. For example, it is obvious that the main structure of a accident document is still the same. The three subelements of the element `accident` in the first row denote that the optional elements `staticObjects`, dynamicObjects, and collisions have to appear in that order. These elements are defined below as well. The rules to define a DTD are explained above on the zoo.dtd example and need no further description. For more detailed information, visit the XML web resources, which are given above. The design concepts of the old accident formalism are explained more in detail in Egges (2000, Chapter 3).

2.3 SAX Parser in Java

The package `javax.xml.parsers` contains the class `SAXParser`. The constructor of this class is *protected*[7]. An instance can only be obtained from a `SAXParserFactory` object. Before the factory creates the `SAXParser` object, some settings can be made. The code in the `FormalismParser` class of CarSim correspond to this:

```
SAXParserFactory saxParserFactory = SAXParserFactory.newInstance();

saxParserFactory.setValidating(true);
saxParserFactory.setNamespaceAware(false);

SAXParser parser = saxParserFactory.newSAXParser();
```

The first line creates a `SAXParserFactory`. The next two lines set properties for the parser, which will be created in the last line. This parser is validating, which means that it checks if every document corresponds to the structural constraints given in the DTD. Furthermore, the parser does not care about namespaces[8].

To parse a file, the program calls now the `parse()` method of the `SAXParser` object. This method needs two input parameters: the file (`java.io.File`) containing the XML document and an object of the `DefaultHandler` class from the package `org.xml.sax.helpers`. The `DefaultHandler` contains the methods, which the parser calls if a new element, character string, comment, or an error occurs in the input file.

```
parser.parse(file, this);
```

The parse method gets the instance of the `FormalismParser` as `DefaultHandler` (addressed by `this` because it is called from inside itself). The `FormalismParser` extends the `DefaultHandler` and overrides the empty methods with CarSim specific instructions. These methods are among others:

```
public void startElement(String uri, String localName, String qName,
                         Attributes attributes) throws SAXException { }

public void endElement  (String uri, String localName, String qName)
                                             throws SAXException { }

public void characters  (char[] ch, int start, int length)
                                             throws SAXException { }
```

The methods `startElement(...)` and `endElement(...)` contain several if-statements which decide what to do if a certain element occurs in the XML document. If the parser reads *<road kind="crossroad">* from the document for instance, it calls the method

[7] A constructor method creates an object instance of a class in Java. If this class is *protected* instead of *public* it is not possible to create an object of this class from outside the class.

[8] Namespaces are another topic within XML, which are not essential to know for this essay. We follow the parser and we set them aside.

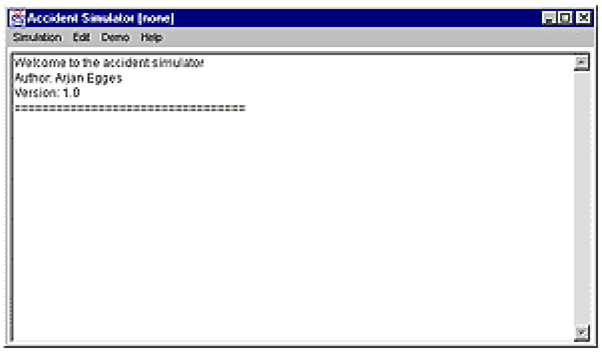
`startElement(...)`. The parameter `qName` contains the value “road“ and `attributes.getValue("kind")` outputs “crossroad“. In this way, the respective objects can be added to the `AccidentInfo` object, which is the internal representation of the accident information.

There is one disadvantage of choosing SAX instead of DOM. The *coords* element appears in several elements (*tree*, *sign*, *trafficLight*, *levelCrossing*, *collision*) as a subelement. If the `startElement(...)` method is invoked and the parameter `qName` is assigned with “coords“, the system does not know where the coordinates belongs to because SAX is memoryless. Therefore, the program stores a `Tree`, `StopSign`, `TrafficLight`, `LevelCrossing`, or `CollisionInfo` object in the variable `tempObject`. It assigns the coordinates to this object and adds `tempObject` finally to the `AccidentInfo` object.

The way the objects are added to the `AccidentInfo` object and represented internally was already given by the first version of CarSim. The new parser adopted the instructions from the old parser. It is described in detail in Egges (2000 Chapter 10).

3 Graphical User Interface (GUI)

The first version of CarSim consisted of two programs that were not integrated under the same interface. The information extraction module was command line driven and the visualization part had a WIMP[9] user interface (see Figure 3). Although this interface already provided the main functions, the clicks, which were needed to get the desired result, were not very intuitive[10]. Finally the program still used the deprecated java.awt classes.



```
Welcome to the accident simulator
Author: Arjan Egges
Version: 1.0
================================
```

Figure 3: Screenshot of the previous GUI

In the next section, I describe the new user interface for CarSim. I tried to create a more practical and more user-friendly interface. I examined several possible sketches and layouts trying to combine usability with an interface structure, which makes the process, flow easier to understand.

3.1 Functionalities of the New GUI

When the program starts first a small window appears. It displays the CarSim logo and at its bottom a progress bar, which shows the actual loading status. This window disappears when the program finished loading and the main window is displayed.

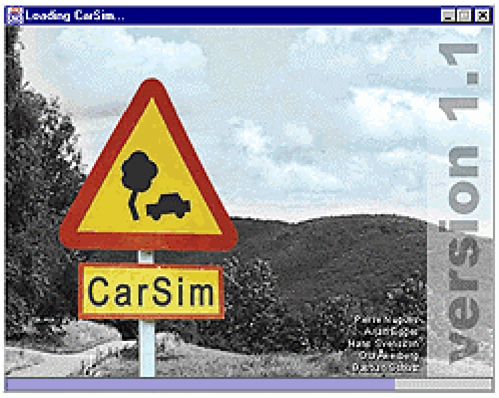
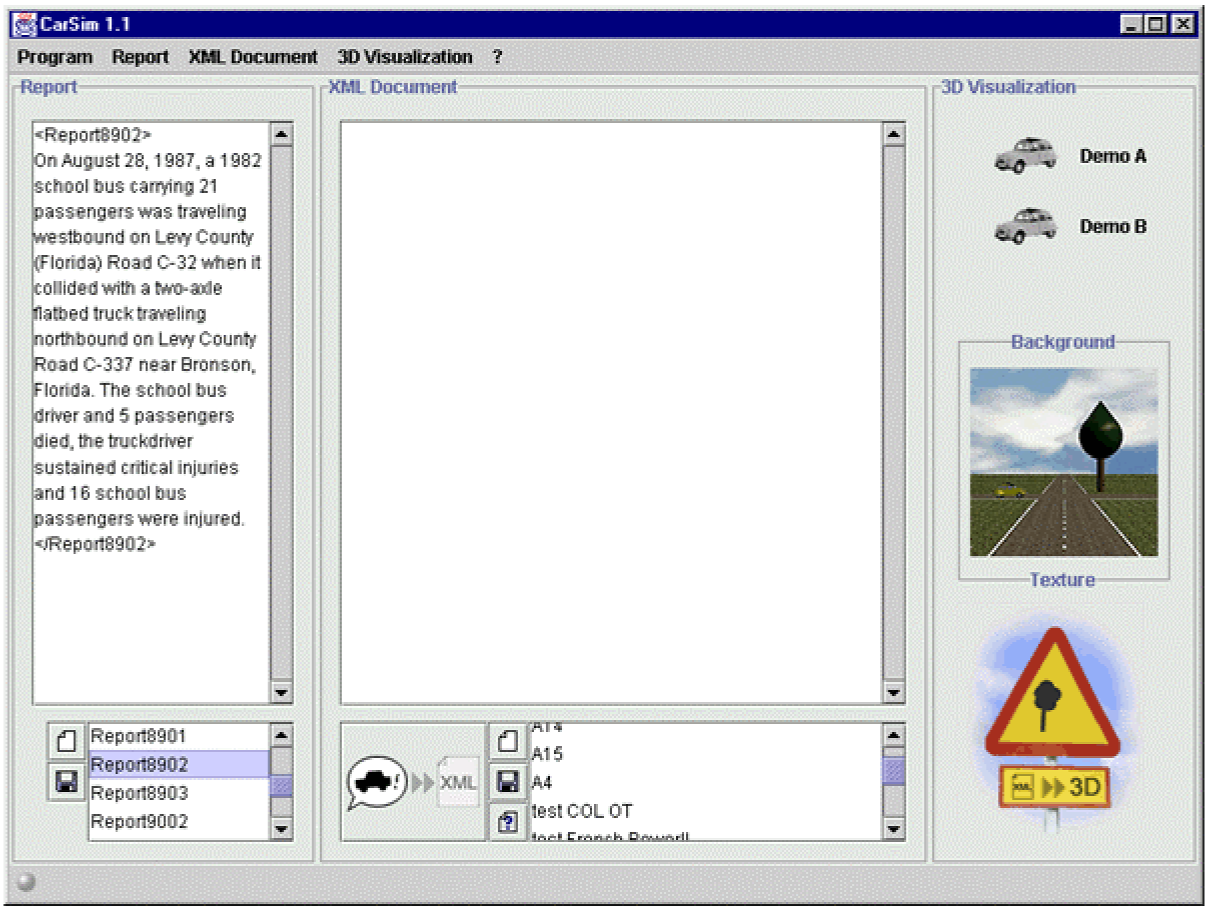


Figure 4: The loading window

The main window is divided into three parts. The left column contains all functions that belong to the accident reports. The middle part is almost identical to the left one. All elements

[9] Windows, Icons, Menus, and Pointers
[10] Of course, this was not the main task when the CarSim system was created.

that belong to the XML documents and their processing can be found in this part. The right side finally contains some of the functions concerning the 3D visualization.



<Report8902>
On August 28, 1987, a 1982 school bus carrying 21 passengers was traveling westbound on Levy County (Florida) Road C-32 when it collided with a two-axle flatbed truck traveling northbound on Levy County Road C-337 near Bronson, Florida. The school bus driver and 5 passengers died, the truckdriver sustained critical injuries and 16 school bus passengers were injured.
</Report8902>

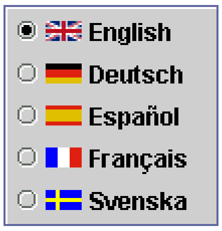
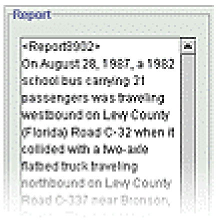
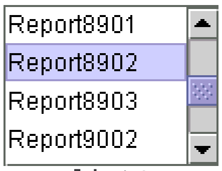
Figure 5: The main window

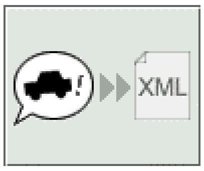
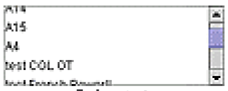
The left column mainly consists of a text area (`textArea1`) and a selection box (`list1`). It is possible to load existing reports into `textArea1` by selecting them in `list1`, to edit them, to save or delete them or to create completely new reports. By clicking on the Generate XML button the program starts the information extraction module with the selected reports as input parameters. It writes the result into `textArea2`, the large text field in the middle part of the interface.

The XML part provides almost the same functionality for the XML documents as described above for the reports. It contains a text area (`textArea2`) and a selection box (`list2`), too.

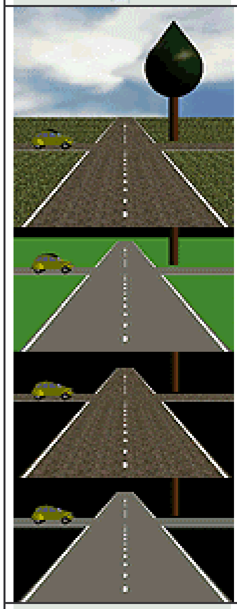
To visualize the accident which is described in the XML document shown in `textArea2` the user has to click on the CarSim sign at the down right corner of the window. A new window appears containing the animation, a start and a stop button.

3.2 Detailed Description of all User Inputs

Screenshot	Menu Entry	Description
	Program > Language	As a little extra feature I built in the option to switch the menu language between English, German, Spanish, French and Swedish. All the menu entries, alternative texts above buttons as well as information and error messages appear in the chosen language. This option was neither an instruction nor really needed, but it was just fun for me to add it to the software. :)
	Program > Exit	The program saves all the session specific data (menu language, 3D scene background, duration of the animation, ...) to a file and exits then the program. It has the same effect like clicking on the X in the upper right corner of the window.
 <Report8902> On August 28, 1987, a 1982 school bus carrying 21 passengers was traveling westbound on Levy County (Florida) Road C-32 when it collided with a two-axle flatbed truck traveling northbound on Levy County Road C-337 near Bronson, textArea1		`textArea1` displays the selected accident reports. Every particular report is bounded by a start and an end tag looking like <reportname> … </reportname> These reports can be edited within this text area.
 list1		`list1` displays the names of all files in the folder “carsimdata” with the file extension “.ana” (accident narration). Each file contains one accident report. The start and end tags visible in the text area are not part of the file. They are added automatically while writing the report to `textArea1`. If the user selects a report, its content will be displayed in the text area together with the other reports. If the user deselects a file name, it will be checked if the file has been modified. In that case, a dialog box will appear asking if the modified text should be saved or not. The deselected report will disappear from `textArea1`.
	Report > New	All the selected reports will be deselected. If some of them have been modified, a dialog box will ask the user if the modifications should be saved or dropped. Thereafter the textarea will be empty.
	Report > Save	The program checks if the selected reports have been modified. In this case all

Screenshot	Menu Entry	Description
		the selected files will be written to the disk. If there was no file selected but a new written text in the textarea, a dialog box with an input field appears and asks the user for typing in the file name for the new report.
	Report > Delete	If there is at least one selected, report a dialog box asks the user if all selected reports really should be deleted. If the user confirms, the textarea will be cleared, the files will be erased and `list1` will be updated.
	Report > Generate XML	Because of that `textArea2` (which contains the XML documents) will be needed for this operation, the program checks first if the `textArea2` contains modified XML documents. In this case the user will be asked if the modified XML documents should be saved or not. The visible reports in the `textArea1` will be converted to corresponding XML documents. The user must be aware that if the texts are modified, the modified versions are taken, not the texts contained in the report files on the disk. The new generated XML documents will use the same name of the original report with the ".ade" extension. If the files already exist, their content will be overwritten.
`textArea2`		`textArea2` displays the selected XML documents. Every particular document is bounded by a start and an end tag looking like <START documentname> … <END documentname> These XML documents can be edited within this text area.
A14 A15 A4 test COL OT `list2`		`list2` displays the names of all files in the folder "carsimdata" with the file extension ".ade" (accident description). Each file contains one XML document. The start and end tags visible in the text area are not part of the file. They are added automatically while writing the report to `textArea2`. If the user selects a XML document, its content will be displayed in the text area together with the other documents. If the user deselects a file name, it will be checked if the file has been modified. In that case a dialog box will appear asking if the

Screenshot	Menu Entry	Description
		modified document should be saved or not. The deselected document will disappear from `textArea2`.
🗋	XML Document > New	If there is at least one selected XML document, all these documents will be deselected. If some of them have been modified, a dialog box will ask the user if the modifications should be saved or discarded. Thereafter the textarea is empty. If no XML document has been selected in the beginning and the `textArea2` has been empty, a XML template will be written into the area. This template enables the user just by removing unnecessary parts to create a new accident description in an easier and faster way than writing the XML tags step by step. If the user clicks now on the New XML Document button without having modified the XML template, the content of `textArea2` will be cleared again.
💾	XML Document > Save	Checks if the selected XML documents have been modified. In this case all the selected files will be written to disk. If there was no file selected but a new written document in the textarea, a dialog box with an input field appears. It asks the user for typing in the name for the new XML document file.
	XML Document > Delete	If there is at least one selected XML document, a dialog box asks the user if all the selected XML documents should really be deleted. If the user confirms, the textarea will be cleared, the files will be erased and `list2` will be updated.
❓	XML Document > Validate Selected Documents	Checks if the XML documents in the `textArea2` correspond to the DTD (Document Type Definition). The status bar shows the message “The XML document is valid” resp. “The XML document is not valid” afterwards.

Screenshot	Menu Entry	Description
	3D Visualization > Start Visualization	Exactly one XML document must be selected, otherwise an error message occurs. The program checks the syntax of the XML document and if no errors occur, a window opens and shows the 3D simulation[11]. The user has only to press on a play button to start the animation. (see Figure 6 (a))
		By clicking on this image the user can switch between four different scene environments. The first one just paints grey roads (from bottom-up), the second one adds an asphalt texture to the road. The third one is without the asphalt, but embeds the roads in a green area. All of them have no background (black colour). In the fourth mode the roads have an asphalt texture, the surrounding area looks like real lawn and the background is a blue sky with some clouds.
	3D Visualization > Demo A	A 3D simulation of an overtaking car is shown in the simulation window.
	3D Visualization > Demo B	A complex traffic situation with priority at a crossing is shown in the simulation window.
	3D Visualization > Configuration	The configuration window appears (see Figure 6 (b)). The user can adjust the following values: - Duration of the animation - Start radius of the circle intersection algorithm (`MultipleAccidentPlanner`) - Precision of the trajectory - Percentage of the time the car stands during a STOP event referring to whole time
	? > Help	A help frame appears.
	? > About	A information box with the software logo appears.

[11] The simulation window was already implemented. I just adopted it from the first version of CarSim.

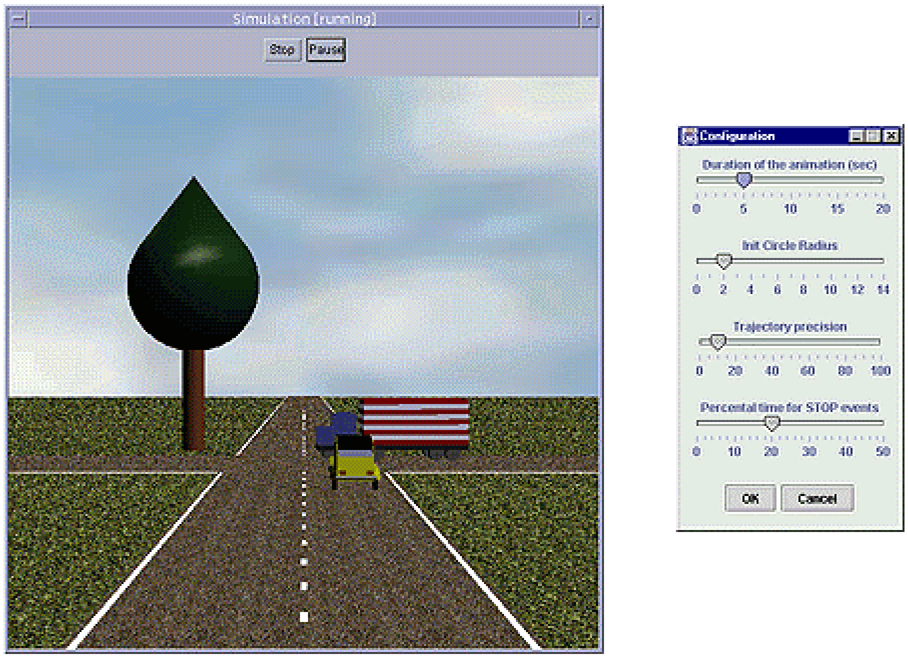


Figure 6: (a) Simulation window, (b) Configuration window

3.3 Implementation

I wrote all the new interface classes and I integrated the CarSim simulation programs. I had to adapt some existing classes so that they fit in the interface. I list and describe all the classes, which are important to know to understand the user interface.

CarSim.java

This class instantiates first the `MainWindow_Loading` window and hides it when the rest of the initialization is completed.

It calls the function `loadProperties()` of the `Messenger` class which loads some values (language, background/texture, duration of the simulation, …) that have been saved in a file after the last session. If this file does not exist, default values are taken.

It initializes the `MainWindow`, fills its list boxes with help of `FileFilter` with the matching file names, initializes `MainWindowManager`, `MainWindowEventListener`, and `MainWindowStatusManager` and sets all menu texts in the chosen language.

MainWindow.java

This class generates the main window and all its buttons, text fields, list boxes, the menu and the status bar.

MainWindowManager.java

It handles all incoming method calls from `MainWindowEventListener`.

Important variables are:

public String[] fileReports

public String[] fileXMLdocs

All the accident reports resp. XML documents are held in these arrays as String objects. They contain always exact the same content as the respective files.

public String[] textAreaReports

public String[] textAreaXMLdocs

All the accident reports resp. XML documents are written temporarily from the textareas into these arrays. The temporary texts have to be compared with the saved ones if they have been modified.

public String[] wasReportSelected

public String[] wasXMLdocSelected

Saves the selection status of each item of the list boxes. If the selection changes, the new status can be compared with the old one.

MainWindowEventListener.java

It adds event listeners to all the elements of the main window. If an event occurs, this class calls the appropriate methods of `MainWindowManager`.

mouseClicked event on `list1`

`list1` contains the file names of the accident reports. If the user clicks on an item of the list box, this event is generated. The general outline of the algorithm is:

If there was at least one list entry selected before:

- Write the reports of the earlier selected list items (saved in `wasReportSelected[]`) to `textAreaReports[]`
- Compare these strings with the corresponding strings of `fileReports[]`
- If some of them have been modified, ask the user if the changes should be saved or not. If YES, write the Strings from `textAreaReports[]` to `fileReports[]` and save them in the files. If NO, overwrite the modified strings from `textAreaReports[]` with the originals from `fileReports[]`
- Set the text of `textArea1` referring to the new selection of `list1`
- Save the current selection of `list1` in `wasReportSelected[]`

If no list entry was selected, it is a little bit different.

- Check if `textArea1` contains characters. In that case, request the user for a name for the new report file. The user can choose between save the report to disk under any wanted name or drop the new report.
- Set the text of `textArea1` referring to the new selection of `list1`
- Save the current selection of `list1` in `wasReportSelected[]`

New Report

- Set all entries of `list1` to unselected
- Write the reports of the selected list items to `textAreaReports[]`
- Compare these strings with the corresponding strings of `fileReports[]`
- If some of them have been modified, open an instance of `JOptionPane` which asks the user if the changes should be saved or not. If YES, write the strings from `textAreaReports[]` to `fileReports[]` and save them in the files. If NO, overwrite the modified strings from `textAreaReports[]` with the originals from `fileReports[]`
- Clear `textArea1`
- Save the actual selection of `list1` in `wasReportSelected[]`

Save Report

If there was at least one list entry selected before:

- Write the reports of the selected list items to `textAreaReports[]`
- Write the strings from `textAreaReports[]` to `fileReports[]` and save them in the files.

If no list entry was selected:

- Open an instance of `JOptionPane` which requests the user to type in a name for the new report.
- Save the file under this name
- Update `list1`, `fileReports[]` and `textAreaReports[]` by loading anew all the report file names and contents
- Set the `list1` as well as the `wasReportSelected[]` entry of the new file name to selected
- Update `textArea1`. The report is still the same thereafter, but it is surrounded by its start and end tag <reportname> … </reportname>

Delete Report

If no report is selected, the status bar displays an error message. Otherwise:

- Open an instance of `JOptionPane` which asks the user if the selected reports really should be deleted. If the user confirms delete the files, clear the content of `textArea1`, and update `list1`, `fileReports[]`, and `textAreaReports[]` by loading anew all report file names and contents

mouseClicked event on `list2`, New / Save / Delete XML Document

They have the same functionality like the versions for the accident reports, just with `list2` instead of `list1`, `textArea2` instead of `textArea1` and XMLdoc instead of every Report.

Other events

I decided not to explain every single event in detail, because the functionality of the particular events often differs just in a few points or in the order of the method calls. The interaction of the important variables in `MainWindowManager` should be well-explained enough at this point. The order of the method calls of the other events can be constructed easily from the description in section 4.2.

4 Extension of the 3D Visualization Module

In the last part of my project, I extended the animation with a new movement. The first version of CarSim took the following movements into consideration: *driving_forward*, *stop*, *turn_left*, *turn_right*, *change_lane_left*, *change_lane_right*. In many of the real accident reports from the corpus of the National Transportation Safety Board (www.ntsb.gov) the vehicles overturn. This movement was not considered in CarSim until now. The problem was that this movement is caused sometimes after a collision, but often as well after leaving the road without being involved in any collision with another vehicle. So I decided to cover both cases and to implement the motions *leave_road_left* resp. *leave_road_right*, too.

Before I start to explain the work I did, it is necessary to give a short overview about the planners of the CarSim system.

4.1 Detailed Overview about CarSim

Chapter 3 gave an overview about the graphical user interface and the methods called by the button events. This section gives a detailed description of what happens if the "Start Visualization" button is pressed.

If the XML document in `textArea2` is valid, the String is passed to the `FormalismParser`. This class generates an AccidentInfo object, which is the intern representation of the information in the XML document. This object is given to the `AccidentPlanner` which computes and completes missing values for the animation using the Java 3D library.

The `AccidentPlanner` consists of five planners: `PrePlanner`, `PositionPlanner`, `GeneralAccidentRoutePlanner`, `MultipleAccidentPlanner`, and `TemporalPlanner` (in that order). They are now explained more in detail. Note: All the classes described in this section were already implemented when I started to work. I just give a description of the status of CarSim that I found.

4.1.1 PrePlanner.java

The first planner fills in unknown values for the initial direction (north, south, east, west) or the part of a vehicle participating in a collision (front, rear, leftside, rightside). The planner is rule-based. Example: Car1 starts to drive northbound and the initial direction of Car2 is unknown. They collide. Car1 hits Car2 in the left side and the involved part of Car1 is unknown. The `PrePlanner` assigns the initial direction "west" to Car2 and "front" as actorPart to the collision (see Figure 7).

4.1.2 PositionPlanner.java

This planner takes the initial directions and computes a "1-dimensional" start and end position of every vehicle. It means that every vehicle gets a float value for the distance of its start position to the middle of the crossroads (zero point of the coordinate system) and a value for the distance between zero point and end position. Example: Car1 starts northbound, makes a turn to the right and ends eastbound. Its start position value is 20.0 (default value in CarSim), its end position value is 20.0, too. Car2 starts westbound and ends westbound. It gets the same values as Car1. Now it becomes a bit more interesting: Car3 starts like Car1 northbound. Because of Car1 already starts 20.0 metres (length units) distant from the crossroads, Car3 cannot get the same value. Car3 has to be placed behind Car1 and starts 6 metres (default in CarSim) behind the front car. Its start position becomes 26.0. Actually, the end position could be 20.0, because no other car drives to the north road. But to keep the same speed for all the vehicles, which depends on the distance to be covered during the animation, Car3 gets the end position value 14.0 (see Figure 8).

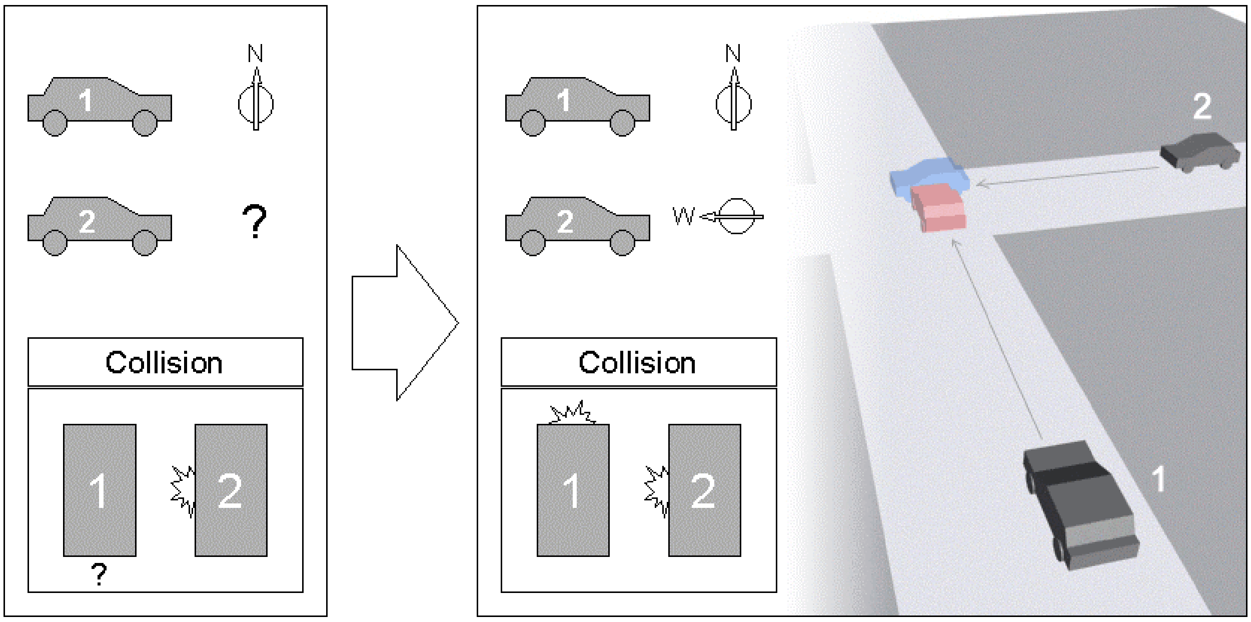


Figure 7: Task of the PrePlanner

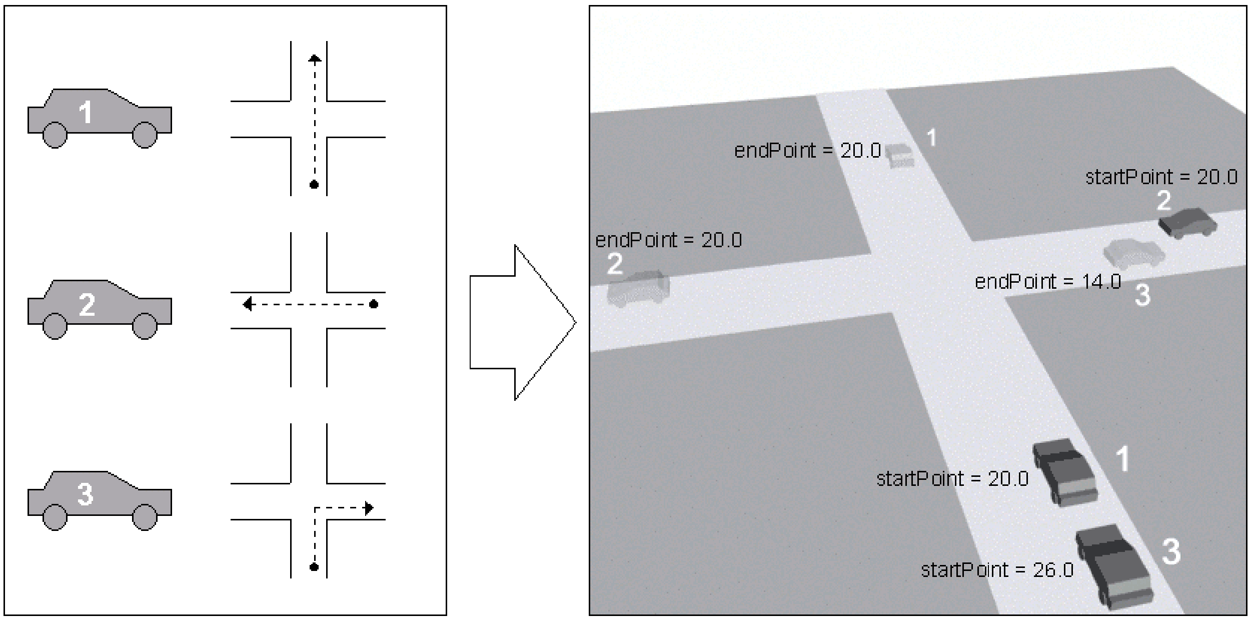


Figure 8: Task of the PositionPlanner

4.1.3 GeneralAccidentRoutePlanner.java

This planner takes the event chain of every particular vehicle and constructs a provisional trajectory for each, which does not care about any collisions. First it tries to detect the used scenario. There are six scenarios known in CarSim:

- Scenario I: (DF | STOP)* TL (DF | STOP)*
- Scenario II: (DF | STOP)* TR (DF | STOP)*
- Scenario III: (DF | STOP)* DP (DF | STOP)*
- Scenario IV: (DF | STOP)* CLL (DF | STOP)*
- Scenario V: (DF | STOP)* CLR (DF | STOP)*
- Scenario VI: (DF | STOP)+

Abbreviations:
DF: driving_forward, STOP: stop, TL: turn_left, TR: turn_right, DP: depass[12]*, CLL: change_lane_left, CLR: change_lane_right*

Examples:
The event chain
driving_forward, stop, driving_forward, driving_forward, change_lane_left, driving_forward
is classified to scenario IV.
The event chain
stop, turn_right, driving_forward, turn_right, stop, driving_forward
does not fit to any scenario. The planner aborts and produces an error message.

If a scenario has been found, the `GeneralAccidentRoutePlanner` determines some scenario typical coordinates. Example: Consider a vehicle with the event chain *driving_forward, stop, driving_forward, turn_left, driving_forward*. Scenario I was detected. The planner sets the start coordinates of the first event, the start and end coordinates of the main part, the left turn, and, finally, the end coordinates of the last events. The end coordinates of the event before the left turn as well as the start coordinates of the event after it are given by the coordinates of the turn itself. Figure 9 shows this state.

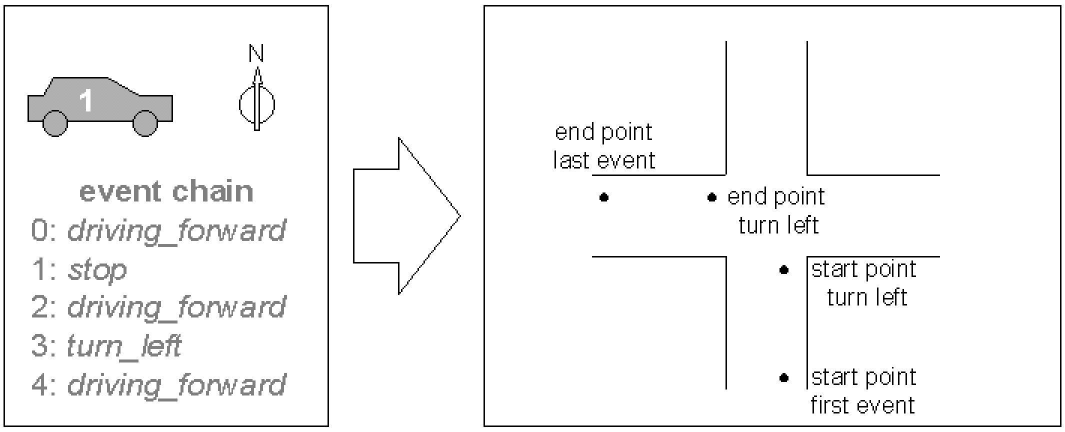


Figure 9: Task of the GeneralAccidentRoutePlanner

The `GeneralAccidentRoutePlanner` calls now the `PathPlanner` to fill in the missing start and end coordinates of some events. In our example, the first *driving_forward* has start coordinates but no end coordinates. The stop event thereafter has neither the one nor the other and so on. The `PathPlanner` completes the route from the start point to the beginning of the left turn and the route from the end of the left turn to the end point of the last event (not necessary in this example). Figure 10 shows the added points.

[12] The movement *depass* (=overtake) is not implemented

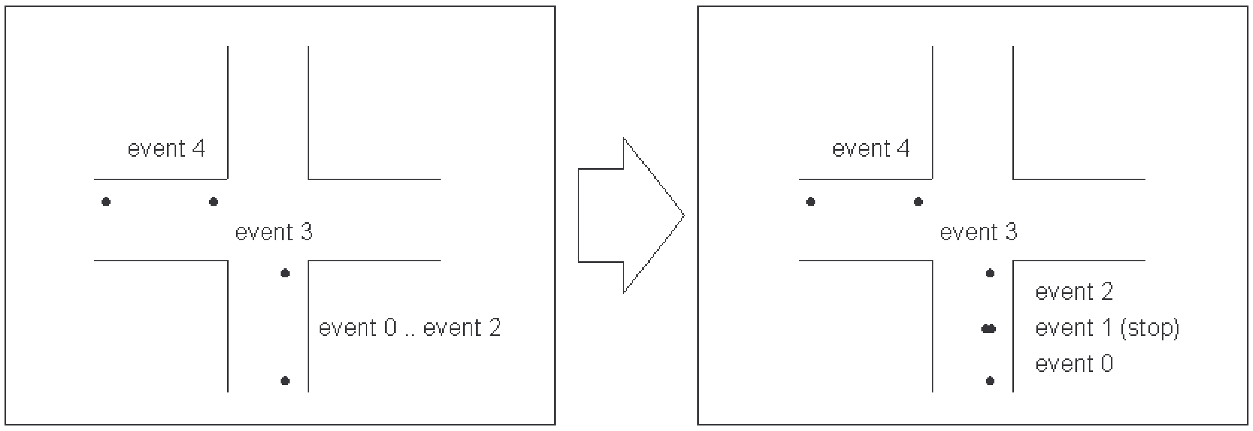


Figure 10: Task of the PathPlanner

If only these calculated coordinates were taken for constructing the trajectory, the straight lines would be modelled correctly, but the turn would not be very realistic. So the next planner is called, the `TrajectorPlanner`. It separates the whole route into many small distances. So the turn is approximated by a couple of small straight pieces. If the precision is high enough, the viewer cannot see the inaccuracy (see Figure 11).

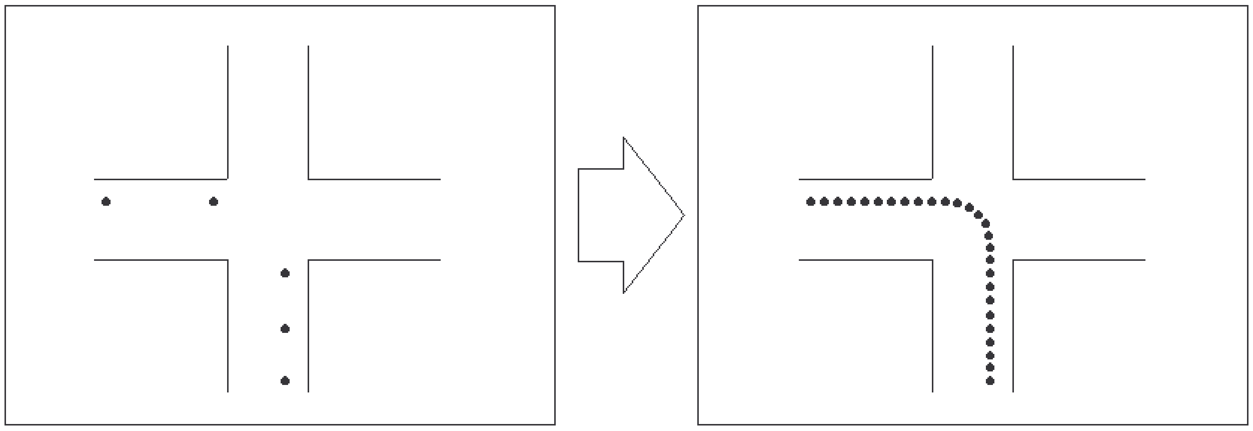


Figure 11: Task of the TrajectoryPlanner

4.1.4 MultipleAccidentPlanner.java

This planner changes all the trajectories of vehicles, which are involved in an accident. It uses a so-called Circle Intersection Algorithm[13] to determine the best position to cut the trajectory and connect it to the first collision point. Further collisions are just connected by a straight line (see Figure 12).

[13] See Egges (2000) for a detailed description.

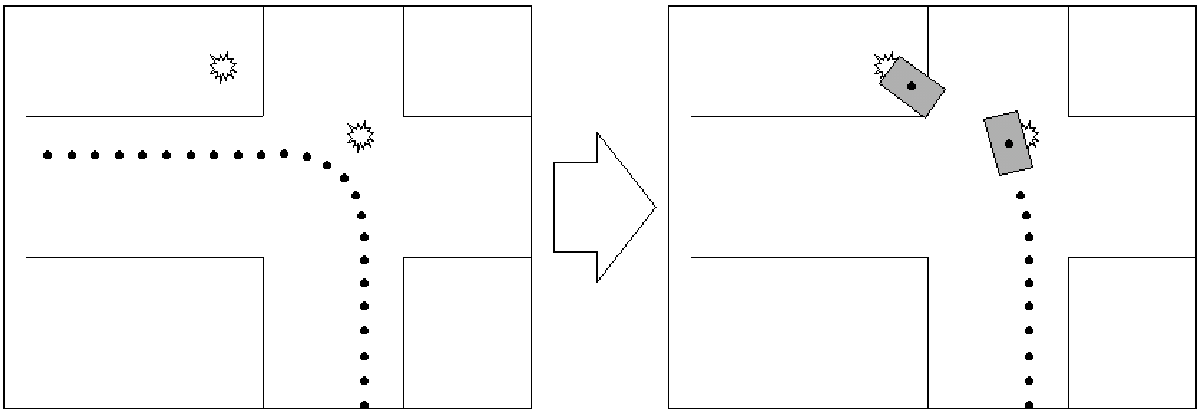


Figure 12: Task of the MultipleAccidentPlanner

4.1.5 TemporalPlanner.java

The last planner takes the complete trajectories and adds a time stamp to every point of a trajectory. The start point is set to 0.0, the end point to 1.0. CarSim distributes the collisions equally over the time slot ranging from 0.8 to 1.0. All the trajectory parts before the first collision are assigned with a duration, which corresponds to the percentage of their length referring to the whole distance (see Figure 13).

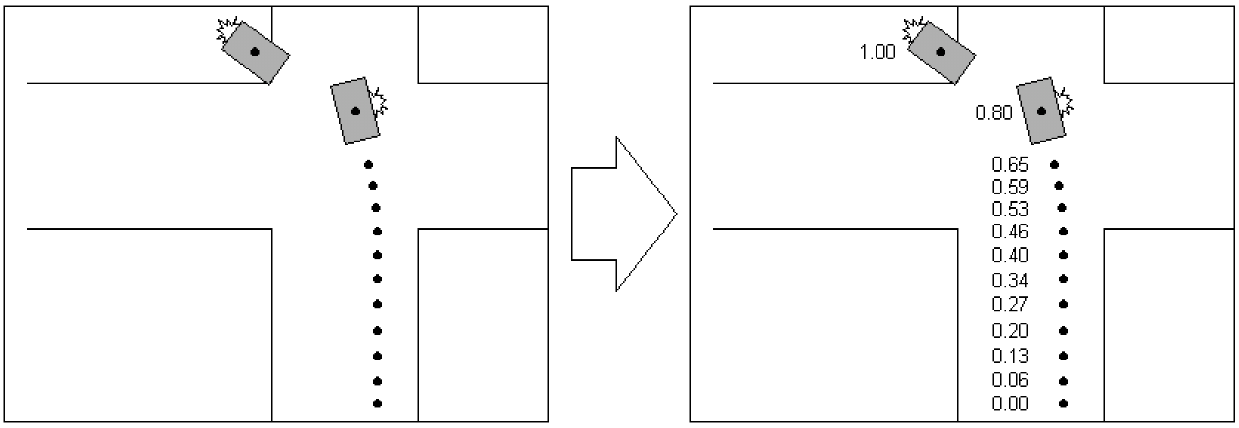


Figure 13: Task of the TemporalPlanner

4.2 The New Event "leave_road"

I considered all the occurrences of leaving a road in the corpus: the shape of the movement depends on the event before. A *leave_road_left* event has the same shape after the events *driving_forward*, *stop*, *turn_left*, *change_lane_right*. But if it follows a *turn_right* or a *change_lane_left* event, the movements look differently (see Figure 14).

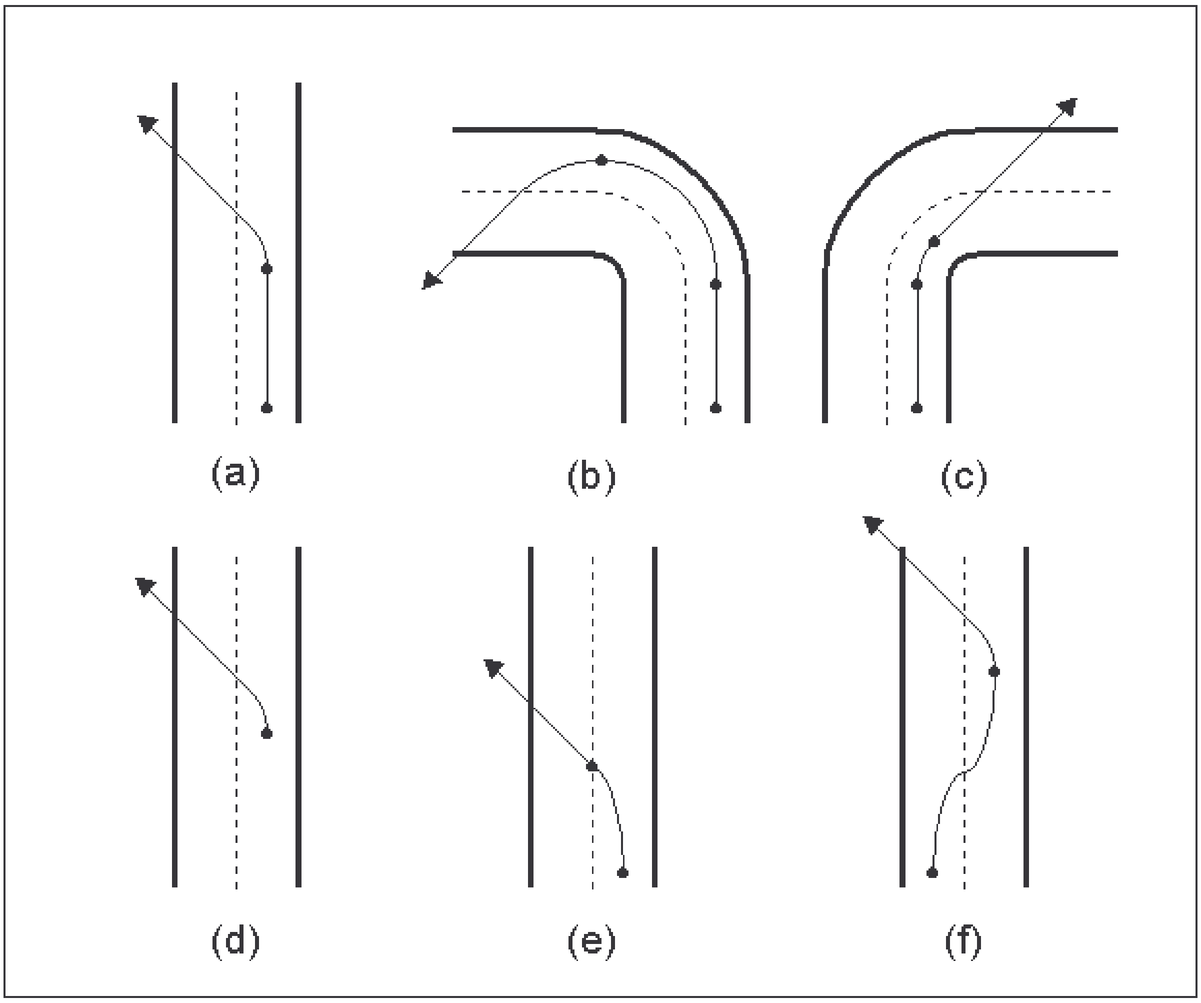


Figure 14: The leave_road_left event after…
(a) driving_forward (b) turn_left (c) turn_right
(d) stop (e) change_lane_left (f) change_lane_right

The hypothesis was that if a sentence like "…the car tried to turn right, but the driver lost control and the car left the road…", occurs in a report, the events *turn_right* and *leave_road* do not happen one after the other, but the car leaves the road during the right turn. So if a *leave_road_left* follows a turn_right (see Figure 14 (c)), some of the trajectory points of the *turn_right* event have to be taken away to let the new event starting within the turn.

If there is a *leave_road_left* after a *change_lane_left* (see Figure 14 (e)), we have the same situation. The half of the trajectory points of the *change_lane_left* event will be taken away to replace them with a straight line of new trajectory points which leads up to leave the road.

If there is on the other hand a *leave_road_left* event after a *turn_left* (see Figure 14 (b)), it makes no sense to leave the road during the turn. If the driver of the car loses control during a left turn, the car would try to continue its way on a straight line[14] and to leave the road to the right side. To leave a road to the left side after a left turn seems in any case not to be very realistic. But it can be understood as forgetting to turn back the steering wheel after the turn. For every combination of an event with the *leave_road_left* resp. *leave_road_right* event the – in a physical view – most realistic trajectories were taken.

[14] Because of the inertia of the mass

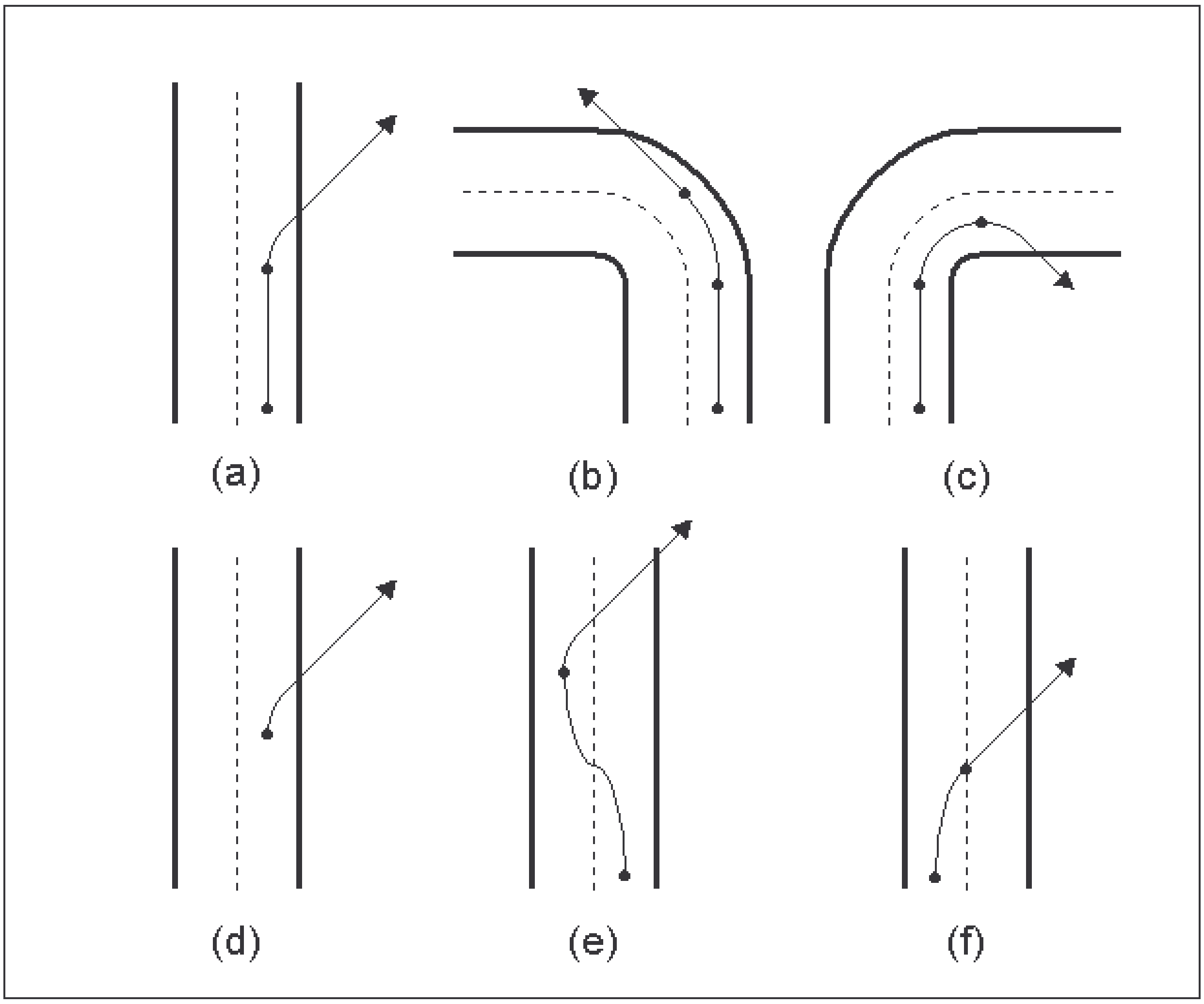


Figure 15: The leave_road_right event after…
(a) driving_forward (b) turn_left (c) turn_right
(d) stop (e) change_lane_left (f) change_lane_right

4.2.1 The Implementation

The DTD had to be extended by a new event (See Figure 16). *leave_road*, *leave_road_left* and *leave_road_right* became new values for the *kind* attribute of the element *event*.

```
...
<!ELEMENT eventChain ( event )+>
   <!ELEMENT event EMPTY>
   <!ATTLIST event
      kind       ( driving_forward | turn_left | turn_right | stop | overtake |
                 change_lane_left | change_lane_right |
                 leave_road | leave_road_left | leave_road_right)    #REQUIRED
      critical   ( yes | no )                                        "no"
   >
...
```

Figure 16: Part of the accident.dtd with the events
leave_road, leave_road_left and leave_road_right

The `FormalismParser`, which generates the AccidentInfo object from the XML document, had to be extended, so that the new events are known there, too.

The `PrePlanner` enlarged its responsibilities as well. In the previous sections, sometimes the events *leave_road_left* and *leave_road_right* were mentioned, but sometimes only *leave_road*. All these three events are possible. If it is not sure in a report if a vehicle leaves the road to the left or to the right side, the event *leave_road* can be indicated. The `PrePlanner` which also fills in unknown values for collision parts and initial directions is responsible to decide if a *leave_road* event means a *leave_road_left* or a *leave_road_right* event. It uses the following rules: If the previous event is *driving_forward*, *stop*, *turn_left* or *change_lane_right*, the *leave_road* event is changed into a *leave_road_right* event. If the previous event is *turn_right* or *change_lane_left*, it becomes a *leave_road_left*. These rules were set up after considering the probability of each direction after the particular events.

The scenarios of the `GeneralAccidentRoutePlanner` had to be extended (see Figure 17).

• Scenario I:	(DF | STOP)* TL [LRL | LRR] [OTL | OTR] (DF | STOP)*
• Scenario II:	(DF | STOP)* TR [LRL | LRR] [OTL | OTR] (DF | STOP)*
• Scenario III:	(DF | STOP)* DP [LRL | LRR] [OTL | OTR] (DF | STOP)*
• Scenario IV:	(DF | STOP)* CLL [LRL | LRR] [OTL | OTR] (DF | STOP)*
• Scenario V:	(DF | STOP)* CLR [LRL | LRR] [OTL | OTR] (DF | STOP)*
• Scenario VI:	(DF | STOP)+ [LRL | LRR] [OTL | OTR]

Figure 17: Scenarios in GeneralAccidentRoutePlanner

The scenerio grammar contains already the *overturn_left* (OTL) resp. *overturn_right* (OTR) events that will be introduced in the next section.

After every main movement of a scenario (like TL, TR, DP, CLL or CLR) an optional *leave_road_left* or *leave_road_right* statement follows. Thereafter an *overturn_left* resp. *overturn_right* is possible. The event chain can be a valid scenario as well if an *overturn* event follows directly the main movement without a *leave_road* event before.

These scenarios contain only the directional version of the events; the direction less *leave_road* and *overturn* were already replaced by the `PrePlanner` as mentioned before.

The `PathPlanner` considers only straight lines of successive *driving_forward* an *stop* events. Changes due to the *leave_road* events were not necessary. The `TrajectoryPlanner` however needed some extensions. Beside the already existing methods

```
private void planStraightRoad(Event e)
private void planStop(Event e)
private void planTurnLeft(Event e)
private void planTurnRight(Event e)
private void planChangeLaneLeft(Event e)
private void planChangeLaneRight(Event e)
```

there are now as well

```
private void planLeaveRoadLeft(Event e, Event ePrevious)
private void planLeaveRoadRight(Event e, Event ePrevious)
private void planOverturnLeft(Event e, Event ePrevious)
private void planOverturnRight(Event e, Event ePrevious)
```

It is conspicuous that the new methods have two `Event` objects as input parameters. The first one is the current event (*leave_road_left* for instance), the second one is the preceding event. Like already mentioned, the *leave_road* (or *overturn*) event sometimes starts after the previous event but sometimes already within it. So the methods needs the previous event object as well to determine its kind and out of this on which kind of trajectory the points to be added should lie. Example: The method `planLeaveRoadLeft(…)` is called. If the preceding event is a *turn_right*, only the first two trajectory points of the right turn will be kept. From this coordinates the method adds a 20.0 metres long straight line, which leaves the turn to its left side. The direction of the straight line is that of the connection of the last two coordinates of the right turn (see Figure 14 (c)). If the preceding event were a *change_lane_left*, half of its trajectory points are kept (the vehicle is placed in the middle of the road and would now start to turn right again). At this point the method adds the same straight line as described above (see Figure 14 (e)). If the preceding event were any other kind of event, the method would not take away any trajectory points, but add a twelfth circle arc and a 10.0 metres long straight line (see Figure 14 (a),(b),(d),(f)). After the `TrajectoryPlanner` has completed its work all the coordinates of the (provisional) trajectory are given. The `MultipleAccidentPlanner` changes the trajectory once again if the vehicle the trajectory belongs to is involved in a collision, and the `TemporalPlanner` computes the timestamps for every trajectory points. Both planners did not need any changes due to the extension by this new movement.

4.3 The New Event "overturn"

An overturn could mean that a car flies after a collision through the air, turns several times around itself, hits hardly the ground and finally stops after turning around further on its top[15]. But in all of the reports I have read the cars or trucks just toppled over and lay then on its right or left side. So I decided to implement only this light-version of an overturn which covers though most of the accidents. The movement of an overturn to the left side is shown in Figure 18. The vehicle turns from picture (b) to (e) 45 degrees clockwise around its vertical axis and 90 degrees anti-clockwise around its length axis. At the same time it moves a few meters in the direction, which is 45 degrees anti-clockwise from the old one. When the vehicle lays on its side it skids a few meters in the same direction.

In the accident reports of the corpus an overturn occurred mainly in two different ways: Caused by a collision or caused by a driver's failure without any influence of another vehicle (for example after an unwanted leaving of the road). In the second case the overturn can be implemented as just another event like *driving_forward*, *stop* or *leave_road_right*. But the first case cannot be implemented as an event. It has to be bound directly to a collision. So I decided to handle these two cases in two separate ways:

- *overturn*, *overturn_left* and *overturn_right* as new values for the *kind* attribute of the element *event*
- "no","unknown","left" and "right" as values for the new attributes *actorOverturn* and *victimOverturn* in the element *collision*

[15] …and explodes immediately!

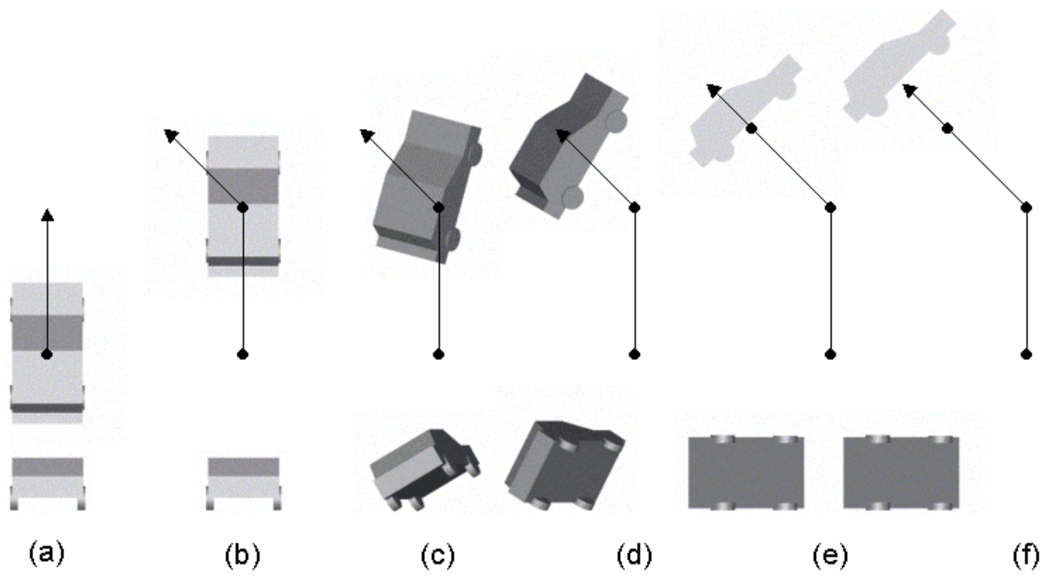


Figure 18: Overturn, top and back view
(a) right before (b)-(e) the vehicle overturns (e)-(f) the vehicle skids on its left side

See the extensions of the DTD in Figure 19.

```
            ...
            <!ELEMENT eventChain ( event )+>
               <!ELEMENT event EMPTY>
               <!ATTLIST event
                  kind       ( driving_forward | turn_left | turn_right | stop | overtake |
                             change_lane_left | change_lane_right |
                             leave_road | leave_road_left | leave_road_right |
                             overturn | overturn_left | overturn_right )  #REQUIRED
                  critical   ( yes | no )                                  "no"
               >

      <!ELEMENT collisions ( collision )+>
         <!ELEMENT collision ( actor, victim, coords? )>
            <!ELEMENT actor EMPTY>
            <!ATTLIST actor
               id            IDREF                                                #REQUIRED
               side          ( front | rear | leftside | rightside | unknown )   #REQUIRED
               overturn      ( left | right | unknown | no )                      "no"
            >
            <!ELEMENT victim EMPTY>
            <!ATTLIST victim
               id            IDREF                                                #REQUIRED
               side          ( front | rear | leftside | rightside | unknown )   #REQUIRED
               overturn      ( left | right | unknown | no )                      "no"
            >
            ...
```

Figure 19: Part of the accident.dtd with the extensions

Due to the similarity of the adaptations in chapter 4.2.1, I first describe the implementation of the *overturn* event.

4.3.1 Implementation of overturn as an event

The `FormalismParser` had to be extended as the integration of the *leave_road* event so that the *overturn* event could be saved internally. The `PrePlanner` which changes already *leave_road* to *leave_road_left* resp. *leave_road_right* decides now as well if an occuring *overturn* should be exchanged by an *overturn_left* or *overturn_right*. Like already mentioned in chapter 4.2.1, the scenarios of the `GeneralAccidentRoutePlanner` were extended by an optional *overturn_left* or *overturn_right* after every optional *leave_road_left* or *leave_road_right* (see Figure 17).

The new methods of the `TrajectoryPlanner` `planOverturnLeft(…)` and `planOverturnRight(…)` were introduced, too. Now it starts to become interesting. Up to this point, all the movements of a vehicle were defined just by an order of 2-dimensional points and a timestamp belonging to it. The system computes automatically the angle of the car so that the front of the car always looks in the driving direction. These data is not enough anymore. The parts from (b) to (f) of Figure 18 show that the vehicle is not turned into the direction it moves but moves sideways. At the same time, it turns around its length axis. To save this information, the class `TrajectoryPoint` had to be extended by two variables:

- `public float overturnXAngle`
- `public float overturnYAngle`

The first variable contains the rotation angle of the vehicle around its length axis. That's why the second variable is needed if the vehicle should move sideways and its front part does not point in the driving direction. Figure 20 shows the position of the two new angles.

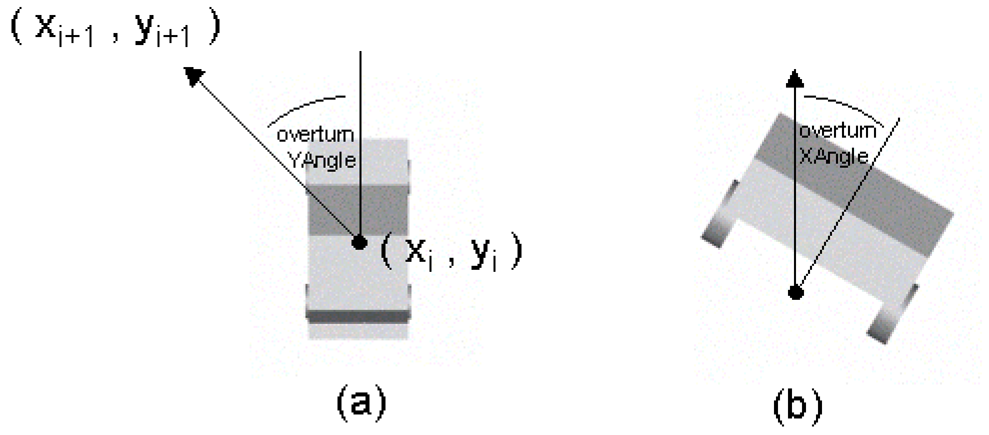


Figure 20: Two (different) trajectory points
(a) top view to demonstrate overturnYAngle
(b) back view to demonstrate overturnXAngle

A new constructor method for the `TrajectoryPoint` class was necessary:

```
public TrajectoryPoint(float xVal,yVal,otxAngle,otyAngle,perc, boolean c)
```

Now it is possible to set the values for these two angles as well[16]. This is the task of the method `planOverturnLeft()`[17] in the `TrajectoryPlanner`. If the vehicle starts to overturn to the left, it changes its driving direction 45 degrees to the left. Normally the vehicle would be turned pointing to this direction. But now it should first stay in the same direction and turn

[16] The old constructors used by all other methods of the TrajectoryPlanner are still available. They do not provide these two angles as input parameters but initialize them internally with default values (0.0).

[17] or planOverturnRight() of course

then by 45 degrees to the right until it is turned by 90 degrees to the right from the moving direction (see Figure 18 (b)-(e)). So the variable `overturnYAngle` which is normally 0.0 is set to -0.25 * π and ends at -0.5 * π. At the same time, the vehicle turns around by 90 degrees to its left side. The variable `overturnXAngle` starts at 0.0 and ends at -0.5 * π. Here is a sample code that adds the points to the trajectory:

```
float overturnXAngle = 0.0f;
float overturnYAngle = -(float)Math.PI/4;
int overturnPrecision = 6;
float anglePart = (float)Math.PI/(2.0f * overturnPrecision);

for (int i = 1; i <= overturnPrecision; i++)
{
      xVal = t.get(currTrajIndex-1).x + deltaX;
      yVal = t.get(currTrajIndex-1).y + deltaY;
      overturnXAngle -= anglePart;
      overturnYAngle -= anglePart/2;

      // Add the point to the trajectory
      t.add(new TrajectoryPoint(xVal, yVal,
            overturnXAngle, overturnYAngle, 0.0f, e.critical));
      currTrajIndex++;
}
```

The turning around is subdivided in `overturnPrecision` (= 6) parts. The overturn angles are adjusted within the for-loop as described above. The `x` and `y` coordinates are increased by delta values so that the vehicle moves each time by a vector which is defined by `deltaX` and `deltaY`. These two values are computed before in that way that the vector has the desired length and points in the desired direction.

Now all necessary data is saved in the trajectory. Finally, we have to say the system what to do with the additional information. The class, which is responsible for the behavior of the vehicle, is called `TrajectoryBehaviour`. The core of this class builds the class `RotPosScaleTCBSplinePathInterpolator`. This class from the package `com.sun.j3d.utils.behaviors.interpolators` computes translations and rotations of an object between given edge values for coordinates, angles, and timestamps. Its constructor needs a `TCBKeyFrame` array. Each `TCBKeyFrame` object contains among other things the following values:

```
float percentage // time value between 0.0 and 1.0
Point3f position // a 3-dimensional point
Quat4f quat      // determines the angles
                 // with a Matrix4d object as input variable
```

The position coordinates are:

```
positions[i] = new Point3f(traject.get(i).x,
                           overturnHeight,
                           traject.get(i).y);
```

with

```
overturnHeight =
   Math.abs(((float)Math.sin(overturnXAngle))*target.getWidth());
```

The y-value of the trajectory becomes the z-value of the Java3D coordinate system because the x-z-plane forms the horizontal plane. Its y-axis is vertical. For a better understanding of the computation of `overturnHeight`, see Figure 21.

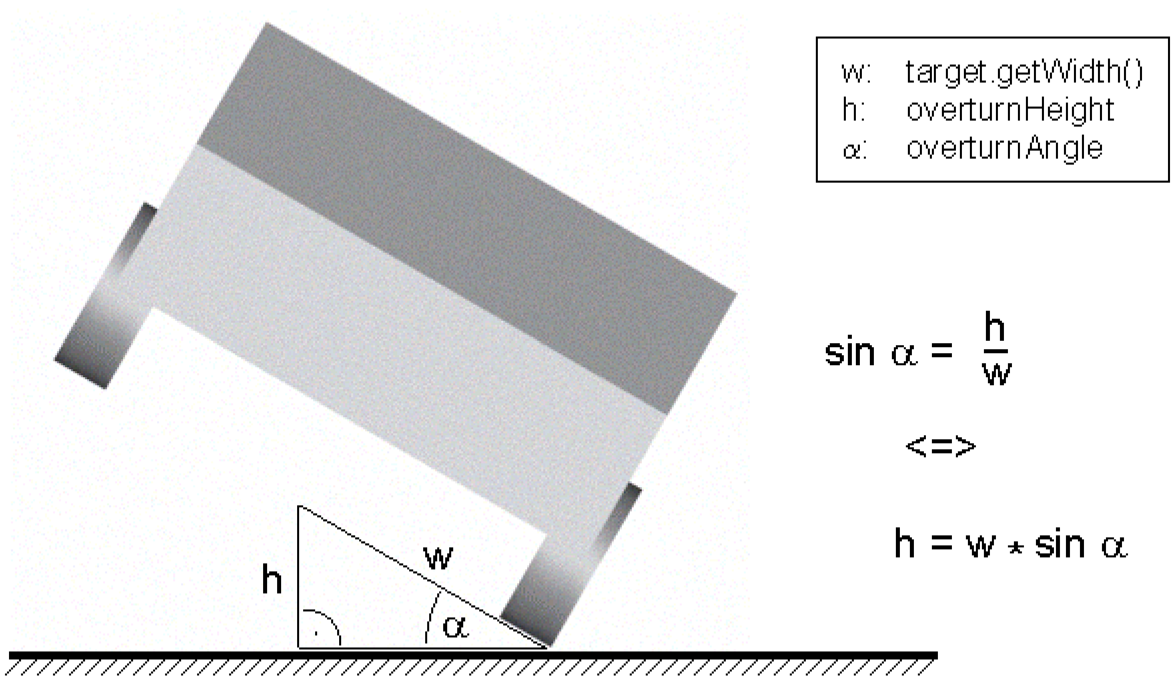


Figure 21: Sketch of overturnHeight

The rotations are defined by a `Quat4f` object. This object is built by a matrix product.

```
Matrix4d mat = new Matrix4d();
Matrix4d rotMat = new Matrix4d();
…
mat.rotY(angle + overturnYAngle);
rotMat.rotX(overturnXAngle);
mat.mul(rotMat);
quats[i] = new Quat4f();
quats[i].set(mat);
```

The matrix `mat` contains the rotation around the vehicle’s vertical axis defined by the driving direction dependent `angle` and the additional `overturnYAngle`. `rotMat` is the matrix that carries out the rotations around the vehicle’s length axis. Both matrices are multiplied with each other and form the final rotation matrix `quats[i]` for the i^{th} TrajectoryPoint.

4.3.2 Implementation of overturn as a collision attribute

The handling of an overturn as an attribute of the collision event is slightly different. First the `FormalismParser` has to be extended to be able to handle the new attributes. It saves its values in the respective `CollisionInfo` object whose class was extended by the two variables `actorOverturn` and `victimOverturn`.

If the overturn attribute has the value „unknown“, the `PrePlanner` decides the direction like for the overturn event.

Though the `GeneralAccidentRoutePlanner` was affected by the overturn event, the overturn attributes have no effect to this planner. The planner considers only events and the overturn attribute is no event of the event chain.

The methods `planOverturnLeft()` and `planOverturnRight()` of the `TrajectoryPlanner` are almost entirely copied into the methods `addOverturnLeft()` and `addOverturnRight()`[18]. These new methods were added to the `MultipleAccidentPlanner`. Each time when a vehicle is involved in a collision in which it overturns one of these new methods will be called.

In contrast to the overturn event, the `TemporalPlanner` needed an extension. The time stamp of the last collision is generally set to 1.0. Therefore, if an overturn occurs after it, the trajectory points thereafter are assigned with values greater than 1. So the time stamps have to be normalized. After all the trajectories have been assigned with time values the `TemporalPlanner` iterates over all the trajectories and takes the greatest time value. If this value is greater than 1, all the time values of all the trajectories are divided by this value. Thereafter the planner checks if all the trajectories end with a trajectory point at the time stamp 1.0. If not, an additional trajectory point is added which has the same location as the previous one and the time value 1.0.

[18] the only part which is missing is the taking away of previous trajectory points

5 Proposals for Extensions and Improvements

During the course of the project, several proposals crossed my mind regarding the improvement of CarSim.

First of all, the **`TemporalPlanner`** needs to be enhanced. For example: If two vehicles first collide together and then each of them bangs into a tree, the XML document contains three collisions. CarSim sets the time stamps of the collisions independently of the distance differences of the two trajectories after the collision: to 0.8 for the first collision, 0.9 for the second one and 1.0 for the third one. If the trees have nearly the same distance from the first collision point, the last car takes twice as much time as the other. In reality, the cars would crash against the trees rather at the same time.

Furthermore, vehicles don't have a constant speed. The `TemporalPlanner` uses a fixed duration for every vehicle[19]. Therefore, the velocity of a vehicle is given by

```
    velocity = duration / trajectory length
```

A vehicle with the event chain *stop*, *turn_left*, *stop* crawls around the turn whereas a vehicle having the event chain *driving_forward*, *turn_left*, *driving_forward* drives much faster. A `TemporalPlanner` of the next generation would take a constant velocity instead of a constant duration.

At this point, another restriction has to be made. It affects one of the movements I added to the system: the *overturn* event. In the current version of CarSim, the overturn looks at little bit clumsy, because the vehicle overturns very fast. This comes from the fact that the `TemporalPlanner` determines the values for the time stamps according to the distance to the last trajectory point. Because of the trajectory points of the overturn movement lie narrow to each other, the system presses the whole overturn in a quite small time slot. Therefore, the vehicle's overturn seems a bit unrealistic.

The next suggestion I do is to reconsider if it is really advantageous to seperate between ***eventChain*** and ***collisions*** in the formalism. As described in Chapter 5.3, the *overturn* movement had to be integrated to the system in two ways: as an event and as an attribute of the *collision* element. Consider, *collision* would be an event like *driving_forward*, *stop* or *overturn*. Then neither the *actorOverturn* and *victimOverturn* attributes were necessary nor the attribute *critical* of the *event* element[20]. But it could entail extensive consequences concerning e.g. `CollisionPlanner`, the `TemporalPlanner` and even the information extraction module.

The last extension is easier to implement. The roads ***straightroad***, ***turn_left*** and ***turn_right*** are restricted to certain directions. A straight road lies only in east-west direction. A left turn comes from the west and goes to the north. A right turn comes from the west and leads then to the south. Other orientations aren't possible in the current version of CarSim. The DTD could be extended e.g. in that way:

```
<!ATTLIST road
  kind ( straightroad | ns | we | turn | nw | ne | sw | se | nesw )
    #REQUIRED
>
```

[19] Although the user can choose the value "duration of the animation" in the property frame before starting the simulation, it is a fixed value in this context. All the trajectories are assigned with values from 0.0 to 1.0. Therefore, every vehicle needs the same time to pass the trajectory, not depending on its length.

[20] The aim of a critical event is to indicate the event where the vehicle is involved in a collision. This is mainly needed for collision with stopped vehicles. See Egges (2002, Chapter 7.3) for details.

The characters *n*, *e*, *s*, *w* indicate the four orientations north, east, south and west. *ns* is a straight road from north to south, *sw* is a turn from south to west and *nesw* is a crossroad, which points to all the directions. Additionally, the T-junctions *nes*, *esw*, *swn*, *wne* would be possible to implement, too. The values *straighroad* and *turn* are default values, which are taken if the orientation is not known from the accident report. The `PrePlanner` tries to derive the orientation by knowing start and end directions of the vehicles. Otherwise, it assigns fixed default options to the variables.

6 Commented Bibliography

Armstrong, E. (2002) *Java API for XML Processing (JAXP) Documentation*
Online-Documentation available at http://java.sun.com/xml/jaxp/docs.html

Deitel, H.M., Deitel, P.J., Nieto, T.R., Lin, T.M. and Sadhu, P. (2001) *XML How to Program* Prentice Hall.

Dupuy S., Egges, A., Legendre, V., Nugues, P., (2001) Generating a 3D Simulation of a Car Accident From a Written Description in Natural Language: The CarSim System, in *Proceedings of The Workshop, on Temporal and Spatial Information Processing*, pp. 1-8, ACL 2001 Conference, Toulouse, 7 July 2001.
Download: http://www.cs.lth.se/~ pierre/Articles/acl2001/acl2001.pdf

Egges, A. (2000) *Generating a 3D Simulation of a Car Accident from a Formal Description: the CarSim System,* Internship report, ISMRA and University of Twente.
The report about the first version of the visualization module of CarSim. It explains the planning strategies and the underlying internal data structure in more detail.
Download: http://www.cs.lth.se/~pierre/memoires/arjan/report.tar.gz

Flanagan, D. (2002) *Java in a Nutshell,* 4th Edition, O'Reilly & Associates

Münz, S. (2001) *SELFHTML 8.0 (HTML-Dateien selbst erstellen)*
The section XML/DTDs was of interest for getting into the topic XML and for creating the DTD for the new CarSim formalism. The documentation is in German and available at http://selfhtml.teamone.de. Unfortunately, there is no English translation. Available translations are in French, Spanish and Japanese, but the chapter about XML is not translated as yet. For English informations, see the officiell XML specification of the World Wide Web Consortium (W3C) at http://www.w3.org/XML/.

Svensson, H. and Åkerberg, O. (2002) *Development and Integration of Linguistic Components for an Automatic Text-to-Scene Conversion System,* MSc Thesis, LTH, Lund.
The report of the MSc thesis about the information extraction module of CarSim.

7 Appendix A

Two documents for the same accident: left the old version, right the new XML version.

```
// Static objects
STATIC [
   ROAD [
      KIND = crossroad;
   ]
   TREE [
      ID = tree1;
      COORD = ( 5.0, -5.0 );
   ]
]

// Dynamic objects
DYNAMIC [

   VEHICLE [
      ID = vehicleB;
      KIND = car;
      INITDIRECTION = east;
      CHAIN [
         EVENT [
           KIND driving_forward;
         ]
      ]
   ]
   VEHICLE [
      ID = vehicleA;
      KIND = car;
      INITDIRECTION = north;
      CHAIN [
         EVENT [
           KIND driving_forward;
         ]
      ]
   ]
]

ACCIDENT [
   COLLISION [
      ACTOR = vehicleB, front;
      VICTIM= vehicleA,leftside;
      COORD = ( 1.0, 1.0);
   ]
   COLLISION [
      ACTOR = vehicleA, front;
      VICTIM = tree1, unknown;
   ]
]
```

```
<?xml version="1.0"?>
<!DOCTYPE accident SYSTEM "accident.dtd">
<accident>

  <staticObjects>
    <road kind="crossroad" />
    <tree id="tree1">
      <coords x="5.0" y="-5.0" />
    </tree>
  </staticObjects>

  <dynamicObjects>
    <vehicle id="vehicleB" kind="car"
             initDirection="east" >
      <eventChain>
        <event kind="driving_forward" />
      </eventChain>
    </vehicle>
    <vehicle id="vehicleA" kind="car"
             initDirection="north" >
      <eventChain>
        <event kind="driving_forward" />
      </eventChain>
    </vehicle>
  </dynamicObjects>

  <collisions>
    <collision>
      <actor id="vehicleB" side="front" />
      <victim id="vehicleA"side="leftside"/>
      <coords x="1.0" y="1.0" />
    </collision>
    <collision>
      <actor id="vehicleA" side="front" />
      <victim id="tree1" side="unknown" />
    </collision>
  </collisions>

</accident>
```