Cooperative Multi Agent Systems in Automobiles

By Mirco Hering

Master of Science

(Artificial Intelligence)

School of Informatics

University of Edinburgh

2004

Abstract

This paper describes the process of designing a scenario and a protocol for a multi agent system in cars as well as the testing process afterwards. The Webots simulation software is used in simulation tests to evaluate the success of the protocol. Another aspect of testing is covered with the usage of the Spin Model Checker. With this model checking tool model simulations and different verifications were performed.

The protocol design was partially successful, but some flaws were found during the testing process. As a more important outcome a totally different approach seems possible which includes the Spin Model Checker in earlier stages of the design process for a Multi Agent System protocol.

Acknowledgements

I would like to thank Dave Robertson and Chris Walton for their support throughout the project. Whenever there was a problem it was possible to discuss it and find a way to solve the problem.

I would also like to thank Cyberbotics and especially Olivier Michel for granting me an evaluation license for the duration of the project. It was possible for me to work from home thanks to this license.

Many thanks go out to my fellow students Shahryar Kashani and Mischa Tuffield. On many occasions we discussed our projects and found new views and ideas which enriched our projects. And also we had a great time during this year.

Last but not least I would like to thank my parents for their financial support, which made the year in Edinburgh possible and for all their support throughout the year.

Thanks to all of you, who made this year a great experience in my life.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(Mirco Hering)

Table of Contents

Table of Figures

1 Introduction

This dissertation investigates the possibility to design a multi agent protocol which allows a multi agent system in a car to make rational decisions in a certain environment and the possibility to evaluate the flaws of this protocol with the aid of simulation and model checking. The protocol is designed for a certain scenario, which is quite popular for projects like this.

The simulated scenario is a three lane motorway with multiple cars on it. The model car has to brake and accelerate on this motorway and change lanes if the car in front of it is coming closer/ too close.

The motivation for this scenario lies in the fast changing technical development in car industry and research. In the last years cars developed from simple mechanical devices towards vehicles with many electrical and computational elements like distance sensors, GPS devices and navigation systems.

As a testing environment the Webots [1] mobile robot simulation software is chosen, which allows to construct a world in which robots can interact with each other and the environment. Webots supplies the functionality to design robots and their sensors. It also allows the simulation of scenarios with the according sensor output and to actively change the behavior of the agents according to a programmed controller. For this project the controller for the model car is merely a proxy, which receives its “actions” from an agent in a Multi Agent System.

The core multi agent system is developed on the MagentA [2] platform, where the single agents are web services. The interaction between those agents/web services is controlled by a protocol based on the MAP language [3], which is used in connection with the MagentA platform. The rational behavior generated by this protocol is based on the interaction between single agents and some simple calculations inside each agent.

The actual protocol is then tested with the Webots scenario, where the sensor output is redirected into the Multi Agent System and the according actions are transmitted back to the Webots simulation.

Further testing of the protocol is achieved by model checking and simulation with the Spin Model Checker [4]. So both practical and theoretical testing are performed with the protocol during this project.

After performing a set of tests with the protocol in simulation using Webots it becomes clear that the protocol still has some weaknesses. In the beginning of each simulation the model car behaves well, but the longer the simulation lasts the behavior of the model gets more and more inappropriate. The Spin Model checker is used to gain insight into the behavior of the model car and discover reasons for the inappropriate behavior.

The model checking showed more weaknesses and reasons for the flaws already found in simulation and would make it possible to develop a better protocol with the conclusions drawn. Even a different approach with using the Model Checker first seems to be appropriate, but more about that in later chapters.

This introduction is intended to give a short overview over the project and the structure of this thesis. The basic layout is like this:

Chapter 2 describes the background of this project, related research and literature for further reading;

Chapter 3 gives a short summary of the scenario;

Chapter 4 is a very brief discussion of physical modeling of the cars and the environment;

Chapter 5 covers the controller for the dummy cars to show how their very simple behavior is designed;

Chapter 6 is a description of the multi agent system architecture and the implementation of this architecture for this project;

Chapter 7 is covering the abstract design of the protocol;

Chapter 8 describes the tests which were performed to evaluate the success of the protocol in a simulated environment;

Chapter 9 is describing the test and simulation results which were achieved by the usage of the Spin Model Checker;

Chapter 10 sums up the results of this project, what was achieved and how;

Chapter 11 recommends things for further research in this area which would be worth to look into.

2 Background and Literature Review

In this chapter some motivation for the scenario topic of this project is given by outlining the development of automobile technology and showing why research in this area is important and interesting. The time line of automobile technology is covered to show how the purely mechanical cars developed into vehicles which include a lot of electronical gadgets. After that, previous projects with Webots are described and it is showed why it seems suitable for this project. One of these projects includes car models and their behavior. In the next section key concepts like agent, multi agent system and environment are discussed. In the following two parts of this chapter the MAP language and the MagentA platform with the according papers are illuminated. MAP and MagentA were developed at the University of Edinburgh, so for this project the newest version of MagentA is used. The last section covers Model Checking Technologies, how they work and what their purpose is.

2.1 Automobiles

This section about automobiles will motivate the scenario of this project from the aspect of current research in automobile technology. At first a short review of history in automobile technology and then a deeper look into the ongoing research is given. This should give an idea why car technology is a growing and interesting field for computer scientists and that there are still many possibilities to make contributions.

2.1.1 The early beginnings

After James Watt developed the steam engine in 1765 it took over a hundred years until in 1886 Karl Benz used an Otto engine to construct the first vehicle that was able to drive only with the help of this engine. [5] From there on the automobile became a very important part of human society. The first stream of improvements for cars covered the reliability and performance of automobiles. Better engines, more comfortable car bodies and many more developments were made in the next hundred years. Of course all those improvements were purely mechanically. Later the first electric devices were introduced like electric light, before the invention of computers and especially reasonable small computing devices made it possible to include computer technology in cars. In addition to making cars more comfortable and powerful security and luxury became a main focus of car development in later years.

2.1.2 Modern Cars

In the last ten, twenty years cars evolved into a system of mechanical, electric and computational devices, which improve safety and comfort of the passengers. Examples for the more common devices nowadays are the stereo, airbag and air conditioning. Those devices are not really interesting for a computer scientist, besides the possibilities to manage those devices. Nevertheless computer science is getting more and more important for the automobile industry, because it offers a variety of possible ways to improve cars to make them more desirable for the audience. The automobile industry introduced computing devices for engine maintenance, theft security, on board communication and other devices for safety and luxury.

One of the key interests in the last few years is the assistance of the drivers while they are driving the automobile. The following section will describe older and ongoing research in the area of Driver Support Systems.

2.1.3 Driver Support Systems (DSS)

Driver Assistance Systems are covering three different tasks:

Information DSS provide appropriate information for the driver, which help the driver to avoid stressful and difficult situations.

Warnings In dangerous situations the driver is informed by the DSS to assure that the driver is not missing a dangerous situation.

Interventions Supporting interventions by the DSS are provided to help the driver in stressful situations and improve the safety of driver.

The most sophisticated scenario for DSS is the autonomous driving on public streets. The first tests with autonomous driving were carried out in the 1980s. One of the first projects in this area was autonomous driving on the motorway with 100 km/h, which was based on the interpretation of single pictures and keeping track of the current and previous situations. [6]

The motorway is a well-fitted scenario for tests in this area, because the street is mostly straight, no cars are driving in the opposite direction and street signs are rare. For example driving in the city is way more complicated, because it involves more participants in traffic like bicycles and pedestrians. The road markings are more complicated and less separable, road crossings are appearing and the traffic signs are more important for successful behavior in this environment.

Most of the early DSS were based on Computer Vision technologies; as this thesis is not using Computer Vision only the work by C.E. Thorpe [7] is mentioned, which gives a summary on Computer Vision projects at Carnegie Mellon University in the 80s. It gives a good overview over approaches in the area of DSS in connection with Computer Vision.

As DSS became more important major projects like the European PROMETHEUS (PROgraMme for a European Traffic with Highest Efficiency and Unprecedented Safety) [8] and the IHVS (Intelligent Vehicles Highway System) were introduced to accumulated research efforts in this area.

One of the first DSS which is marketed for a broad but exclusive audience and which includes all three aspects of DSS is the Distronic technology by

DaimlerChrysler [9]. Distronic is an advanced version of the nowadays quite common Tempomat, which allows the driver to choose a traveling speed and the car holds this speed while the system is active. In difference to the Tempomat Distronic also includes a radar sensor which checks the distance to the car driving in front. If the car in front comes to close the Distronic system activates the brakes to adjust the distance. If the car is coming closer too fast for normal braking the DSS warns the driver with a signal light and an acoustic signal to mark the need of an intervention. [10]

Of course this is just one of the new technologies which are used nowadays or are in development at universities and automobile companies. There are devices for improved navigation, driving in large groups (like lorries) and many more, which are not described further here.

This project bases the information about the outside world on distance information obtained by infrared sensors, so the Distronic system is a good example for the relevance of this approach.

This should conclude this chapter about the motivation for this project. It should have given you an idea, why research in this area is interesting and that DSS and computer based systems in automobile technology are still an area where a lot of research is going on and that self-driving cars like those in the movie "Minority Report" are still dreams of the future.

2.2 Webots

"The Webots mobile robotics simulation software provides you with a rapid prototyping environment for modeling, programming and simulating mobile robots. The included robot libraries enable you to transfer your control programs to many commercially available real mobile robots." [1]

The simulation environment, which is used in this project to simulate a "realistic" traffic scenario, is Webots by Cyberbotics [1], a widely used software to simulate mobile robots and their sensors. It allows modeling a 3D world and robots. The behavior of the robots is defined by a controller implemented in JAVA or C. It became widely known in the Informatics community with a competition they started, called Roboka [11], which simulates a humanoid wrestling robot, for which a controller has to be designed. The controller driven robots are fighting against each other to earn points and climb in the ranking. This competition attracts computer science students from around the globe to compete against each other on the basis of their knowledge in programming and their skills in robotics.

A lot of common robot types like AIBO, Hemission, Khepera and Koala are already included in the software package, while it is also possible to design your own robot models.

A project with a closely related topic was carried out at the California Institute of Technology. Its aim was to use the principle of Swarm intelligence and evolutionary ideas to explore the traffic situation and optimize the control system in the simulation. [12] The scenario was, as in this project, a three lane motorway. They were using small, circular, unicycle robots to make it possible to test the results with their robots in a real life scenario.

This dissertation uses a different approach to the problem which is based on a Multi Agent System; therefore the following section is covering ideas about Agents and Multi Agent Systems in particular.

2.3 Agents and Multi Agent Systems

In this section some basic principles of agents and multi agent systems should be mentioned to describe what the meaning of those words is. The first part is used to introduce different kinds of agents and environments and to classify this project with respect to those different kinds. In the middle bit of this section multi agent systems are introduced with their features, strengths and weaknesses. In the last bit of this section it is stated why it seems to be appropriate to use an agent/multi agent system approach to the problem of autonomous driving.

2.3.1 What is an agent?

An agent can be any autonomous software system that interacts with its environment through sensors and actuators, in which some kind of reasoning is performed to decide its behavior. The sensors can be seen as input into an agent function, while the actuators can be seen as output. The task of the agent is to find the appropriate outputs to a given set of inputs. So the key task to develop a successful agent, where successful means he acts rational, is to find an agent function that allows the agent to find those outputs with the help of the inputs.

There are different kinds of agent functions, which can be used to classify agents in five groups:

i) Simple Reflex Agents

This is the simplest kind of agent. Its agent function just uses the current perception of the environment, the current inputs, to decide which actions should be executed. It totally ignores previous data like previous perceptions, actions or achievements. Obviously those agents cannot be very intelligent on its own, but the combination of many might still be successful.

ii) Model Based Reflex Agents

This type of agent is a bit more sophisticated as it maintains an internal state of the world to keep track of events that happened earlier in his lifecycle. This internal state is based on a model of the world, which contains the information of how the agent affects the world and how the world evolves without interaction with the agent. Creating a model of the world is the part which makes this kind of agent more difficult to implement than the simple reflex agent.

iii)Goal Based Agents

Goal based agents base their actions on the aim to reach a certain goal instead of just evaluating the current state they are in. Those agents are more flexible than the previous ones. If the world is changing, all states are changing and have to be reevaluated in the reflex agents and little or no information can be reused. The goal based agents would still have the same goal and therefore are able to use the same principles to reach this goal in the changed world.

iv) Utility Based Agents

For agents of this kind a utility function has to be designed, which represents the degree of happiness or unhappiness a certain state will provide for the agent. The decisions of the agent function will be based on this utility function to evaluate the best action for the agent.

v) Learning Agents

The most sophisticated kind of agent is the learning agent. This kind of agent uses its performance in earlier steps to evolve its behavior to a more successful one. Obviously this kind of agent needs a much more complicated implementation than all other kinds of agents, as this agent type needs to evaluate its performance, explore new behaviors and try to find the best possible way to response to every situation.

All these kinds of agents are more thoroughly covered in Chapter two of Russells and Norvigs book "Artificial Intelligence – A Modern Approach" [13].

For this project mainly the first kind of agent, the simple reflex agent, was used, as the intelligence or rationality of the model car (model driver) is arising out of the interaction between multiple agents, a so called multi agent system. Only in a few cases information from previous steps is used in the evaluation of the next action(see Chapter 6 The System Architecture).

But before multi agent systems are described, some words about the environment used in this project should be mentioned.

2.3.2 Agent Environments

The before mentioned book by Russell and Norvig [13] classifies agent environments in 6 dimensions. The classification here will follow this recommendation to classify the properties of the environment of the model car.

The six dimensions, agent environments can be classified in, are:

i) Observable

An environment is fully observable if the sensors can access the complete state of the environment at every point in time. Otherwise the environment is only partially observable.

ii) Deterministic

A deterministic environment would mean that every action an agent is executing would result in one completely determined new state of the world. If this is not the case the environment is called stochastic, in which case the agent cannot be sure which state of the world his action will result in.

iii) Episodic

In an episodic environment the agent's lifecycle is divided in reoccurring episodes, where each episode is totally independent of the previous one. In the case that there are no episodes or the episodes are influencing each other, the environment is called sequential.

iv) Static

In a static environment the agent can basically take all the time it needs to evaluate its next action, because the environment is not changing over time. Otherwise, if the environment is dynamic the agent has to deal with the possibility that if he takes too much time long to decide for a certain action, the environment might have changed to a state where his action is totally inappropriate.

v) Discrete

Discreteness of the environment is dealing with the states of the world and the sensory input of the agent. If those are discrete, which means only a certain amount of different states or values can appear, the world is discrete, otherwise it is continuous. Of course if agents are implemented on a computer everything is in one way or the other discrete, but if the number of different values is quite big, e.g. float values, then it is treated as continuous.

vi) Agents

If the agent is alone in his environment then it is a single agent environment, otherwise if more then one interacting agent is involved it is a multi agent environment.

After clarifying what each dimension means, we can now try to classify the environment in which the model car is acting in: the three way motorway with multiple cars on it.

It is obviously a multi agent environment as it involves other cars respectively driver agents and it is continuous because the amount of states and values for the sensory inputs are surely uncountable.

The environment is highly dynamic, because other agents are changing the environment all the time, obstacles are moving and so the agent has to decide for an action quite fast to assure that the action is still accurate for the situation. Although we assume for this project that every decision is independent from the previous action (with one little exception when changing the lane) the whole environment is sequential because previous decisions influence the state of the environment for the next decision.

Even if you would like to assume that driving a car is deterministic, you have to take into account that sometimes errors or differences in behavior can occur while driving a car. For example rain can influence the braking behavior or some parts of the car might experience a malfunction. So the environment is stochastic.

The last dimension to be covered is whether the environment is observable or only partially observable. As cars might be hidden behind other cars and not the whole motorway in front of the model car and behind the model car can be covered with sensors it is obviously only partially observable.

The conclusion of this classification is that this environment is in the hardest category for agent behaviors, because it is a partially observable, stochastic, sequential, dynamic and continuous multi agent environment.

To tackle this problem some simplifications have been made and a multi agent system environment is used to achieve some success.

2.3.3 Multi Agent Systems

Multi Agent Systems (MAS) are defined in [14] as "the subfield of AI that aims to provide both principles for construction of complex systems involving multiple agents and mechanisms for coordination of independent agents' behaviors".

A Multi Agents System consists of multiple autonomous agents, that can solve together bigger problems than they would be able to solve individually. The key advantages of an MAS are that each agent can be very simple to implement (for a example a simple reflex agent) and that every agent can be implemented independently as long as its specification is respected to assure the correct interaction with the other agents. Another nice aspect of MAS is that they are scalable because including a new agent in a MAS is easier than changing a large system.

The intelligent behavior of the MAS evolves out of the organized interaction between the agents. Therefore it is important to assure that this interaction is organized in the right way through a protocol or some other way of sharing information. One task in this project is to design such a protocol for a MAS consisting of sensors and other agents in an automobile.

Ideally every agent would have its own computation power to solve its own task, which is one of the reasons that an MAS can be more powerful than one single large system. In this project this aspect cannot be reflected as everything is run on one machine, but in real life the system can be ported on a distributed system to allow that behavior.

In an MAS the designer has to decide how much intelligence is implemented into the agents and how much is included in the communication. It is possible to use only some quite powerful agents or a larger amount of simpler agents. The trade off has to be made according to the problem, which has to be solved, because too little intelligence in the communication would contradict the idea of MAS while too much communication might cause delays because of the entire message passing procedures.

That a modern car actually can be described as an MAS from an abstract viewpoint is discussed in the next section.

2.3.4 Modern Cars – a multi agent system?

As described earlier in this chapter, a modern car includes of lot of high-tech gadgets like stereo, navigation system, electronic engine control, braking assistance and so on. Each for itself is useful for the human driver, but if you look at all those elements you can see that they can all be seen as agents. If you would be able to connect those agents in a way that they can communicate and share their information about the world in an organized way it might be possible to achieve a multi agent system that is able to drive a car on its own. Surely we are still quite far away from that although some progress has been made. But this idea is motivation for this project: the modern car as a multi agent system. Of course for this project an abstract view of the car is used, and all sensory agents are of the same kind (Infra Red Sensors), but nevertheless it is worth using this special scenario.

2.4 Multi Agent Protocol (MAP) Language

This section only gives a very brief introduction to the MAP language and mentions the works of Christopher Walton about it.

“The MAP language is a lightweight dialogue protocol language” [15] that allows designing a multi agent protocol in an easy, convenient way. The following graphic shows the abstract syntax of the MAP language.

$$
\begin{array}{lcll}
P & ::= & n\,[\mathcal{A}] & \text{(Protocol)} \\
A & ::= & r\,[\mathcal{M}] & \text{(Agent Role)} \\
M & ::= & \texttt{method}(\phi^{(k)}) = op & \text{(Method)} \\
op & ::= & \alpha & \text{(Action)} \\
 & \mid & op_1\ \texttt{then}\ op_2 & \text{(Sequence)} \\
 & \mid & op_1\ \texttt{or}\ op_2 & \text{(Choice)} \\
 & \mid & op_1\ \texttt{par}\ op_2 & \text{(Parallel)} \\
 & \mid & \texttt{waitfor}\ op_1\ \texttt{timeout}\ op_2 & \text{(Iteration)} \\
 & \mid & \texttt{call}(\phi^{(k)}) & \text{(Recursion)} \\
\alpha & ::= & \epsilon & \text{(No Action)} \\
 & \mid & v = p(\phi^{(k)}) & \text{(Decision)} \\
 & \mid & M \texttt{ => agent}(\phi^1,\ \phi^2) & \text{(Send)} \\
 & \mid & M \texttt{ <= agent}(\phi^1,\ \phi^2) & \text{(Receive)} \\
M & ::= & \rho(\phi^{(k)}) & \text{(Performative)} \\
\phi & ::= & \_ \mid a \mid r \mid c \mid v & \text{(Terms)}
\end{array}
$$

Figure 1: MAP Abstract Syntax [17]

A protocol written with the MAP language defines the way agents communicate with each other. Each kind of agent is described by an agent role which can contain different methods with input parameters (the initial method for each agent has no input parameters). Each agent has a fixed role for the duration of the protocol. The protocol describes who sends and receives which kind of message. Also logical operations and procedures are allowed. Those procedures allow to evaluate values for messages or to check constraints, where a failure in the procedure would cause backtracking in the protocol.

There are some papers about MAP; an introduction to MAP is given in [15] and [16], [17] and [18] are describing MAP in connection with MagentA and model checking, where [17] is focusing on model checking and [18] is using a view on e-science to introduce MAP and MagentA.

We will describe the before mentioned MagentA Platform briefly in the next section.

2.5 MagentA Platform

The MagentA (Multi-agent architecture) platform is using web services as agents to allow experiments with Multi Agent Systems.

"A web service is viewed as an abstract notion that must be implemented by a concrete agent. The agent is a concrete entity (a piece of software) that sends and receives messages, while the service is the set of functionality that is provided." [19]

A nice introduction to web services can be found on the SUN JAVA webpage [20].

The four main tasks of MagentA are shown in the following picture.

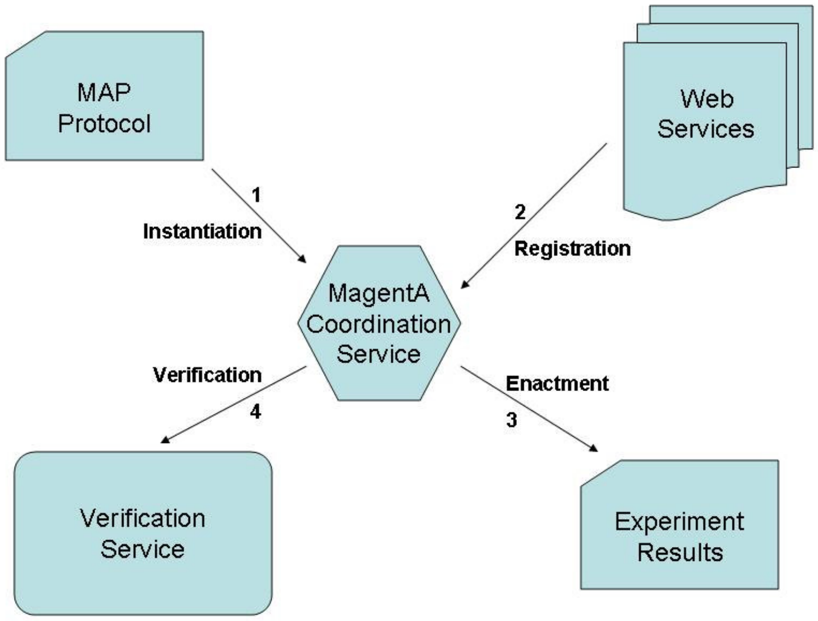

Figure 2: MagentA Platform Overview [18]

The core of the MagentA platform is the coordination service, which takes a protocol written in the MAP language to coordinate the web services, while the web services are implemented in the JAVA language.

Each role defined in the MAP corresponds to one or more web services.

After instantiation every web service which will be used in the Multi Agent system has to be registered with the MagentA platform. Of course for every role in the protocol at least one service has to be assigned to make it work properly. During the registering process an agent is automatically generated within the MagentA service to act as a proxy for the web service. After all web services have been assigned it is possible to enact the protocol. It finishes when either all steps of the protocol are enacted or the protocol fails (either the protocol itself or an assigned web service).

"MagentA has been implemented using JWSDP and utilizes the XML representation of MAP, and the web service WSDL and SOAP protocols." [18]

Another nice feature of MagentA is that it can produce WSDL specifications for roles defined in a MAP protocol, which then can be used to implement the web service. It also produces a Promela file, which can be used for model checking, a process which is described in the following section.

2.6 Model Checking

This section describes the process of model checking and its key features.

"Model checking is an automatic technique for verifying finite state concurrent systems."[21]

But how is it working?

2.6.1 The model checking process

To model check a certain system consists of three tasks which have to be done during the process:

Modeling

The first thing to do is to convert the design of the system into a formalism that can be used by a model checking tool. In this project the modeling is based on the output of the MagentA platform according to the PROMELA (PROcess MEta Language) formalism for the Spin Model Checker [4] and other models on different abstraction levels. A very good introduction and manual for the Spin Model Checker is written by Gerard J. Holzmann. [22]

Specification

The second step is to specify the properties that the design must satisfy. This specification is commonly done in LTL (Linear Temporal Logic), which allows defining how the behavior of the system evolves over time. A key issue for this specification is completeness. Model checking may be able to automatically check a specification, but if a specification is not complete the whole model checking process is in vain.

Verification

The verification of the specification is automatically done by the model checker. Nevertheless the outcome has to be interpreted by a human being. In case of a negative result, an error trace is given, which is helpful as a counterexample for finding the problem in the model.

The Spin Model Checker can also be used to simulate the behavior of the system and gain insight in the message passing procedure.

2.6.2 Automaton and Linear Temporal Logic

The Spin Model checker generates a finite state automaton for every thread defined in the Promela code.

"A finite state automaton is tuple (S,s_0,L,T,F), where

S is a set of states,

s_0 is a distinguished initial state , $s_0 \in S$,

L is a finite set of labels,

T is a set of transitions, $T \subseteq (SxLxS)$, and

F is a set of final states, $F \subseteq S$."[22]

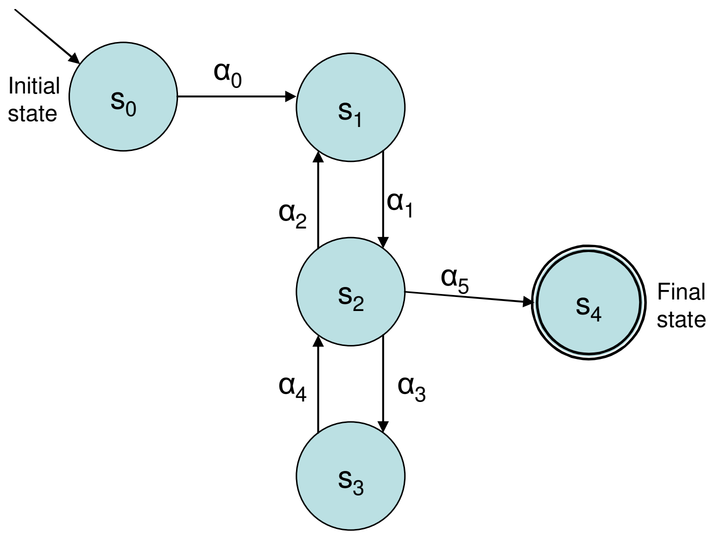

Figure 3: A simple Finite State Automaton[22]

Linear Temporal Logic is defined on such automatons. It allows the formalization of properties during a run, where a run is an ordered set of transitions.

It captures temporal event properties, which means that it is possible to formalize that after a state ***a*** is reached another state ***b*** will be reached at some point (***a*** -> <> ***b***), where ***a*** and ***b*** are state formulae like:

a: "the value of x is odd" and ***b***: "the value of x is 13".

The following table shows frequently used LTL formulae:

Formula	Pronounced	Type/Template
[]p	Always p	Invariance
<>p	Eventually p	Guarantee
p -> <>q	p implies eventually q	Response
p -> q U r	P implies q until r	Precedence
[]<> p	Always eventually p	Recurrence (Progress)
<>[]p	Eventually always p	Stability (Non-progress)
<>p-><>q	Eventuall p implies eventually q	Correlation

Figure 4: Frequently Used LTL Formulae[22]

2.6.3 The Verification process

The Spin Model Checker uses search algorithms (like depth-first or breadth-first search) on the system state space to verify LTL specifications. A system state is defined by the local states of all processes, the values of global variables and the content of the message channels. During verification it searches through this system state space and tries to find counterexamples of the specification given. If it cannot find such a counterexample then the specification is sound otherwise a counterexample is found and can be used to detect the problem is the system model. To detect such a counterexample the Spin Model Checker uses a so-called never claim. A never claim is a thread that should never be able to terminate. An example for the never claim thread for [] p is given in the next figure:

```
never {
T0_init:
    if
    :: (!p) -> goto accept_all
    :: (1) -> goto T0_init
    fi;
accept_all:
    skip
}
```

Figure 5: The never claim for []p

The never claim process is executed at every step of the system. In the example claim the process terminates as soon as !p becomes true. As soon as the never claim process terminates the Spin Model Checker has found a contradiction and will report this.

There are ways of detecting a wrong never claims which use different methods like the detection of acceptance cycles, but explaining this would go beyond the scope of this introduction.

The Spin Model Checker generates the never claim automatically for a given LTL specification, so that the never has not to be hand coded.

Another way to verify specification is the usage of assertions. Assertions can be put in the Promela code to assure that a certain variable satisfies a certain specification. For example assert(p==true). Whenever the system reaches this line of code it checks that the assertion is true. If the assertion is not true the system breaks and has found a counterexample.

This should conclude this chapter about the background and the literature review.

3 The Scenario

The scenario for this project is a three lane motorway with several cars on it. One of those cars is our model car which has to navigate on this motorway, avoiding other cars and trying to maintain a certain preferred speed by changing lanes and acceleration and braking. If a car comes closer in front it has to brake or change lanes to a free lane, while a closer coming car from behind will make it drive faster, but only until a certain maximum speed is reached. If no obstacles are in front or behind it will try to adjust its speed to the preferred one. It will also try to stay on the current lane if no process of lane departure is invoked.

The other cars have mainly the same target, but they have a different controller and sensor setting. They are not reacting to cars coming up from behind or cars on other lanes. So that they will occasionally cause accidents (Avoiding this problem was not implemented in this project to keep the controller for the dummy cars as simple as possible). But the main targets are also trying to stay on the lane, maintaining the preferred speed and changing lanes if its speed is below its preferred speed and also on random to make the environment more dynamic.

More details about the controllers and tasks of the cars are described later in this paper.

The actual motorway is a plain white surface, with three black straight lines representing the three lanes. The rather unusual design of the motorway surface (black line on white ground) is chosen because of the color sensitivity of the Infra Red Distance sensors and is inspired by another Webots Project called “Rover”, which is included in the Webots distribution [1]. The following picture shows the motorway.

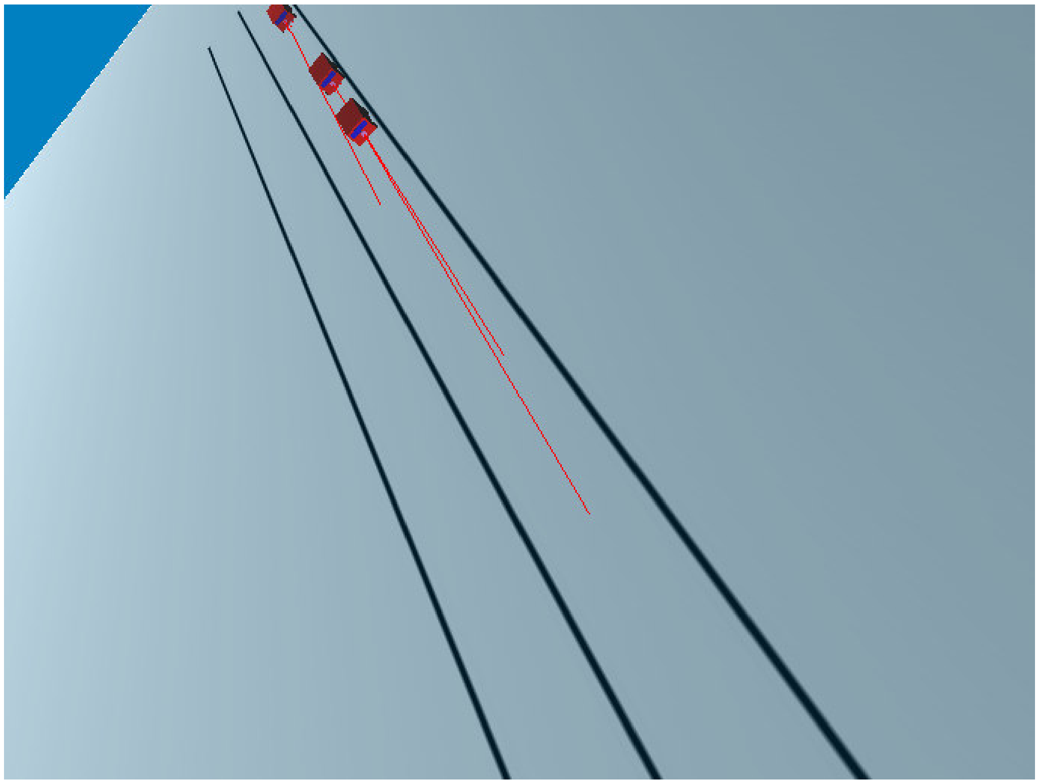

Figure 6: The motorway scenario

4 The Modeling of the Cars

This section covers how the cars of the scenario and the model car for our experiment were modeled in Webots, with their outward appearance and their sensors.

The first part describes the common design of both types of cars, the dummy cars and the model car. The second part describes the specific features of the dummy cars with their sensors and the last part the specific features of the model car.

4.1 The physical features of the cars

This section will describe briefly how the physical features of the cars where developed in Webots. For further details I refer to the Webots User Guide [23] and the Webots Reference Manuel [24].

The details of implementation are left out, but the interested reader will have no problem to figure that out with the two above mentioned manuals.

A fairly simple design was chosen for the cars. They consist of a cuboid of the size 0.1m x 0.045m x 0.032m (length x breadth x height) as the car body and small cylinders with the radius of 0.006m for the wheels. Those proportions are downscaled with factor 40 from the specifications of a real car; in this case a Mercedes SLK 200 Kompressor (Original size: 4.089m x 1.777m x 1.296m with wheels of 24cm radian). [25] The size of the wheels is important as the speed of the car is defined by wheel rotations. The speed value is defined as rad/s. For example a speed of 100 means that the wheel rotates approximately 16 times per second (In real life this would correspond to a speed of approximately 50 m/s or 180 km/h which is about right as a maximum speed on a motorway). The steering of the car is accomplished by assuming that there are 2 wheels in the middle of the car with a customized axle length; in this case 0.04m. Those wheels can be turned independently, so that the car will turn left, if the right wheel rotates faster and turn right if the left wheel rotates faster.

The outside of the car is enclosed in another cuboid which defines the physical boundary of the car. This cuboid cover is used to determine whether the car can still move or is blocked by another obstacle, and for the distance evaluation of the infra red sensors.

Figure 7 shows a picture of the dummy car. As you can see for orientation and a nicer appearance a front window was also modelled.

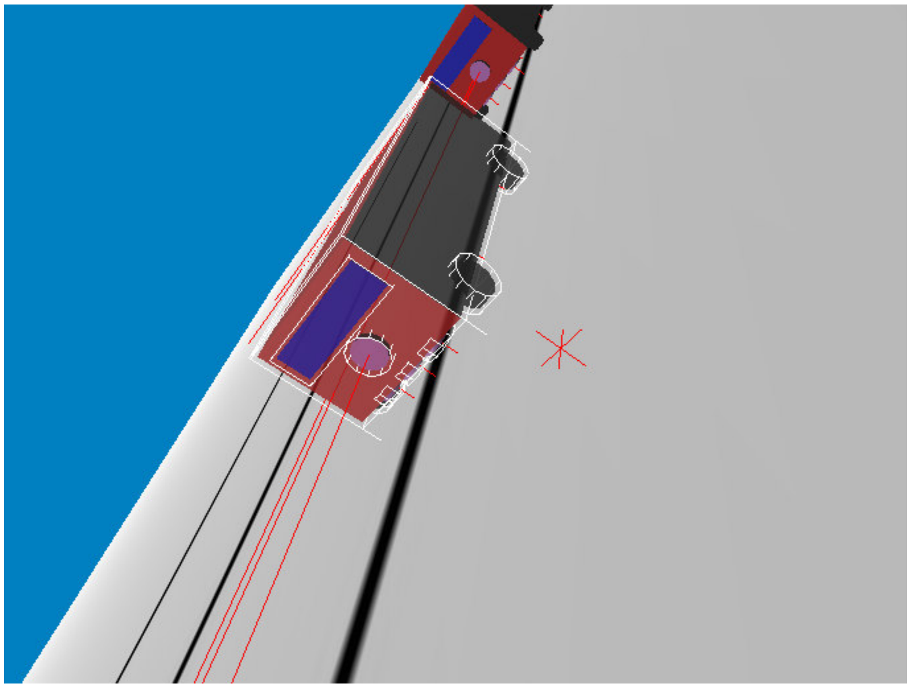

Figure 7: The dummy car

4.2 The sensor equipment of the dummy cars

In this section the sensor equipment of the dummy cars is explained.

All dummy cars have the same sensor equipment which consists of ten Infra Red Distance (IRD) sensors. Those IRD sensors have another property; they are color sensitive. They perceive lighter and red objects better then dark or black ones, which is used for the lane detection. The dummy cars have nine IRD sensors on the bottom for lane detection and one in front for distance measurements to obstacles in front of it. The front distance sensor can detect obstacles within one meter, which is about 10 car length or 40 meter in real life. The sensor will return a maximum value of 1000 if no obstacle is in its reach or an obstacle just entered its reach and null if the obstacle is touching it. Nevertheless the sensor has a certain noise, which is less than 10% depending on how close the obstacle is. The noise is null at distance zero and about 10% if the obstacle is at the maximum reach of the sensor.

The return value of the distance sensor is therefore approximately the distance in millimeter.

The nine sensors on the bottom are organized in three rows with three sensors each. This should help to detect the orientation of the car to the lanes. More details about the lane detection process is given in Chapter 5 The Controller for the Dummy Cars.

4.3 The sensor equipment of the model car

The sensor equipment for the model car is somewhat different to the one the dummy cars have. The details will be described in this section.

As the dummy cars the model car has a front distance sensor with the same properties as the dummy cars to detect obstacles in its way, but it also has a distance sensor of the same kind in the back to detect upcoming cars.

For lane departure warning the model car uses just one row of three IRD sensors on the bottom in the middle of the car, as its function is somewhat different to the function in the dummy cars. In the dummy cars those sensors are used to keep the car on lane, while here they are just used to warn if the car is leaving the current lane, so three is enough to detect the departure of the car from the current lane.

There is another set of four IRD sensors in the model car. Those sensors are on the side of the model car, two on the left and two on the right side. On each side one sensor is at the front and one in the back. As in our scenario each car has the same length those sensors can be used to detect whether there is a car on the adjacent lane to the left or right, because no car can be missed by those sensors. If the cars would not be of equal length a car might be just between those sensors and therefore important information is lost. This is a simplified idea of checking the possibility to overtake as it is still possible that a slower car on the adjacent lane in front of the model car will intervene during the overtaking process or a faster car further back on the adjacent lane. But for this project we stick to this simplified overtaking control idea.

Also a camera is installed in the front window and a GPS device on the bottom of the car for further research, but for this project those two devices were disregarded.

5 The Controller for the Dummy Cars

The controller of the dummy car has to take care of the tasks the dummy cars should carry out. Those are mainly two: The steering to stay on the current lane and braking/accelerating according to the obstacles in front of it. Another small task is to decide whether or not to change the lane.

After every simulation cycle which lasts 60 milliseconds the values of all sensors are collected and the controller is run to calculate changes in the behavior of the car.

The controller is implemented in the JAVA language and is the same for all dummy cars. The only differences between those controllers are the different preferred speeds which are set.

5.1 Accelerating and Braking

The process of accelerating and braking is dependant on the values, which are provided by the distance sensor at the front of the car. The dummy car will accelerate if there is no obstacle in front of it (value of the front sensor about 1000), the obstacle in front of it is still far away (value of the front sensor above 700) or if the obstacle is not dangerously close (value above 200) and the obstacle is getting further away (last value smaller than current value). In these cases the speed is increased by 1 as long as the current speed is below the preferred speed.

The car is neither accelerating nor braking if the obstacle is in a medium range and not getting further away (value between 400 and 700 and the current value is smaller or equal to the last value).

The speed of braking is also dependant on the distance of the nearest obstacle in front. A slight breaking by decreasing the speed by 1 is initiated in the case that there is an object between 20cm and 40cm in front (value between 200 and 400). The speed is decreased by 2 if it is even closer (value between 100 and 200). And the strongest braking by decreasing the speed value by 3 is for all cases where the distance is even closer than that (value below 100). Of course no negative values of speed are allowed (as driving backwards is not allowed on motorways).

5.2 Steering and Staying on the Current Lane

The steering is done in the dummy cars by evaluating the sensor data from the IRD sensors on the bottom of the car. There are nine of those organized as showed in the following picture:

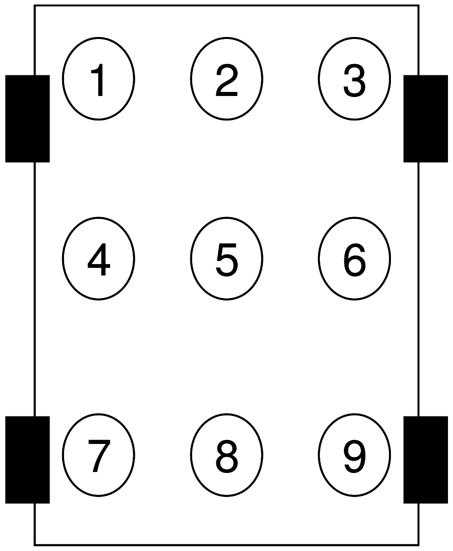

Figure 8: IRD sensors for lane detection

The first thing which is done in the controller is the evaluation of each row of sensors on its own to decide where the lane mark is. So each row will evaluate to a value of left, middle, right or not decidable. So the decision on the steering has to be made on a combination of values out of $4^3=64$ possible combinations. In the controller there are two tables which have 64 entries each; one table for the speed of the left wheel and one table for the speed of the right wheel. Not all of those entries should be explained here, but the general idea is to identify the general trend of the car and to steer it in the opposite direction (For Example: If the first two rows detect that the lane mark is on the right side of the car and the last row detects that the lane is in the middle, then this means that the dummy car is about to leave the lane to the left side, so it will steer right to stay on the lane). If only the front IRD sensors detect the lane mark then this is a sign for a changed lane, so it will steer to get on this new lane. (Again an example: If only the front row detects the lane mark on the left side, then it means that the dummy car is approaching from right and therefore has to steer a bit to the right to avoid over steering and make it possible to get on the new lane.) The following pictures show some examples of the steering decissions.

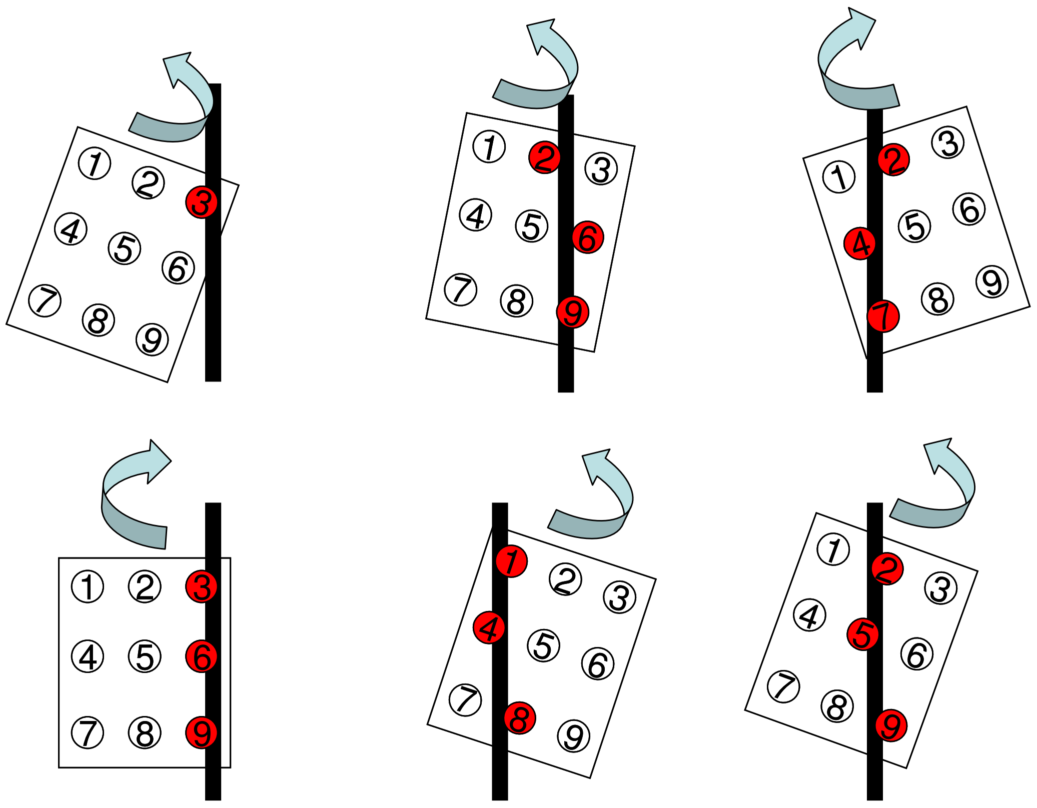

Figure 9: Examples for steering decisions

There are examples where the steering indicated by this simple lookup table is not right, but the ones in the table are the most probable ones, which is good enough. As the steering decision is reviewed in every simulation step a wrong decision at one point will be revised at one of the next steps.

The whole process of changing the lanes needs some steering which is a bit different to the behavior explained here. It will be described in the next section.

5.3 Changing Lanes

Changing lanes is a somewhat different process than staying on the lane so this is described in some details here. But before explaining the details of this process the decision making process is described to show when this process of changing lanes is initiated.

5.3.1 When to change the lane

A dummy car changes its lane at each simulation step with a certain probability. If another car in front of the dummy car is forcing the dummy car to drive slower than its preferred speed, then this probability is increased with every simulation step until the dummy car can either accelerate again because the car in front is getting further away or the probability decision makes the dummy car change its lane.

5.3.2 How to change the lane

After the decision is made that the dummy car should change its lane, the dummy will steer in the direction of the new lane. Of course only valid lanes are considered (it is not possible to change to the left from the first lane or to the right from the third lane).

It then goes through different phases, the first one is just steering in the direction until the sensors of the other side of the car show that it is about to leave the current lane. Then it continues to drive in that direction until the sensors detect that it is approaching the new lane and finally when it is fully on the new lane (meaning that the middle sensor detect the lane mark) it is changing back to the Stay-On-Lane modus.

This should conclude the chapter about the dummy controller. This controller is really simple but it is good enough to generate the appropriate behavior.

6 The System Architecture

From this chapter onwards the focus changes from the environment towards the actual model car with its multi agent system and the according protocol. In the first section of this chapter the architecture of the multi agent system will be described. The second section will cover the whole system as it used here with the connection of Webots and MagentA, which can be seen as an implementation of the abstract architecture described in the first section.

6.1 The Multi Agent System

This section will describe what kinds of agents are involved, what tasks they fulfill and what information they use to accomplish their task. The description will not focus on the agent as a web service but on an abstract point of view. A more implementation focused description of the system will be provided in the later section of this chapter.

6.1.1 System Overview

The multi agent system of the model car consists of 15 agents on three different levels.

The lowest level is the sensor level; it contains all the IRD sensors in the system. There are the three sensors on the bottom for the lane detection, the four sensors on the sides for a safe lane changing and the sensor in front and back for detecting obstacles. So in total this lower level consists of 9 sensors.

The middle level is the system level for the warning agents; those agents collect the data from the according sensors and use this data to evaluate whether there is problem to report to the driver agent or not. There is one agent for each side to warn about a car on the other lane, one agent each for the front and back sensor and one final agent for the lane departure warning. So there are five agents in this middle level.

On the top level there is only one agent, the driver agent. This agent is doing all the steering and accelerating and the according evaluations of the data which is contributed by the agents of the middle level.

The following picture shows the system with all agents and their dependencies.

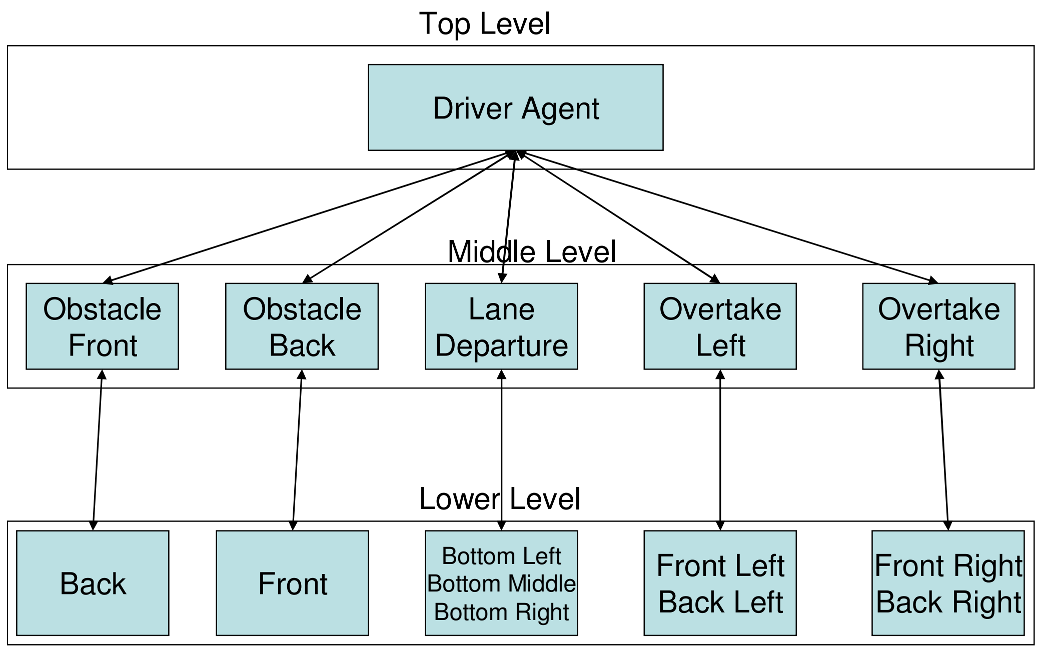

Figure 10: System Architecture

6.1.2 The Agents in More Detail

In this section all agents are described closer to show what tasks they perform and briefly describe how they perform their task with respect of the data provided.

The Lower Level Agents

The lower level agents are all just sensors. Their only task is to provide the data they collect from the environment to other agents. It may be asked why even use these agents if they are nothing else than sensors. It would be possible to pass the sensor data directly to the middle level, but as in real life those sensors have to be respected; they are included here to assure a full architecture model.

The Middle Level Agents

The middle level agents are a bit more complicated than the sensors. They use the data they get from the sensor agents of the lower level to perform their specific task.

Those middle level agents can be grouped in three different groups of agents:

i) Lane Departure Control

There is only one agent, which is performing the lane departure control. It takes the sensor data from the bottom IRD sensors and uses them to detect where the lane mark is and to warn the driver agent if it detects that the lane mark is to the right or to the left of the car, which would indicate a lane departure. The warning to the driver agent includes the direction in which the car is about to leave the lane, so that the driver agent can steer in the appropriate direction.

ii) Overtake Control

The overtake control agents have the task to secure a safe lane change.

There is one for a change to the right (using the data from the sensor agents on the right side of the car) and one for a change to the left (using the data from the sensor agents on the left side of the car).

It checks whether one of the sensors on the side detects an obstacle on the other lane. If it is so, it will warn the driver agent that it is not safe to change the lane to the next lane in that direction.

iii) Obstacle Detection

The two agents responsible for the detection of obstacles on the same lane just use the value of the corresponding sensor agent. But it keeps the last value as well to evaluate whether the object is coming closer or getting further away. The warning which is given to the driver agent consists of the information about the distance of the nearest object and the tendency, whether it gets further away or is coming closer, to allow the driver to react accordingly.

The Driver Agent

The driver agent is the most sophisticated agent. It is responsible for all actions the car is performing, meaning all steering and speed changing actions. To make rational choices for those actions it uses the information which is given by the agents of the middle level. Those agents are providing warnings to which the driver has to react accordingly. In the case of a lane departure warning the driver agent has to steer to stay on the lane. If a car is coming closer in front the driver agent has to brake or to decide to change the lane to avoid crashing into this car. If the driver agent is deciding whether it is going to change the lane or not it has to evaluate whether there is another car on the other lane which will prevent the driver agent from changing the lane. Of course all those tasks are simple if the driver agent has enough time to evaluate all possibilities, but in a real scenario time is an issue. So in the case that all warnings are coming in at the same point of time, there has to be a way to assure that the driver agent can still perform its task and not get stuck. Here the protocol comes into play, how this problem is approached in this project will be described in the next chapter. But before the protocol design is discussed, the actual implementation of the architecture should be covered briefly.

6.2 The Webots and MagentA Implementation

After describing the abstract structure of the Multi Agent System in the model car, we will now describe some details about the actual implementation and especially about the connection to the Webots simulation software, which is used to test the protocol in a simulated scenario.

6.2.1 The connection between MagentA and Webots

The data which is sent between MagentA and Webots is transferred via the TCP/IP protocol and uses the socket technology of the JAVA language. The server socket is implemented in an external thread which is just listening for incoming data. At every point in time where data is needed from this thread the current value is read and used.

Every sensor agent in the MagentA implementation of the multi agent system has its own server socket which is receiving the actual sensor data from the Webots simulation software. On the startup of the Webots simulation the car controller initializes nine threads, one for each IRD sensor. Those threads then establish a connection to the according server socket threads on the MagentA multi agent system. The driver agent on the MagentA side has two threads, which, after it is assured that the connection with Webots is established and that the simulation is started, establish a connection with the car controller. One of these threads will transmit the current speed value and the second thread will transmit the different percentages of speed for the left and the right wheel (the steering). On the other side two server sockets are implemented in two threads to receive those values and make them available to the car controller. The thread for the current speed is just a proxy to provide the value to the car controller, but the other thread also has the function to adjust the steering if no new steering value combination is coming. So if a value combination of for example 90% and 100% was sent and then in the next step no new steering combination is coming, then it will change the values to 91% and 100% until the values are both at 100% again. So every steering action is just a steering impulse. This assures that the car is not steering all the time and bouncing from one border of the lane to the other, but has the chance to stay on the lane stable.

The following picture shows the connection between the agents in MagentA and the controller in Webots:

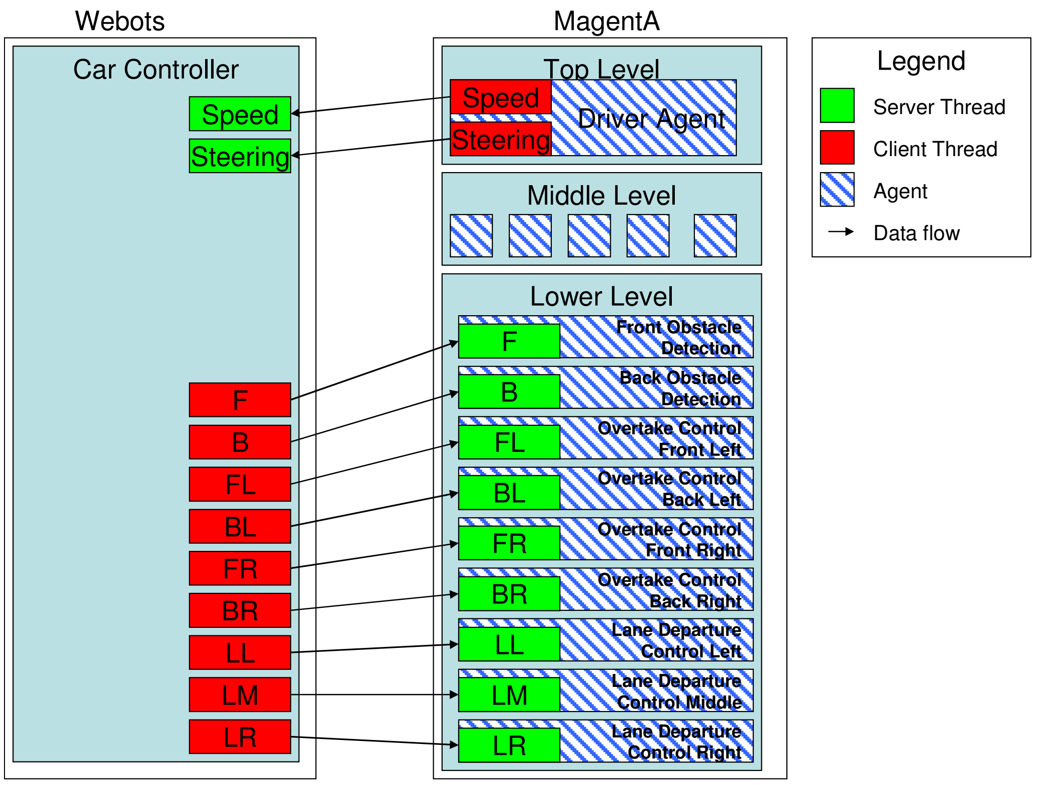

Figure 11: Data Connection between MagentA and Webots

6.2.2 The car controller in Webots

The car controller for the model car is only a proxy for the multi agent system on MagentA. All sensory data is transmitted to the multi agent system in every step of the simulation and for all the actions, which are represented by setting the current speed of each wheel, it will use the information of its two receiver threads and calculate the speed of each wheel with this information.

6.2.3 The MagentA implementation

Every agent is implemented as a web service in the JAVA language. All the procedures which are called in the protocol are methods in the web service. The sensor agents all have a separate thread, which receives the sensor data from the Webots controller. The value which is stored and updated in this thread is used by the agent in the transmission to other agents. The agents in the middle level only have methods to evaluate the warnings which will be sent to the driver agent, while the driver agent itself has two separate threads to transmit the action it has decided on to the Webots controller. Some more details about the methods which are implemented in the single agents will be given in the next chapter, which describes the protocol design.

7 Designing the Protocol

In this chapter the design of the protocol will be described using a bottom-up structure. At first the protocol definition for the sensor agents will be described with a special section for one sensor agent which is used for the initialization of the testing environment. After that the middle level agents will be covered and described in connection with the other agents. And at last the driver agent protocol definitions will be described, which is the most complicated and important bit of the protocol. This chapter will describe the protocol as abstract as possible, but will give sections of the MAP definition to illustrate the implementation.

(Remark: To make the MAP sections in this chapter more readable, the naming of agents and variables are following the recommendations in other MAP papers. So variable names will have $ as a prefix, role names a % and agent names a !.)

7.1 The sensor agents

The task of the sensor is really easy; whenever another agent asks for information this agent will send the current value of the according sensor to this agent. It is merely a proxy for the sensor. In real life this agent would be the actual sensor, so it makes sense to define this agent as it is.

In the protocol there is a loop, where at the beginning the newest value is read by a procedure and then a waitfor-loop is waiting for an incoming request for data. If this request occurs the agent has to send the value to the requesting agent; otherwise it will jump back to the beginning of the loop and read the newest value before waiting again for a request.

The MAP definition for a sensor would look like that:

```
%sensor{
Method() =
    $val = getvalue() then
    Waitfor
            (( inform(sendval) <= agent($aname, %role) then
            inform(retval, $val) => agent($aname,%role) then
            Call())
    Timeout
            Call()
}
```

Figure 12: Map definition for sensor agents

In this bit of the protocol the %sensor represents the actual sensor like IRD sensor front. The %role is depending on the actual sensor as every sensor is sending to different agent roles as showed in the system structure. The procedure getvalue() is receiving the return value from the separate thread, which represents the newest sensor data from Webots. Each sensor has its own agent role in this protocol, because each agent uses a different port for the connection to Webots. The easiest way to do that is with different web services and therefore different agent roles.

7.2 Sensor agent for Initialization

To assure that the Webots simulation software is up and running before establishing a connection to the server threads in the Webots controller, one sensor is used to trigger the handshake. The driver agent which will eventually establish the connection to the Webots server threads is requesting information from this special sensor and evaluating whether it is still in its initial state, which would return the value -1. Every sensor is initialized with this value which will never be transmitted from Webots, so as soon as it changes, the system can be sure that Webots is up and running. Therefore the driver agent will change from the initialization to its normal behavior after receiving the first real value. The sensor is then informed that the system is up and running and the sensor can change to its normal sensor behavior as defined above.

The MAP definition for this special sensor looks like this:

```
%specialsensor{
Method() =
$val = getvalue() then
    Waitfor
            (( inform(sendval) <= agent($aname, %driver) then
            inform(retval, $val) => agent($aname,%rdriver) then
            Call())
            Or
            (inform(isinit) <= agent (_,%driver) then
            Call(normal))
    Timeout
            Call()

Method(normal)=
$val = getvalue() then
    Waitfor
            (inform(sendval) <= agent($aname, %role) then
            inform(retval, $val) => agent($aname,%role) then
            Call(normal)
    Timeout
            Call(normal)
}
```

Figure 13: Map definition for the Handshake agent

It does not matter which sensor is chosen for this initialization handshake.

7.3 The back/front obstacle detection agent

The agents for obstacle detection in front and back have the same protocol design, but they differ in the implementation of the procedures.

They request data from the according sensor agent and use the return values for the evaluation in their procedure. If the driver has asked for information the evaluation values are forwarded to the driver.

The MAP definition for both of them is similar, so the definition for the back obstacle detection would look similar with the according changes for agent roles.

```
%frontdetection{
Method() =
    Inform(sendval) => agent(_,%frontsensor) then
    Waitfor
            ( inform(retval,$val) <= agent(_,%frontsensor) then
            $evalval = fronteval($val) then
            Waitfor
                    ( inform(sendval) <= agent($aname,%driver) then
                    inform(retval,$evalval) => agent($aname,%driver))
            Timeout
                    Call())
    Timeout
            Call()
}
```

Figure 14: MAP definition for obstacle detection agents

The procedure fronteval produces a return value which is the coding for obstacles in front. It is a code for the distance of the obstacle and the tendency, whether it is coming closer or getting further away. It stores the last value of the sensor in a temporary field and uses that value together with the new value to evaluate the code.

7.4 The lane departure control agent

To evaluate a lane departure warning this agent needs the values of three sensors. So the first step in the protocol design is to send those requests to the according sensors. It then waits for the return values to invoke the evaluation procedure. If the driver agent is requesting the evaluation value, which is either a warning with the direction of the lane departure or a non-warning message, the lane departure control agent is transmitting the message to the driver.

The straightforward MAP definition is shown below:

```
%lanedeparture{
Method() =
    inform(sendval) => agent(_,%laneleft) then
    inform(sendval) => agent(_,%lanemiddle) then
    inform(sendval) => agent(_,%laneright) then
    Waitfor
            ( inform(retval,$lval) <= agent(_,%laneleft) then
            inform(retval,$mval) <= agent(_,%flanemiddle) then
            inform(retval,$rval) <= agent(_,%frontright) then
            $evalval = depwarning($lval,$mval,$rval) then
            Waitfor
                    ( inform(sendval) <= agent($aname,%driver) then
                    inform(retval,$evalval) => agent($aname,%driver))
            Timeout
                    Call())
    Timeout
            Call()
}
```

Figure 15: MAP definition for the lane departure agent

The procedure depwarning is using the color sensitivity of the IRD sensors to detect the lane markings. If the marking is under one of the sensors, this sensor will return a higher value than the other agents. If no marking can be detected then no warning is produced.

7.5 The overtaking control agents

In order to assure that a "safe"[1] overtaking process can be invoked these agents need to request information from two other agents, which is the first step in the protocol design. Then the familiar structure of the middle level agents appears. After receiving these values a procedure is called to evaluate the warning message. If the driver agent requests this message, it is sent to him; otherwise a new evaluation cycle starts.

[1] The limitiations of this overtaking control are discussed in the chapter 4.3 The sensor equipment of the model car.

```
%otcontrolleft{
Method() =
    inform(sendval) => agent(_,%frontleft) then
    inform(sendval) => agent(_,%backleft) then
Waitfor
            ( inform(retval,$fval) <= agent(_,%frontleft) then
            inform(retval,$bval) <= agent(_,%backleft) then
            $evalval = otwarning($fval,bval) then
            Waitfor
                    ( inform(sendval) <= agent($aname,%driver) then
                    inform(retval,$evalval) => agent($aname,%driver))
            Timeout
                    Call())
    Timeout
            Call()
}
```

Figure 16: The MAP definition for the overtaking control agents

The procedure otwarning is using the two values from the IRD sensors to check whether there is a car on the other lane. If one or both of the values is too small, then a warning message is produced otherwise a non-warning message.

7.6 Remark for middle level agents

There is one special property within the middle level agents. The waitfor loop for the receiving of the driver request should be shorter than the outer waitfor loop. This assures that the agent can evaluate the newest values and is not staying in the request waitfor loop for too long.

7.7 The driver agent

Now in the last section of this chapter the protocol design for the most sophisticated agent is described.

The first method in this protocol part is the initialization which was described from the sensor point of view earlier in this chapter. After this initialization handshake the driver agent establishes the connection to the Webots controller and goes into the main loop which is also the loop for staying on the current lane.

7.7.1 The handshake

The handshake method is the counterpart to before mentioned handshake method in the sensor agent that is specified for the handshake. The driver agent requests information from one special sensor and checks whether it is a valid value. If so, it establishes the connection to Webots threads which control the speed and steering of the car in the connect procedure. Afterwards it calls the keeplane method setting the initial lane to 1. If the value is still not a valid value it loops until this constraint is satisfied. The procedure checkval will fail in this case.

7.7.2 The keeplane method

The keeplane method of the driver agent is the part where the most important decisions have to be made. The idea was to introduce a kind of priority system, which uses the given information in a certain order. At first the driver agent requests information from the three main middle level agents. In a waitfor loop the driver agent checks whether information is available and uses this information for steering and accelerating/braking. At first it checks whether information from the front obstacle detection is available. If so, it requests additional information from the overtaking control agents. The waitfor loop which is waiting for this information is a short one. If the information is not coming back fast the information is ignored. The procedure uses the information from the front obstacle detection agent and the additional information (if available) to decide on the braking or overtaking. If no overtaking process is invoked, then the procedure fails after the braking information is transmitted to Webots in order to allow additional steering.

Next in priority is the information from the lane departure control agent. If no overtaking process is invoked or no information from the front obstacle detection agent is available this information is received and used in the steer procedure to steer in the appropriate direction. This procedure fails if no steering is needed, so that the lowest priority warning can be checked.

The lowest priority warning comes from the back obstacle detection agent. It is only checked if the other information is not available or no major actions were performed. In this case the accelerate procedure accelerates according to the information given.

If no information is available at all, the default action is to just accelerate the car by one (if not yet at the preferred speed). This assures that a standing car will accelerate and provoke warning messages at some point. It also ensures that the car is able to reach its preferred speed if no other actions are needed.

7.7.3 The changelane method

The changelane method is used for the controlled change of lanes. It just requests information from the lanedeparture agent. This information is used in the checkot procedure to send the steering information to the model car and evaluate when the overtaking process is finished. In this case it uses the parameter given to the changelane method (the current lane and the direction of change) to return the new lane, which is then used for the call of the keeplane method.

Until the overtaking process is finished the procedure checkot fails after doing its evaluation and sending the steering information to stay in this method.

7.7.4 The MAP definition

```
%driver{
    Method() =
        inform(sendval) => agent(_,%sensor) then
    Waitfor
                (( inform(retval,$val) <= agent(_,%sensor) then
                Checkval($val) then
                Connect() then
                Call(keeplane, 1))
                Or Call()
        Timeout
                Call()

    Method(keeplane, $curlane)=
        inform(sendval) => agent(_,%lanedeparture) then
        inform(sendval) => agent(_,%frontdetection) then
        inform(sendval) => agent(_,%backdetection) then
        Waitfor
                (( inform(retval,$fval) <= agent(_,%frontdetection) then
                inform(sendval) => agent(_,%otcontrolleft) then
                inform(sendval) => agent(_,%otcontrolright) then
                Waitfor
                  ( inform(retval,$leftval) <= agent(_,%otcontrolleft) then
                  inform(retval,$rightval) <= agent(_,%otcontrolright) then
                  $dir = fronteval($fval,$leftval,$rightval) then
                  call(changelane,$curlane,$dir))
                Timeout
                  $dir = Fronteval($fval,-1,-1)
                Or(
                  (( inform(retval,$lval) <= agent(_,%backdetection) then
                  Steer($lval) then
                  Call(keeplane,$curlane))
                  Or(
                        (( inform(retval,$bval) <= agent(_,%backdetection) then
                        Accelerate($bval) then
                        Call(keeplane,$curlane))
                        Or accelerate(1)
        Timeout
                Call(keeplane,$curlane)

    Method(changelane,$curlane,$direction)=
        inform(sendval) => agent(_,%lanedeparture) then
        Waitfor
                (( inform(retval,$val) <= agent(_,%lanedeparture) then
                $newlane = checkot($val,$curlane,$direction) then
                Call(keeplane,$newlane))
                Or Call(changelane,$curlane,$direction)
        Timeout
                Call(changelane,$curlane,$direction)
}
```

Figure 17: MAP definition for the driver agent

8 Evaluation of the Protocol in Tests and Simulation

8.1 Testing the Protocol in Simulation

After the protocol is designed we want to describe the results of the tests with the Webots simulation software. As the behavior of the dummy cars is probabilistic, the best way to do testing is to run many tests and analyze the behavior. The only parameter which can be changed is the initial setup of the environment. In all cases four dummy cars are used which have different preferred speeds(40,60,80 and 100) and are placed on different spaces on the first lane. During the simulation those cars brake, accelerate and change lanes. How successful the protocol is, can be evaluated in looking how often the model car is behaving inappropriate in the interaction with the environment and the other cars.

A series of tests showed that initially the model car is behaving appropriate: it stays on the lane and avoids other cars. Even the first overtaking processes take place. But later on different inappropriate behaviors appear. Sometimes the model car misses the next lane and keeps steering in the same direction. This is especially bad when the model car is leaving the motorway completely. Another problem is that the model car sometimes leaves the lane without invoking an overtaking process. The longer the simulation runs the more probable it gets that the model car misbehaves.

As a result of the testing in simulation it has to be stated that the protocol was not as successful as hoped. So the next very important step is to find the flaws of this protocol.

The problem of finding the difficulties in the protocol is hard, because the log files of the agents and the outputs of the Webots simulation do not allow a closer evaluation. There are 15 agents and therefore 15 log files which have to be checked and compared individually. But how is to possible to tackle this problem? In this project we chose the Spin model checker to do a closer examination of this protocol. This should allow discovery of the reasons behind these misbehaviors.

8.2 Further Testing with the Aid of Model Checking

After the simulation showed that there exist problems within the protocol definition, those problems have to be found. This problem is tackled here with the help of the Spin Model Checker which allows simulating and verification of the protocol. But before this simulation or verification can be done, the protocol or the whole system has to be modeled.

Different models were used to find the problems in the protocol. The results will be discussed below, organized by the different kinds of models.

Not only will the results for the protocol be discussed, but also how this model can be used and the abstraction level which is used for this model. This discussion will help to see how the Spin Model Checker can be used for protocol evaluation in general as well as the results achieved in this special case.

It has to be mentioned that in all cases weak fairness was used, that means that if a thread has an executable command it will at some point execute this command. If this is not used, then in all verifications it will find faulty behaviors, which is just a loop inside one agent.

The Promela code for the models besides the MagentA generate coded are listed in the appendix.

8.2.1 The MagentA generated Promela Code

For a first model the Promela Code produced by the MagentA platform was tested for the usage of evaluation. As the Promela code is a direct translation from the MAP definition, it contains no information about the procedures. Also the non blocking character of the MAP implementation in MagentA is modeled, which increases complexity, because of the waitfor loops. A third point worth to be mentioned is, that this model is creating new threads for each call in the MAP definition. This increases the number of threads heavily, so that after a short while the maximum number of threads is reached, which stops the invocation prematurely.

All in all, the Promela code which is generated by MagentA platform is not a good model for complicated systems like this one. Another model has to be defined to find the flaws of the protocol.

8.2.2 A full model for simulation

In order to effectively simulate the multi agent system with the protocol, a different model was used. The Promela code is listed in the appendix(see Chapter 12.1). For every agent one thread is used, which has its own message channel. Instead of the non blocking behavior of MAP a blocking behavior is used, which reduces complexity in disregarding the variables which would normally keep track of the loop iteration. The message sending and receiving pattern can be directly modeled from the protocol design. But to assure that the model is close to the real system also the procedures which generate return values are modeled. The procedure which establishes the connection with Webots is disregarded as it adds no new information to the system messages. All other procedures produce a random binary value (0 or 1). This is enough to simulate the system with the influence of the procedures.

The methods which are defined in the protocol are modeled with jump labels and the calls with goto commands. This model is still very complex as it has a lot of different variables for all the agents. So the model checking process would still produce no good results as the state space is still too big. If you regard that every binary variable doubles the state space, only the 9 binary values for the sensor agents multiply the number of states by 512. There are also 15 agents in total which might be in different states each. If you assume that every agent has just 2 states one for receiving and one for sending messages, of course they have more, then only this system with those agents will have 32768 states. So only regarding those two simplifications will end in a system with more than 10,000,000 states. As every additional variable and state inside an agent will multiply this number it is easy to imagine that the real system has far more states. So this model can only be used for simulation purposes. But nevertheless the simulation of the system with this model might give us some insight in the problems of the protocol.

In the simulation with the Spin Model Checker it is possible to keep track of the message channels and global variables. After running the simulation for a while it is now possible to see one of the problems with this protocol. The idea of warning messages from agents was inspired by the hardware design of embedded controllers, where an electrical signal at a channel will represent a warning. This signal will only be there until the warning is no more present. But with messages in a multi agent system of this kind the message will stay in the message queue until it is received. This is no problem for the sensor agents, as they have a direct connection between receiving and sending messages. For every message that requests an answer it will send exactly one message with the newest value. Even for the middle level agents the problem with message queues is not present, as every agent gets one message for each request to the sensor agent and will send exactly one message for each request by the driver agent. But the problem becomes apparent if you look at the message channel for the driver agent. As it is not assured in the driver agent that every message passed through to the driver is received on short term, some messages stay in the queue too long, so that when they are finally received the information is no more valid. With every invocation of the keeplane method in the driver agent, he requests information from three middle level agents. But if, for example, the driver agent does not get the answer from the overtake control agents fast enough, those messages are nevertheless sent to him. The next time the front obstacle detection agents sends a warning the driver will receive the old value from the overtake control agents and the newer message gets stuck in the queue. The following figure illustrates this problem:

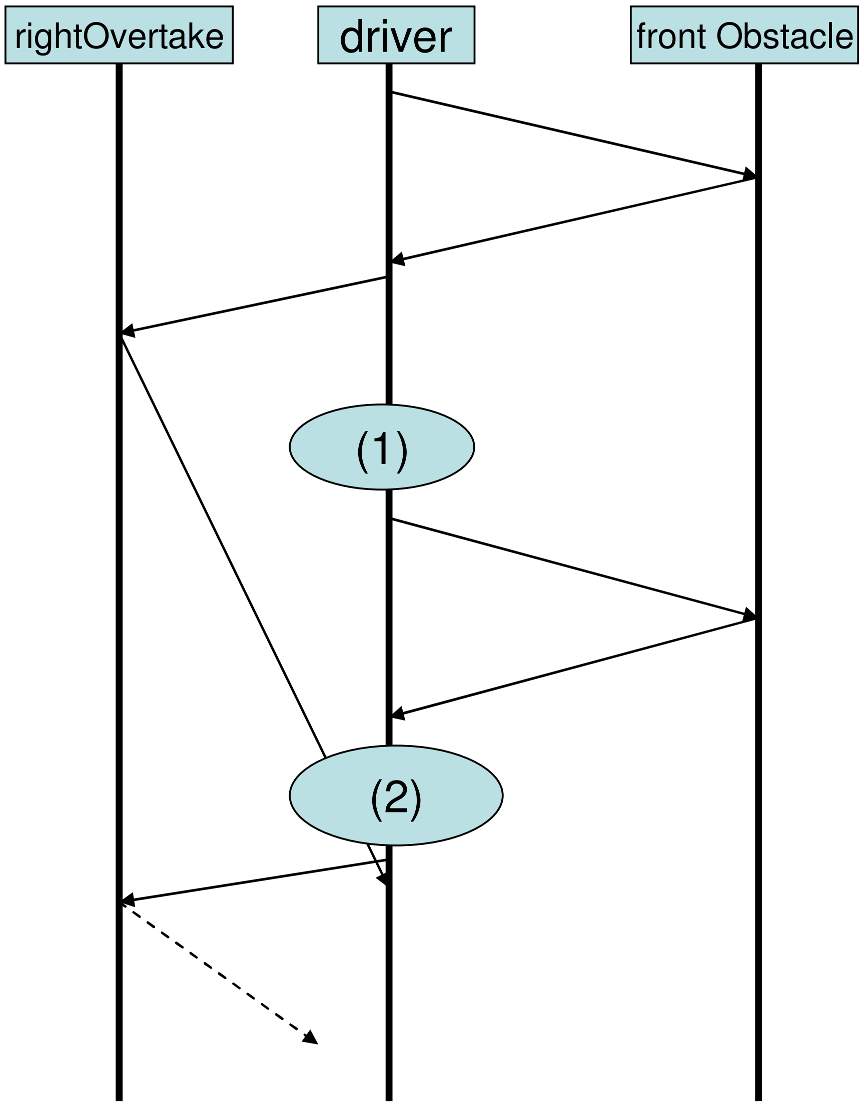

Figure 18: An example for faulty behavior

The picture is a simplified version of message flow, where only three agents are shown. The driver agent requests information from the agent that is responsible for the detection of obstacles in front. This agents sends a warning and therefore the driver requests information from the overtake control agent. The answer is not coming back fast enough and therefore in (1) the driver agent just brakes and requests new information form the obstacle detection agent. This agent is sending a warning again and after the driver agent requests information from the overtake control agent again. Meanwhile the overtake control agent has sent the requested information for the last request. This information is now received by the driver agent. If the time between (1) and (2) is too long, which would happen for example if the front obstacle detection agent is sending some non-warning messages before sending another warning, this information can be totally inappropriate.

Also one message from the overtaking control agent is still pending in the queue. As this can happen repeatedly the message queue fills up over time, so that eventually the message buffer is full; a behavior that is absolutely not intended. This faulty behavior can be the reason for the misbehavior of the model car. It might especially be responsible for the vulnerability over time, as the probability for wrong received warnings grows with simulation time.

It is not possible to show that the problems of the simulation is provoked by this fault in the design as no real data is used in this simulation, but it shows one possible reason for the faulty behavior. And as this problem was not seen in the initial design of the protocol, the Spin Model Checker simulation pinpointed this flaw. Besides the problem with the message queues the simulation showed the expected behavior, it reaches every possible reaction by the driver eventually and uses the right type of messages. This indicates that the design is not totally wrong; the problem would be solved if messages of the same type would be overwritten by a new message with new information. Of course this is not possible in the real MAP protocol, but it is possible to model such a system with the Spin Model Checker and the aid of shared message channels. So after this initial usage of the Spin Model Checker we already have a probable reason for the faulty behavior of the protocol. The next step would be to use model checking to test more specifications of the system. Therefore other models with different model abstraction levels are used.

8.2.3 A simplified model for model checking the handshake

As the protocol can be divided into two parts, the one before the handshake and the one after the handshake, a first model is used to prove that eventually after the sensor agents receive values from the environment the protocol establishes the handshake. For this model all sensors besides the one which is used for the handshake can be disregarded. Also all middle level agents can be disregarded, as they are not communicating with the driver. The communication between the middle level agent and the sensor will just result in a message in the message queue of the sensor, which will not be received until the handshake is done, so it is safe to reduce the model to the sensor agent and the driver agent and disregard all other agents. The Promela code is listed in the appendix under 12.2. In the sensor agent the initial value for the messages is set to -1 and a random procedure assigns a new binary value on random or leaves it as it is. To show that the handshake eventually takes place the model checking has to test, whether after the sensor value is unequal -1 the driver agent eventually establishes the connection, which is represented as the assignment of 1 to a defined variable. The LTL specification and the correspondent never claim are shown in the following figure:

```
LTL:
((<>  p) -> ( <>  q))  || []  !p  should hold for all executions
#define p (warnsend ==1)
#define q (initv ==1)
never {    /* !(((<>  p) -> ( <>  q))  || []  !p) */
T0_init:
    if
    :: (! ((q)) && (p)) -> goto accept_S4
    :: (! ((q))) -> goto T0_init
    fi;
accept_S4:
    if
    :: (! ((q))) -> goto accept_S4
    fi;
}
```

Figure 19: LTL specification and never claim for the handshake

The model checking was able to show that exactly this specification holds for the model.

The message flow in an example simulation is shown in the figure on the next page. In this example simulation the sensor is sending the initial value twice before a “real” value is sent. The driver agent answers with the “isinit”-message and would initialize the connection to Webots.

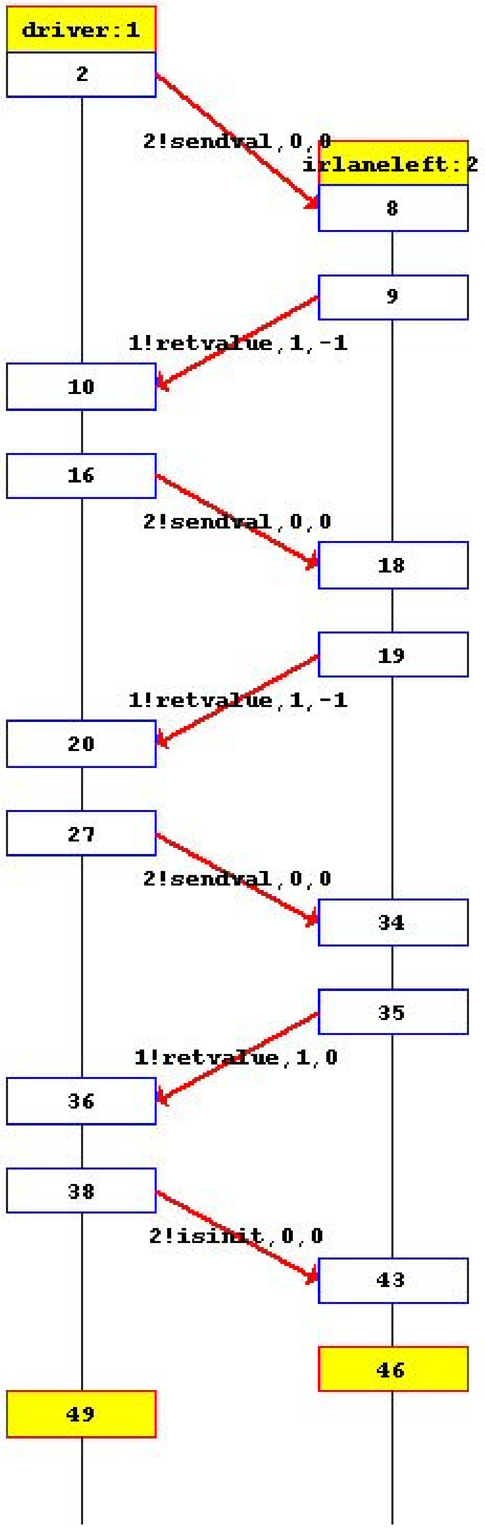

Figure 20: Message flow example for the handshake

8.2.4 A simplified model for model checking

In order to perform some more tests with model checking a different model was used, which models the second part of the protocol. In this model all sensor agents are disregarded. Only the middle level agents and the driver agent are used in this model (see 12.3). The middle agents answer a request with a warning or non-warning message on random. With this model it was checked that it is not possible to reach a not allowed lane number, which was a positive test. An assertion was used for this test. At the beginning of the keeplane method in the driver agent the assertion

assert(curlane == 1 || curlane == 2 || curlane ==3)

was introduced that checked that the curlane variable holds a valid value. The other tests were made to check whether a warning is eventually handled in the driver agent.

The LTL specification and the never claim is similar to the one shown in the previous chapter. The commands of the warnsend=1 and the initv=1 were used in the corresponding positions(warnsend after the warning has been sent and initv where the warning is handled) for each warning message.

It turned out that a warning from the back sensor might never be handled but stay in the message queue and blocks it eventually when more messages of this kind are added to the queue. That it might never be handled was assumed in the initial design, but it was noticed that this might cause a spamming problem in the message queue. The messages of the front sensor agent and the lane departure control agent are eventually handled if it is assured that the messages from the back sensor are not spamming the system.

Just these simple model checking specifications showed another faulty behavior of the protocol which might cause errors in the simulation with Webots.

This should conclude the discussion of the usage of model checking to assess the problem of this protocol. Major insights were gained by using the Spin Model Checker, which showed problems with the protocol which were not noticed while designing the protocol. What all this means for the process of designing and testing a protocol will be summarized in the next chapter: the conclusion of this project.

9 Conclusion

As this project is finished some of the major results should be mentioned here. The task of designing a successful protocol for the scenario of a model car on a three lane motorway was only partially successful as the protocol has still errors and flaws, which was shown in tests with the simulation software. This is not surprising as designing a protocol from scratch is very difficult and the hardware inspired approach with warnings was misleading. But nevertheless the flaws in the design of protocol had one advantage: Testing and Verification became more important and different approaches could be used.

The testing in simulation might help to see whether the protocol is successful, but it is hard to find errors if the protocol is not successful in some situations. Even log files and print statements cannot be used to gain an easy and intuitive insight in the message passing. You have to use other methods to find errors in a protocol, if you want to find most of the errors (As long as you cannot model check the full system there is no way of knowing that you have found all errors).

With the aid of model checking and model simulation in the Spin Model Checker it is possible to gain insight in the message passing more intuitively. The modeling of the system is very important to get good results. It has to be assured that the model is actually a model of the system and the used protocol; otherwise the whole testing and verification is useless as the results are for a totally different system, but nevertheless abstraction has to be used to allow model checking in a reasonable time and memory usage. As shown in this project even different models with different abstraction levels might be useful to respect the different characters of different tests. After this project is done now it even seems appropriate to use the Spin Model Checker in the protocol design phase to get a first glimpse of the system behavior even before it is implemented. One key problem with this idea is that it is possible to design systems with the Spin Model Checker which cannot be implemented in a certain system architecture and protocol language. So the knowledge of the boundaries of the architecture and language has to be respected in the modeling process.

The reasonable good results of this protocol, even though it has flaws, might partially be provoked by the simple scenario, so that conclusion towards the usefulness of MagentA in real time environments have to be postponed to further research with a more complicated scenario and architecture. The different amount of cycles in the waitfor-loops might be a good step in the right direction, but whether that is enough to ensure an appropriate behavior under race conditions of the system is still in question.

10 Further recommendation

This project opens a lot of possibilities for further research in different directions. The first and most obvious continuation would be the design of a more successful protocol, with the knowledge gained in this project. The description of the problems shows what has to be corrected. Another way would be the addition of more agents to the model car. With the usage of more sensors, for example the already modeled GPS device and the camera, it is possible to regard the typical Multi Agent Task of negotiation. For example the camera could be used as obstacle detection and lane departure control and could negotiate with the already used sensors about the conclusions to be drawn by the sensor data. Another typical Multi Agent attribute, the fall back possibility, can also be modeled with more agents. These are just some possibilities to continue the protocol design from this project on.

Also the MAP language and the MagentA platform could be revised with the results from this project. For simulation scenarios like the one used for this project, real time constraints are really important. Until now it is not possible to use real time constraints, but only different numbers for the loop iterations in the waitfor loop. It would be interesting to design an environment which provides more real time abilities and design a protocol for this new environment.

Another interesting idea for further research is a different approach on the protocol design methodology. The Spin Model Checker is an excellent tool to use even in the design period. One has to keep in mind that it is possible to model system with the Spin Model Checker which cannot be translated into a protocol and a Multi Agent System with MAP and MagentA. But if the characteristics of MAP and MagentA are respected it should be possible to design a protocol more effectively than from scratch. Of course the Spin Model Checker is not able to substitute test in simulation with real data and the time constraints.

11 Bibliography

[1] Webots; www.cyberbotics.com

[2] MagentA; http://homepages.inf.ed.ac.uk/cdw/magenta.html

[3] Christopher Walton, David Robertson; „Flexible Multi-agent Protocols";http://homepages.inf.ed.ac.uk/cdw/magenta.html

[4] Spin Model Checker; http://spinroot.com/spin/whatispin.html

[5] Manuel Schönfeld; "Die Geschichte des Automobils" http://www.learnline.de/angebote/automobil/info/die.htm

[6] E.D. Dickmanns, A. Zapp; „A curvature-based scheme for improving road vehicle guidance by computer vision"; SPIE Conference on Mobile Robots, Volume 727; 1986

[7] C.E. Thorpe (ed.); *"Vision and Navigation -- the Carnegie Mellon Navlab";* Kluwer Academic Publishers; 1990

[8] H.-H. Braess und G. Reichart; „Prometheus: Vision des `intelligenten Automobils' auf `intelligenter Straße' ? Versuch einer kritischen Würdigung"; ATZ Automobiltechnische Zeitschrift; 1995

[9] www.mercedes-benz.de (commercial information) and

[10] "DISTRONIC proximity and cruise control system: now fitted in more than 40,000 cars worldwide";

http://www.new-cars.com/news/030225-distronic-system.html

[11] Roboka; http://roboka.org

[12] Yizhen Zhang, Alcherio Martinoli; "Swarm Intelligence and Traffic Safety";

http://www.cnse.caltech.edu/Research02/reports/zhang1full.html

[13] Stuart Russell, Peter Norvig; "Artificial Intelligence – A Modern Approach (Second Edition)"; Pearson Education International, 2003

[14] Peter Stone, Manuela Veloso; "Multiagent Systems: A Survey from a Machine Learning Perspective"; Autonomous Robotics, Vol. 8; 2000

[15] Christopher Walton; "Multi-Agent Dialogue Protocols"; http://homepages.inf.ed.ac.uk/cdw/magenta.html

[16] Christopher Walton; "Dialogue Protocols for Multi-Agent Systems";

http://homepages.inf.ed.ac.uk/cdw/magenta.html

[17] Christopher Walton; "Model Checking Multi-Agent Web Services";

http://homepages.inf.ed.ac.uk/cdw/magenta.html

[18] Christopher Walton, Adam Barker; "An Agent-based e-Science Experiment Builder"; personal communication

[19] David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Champion, Chris Ferris, David Orchard; "Web Services Architecture" ; W3C; http://www.w3.org/TR/ws-arch/

[20]
http://java.sun.com/webservices/docs/1.0/tutorial/doc/IntroWS.html

[21] Edmund M Clarke, Jr., Orna Grumberg, Doron A. Peled; "Model Checking"; MIT Press; 1999

[22] Gerard J. Holzmann; "The Spin Model Checker"; Addison-Wesley; 2004

[23] "Webots User Guide", Release 4.0.21; www.cyberbotics.com

[24] "Webots Reference Manual", Release 4.0.21; www.cyberbotics.com

[25] www.mercedes-benz.de

12 Appendix

12.1 The full model (Promela Code)

```
#define DRIV 0
#define LANEDEP 1
#define FDIST 2
#define BDIST 3
#define LDIST 4
#define RDIST 5
#define IRB 6
#define IRBL 7
#define IRBR 8
#define IRF 9
#define IRFL 10
#define IRFR 11
#define IRLL 12
#define IRLM 13
#define IRLR 14
#define MBUFFER 10
#define AGENTS 15
#define TIMEOUT 50

mtype = { sendval, retvalue,isinit};


/* Global Variables */
chan messages[AGENTS] = [MBUFFER] of {mtype , int , int};
int state = -1;
int steer = 0;
int acc = 0;
int curlane = -1;
int checkone = -1;
int irllval = -1
int lanedepval = 0;

proctype driver()
{

  /* normal vars */
  int valirll = -1;
  int valfdist = -1;
  int valbdist = -1;
  int vallanedep = -1;
  int valldist = -1;
  int valrdist = -1;
  int modus = 0;
  int dir = -1;
  int flagot = 0;
  int itera = 0;
  chan ch1 = messages[DRIV];
```

```
xr ch1;

loopinit:
  state=0;
  messages[IRLL]!sendval,DRIV,0;
  ch1??retvalue,IRLL,valirll;
  if
    :: valirll != -1 ->
      messages[IRLL]!isinit,DRIV,0;
      curlane=1;
      printf("INIT");
      goto keeplane;
    :: else ->
      goto loopinit;
  fi;

keeplane:
steer = 0;
acc = 0;
  assert(curlane == 1 || curlane == 2 || curlane ==3);
  state=1;
  messages[FDIST]!sendval,DRIV,0;
  messages[LANEDEP]!sendval,DRIV,0;
  messages[BDIST]!sendval,DRIV,0;
modus1:
  if
    :: (modus == 0 && ch1??[retvalue,FDIST,valfdist]) ->
      ch1??retvalue,FDIST,valfdist;
      messages[LDIST]!sendval,DRIV,0;
      messages[RDIST]!sendval,DRIV,0;
      /* timeout missing */
      do
        :: (ch1??[retvalue,LDIST,valldist] && ch1??[retvalue,RDIST,valrdist]) ->
          ch1??retvalue,LDIST,valldist;
          ch1??retvalue,RDIST,valrdist;
          itera = 0;
          if
            :: valfdist ==1 ->
              progress0:
              acc++;
              if
                :: curlane==1 ->
                  if
                    :: valrdist ==1 ->
                      dir = 1;
                      steer++;
                      goto overtake;
                    :: else ->
                      modus=1;
                      goto modus1;
                  fi;
                :: curlane==3 ->
                  if
                :: valldist ==1 ->
                  dir = -1;
                  steer++;
                  goto overtake;
                :: else ->
                  modus=1;
                  goto modus1;
              fi;
            :: curlane==2 ->
              if
                :: valrdist ==1 ->
                  dir = 1;
                  steer++;
                  goto overtake;
                :: valldist ==1 ->
                  dir = -1;
                  steer++;
                  goto overtake;
                :: else ->
                  modus=1;
                  goto modus1;
              fi;
          fi;
        :: else ->
          modus=1;
          goto modus1;
      fi;
    :: else -> itera++
    :: itera == 10 ->
      if
        :: valfdist ==1 ->
          progress1:
          acc++;
        :: else ->
          modus =1;
          itera =0;
          goto modus1;
      fi
  od
:: else ->
modus2:
  if
    :: ((modus == 0 || modus ==1) && ch1??[retvalue,LANEDEP,vallanedep]) ->
      ch1??retvalue,LANEDEP,vallanedep;
      if
        :: vallanedep==1 ->
          progress2:
          steer++;
          modus =0;
          goto keeplane;
        :: else ->
          modus =2;
          goto modus2;
```

```
            fi;
          :: else ->
            if
              :: ((modus == 0 || modus ==1 || modus==2) && ch1??[retvalue,BDIST,valbdist]) ->
                ch1??retvalue,BDIST,valbdist;
                progress3:
                acc++;
                modus ==0;
                goto keeplane;
              :: else ->
                progress4:
                acc++;
                modus ==0;
                goto keeplane;
            fi;
        fi;

    fi;
    modus=0;
    goto keeplane;

  overtake:
    state=2;
    messages[LANEDEP]!sendval,DRIV,0;
    ch1??retvalue,LANEDEP,vallanedep;
    if
      :: flagot == 0 && vallanedep==1 ->
        flagot =1;
        goto overtake;
      :: flagot == 1 && vallanedep==0 ->
        flagot =0;
        curlane= curlane + dir;
        goto keeplane;
      :: else ->
        goto overtake;
    fi
}

proctype lanedep()
{
  int val = -1;
  int val_l = -1;
  int val_m = -1;
  int val_r = -1;
  chan ch1 = messages[LANEDEP];
  xr ch1;

  loopa:
      messages[IRLL]!sendval,LANEDEP,0;
      messages[IRLM]!sendval,LANEDEP,0;
      messages[IRLR]!sendval,LANEDEP,0;
      ch1??retvalue,IRLL,val_l;
      ch1??retvalue,IRLM,val_m;
```

```
        ch1??retvalue,IRLR,val_r;
        if
          :: lanedepval = 1
          :: lanedepval = 0
        fi;
        if
          :: ch1??[sendval,DRIV,0] ->
            ch1??sendval,DRIV,0;
            messages[DRIV]!retvalue,LANEDEP,lanedepval;
            goto loopa
          :: else -> skip
        fi;
        goto loopa

}

proctype rightdist()
{
    int val = -1;
    int val_f = -1;
    int val_b = -1;
    chan ch1 = messages[RDIST];
    xr ch1;
    int rvalue = -1;

    loopa:
        messages[IRBR]!sendval,RDIST,0;
        messages[IRFR]!sendval,RDIST,0;
        ch1??retvalue,IRBR,val_b;
        ch1??retvalue,IRFR,val_f;
        if
          :: rvalue = 1
          :: rvalue = 0
        fi;
        if
          :: ch1??[sendval,DRIV,0] ->
            ch1??sendval,DRIV,0;
            messages[DRIV]!retvalue,RDIST,rvalue;
            goto loopa
          :: else -> skip
        fi;
        goto loopa

}

proctype leftdist()
{
    int val = -1;
    int val_f = -1;
    int val_b = -1;
    chan ch1 = messages[LDIST];
    xr ch1;
    int rvalue = -1;
```

```
 loopa:
    messages[IRBL]!sendval,LDIST,0;
    messages[IRFL]!sendval,LDIST,0;
    ch1??retvalue,IRBL,val_b;
    ch1??retvalue,IRFL,val_f;
    if
      :: rvalue = 1
      :: rvalue = 0
    fi;
    if
      :: ch1??[sendval,DRIV,0] ->
        ch1??sendval,DRIV,0;
        messages[DRIV]!retvalue,LDIST,rvalue;
        goto loopa
      :: else -> skip
    fi;
    goto loopa

}

proctype backdist(){

   chan ch1 = messages[BDIST];
   xr ch1;
   int val = -1;
   int rvalue = -1;

 loopa:
   messages[IRB]!sendval,BDIST,0;
   ch1??retvalue,IRB,val ;
   if
     :: rvalue = 1
     :: rvalue = 0
   fi;
   if
     :: ch1??[sendval,DRIV,0] ->
       ch1??sendval,DRIV,0;
       messages[DRIV]!retvalue,BDIST,rvalue;
       goto loopa
     :: else -> skip
   fi;
   goto loopa


}


proctype frontdist(){

   chan ch1 = messages[FDIST];
   xr ch1;
   int val = 0;
```

```
    int rvalue = 0;

  loopa:
    messages[IRF]!sendval,FDIST,0;
    ch1??retvalue,IRF,val ;
    if
      :: rvalue = 1
      :: rvalue = 0
    fi;
    if
      :: ch1??[sendval,DRIV,0] ->
        ch1??sendval,DRIV,0;
        messages[DRIV]!retvalue,FDIST,rvalue;
        goto loopa
      :: else -> skip
    fi;
    goto loopa


}

proctype irlaneleft()
{
  chan ch1 = messages[IRLL];
  xr ch1;

  loopinit:
      /* Proc Rec_VAL */
      if
        :: irllval =0
        :: irllval =1
        :: skip
      fi;
      if
        :: ch1??[sendval,DRIV,0] ->
          ch1??sendval,DRIV,0;
          messages[DRIV]!retvalue,IRLL,irllval;
          goto loopinit
        :: ch1??[isinit,DRIV,0] ->
          ch1??isinit,DRIV,0;
          goto loopa
        :: else -> skip
      fi;
      goto loopinit;

  loopa:
      /* Proc Rec_VAL */
      if
        :: irllval =0
        :: irllval =1
        :: skip
      fi;
  end:  if
```

```
        :: ch1??[sendval,LANEDEP,0] ->
          ch1??sendval,LANEDEP,0;
          messages[LANEDEP]!retvalue,IRLL,irllval;
          goto loopa
        :: else -> skip
      fi;
      goto loopa
}

proctype irbackright()
{
  int val = 0;
  chan ch1 = messages[IRBR];
  xr ch1;

  loopa:
      /* Proc Rec_VAL */

  end:  if
        :: ch1??[sendval,RDIST,0] ->
          ch1??sendval,RDIST,0;
          messages[RDIST]!retvalue,IRBR,val;
          goto loopa
        :: else -> skip
      fi;
      goto loopa
}

proctype irlanemiddle()
{
  int val = 0;
  chan ch1 = messages[IRLM];
  xr ch1;

  loopa:
      /* Proc Rec_VAL */

  end:  if
        :: ch1??[sendval,LANEDEP,0] ->
          ch1??sendval,LANEDEP,0;
          messages[LANEDEP]!retvalue,IRLM,val;
          goto loopa
        :: else -> skip
      fi;
      goto loopa
}

proctype irlaneright()
{
  int val = 0;
  chan ch1 = messages[IRLR];
  xr ch1;
```

```
 loopa:
     /* Proc Rec_VAL */

 end:  if
        :: ch1??[sendval,LANEDEP,0] ->
          ch1??sendval,LANEDEP,0;
          messages[LANEDEP]!retvalue,IRLR,val;
          goto loopa
        :: else -> skip
     fi;
     goto loopa
}

proctype irfront()
{
 int val = 0;
 chan ch1 = messages[IRF];
 xr ch1;

 loopa:

 end:  if
        :: ch1??[sendval,FDIST,0] ->
          ch1??sendval,FDIST,0;
          messages[FDIST]!retvalue,IRF,val;
          goto loopa
        :: else -> skip
     fi;
     goto loopa
}

proctype irback()
{
 int val = 0;
 chan ch1 = messages[IRB];
 xr ch1;

 loopa:

 end:  if
        :: ch1??[sendval,BDIST,0] ->
          ch1??sendval,BDIST,0;
          messages[BDIST]!retvalue,IRB,val;
          goto loopa
        :: else -> skip
     fi;
     goto loopa
}

proctype irbackleft()
{
 int val = 0;
 chan ch1 = messages[IRBL];
```

```
  xr ch1;

  loopa:
      /* Proc Rec_VAL */

  end:  if
        :: ch1??[sendval,LDIST,0] ->
          ch1??sendval,LDIST,0;
          messages[LDIST]!retvalue,IRBL,val;
          goto loopa
        :: else -> skip
      fi;
      goto loopa
}

proctype irfrontright()
{
  int val = 0;
  chan ch1 = messages[IRFR];
  xr ch1;

  loopa:
      /* Proc Rec_VAL */

  end:  if
        :: ch1??[sendval,RDIST,0] ->
          ch1??sendval,RDIST,0;
          messages[RDIST]!retvalue,IRFR,val;
          goto loopa
        :: else -> skip
      fi;
      goto loopa

}

proctype irfrontleft()
{
  int val = 0;
  chan ch1 = messages[IRFL];
  xr ch1;

  loopa:
      /* Proc Rec_VAL */

  end:  if
        :: ch1??[sendval,LDIST,0] ->
          ch1??sendval,LDIST,0;
          messages[LDIST]!retvalue,IRFL,val;
          goto loopa
        :: else -> skip
      fi;
      goto loopa
```

```
}

init {
  run driver();
  run irfrontright();
  run irbackright();
  run rightdist();
  run leftdist();
  run irfrontleft();
  run irbackleft();
  run backdist();
  run frontdist();
  run irback();
  run irfront();
  run lanedep();
  run irlaneright();
  run irlanemiddle();
  run irlaneleft();
}
```

12.2 The handshake model (Promela Code)

```
#define DRIV 0
#define IRLL 1
#define MBUFFER 10
#define AGENTS 2
#define TIMEOUT 50

mtype = { sendval, retvalue,isinit};


/* Global Variables */
chan messages[AGENTS] = [MBUFFER] of {mtype , int , int};
int curlane = -1;
int warnsend=0;
int warn1=0;
int initv = 0;
int irllval = -1;

proctype driver()
{

  /* normal vars */
  int valirll = -1;
  initv = 0;
  chan ch1 = messages[DRIV];
  xr ch1;

  loopinit:
        warn1=0;
    messages[IRLL]!sendval,DRIV,0;
    ch1??retvalue,IRLL,valirll;
  if
    :: valirll != -1 ->
      messages[IRLL]!isinit,DRIV,0;
      curlane=1;
      printf("INIT");
      warn1=1;
      goto keeplane;
    :: else ->
      goto loopinit;
  fi;

 keeplane:
 initv=1

}


proctype irlaneleft()
{
          irllval = -1;
 chan ch1 = messages[IRLL];
 xr ch1;

 loopinit:
          warnsend=0;
    /* Proc Rec_VAL */
    if
      :: skip
      :: irllval =0
      :: irllval =1
    fi;
    if
      :: ch1??[sendval,DRIV,0] ->
        ch1??sendval,DRIV,0;
        messages[DRIV]!retvalue,IRLL,irllval;
        if
          ::irllval != -1 ->
                  warnsend =1;
                                                      ::else ->
                                                              skip;
        fi;
        goto loopinit
      :: ch1??[isinit,DRIV,0] ->
        ch1??isinit,DRIV,0;
        goto loopa
      :: else -> skip
    fi;
    goto loopinit;

 loopa:
 skip;

}
```

```
init {
  run driver();
  run irlaneleft();
}
```

12.3 The simplified model (Promela Code)

```
#define DRIV 0
#define LANEDEP 1
#define FDIST 2
#define BDIST 3
#define LDIST 4
#define RDIST 5

mtype = { sendval, retvalue};


/* Global Variables */
chan chdriver =[10] of {mtype,int,int};
chan chldist =[1] of {mtype,int,int};
chan chrdist =[1] of {mtype,int,int};
chan chlanedep =[1] of {mtype,int,int};
chan chfdist =[1] of {mtype,int,int};
chan chbdist =[1] of {mtype,int,int};
int state = 1;
int steer = 0;
int acc = 0;
int warnsend =0;
int initv=0;
int curlane = 1;
int lanedepval = 0;
int rightdistval = 0;
int leftdistval = 0;
int frontdistval =0;
int backdistval =0;
int frontwarning =0;
int lanewarning =0;
int backwarning =0;

proctype driver()
{

  /* normal vars */
  int valfdist = 0;
  int valbdist = 0;
  int vallanedep = 0;
  int valldist = 0;
  int valrdist = 0;
  int modus = 0;
  int dir = 0;
  int flagot = 0;
```

```
xr chdriver;


keeplane:
lanewarning=0;
backwarning=0;
frontwarning=0;
steer = 0;
acc = 0;
  assert(curlane == 1 || curlane == 2 || curlane ==3);
  state=1;
  chfdist!sendval,DRIV,0;
  chlanedep!sendval,DRIV,0;
  chbdist!sendval,DRIV,0;

modus1:
          initv=0;
  if
    :: (modus == 0 && chdriver??[retvalue,FDIST,valfdist]) ->
      chdriver??retvalue,FDIST,valfdist;
      chldist!sendval,DRIV,0;
      chrdist!sendval,DRIV,0;
      frontwarning= 1;

      if
        :: (chdriver??[retvalue,LDIST,valldist] && chdriver??[retvalue,RDIST,valrdist]) ->
          chdriver??retvalue,LDIST,valldist;
          chdriver??retvalue,RDIST,valrdist;
          if
            :: valfdist ==1 ->
              progress0:
              acc = 1;
              if
                :: curlane==1 ->
                  if
                    :: valrdist ==1 ->
                      dir = 1;
                      steer = 1;
                      goto overtake;
                    :: else ->
                      modus=1;
                      goto modus1;
                  fi;
                :: curlane==3 ->
                  if
                    :: vaIldist ==1 ->
                      dir = -1;
                      steer = 1;
                      goto overtake;
                    :: else ->
                      modus=1;
                goto modus1;
              fi;
            :: curlane==2 ->
              if
                :: valrdist ==1 ->
                dir = 1;
                steer = 1;
                goto overtake;
                :: valldist ==1 ->
                dir = -1;
                steer = 1;
                goto overtake;
                :: else ->
                modus=1;
                goto modus1;
              fi;
          fi;
        :: else ->
          modus=1;
          goto modus1;
      fi;
    :: else ->
      if
        :: valfdist ==1 ->
          progress1:
          acc = 1;
        :: else ->
          modus =1;
          goto modus1;
      fi
  fi
:: else ->
modus2:
  if
    :: ((modus == 0 || modus ==1) && chdriver??[retvalue,LANEDEP,vallanedep]) ->
      chdriver??retvalue,LANEDEP,vallanedep;
      lanewarning= 1;
      if
        :: vallanedep==1 ->
          progress2:
          steer = 1;
          modus =0;
          goto keeplane;
        :: else ->
          modus =2;
          goto modus2;
      fi;
    :: else ->
      if
        :: ((modus == 0 || modus ==1 || modus==2) && chdriver??[retvalue,BDIST,valbdist]) ->
          chdriver??retvalue,BDIST,valbdist;
          backwarning= 1;
          initv=1;
```

```
            progress3:
            acc = 1;
            modus =0;
            goto keeplane;
           :: else ->
            progress4:
            acc = 1;
            modus =0;
            goto keeplane;
          fi;
        fi;

    fi;
    modus=0;
    goto keeplane;

  overtake:
    state=2;
    chlanedep!sendval,DRIV,0;
    chdriver??retvalue,LANEDEP,vallanedep;
    if
      :: flagot == 0 && vallanedep==1 ->
        flagot =1;
        goto overtake;
      :: flagot == 1 && vallanedep==0 ->
        flagot =0;
        curlane= curlane + dir;
        goto keeplane;
      :: else ->
        goto overtake;
    fi
}

proctype lanedep()
{
  xr chlanedep;

  loopa:
      if
        :: lanedepval = 1
        :: lanedepval = 0
      fi;
      if
        :: chlanedep??[sendval,DRIV,0] ->
          chlanedep??sendval,DRIV,0;
          chdriver!retvalue,LANEDEP,lanedepval;
          goto loopa
        :: else -> skip
      fi;
      goto loopa

}
```

```
proctype rightdist()
{

  xr chrdist;

  loopa:
      if
        :: rightdistval = 1
        :: rightdistval = 0
      fi;
      if
        :: chrdist??[sendval,DRIV,0] ->
          chrdist??sendval,DRIV,0;
          chdriver!retvalue,RDIST,rightdistval;
          goto loopa
        :: else -> skip
      fi;
      goto loopa

}

proctype leftdist()
{
  xr chldist;

  loopa:
      if
        :: leftdistval = 1
        :: leftdistval = 0
      fi;
      if
        :: chldist??[sendval,DRIV,0] ->
          chldist??sendval,DRIV,0;
          chdriver!retvaIue,LDIST,leftdistval;
          goto loopa
        :: else -> skip
      fi;
      goto loopa

}

proctype backdist(){

    xr chbdist;

  loopa:
    if
      :: backdistval = 1
      :: backdistval = 0
    fi;
    if
      :: chbdist??[sendval,DRIV,0] ->
        chbdist??sendval,DRIV,0;
        chdriver!retvalue,BDIST,backdistval;
        if
             ::backdistval != -1 ->
                          warnsend =1;
                                                              ::else ->
                                                                          skip;
          fi;
        goto loopa
      :: else -> skip
    fi;
    goto loopa


}


proctype frontdist(){

    xr chfdist;

  loopa:
            warnsend=0;
    if
      :: frontdistval = 1
      :: frontdistval = 0
    fi;
    if
      :: chfdist??[sendval,DRIV,0] ->
        chfdist??sendval,DRIV,0;
        chdriver!retvalue,FDIST,frontdistval;

        goto loopa
      :: else -> skip
    fi;
    goto loopa


}

init {
  run driver();
  run rightdist();
  run leftdist();
  run backdist();
  run frontdist();
  run lanedep();
}
```