Driver Support System for Traffic Manoeuvres

Jacek Malec and Per Österling
Department of Computer and Information Science
Linköping University
S–581 83 Linköping, Sweden
jam@ida.liu.se  peros@ida.liu.se

Abstract

The main topic of this paper is a methodology of designing driver support systems. A driver support system (co-pilot) is required to warn the driver about immediate dangers, such as unexpected obstacles on the road, to provide him with help when he is performing complex manoeuvres, such as overtaking or passing an intersection, and maybe to intervene in some extreme cases, when there is no other possibility of avoiding an accident.

An intelligent co-pilot must possess extensive knowledge about possible traffic situations and scenarios. It must also be able to evaluate and predict drivers’ behaviour in these situations in order to provide meaningful support. Therefore, the task of a co-pilot is first, to recognize the current traffic situation and the driver’s activity (and possibly also his intentions), and secondly, to match this activity against the expected one. If there is any discrepancy between the two then possibly there is a room for some supportive action of the co-pilot.

Knowledge about the dynamically changing road situations has to be continuously extracted from sensor data. The process of forming high-level, symbolic descriptions of traffic situations in real-time, called here characterization, is one of the crucial elements of the analysis and evaluation of the current traffic scenario.

The paper focuses on some chosen aspects of driver support system design methodology, namely on:

1. a solution for the real-time characterization problem;
2. a choice of knowledge representation formalisms required by various co-pilot functions;
3. adoption of systematic software engineering principles, and especially of layered software architecture and software engines facilitating the design and prototyping phases.
### Contents

1 Introduction 3

2 Support needs of the driver 5

3 Characterization 6
   3.1 Understanding the scenario 6
   3.2 The characterizer 7
      3.2.1 Sensory information 7
      3.2.2 Positions 7
      3.2.3 Configuration determination 7
      3.2.4 Properties and trends of individual cars 8
      3.2.5 Properties and trends of configurations 8
      3.2.6 Significant events 8
   3.3 The configuration space 8
      3.3.1 The Own and Meeting configurations 8
      3.3.2 The Merging and Crossing configurations 9
      3.3.3 An example 9
      3.3.4 Relative positions and closeness 10

4 Representation 12
   4.1 Scenario conceptualization with Process Transition Networks 13
      4.1.1 Process Transition Networks 13
      4.1.2 Conceptualization of intersection passing 14
   4.2 Formal description with a Temporal Feature Logic 18
   4.3 Configuration description 21

5 Implementation 22
   5.1 Architecture of the system 22
      5.1.1 The Sensor Simulator Module 22
      5.1.2 The Process Layer Executive 23
      5.1.3 The Rule Layer Executive 24
      5.1.4 Inter-layer Communication Package 24
      5.1.5 The Road Segment Data Base Manager 25
   5.2 Implemented prototype of a Driver Support System 25
   5.3 Examples 27

6 Conclusions 28
1 Introduction

The increasing density of road traffic together with the continuous improvement of cars’ performance make the driving task more and more demanding. The driver is forced to make more decisions and has to make them faster than before. On the other hand, the availability of a wide range of advanced electronic equipment, both sophisticated sensors and advanced communication devices, create among the car manufacturers the temptation to provide the driver with all the information available in the car in order to help him to make the best (or at least an acceptable) decision in given circumstances. Other obvious factors affecting the design decisions are the needs to increase driving comfort, to improve economy and to decrease environmental effects. Within several research programmes (PROMETHEUS and DRIVE in Europe, IVHS in U.S.A., to name some of them) a multitude of systems are being created to meet these ends. For example, one can mention driver support systems, intelligent cruise control systems, route guidance systems, or dynamic autonomous car control systems as the relatively advanced project areas.

The topic addressed in this paper is how to support the driver performing traffic manoeuvres. This support might require warning the driver about some immediate danger, such as an unexpected obstacle on the road, and can also include the possibility of an autonomous intervention (in most of the imaginable circumstances this would be braking with maximal force). Another type of support is needed when the driver is performing some difficult manoeuvre, such as overtaking or crossing a crowded intersection.

An intelligent driver support system (DSS, sometimes also called co-pilot) must possess extensive knowledge about possible traffic situations and scenarios. It must also be able to evaluate and predict drivers’ behaviour in these situations in order to provide meaningful support. Therefore, the task of a co-pilot is first, to recognize the current traffic situation and the driver’s activity (and possibly also his intentions), and secondly, to match this activity against the expected one. If there is any discrepancy between the two then possibly there is a room for some supportive action of the co-pilot.

Knowledge about the dynamically changing road situations has to be continuously extracted from sensor data. The process of forming high-level, symbolic descriptions of traffic situations in real-time, called here characterization, is one of the crucial elements of the analysis and evaluation of the current traffic scenario and therefore receives much attention in this paper.

The task of modelling the intended behaviour of the driver participating in different traffic scenarios under various circumstances is another major research topic covered by this paper. Artificial intelligence concentrates, among other things, on creating representation mechanisms for describing activities of intelligent agents (such as drivers, for example). There exists a multitude of formalisms suitable for this purpose. In this paper we present some of them together with an explanation of why we consider them as best suiting our purpose, i.e. the design of an intelligent driver support system.

The next problem faced by a designer of a DSS is that of recognition of the actual activity performed by the driver. Obviously this is not an easy task, requiring a lot of additional knowledge, both about the typical human driving behaviour (which need not fully conform to the activity prescribed by the law and by idealistic traffic scenarios) and about the particular driver, his preferences, intricacies or peculiarities. Although this challenging area is aptly suited the knowledge-based approach of AI we are presenting and advocating here, we are not going to focus on this aspect of DSS design. Rather, we will simply assume that a DSS has some possibilities of observing the driver via the actions he performs and thus it is able to infer the driver’s current activity and, to some degree, also his intentions.

Finally, the problem of comparing the intended and the actual activity of the driver has to be addressed. The comparison has to be made on several levels of abstraction. One can ask whether the driver performs the intended action at all. Another question might be whether the action is performed at the proper moment in time. Finally, the numerical parameters describing an action may be analyzed, e.g. whether the degree of rotation of the steering wheel is appropriate, whether more intensive braking should be applied, etc.

We have separated continuous from discrete analysis and have put them in different layers of the advocated DSS architecture. This allows us to use purely discrete, symbolic models for the purpose of reasoning, while retaining the ability to analyze and process numerical data when it happens to be necessary.

Because a DSS or, in fact, any of the systems mentioned earlier generates a lot of data, the amount of information potentially available to the driver is very large. In this situation there is an immanent danger of overloading the driver’s perceptual capabilities. Clearly, there is a strong need for systematic handling of the output to the driver from different subsystems in the car in order to guarantee that the driver gets the relevant information at the right time. This task is complicated by a number of circumstances, for instance:
• Output requests may come from many different sources (support systems) operating independently, or almost independently, without a coherent view of the information available in the whole system.

• Information is time-dependent, so it might prove useless, or even dangerous, if presented to the driver at the wrong time.

• The amount of information the driver can handle is not static, but varies dynamically depending on, among other things, the traffic situation, the weather conditions and the kind of manoeuvre the driver is performing.

• There may be several output channels available (for example displays, voice output, and haptic actuators) whose appropriateness may change dynamically.

• Drivers may prefer some output channels to others, or may want to suppress some kinds of information.

Although challenging, this topic of information relevance will not be further discussed in this paper. The interested reader is referred to [MM93] for a description of research on this interesting issue.

Our goal is to provide a sound methodology for designing reliable (or, in more technical terms, predictable) DSSs. In order to achieve predictability we need the following:

• formal specification of the domain (traffic situations and scenarios);
• proper implementation procedures;
• formal verification tools;
• extensive simulation and testing facilities.

In the paper we focus on the first three items from this agenda. The fourth item, proper simulation and testing, cannot be done until the full conceptualization of the traffic domain has been completed. The work described here is, hopefully, a step towards providing such a conceptualization and, more generally, a sound design methodology. This methodology is closely linked with a series of software tools which together form an environment in which we have now implemented the first support prototypes.

The software components form a layered architecture whereby the input data is analyzed and classified in successive steps. First by fast low level routines in a level called the process layer, and then in a rule-based environment called the rule layer. The software modules included in the integrated environment consist of:

• A sensor simulation package, providing pre-recorded sequences of sensor values for a number of cars in the environment.

• A process layer executive [MS90], supporting the real-time processing of the pseudo-continuous recorded data in the input streams, where the aim of processing is extraction of certain characteristics of the dynamic input stream for reporting to the rule layer.

• A rule layer executive [Mor92a] which supports the identification and classification of interesting or critical traffic situations based on the rules provided in a knowledge base.

• A number of communication routines [˚Ost91] for linking the above components by sending pseudo-continuous or discrete data flows.

• A road segment data base management module [Far91], which creates and maintains a road map data base, and provides an interface to the database for application programs.

The unifying concept between the various software modules is a fluent: an abstraction of time-dependent data flow. Each piece of the software environment can be seen as a tool facilitating particular types of transformations on fluents.

Our current prototype processes the simulated sensor information [Far91] by characterizing the current situation [NTÖ91, Øst93], sending the relevant discrete messages to the rule layer, and produces a number of information and warning messages based on further processing of the information in the rule layer. The system provides support to the driver in a selected variety of traffic scenarios. These include approaching curves in the road at different speeds and in various traffic and weather conditions, overtaking the car in front in the presence of an arbitrary number of cars in the environment, approaching and passing different types of intersections in the presence of other cars, etc.
The structure of the paper is as follows. In section 2 we provide an overview of the work done within PROMETHEUS on identification of driver’s support needs. Section 3 focuses on the characterization as the necessary and therefore primary element in the assessment of the current traffic situation. Then section 4 presents several formal representation languages used for describing traffic scenarios, expected behaviour of the driver, and, finally, the current traffic situation at an intersection. Section 5 describes the software architecture for the proposed solution and gives some examples extracted from the prototype implementation of the DSS. Finally, section 6 contains a summary of the paper.

2 Support needs of the driver

Before the functionality of a driver support system can be fully specified it is necessary to know what kind of needs the underlying systems are intended to fulfill and, furthermore, what type of support can actually be extracted assuming some realistic availability and quality of sensor data. Also, constraints imposed on the design of the system architecture have to be carefully considered. A thorough analysis of the driver’s needs has been reported in [MFvE92]. Table 1 provides a list of the identified needs, as presented there.

| N1 | driver status       |
| N2 | vehicle status      |
| N3 | detecting a road-related difficulty |
| N4 | obstacle detection  |
| N5 | detecting an oncoming user |
| N6 | detecting a user on an intersecting course |
| N7 | detecting a user outside the frontal view of vision |
| N16 | detecting a pedestrian |
| N8 | adapting speed to road conditions |
| N9 | catching up on a slower road user |
| N10 | estimating a collision course of another user |
| N11 | assessing gaps when overtaking or changing lane |
| N12 | assessing gaps when joining or cutting across the traffic flow |
| N13 | predicting that another user will move off or fail to stop |
| N14 | predicting the manoeuvre of another user |
| N17 | predicting pedestrian behaviour |
| N15 | vehicle control |

Table 1: A list of driver’s needs.

Indeed, to provide the full support entailed by N1 — N17 the system has to consist of a number of powerful modules processing information originating in different sources and related to different needs. In order to limit the discussion to realistic possibilities, a set of selected support functions has been defined [MFvE92] (see Table 2), and a correspondence table providing functions suitable for satisfying particular needs is given.

Almost all the functions listed in Table 2 are currently being subject to intensive research in the Swedish Road and Traffic Informatics programme and, in a larger setting, within the European Prometheus project.

The research described in this paper is relevant for the support functions F9–F14. A realization of functions F10–F12 (intelligent: manoeuvres, cruise control and intersection control) has to rely on proper assessment of the current road situation and on the system’s knowledge about traffic scenarios. Functions F9 and F14 (supportive driver information and emergency warning) depend moreover on proper recognition of the driver’s current behaviour and intentions. The function F13 is less dependent on an accurate recognition of driver’s intentions, however, such possibility would allow to build more widely accepted systems (i.e. the ones that pass only the relevant information to the driver).
Table 2: The support functions.

| F1 | obstacle detection |
| F2 | monitoring environment/road |
| F3 | monitoring driver |
| F4 | monitoring vehicle |
| F5 | vision enhancement |
| F6 | safety margin determination |
| F7 | critical course determination |
| F8 | dynamic vehicle control |
| F9 | supportive driver information |
| F10 | intelligent manoeuvres |
| F11 | intelligent cruise control |
| F12 | intelligent intersection control |
| F13 | medium range pre-information |
| F14 | emergency warning |

3 Characterization

3.1 Understanding the scenario

The task of a driver support system is — to say the least — not an easy one. The conditions for safe overtaking and safe intersection passing are by nature fairly complex, and dependent on a variety of rules for safe traffic behaviour as well as a large number of properties of the traffic environment, the cars involved and the own car and driver\(^1\). Furthermore, any response given by the DSS is subjected to rigorous temporal constraints; the validity of a response is not only dependent on its content but also on the time point or interval at which it is given.

Reasoning in a continuous, dynamic and complex environment can soon become intractable if each individual change in the environment is taken account of. One way of getting around the complexity is to make simplifying assumptions about the environment. Another approach avoids restricting the application domain but attempts to extract significant events from the raw sensory information prior to the reasoning stage, and to provide each type of reasoning with suitable data. We will adopt the latter approach and minimize the restrictions put on the environment and on the behaviours of cars and drivers.

To be able to arrive at support decisions at all, the support system would need to “understand” the significant characteristics of the traffic situation. This understanding must be based on available sensory information and has to be built and maintained continuously during the ongoing scenario. In a DSS realized by a layered software architecture, each layer must base its decisions on its current interpretation of the scenario. Although the layers need models of the environment, each layer has different demands on the level of abstraction and the temporal resolution of its model. This leaves us with two possibilities. The first is to have one comprehensive model of the environment, complex enough to meet the requirements of all layers. The second approach, which we have adopted, is to have a multitude of coherent models, ranging from numeric to symbolic, each intended to fulfill the requirements of a particular layer. No part of the supportive autonomous agent has access to all models. Nevertheless, these models jointly constitute the agent’s overall understanding of the surrounding environment. Moreover, the requirement put on the generated response to satisfy the time constraints of the application necessitates explicit modelling of the temporal aspects.

As an example of how models can be built from a continuous flow of sensory data via subsequent abstractions, let us consider the following. Some physical properties, such as positions, are directly observable (via sensors) in the environment. These properties constitute the “raw” quantitative model. One can then identify a class of derived conceptual properties which are based on a number of observable (and possibly other derived) properties, although not necessary on a momentary basis. For example, a “closeness” property could be based on positions and velocities of two cars being considered as potentially “close”. Yet another class of properties, referred to as trends, monitors the development of derived properties over time (e.g. “getting closer”). Finally, a vital change in a trend may constitute a significant event which in turn can lead to the update of a qualitative model. By defining significant events in terms of trends, we are able to restrict the number of considered occurrences.

\(^1\)The terms own car and own driver will be used to denote the car in which the DSS is mounted and its driver, respectively.
To cope with the complexity of a dynamic environment, we identify the need to restrict the focus of attention to a number of objects in the nearest environment of the own car. Obviously, some of the surrounding cars are more interesting for us than others. In particular, we see that both the degree of importance and the interesting properties of a specific car are determined by this car’s position relative to the own car. This interest is independent of the identities of the cars occupying the relative positions (i.e. regardless of whether a car is actually a blue Volvo or a red Saab). It is therefore advantageous to be able to refer to the cars symbolically through their location relative to the own car rather than through their actual identities. Such references, known as deictic references [AC87], are convenient means for maintaining references to objects in a dynamic environment. Moreover, to facilitate reasoning about the environment at a higher level of abstraction while still being able to refer to each individual explicitly, the deictic references can be grouped into aggregates of related objects. As an example, references to all relevant cars in the same lane as the own car can be grouped together and then be treated as a whole. Hence, one of the main tasks of the process of building a model of the environment would be to classify and “name” all the cars according to those relations.

To conclude, we need a multitude of models “large” enough to enable the support system reach the desired response, but “small” enough to make real-time reasoning computationally tractable.

### 3.2 The characterizer

The task of extracting significant events in a dynamic environment is the objective of a dynamical classification component referred to as the characterizer. The characterizer resides in the process layer of our architecture (cf. section 5.1).

The approach is to transform “raw” continuous-valued input to a discrete-valued output, where both input and output change over a continuous (or evenly sampled) time. The characterizer arrives at a model of the environment by a series of abstractions ultimately resulting in the recognition of significant events. The abstractions and the transformations from continuous to discrete typically depend on the past history.

The choice of properties and the nature of snapshots are of course dependent on the type of support to be provided. Different types of support require recognition of different types of situations, where each situation can be described in terms of a finite set of properties.

#### 3.2.1 Sensory information

From our standpoint, the output of the sensory systems is our input, independently of the actual sensory devices used. We can imagine using processed vision information, active inter–vehicle and vehicle–roadside communication, or any other sensory information about the cars and the environment. Accordingly, the characterizer receives as its input a (compound) piecewise continuous fluent (signal) from a number of sensors. This fluent includes measurements for a number of cars which happen to be in the range of the sensors, together with some information about their current signalling status.

#### 3.2.2 Positions

The position of a car is defined using so-called road segment coordinates i.e. relative to the current road segment, which is an element of a road-segment-based map of the road network. The reason for choosing road segment coordinates instead of cartesian is that the former correspond in a more natural way to the traffic behaviour; that is, cars normally follow a lane, possibly with some aberrations from its center line (axis). As a separate project within our Prometheus group, a database management system for road segments has been developed [Far91]. The system provides an implementation of a map consisting of road segments, as well as transformation routines from cartesian coordinates to road segment coordinates and vice versa.

#### 3.2.3 Configuration determination

In order to reason about the environment at a higher level of abstraction while still being able to refer to each individual explicitly, deictic referrals to the surrounding cars are grouped into aggregates called configurations. Therefore, a configuration consists of agents or objects which are related to each other in specific way over a (possibly limited) period of time. Configurations allow us to divide the objects in

---

2 A road segment can be understood as a homogeneous element of the road network, e.g. a curve, or a straight road element with a constant speed limit would be coded as separate road segments.
the environment into two classes: the In class consisting of the objects belonging to the configuration, and the Out class consisting of the remaining objects. This classification suggests a policy of using the sensing capabilities: concentrate on the In objects and check sporadically whether any of the Out objects is going to become an In. This policy silently assumes that the In/Out classification reflects the relevance of the objects in the environment. This is obviously dependent on the characterization algorithm and on the definition of configuration.

We distinguish the Own and the Meeting configurations which are always present. In addition, Merging and Crossing configurations are defined in the vicinity of an intersection.

Configurations are determined as follows. First an object characterizer determines individual properties of each car detected by the sensors in the environment. One such individual property can be the expected turning intention. Another example is the expected path — a sequence of road segments — which is likely to be traversed next. Recognition of both of these properties relies on the current status of the indicators, the current position, and the past history of the car.

Then a relation characterizer uses the individual paths of cars to determine which cars are in given binary relations with the own car. These relations are of the type “following”, “meeting”, and so on. Thus, the relation characterizer recognizes cars which are the candidates for playing some particular role in the current configuration.

In the final stage of the configuration determination the configuration characterizer checks which cars in the set of candidates are close enough for being considered as belonging to a particular configuration.

3.2.4 Properties and trends of individual cars

Next the characterizer determines properties of individual cars and trends associated with those properties. The reason for not determining them already in the object characterizer above is that they are dependent of how the cars are related to each other and, in particular, how they are related to the own car.

3.2.5 Properties and trends of configurations

The next stage of characterization consists of determining properties assigned to the environment on the basis of the properties of individual cars. Since the degree of being free (“freeness”) of the road in either lane is a useful property for e.g. overtaking support, we specify “freeness” of each lane as a function of time with the range of values free, soon-blocked, blocked and soon-free. To define the concept of “freeness” we use the results of previous characterization phases, i.e. the current configuration, the properties of individual car, and the recognized trends. For example, the blocked property of the left (meeting) lane depends on existence, properties and trends of the cars in the meeting lane, the expected path of the cars in the right lane, and so on.

3.2.6 Significant events

The information relevant to the decisions to be made by the DSS (whether and how to act) is sent to the event-driven rule layer of our system. Each change of the degree of freeness from one value to another should be recognized as a significant event by the characterizer. Examples of significant events used in safety determination are: lane changes by the own car, changes in the degree of freeness of each lane, changes of speed limit, and changes of configurations. The intention of the own driver is either recognized from the actually observed driver’s manoeuvres or communicated directly by the driver. The output of the characterizer is the a description of the whole traffic scenario in terms of a sequence of significant events.

3.3 The configuration space

In this section we will discuss our choice of configuration space and clarify the rationale behind our decisions. The Own and Meeting configurations are assumed to be always present whereas the Merging and Crossing configurations are present (or relevant) only when the car is approaching an intersection.

3.3.1 The Own and Meeting configurations

The Own configuration space ranges over all subsets of cars having a relative position to the own car (referred also to by A) as depicted in figure 1(a), and includes the own car. Moreover, a car in position Co can be present only if a car in position Bo is. The Meeting configuration space consists of cars occupying
the positions depicted in figure 1(b), but excluding the own car. In this case $F_m$ can be present only when $E_m$ is present. The fact that the Own configuration space includes cars in all the positions surrounding

![Diagram of Own configuration space (a) and Meeting configuration space (b).](image)

the own car’s position should not come as any surprise. The reason for including $Co$ is twofold. Firstly, it is used by $A$ as a look ahead, to be able to adjust its speed in a more smooth manner than would be possible if only $Bo$ were included. Secondly, it is needed for the overtaking support, for instance in order to determine whether there is an empty space in front of $Bo$ or whether we would have to overtake more than one car.

3.3.2 The Merging and Crossing configurations

When approaching an intersection, the Merging and Crossing configurations become very relevant. The candidates for the Merging and Crossing configurations are shown in figure 2. The expected future path of the own car is indicated by a thick arrow, while turning directions of other cars should be understood from the status of their indicators.

For the Merging configuration, we distinguish between left and right merging cars. If $A$ is turning right at an intersection then there are no right merging cars. Similarly, if $A$ is turning left there are no left merging cars, while for a non-turning $A$ there can exist both the left and the right merging members of the configuration.

The Crossing configuration is also divided into two (left and right) parts. In this case we can observe that for a right-turning own car the Crossing configuration is empty, while for a left-turning or non-turning $A$ there might exist both left and right crossing members of the configuration.

It should be noted that this choice of configuration space for merging and crossing cars relies on complete knowledge of the turning intentions of the other cars. When such knowledge can not be assumed to be complete, we have to take the worst-case behaviour of each of the surrounding cars into account.

3.3.3 An example

As an example of how the Own configuration can change over time, consider figure 3 where four snapshots of an overtaking manoeuvre performed by the shadowed car are shown. The figure also illustrates how
Figure 2: The candidates for the merging and crossing configurations when A is not turning (a), turning right (b), and turning left (c). RM and LM denotes a right and left merging car respectively, whereas RC and LC denotes right and left crossing cars.

the deictic reference to the overtaking car is altered as the manoeuvre is carried out. This is of course not the only possible configuration development beginning from (a). If for instance, there were a car behind Do on figure (a), it would have become the new Do in figure (b).

In addition, it is possible to define properties of configurations, that is, properties of some aspect of the surrounding environment. One such property, useful for the overtaking support, is the degree of being free ("freeness") of each lane. If we assume in our example that the Meeting configuration is empty then the "freeness" of the left lane ($llf$) evolves as follows. Beginning from figure 3(a), $llf$ is equal to free until the car signals to indicate an intention to overtake. $llf$ then becomes soon-blocked. When the car changes lane to the left, figure (b), $llf$ becomes blocked. It stays blocked until the car (in figure (c)) signals right before merging with the right lane traffic. At this point $llf$ becomes soon-free. Finally, when the car moves back to the right lane, $llf$ is again considered as free.

### 3.3.4 Relative positions and closeness

After this informal discussion of the configuration space, it remains to specify formally the properties required from a car to be considered as a configuration member. In this context there are two concepts which need to be further specified, namely:
1. The concept of being in some fixed relative position to a car. An example is the relation Following which is used to determine the candidate members for the Own and Meeting configurations. How should this relation be defined, and hence how should the configurations be determined in the presence of splitting and merging points?

2. The concept of being close enough, i.e. how close must a car be to be considered as a configuration member?

**Relative positions**  Is it always clear which is the car in front of our car? Consider an example where A is approaching an intersection and there is one car Bf in the same lane but on the other side of the intersection, one car Br to the right of the intersection and one car Bl to the left. Which one of those is then in front of A and thereby a candidate for being a member of the Own configuration? Of course we could consider all of them being in front of us, but that is somewhat contradictory to the idea of unique configurations. If A is turning right at the intersection, then clearly Br is the one which may influence our behaviour, Bf and Bl are not even on the lanes ever traversed by our car. In a similar way we can see that Bf is the car in front when A is not turning, and that Bl is the relevant car when turning left. This discussion proves that we have to consider the future paths of the cars in order to be able to determine how are the cars spatially related. Since this information is at least partially unknown, we have to either define the paths in terms of the car’s future behaviour, or to rely on educated guessing.

**Closeness**  The second interesting concept pointed out above is that of being close enough. For this reason we need a distance measurement which takes the dynamics and reaction times into account. We therefore define a distance-factor between two cars going in the same direction as the ratio between their longitudinal distance and their minimal safe distance, as follows.

Suppose that the car A is driving \(d\) meters behind car B, and that B suddenly brakes as hard as possible. The velocities of the cars are denoted \(v(A)\) and \(v(B)\), and their maximal deceleration capacity are denoted \(Maxd(A)\) and \(Maxd(B)\), respectively. Then B will stop completely at a position

\[
\text{stop}(B) = d + \frac{v(B)^2}{2 \cdot Maxd(B)}
\]

meters ahead of A’s current position. If the driver of car A needs \(rt\) seconds to react to the braking of B
and then brakes as hard as possible, the car will stop at a position

\[
\text{stop}(A) = rt \cdot v(A) + \frac{v(A)^2}{2 \cdot \text{Maxd}(A)}
\]

meters ahead of its current position. The requirement for a distance to be safe is of course that \( \text{stop}(A) < \text{stop}(B) \). If we suppose that \( \text{Maxd}(A) = \text{Maxd}(B) \), we can derive the minimal safe distance \( \text{MSD} \) from the two above equations by letting \( \text{stop}(A) = \text{stop}(B) \).

\[
\text{MSD} = rt \cdot v(A) + \frac{v(A)^2 - v(B)^2}{2 \cdot \text{Maxd}}
\]

A problem arises when the own car goes significantly slower than the car in front. In this case the \( \text{MSD} \) might be negative, suggesting that we have enough time to stop even if we are in front of the other car. To avoid this seemingly odd interpretation, we consider the maximum of the above formula and 1.0m.

The distance-factor of car A with respect to car B is then defined as

\[
\text{distance-factor} = \frac{d}{\max(\text{MSD},1.0)}
\]

A similar approach to compute a relative safe distance is proposed in [?]. However, their safety factor is included in the distance formula and applied solely to the deceleration capability of the own car. This implies that the actual safety factor decreases (the situation becomes less safe) as the velocity of the own car increases.

The distance factor can then be used as follows.

- \( \text{distance-factor} \leq 1 \Rightarrow \text{Danger! Too close.} \)
- \( \text{distance-factor} > 1 \Rightarrow \text{OK.} \)

The main advantage of the distance-factor as distance measurement is that it can be compared with a constant, even though it includes an estimate of the dynamics of cars and of reaction times of drivers.

In this manner a car is considered as close enough to be included in a configuration if its distance-factor with respect to the own car is below some limit. This limit has to be chosen in such a way that each car which is interesting for some support service, actually becomes a configuration member. Since we prefer stable configurations with as few transformations as possible, two thresholds are used: one for “catching up” with a car and another longer distance for “letting it go”.

Since the braking of a meeting car is more of an advantage than a disadvantage to the own car, the distance-factor of a meeting car has to be defined differently. In this case the definition is based on the (imagined) time to impact. However, since we want to use the distance-factor as a uniform distance measurement for determining closeness between cars, we divide the time to impact by a constant time limit. This time limit is to be chosen in such a way that the same “catching up” threshold as for cars going in the same direction can be used. If \( A \) is meeting with \( E \), the longitudinal distance is \( d \) meters and the velocities of the cars are \( v(A) \) and \( v(E) \), respectively, then the distance-factor of car \( A \) with respect to car \( E \) is defined as

\[
\text{distance-factor} = \frac{d}{t_l \cdot (v(A) + v(E))}
\]

where \( t_l \) is the abovementioned time limit.

Note that only one threshold is needed for determining whether a meeting car is close enough to be included in a configuration.

## 4 Representation

In this section we are going to address the following question: What is the proper way of representing knowledge about traffic behaviour? The meaning of the adjective “proper” can vary depending on who is asking this question. The knowledge engineer will look for a representation method facilitating the process of scenario conceptualization and communication with traffic experts. The software engineer will look for a formalism which would ease the implementation process. The systems engineer will look for a specification formalism enabling efficient verification of the design. Finally, the programmer will look for a language giving as concise and efficient code as possible.

We have developed a series of closely coupled representation formalisms which can serve as tools for all the mentioned purposes. These formalisms have been created on the basis of the vast library of methods
and tools available in Artificial Intelligence and Computer Science. Although the languages described here are fairly generic, they have been created having in mind the particular application area, namely the road traffic.

The formalisms and their intended usage are the following:

1. Process Transition Networks (PTNs) [Mal92b]: a graphical, discrete-event-based formal language with explicit causal dependencies. It is intended to help the knowledge engineer with scenario conceptualization and with communication with traffic experts.

2. Metric Temporal Feature Logic (MTFL) [NT92, Öst93] is a formal language intended for specification of time-dependent systems. It is orthogonal to other formalisms mentioned here.

3. Restricted Elementary Temporal Feature Logic (RETFL) [Mal92b]: a formal rule language. It can be used both for analysis (i.e. verification) of the system, and for the implementation. Its basic advantages are availability of procedural semantics for its formulae, what makes RETFL a perfect programming language (in that sense it can be compared to Prolog, although RETFL is much more restricted in most aspects.) and the full-fledged declarative semantics what makes it suitable as a verification tool for knowledge bases created using PTNs.

4. Configuration Language (CL, [Mal91b]): a regular language for describing configurations of cars at an intersection.

In this section we are going to focus on PTNs and RETFL, and shortly mention the intended usage of CL.

4.1 Scenario conceptualization with Process Transition Networks

In this section we will present Process Transition Networks and the way they are used in scenario conceptualization. The description of the language is rather informal; for more details cf. [Mal92b].

4.1.1 Process Transition Networks

We assume that the world consists of objects (some of which we shall call agents), and that the state of the world (including the state of the agent or agents) may be described using sets of object properties. Values of object properties may change with time. In most cases one of the following two statements is true:

- The set of property values is finite, and the value function is piecewise constant.
- The set of property values is infinite, and is usually continuous. The property’s value may change with time, but except for some distinguished time points, the change is continuous. Thus the value function is piecewise continuous.

A set of adjacent time tokens during which a property value continuously changes (this includes values that remain constant), will be called a circumstance. So, for example, the state of a robot performing some primitive operation (such as polishing a nearly-finished kitchen sink, or searching for an empty soda can), traffic lights being red, a car following another one, an airplane following some air corridor, or a computer being hung, are all examples of a circumstance.

In the PTN language we introduce the notion of a process, corresponding to some circumstance. A process will be depicted by a box. We are interested in those moments in which discontinuities of a property occur, because discontinuities convey important information about changes of the current situation.

We will call such discontinuities events. Events are represented in the PTN language by hexagonal boxes. Since discontinuities (by definition) may only appear between periods of continuous change of some property, events appear only between two processes. Such a transition (from process to event to another process) is denoted by two thick arrows, the first pointing from the “source” process to the event, and the second pointing from the event to another process.

Causal dependencies between discontinuities are represented by enabling arrows, denoted by thin lines and pointing either from an event alone or from an event and its subsequent process, to some other event. The first case corresponds to the situation where one discontinuity forces another one to occur; the second case describes the situation of an actual enabling. Somewhat unfortunately, both are referred to as enabling arrows.
A process that describes the behavior of some agent is called an activity. Activities that have implicit termination conditions (i.e., their termination condition is not explicitly stated in the PTN) are called actions. Activity boxes are distinguished by doubling their vertical borders; each action contains an additional internal box denoting its termination condition. Actions are not followed by events, rather, the event is “encoded” in the termination conditions of the action. However, if it is desired that this event enable some other event, an enabling arrow may be drawn directly from the box representing the termination condition. Figure 4 presents a simple PTN with activity and action boxes. It illustrates the most coarse overview of passing through an intersection.

Activities and actions may be distinguished in an arbitrary way. They are used to focus attention on processes ascribed to some agent or agents. Later on we will assume that every PTN cycle containing an activity consists only of activities. The rationale behind this is that if some process (circumstance) is associated with some property of an agent, then all the subsequent values of that property (and thus all the subsequent processes or circumstances) are also associated with the agent.

Every action box can be seen as a way of encoding an activity and its subsequent event, together with the whole PTN cycle describing the triggering conditions for the event. The main reason for distinguishing actions is to keep PTNs as simple as possible. Moreover, making such a distinction allows us to draw the enabling arrows in an intuitively clear fashion.

In the PTN language, one may combine enabling arrows using both and and or connectives. The and of two enablings is expressed by two consecutive events on one transition between two processes. The or may be represented with two different paths leading from one process to another, but is usually depicted by merging the enabling arrows themselves. See Figure 5.

Another primitive of the PTN language is decision. It is an action performed by an agent faced with several possibilities. Like actions, decisions encode some external condition which is not shown in the network. Thus, by making this condition explicit, a decision box can be rewritten using the primitives defined previously. See Figure 6.

4.1.2 Conceptualization of intersection passing

This section describes a process of conceptualization of the intersection passing scenario. It starts with very simple descriptions, and ends with a relatively complex network of dependencies between various possible “states of affairs”. The final result may be still seen as unsatisfactory for “real” applications, but our main intention here was to exemplify the process of creating such a description rather than to get one ready to be implemented in a new model of Volvo or Peugeot. In order to make the flaws of current conceptualization explicit, we have tried to trace and list all the assumptions being made during the process.

Due to the space constraints there are some gaps in the description provided below. The reader
interested in more details may consult the report [Mal91a], keeping in mind however that it has been written in the early stage of the work reported here.

The present conceptualization relies on the following assumption:

Assumption 1 We are not interested in the behavior of other cars driving in our lane.

When the intersection becomes visible the system switches from the Drive Freely activity to the Approach action. It consists of appropriate slowing down (up to stopping, if needed), looking around in order to determine if passing is possible, and preparing to choose the outgoing road segment (left, right, or straight). After concluding the Approach (ie. coming to the meeting point of the road segment the car is on and the intersection’s virtual segment the driver plans to enter) the system switches to the Pass action, which concludes passing through the intersection and “causes” the intersection to vanish.

Assumption 2 There is at most one intersection visible at any moment.

After Passing the intersection the system switches to the Drive Freely activity again, until it meets the next intersection.
Figure 7: Passing straight.

Figure 7\(^3\) presents a bit enriched analysis, which takes into account type of the intersection and the crossing cars (although in a very simplified way), but restricts the own car’s direction to “forward”.

After recognizing an intersection the system gathers information about its type (either from the on-board vision system, or from some local or remote electronic information system, e.g. a database of road segments). After receiving this information, it switches from Approach (which is now an activity) to some appropriate Pullup activity. As different intersections require different pull-up tactics, we have distinguished several Pullup activities, one for each kind of intersection. Pullup is also an activity, as there is no built-in termination condition for it — it can be finished either by stopping before the intersection, or by passing it, depending on the data gathered during the Pullup from sensing devices.

When the intersection the own car approaches is busy and the own car has to give way to other cars, Pullup is changed to Stop activity (which consists of actual stopping the car at the stop line, and then of continuously observing the environment in order to determine the conditions for Free Pass). If the intersection is free or if the own car has rights of way, it switches to the Pass action, which is terminated by entering the road segment outgoing from the intersection.

One of the paths in the PTN should describe passing through the intersection equipped with functioning traffic lights. As a relatively deep analysis of behavior in presence of traffic lights has been done elsewhere, we omit this path in further diagrams, assuming that its inclusion is straightforward.

**Assumption 3** Traffic lights do not exist.

Until this moment the crossing traffic has been conceptualized as a two-state loop: Free Pass and Crossing Cars. Figure 8 presents an enhancement distinguishing cars coming from left hand side (including cars coming from the opposite direction and intending to turn left), and cars coming from the right hand side. This is important in order to determine the own car’s priority.

---

\(^3\)On this and the following figures we have omitted the (usually redundant) event hexagons except those cases where it could lead to a misunderstanding of a figure.
Figure 9 presents some details of passing different kinds of intersections according to different kinds of crossing traffic. Namely, on the intersection with equal priority for each direction the system should give way to the cars coming from the right-hand side only. On the intersection with “Give way” and “Stop” signs, every crossing car should be given the way, independently of direction it comes from.

The rationale given above reminds about an assumption, which has been hidden until now:

**Assumption 4** We expect that all the cars involved in the crossing scenario are behaving according to the rules of traffic, e.g. we expect that no car can enter an intersection on the meeting lane (in such a case we would have had to consider it when turning right), cars coming from subordinate roads will give way, there are no pedestrians, etc.

Figures 10 (a) and (b) and 11 show the result of allowing the own car to leave the intersection by
whichever exit has been chosen. This reminds us about

**Assumption 5** Intersections are places, where two two-lane roads cross each other.

This means that we deal neither with one-way road cases, nor with T- or Y-shaped intersections. Another assumption concerns road priorities.

**Assumption 6** The main road meets in the intersection two subordinate roads (except the + case, where both roads can be considered main), one coming from left, and one coming from right.

This means, that we exclude K-shaped crossings with subordinate roads lying on the same side of the main road, and their topological equivalents (e.g. + shaped crossing, with main road turning right or left).

Returning to our analysis, we distinguish nine different classes of cars which potentially can cross the intersection in the time the own car is going to do it. This classification is shown on Figure 10 (a).

It is not a configuration characterization — it simply lists all the possible states of incoming cars, as computed by the object characterizer. At this moment we will assume that the configuration can be seen as a list of such object characterizations; in the actual implementation it is done according to the guidelines presented in section 3.

Figure 10 (b) presents the conceptualization of driver’s intentions based on observation of the own car’s blinkers.

Finally, Figure 11 presents the most complete (in this paper) version of the APPROACH—PULLUP—STOP—PASS—DRIVEFREELY path of the constructed PTN. The labels of triggering events (configurations) corresponding to different kinds of crossing cars should be read according to Figure 10 (a).

This concludes the construction of the PTN. As the next step we will show how do we translate it into RETFL formulas, but first we have to define RETFL logic itself.

### 4.2 Formal description with a Temporal Feature Logic

Below we introduce a logic which is a modification of Elementary Feature Logic (EFL), defined in [San94]. The modification consists of both an extension, and a restriction, hence its name. EFL is augmented with a simple temporal operator able to distinguish “now” from “just before now”. The restriction removes
Figure 11: How to pass through an intersection.

most of the classical logical connectives, what constrains the shape of formulae. The resulting logic, Restricted Elementary Temporal Feature Logic (RETFL), appears to be still complex enough to express any possible PTN, but on the other hand is simple enough to be treated as a programming language. In particular, we have implemented an interpreter and a compiler for this language.

The ontology underlying RETFL is that of the PTN language: The world consists of objects having certain properties, and involved in different relationships with other objects. Both the properties of an object and the relations it is involved in may change with time. Therefore a description of the world
at a particular time point consists of data structures describing the objects, their properties, and their relations with other objects. Because contents of these data structures may change with time, they can be treated as fluents, i.e., functions mapping time into data structures. In a general case, values of different fields of such a data structure may belong to arbitrary value spaces. However, many of the fields characterizing an object will be numerical, and moreover, their value space will usually be dense (e.g., rational or real numbers, or vectors). In such cases we may distinguish certain time intervals during which value of some field will change in a continuous manner, and breakpoints between these intervals when discontinuities occur. In the next step we introduce circumstances: functions which are constant on those intervals during which the corresponding field value changes in continuous manner. Circumstances may change values only at breakpoints, so they are piecewise constant. Obviously, PTN processes correspond to circumstances, as do the features introduced in EFL. The value space of a feature corresponds to the set of possible values of the circumstance. Value domains are assumed to be finite sets of objects.

A similarity type for RETFL is a mapping: $\sigma = \{ f_1 : D_1, f_2 : D_2, ..., f_n : D_n \}$ where $n \geq 1$, each $f_1, f_2, ..., f_n$ is a feature symbol, and each of $D_1, D_2, ..., D_n$ is a corresponding non-empty, finite feature domain consisting of items. Items will be written here in lowercase; feature names will be capitalized. $X$ will usually denote a set of feature items, i.e., $X \subseteq D_i$ for some $D_i$.

An elementary formula in RETFL with similarity type $\sigma$ is an expression either of the form $f \in X$ or of the form $f \in X'$ to be read “value of $f$ belongs to $X$”, or “value of $f$ becomes a member of $X$”, respectively, where $f$ is a feature symbol defined in $\sigma$ and $X'$ is a subset of the domain assigned to $f$ by $\sigma$.

An and-type formula in RETFL with similarity type $\sigma$ (or and-formula for short) is either:

- an elementary formula of the form $f \in X$, or
- a formula composed from two and-formulae using the logical connective $\land$ (in the standard way), or
- a formula built from another and-formula by preceding it with unary connective $\lambda$, i.e., if $F$ is an and-formula, then $\lambda(F)$ is also an and-formula.

An arrow-free formula in RETFL with similarity type $\sigma$ (or af-formula for short) is either:

- an elementary formula $f \in X$, or
- an and-formula, or
- a formula composed from af-formulae using the logical connective $\land$ (in the standard way).

An or-type constituent in RETFL with similarity type $\sigma$ (or or-constituent for short) is either:

- an elementary formula of the form $f \in X$, or
- a formula composed from two or-constituents using the logical connective $\lor$ (in the standard way).

Finally, a well-formed formula in RETFL with similarity type $\sigma$ (logic formula, or simply formula) is defined as follows:

1. An arrow-free formula is a logic formula;
2. If $F$ is an af-formula, and $G$ is an elementary formula, then $F \Rightarrow G$ is a logic formula;
3. If $F$ is an elementary formula of the form $f \in X$ and $G$ is an or-constituent, then $F \Rightarrow \lambda(G)$ is a logic formula;
4. Only the formulae constructed according with (1)—(3) are logic formulae.

The semantics of the language has been defined in [Mal92b]. Speaking informally, the meaning of the basic constituents of the language are as follows:

- if $F$ is an elementary formula of the form $f \in X$, then it means “feature $f$ has now a value belonging to the set $X$”;
- if $F$ is an elementary formula of the form $f \in X'$, then it means “feature $f$ obtains now a value belonging to the set $X'$”, so in addition to the previous case, we know that the value of $f$ in previous moment of time has been different from the current value;
- if $F$ is a formula of the form $\lambda(G)$, then it should be read as “previously $G$”
A PTN can be transformed to RETFL formulae in two phases [Mal92b]. The first one is simplification, defined as redrawing the PTN using only limited set of primitives. The second phase is the actual translation algorithm, taking as input a simplified PTN, and yielding a set of RETFL formulae. Several examples of the formulae resulting from the translation of the PTN describing the intersection passing scenario will be given in the last section. It can be proven that the resulting RETFL formulae have the same meaning as the original PTN.

4.3 Configuration description

One of the very specific problems to be solved during the design of an autonomous vehicle was choosing an appropriate way of describing a configuration of cars at an intersection. Such a description is stored as a value of some slot, say configuration, of an intersection frame. This value should inform about the current configuration of cars coming from each direction and intending to make some specific manoeuvre at the intersection. In order to be able to describe an arbitrary configuration of cars approaching an intersection, we have defined a special regular language which we call configuration language (CL).

Assuming that \( C = \{ l_1, l_r, l_s, r_t, r_t, r_s, f_t, f_r, f_s \} \) is the alphabet, the configuration language \( CL \) can be defined in the classical recursive way:

1. \( \emptyset \in CL \);
2. \( a \in CL \) for every \( a \in C \);
3. If \( a_1, a_2 \in CL \), then also \( (a_1 + a_2) \in CL \) (called alternative or sum and denoting “either \( a_1 \) or \( a_2 \)”);
4. If \( a_1, a_2 \in CL \), then also \( (a_1 * a_2) \in CL \) (called concatenation and denoting “\( a_1 \) and then \( a_2 \)”);
5. If \( a \in CL \), then also \( (a^+) \in CL \) (called iteration and denoting “concatenation of arbitrary number of \( a \)-s, including 0”);
6. Only the strings constructed according to \( 1 - 5 \) belong to \( CL \).

We will omit parentheses when that will not lead to ambiguity.

In addition, we introduce the standard abbreviation \( a^+ = a(a)^+ \), plus several useful, domain-specific abbreviations: \( l = (l_1 + l_r + l_s) \), \( r = (r_t + r_t + r_s) \), \( f = (f_t + f_r + f_s) \), \( q = (l + r + f) \).

Having defined the syntax of \( CL \), let us exemplify the intended meaning of the language.

- \( \lambda \) — the roads coming to the intersection are empty;
- \( l \) — a car is coming from the left side;
- \( l_t \) — a car is coming from the left side and intends to turn to its left;
- \( l_s, l_r, f_t, f_s, f_r, r_t, r_r, r_f, f \) — similarly;
- \( l_f r r, r f r l, f r r l \) — there are four cars approaching the intersection: one from the left side, one from the front, and two from the right side;
- \( l_r l \) — there are two cars coming from the left side, and the first one is going to turn to its right;
- \( (r + f)^* \) — there is no car coming from the left side;
- \( (r + f)^+ \) — there is no car coming from the left side, but there is at least one car approaching the intersection from some other direction.

One can even state interesting properties of some particular car. For example, “The second car from the left side intends to turn right” would be written as: \( (r + f)^* l (r + f)^* l q^* \).

Saying something about both the second car from the left side and the second car from the right side becomes a bit complicated. “The second car from the left side intends to turn right, and the second car from the right side intends to turn right” would have to be written as follows: \( f^* r f^* r, f^* l f^* l q^* + f^* r f^* l f^* l f^* r q^* + f^* l f^* r f^* r q^* + f^* l f^* r f^* r f^* r q^* \). However, in this particular application we usually state the properties of all the cars coming from some direction, so the problems with saying something specific about the n-th car in a row can be neglected.

We have designed and implemented an algorithm that constructs a nondeterministic finite state automaton accepting a given regular language. It allows the rule interpreter to check, for any given configuration, whether it belongs to the set of “acceptable” ones (in terms of safeness, or the rights of way), or
not. This approach allows us to parse incrementally a description of the current configuration: if some additional information about the configuration becomes known (some letter is appended to the end of the appropriate expression) then we can proceed with parsing it, instead of re-parsing the first part of the description.

5 Implementation

5.1 Architecture of the system

The design of the driver support system is based on the three-layered software architecture (depicted in Figure 12), for intelligent autonomous systems. It has been developed in our group since 1988 [SHT88, HNS89, San90, MS90, MNTOS92]. The three layers are separated on the basis of the type of computations they are intended to perform. The first layer (process layer) performs periodic numerical computations at predetermined frequencies and would typically host the algorithms of control engineering, included both for identification and for control. Also, the process layer accommodates software that filters sequences of sampled sensor data to recognize significant events that should be sent to the other layers. The middle layer is called the discrete response layer (or in some of our previous publications the rule layer), and computes the response to asynchronous events which are generated in the process layer. One type of response is to change the mode of the process layer due to the change of mode in the environment. The last layer (analysis layer) handles symbolic reasoning such as prediction, planning, and replanning. This layer would also host eventual learning mechanisms which might adapt the discrete response layer while maintaining its critical response requirements.

The computational model assumed for discrete response layer is that of discrete event systems (DES). There exist several equivalent DES formalisms: automata, transition systems, rule-based systems, etc. All of them distinguish the notions of state and transition as central, although the details vary from model to model. We have adopted the rule-based approach as the most convenient for the purpose of specification of a knowledge-based DES. This does not preclude usage of other approaches, what has been shown in [Mal92a, SM91, Mal92b].

During our research done within the Prometheus programme we have thoroughly studied this architecture and its implications on software engineering issues [MNTOS92]. One of the conclusions was that it facilitates prototyping of systems, especially because it allows development of generic software tools which can be used for implementation of each particular application system. Along this line we have developed software kernels, or engines, for development and implementation of the process layer and the discrete response layer. The set of tools includes:

- Process Layer eXecutive (PLX) [Mor93]: a multi-threaded time-triggered real-time engine for implementation of process layer software. It has been implemented on a VME-bus system equipped with the pSOS+ real-time operating system, and on a PC-compatible with VDX operating system. There also exists a simulator of the PLX which runs under Unix;
- Process Layer Configuration Language (PLCL) [Mor91b, Mor90] and its compiler: a language for specification of process layer module interconnections and interfaces to both sensors and actuators on one side, and to the rule layer software on the other side. The modules themselves are programmed in some conventional language such as C or C++;
- Rule Layer eXecutive (RLX) [Mor92b]: an engine for implementation of rule-based discrete-event systems. It has been implemented both on a PC with VDX and as a LISP program on a Unix workstation;
- Rule Layer Configuration Language [Mor92a]: a logic-based rule language for declarative description of discrete-event control imposed on process layer by the discrete response layer.

Similar layered architectures have recently been used for implementing autonomous real-time systems by Bonasso [Bon91], Connell [Con92], Gat [Gat91] or Steels [Ste93]. Although there are differences both in the way of assigning various tasks to different layers and in the way the overall control of the system is executed, the general conclusion is that such layering is beneficial, if not necessary, in designing autonomous systems.

Below we shall describe some of the modules of the DSS development system in more detail.
5.1.1 The Sensor Simulator Module

In order to provide stable inputs for repeatable tests of the driver support system software, a graphical sensor simulation tool has been developed. The task of the simulator is to create a periodic fluent containing the outputs (e.g. position, signals) of the sensor system to be simulated. Note that we are only concerned with the outputs of the sensor system, not with how the results are obtained.

Each run of the simulator generates a file of simulated sensor data for one car. Scenarios with several cars are created in an incremental fashion. Previously recorded data is reused by the simulator to represent the presence of other cars.

The simulator is interactive in the sense that the user controls the currently simulated car by pressing keys on the keyboard at the same time as the simulated car and its environment are animated on a graphical display.

5.1.2 The Process Layer Executive

The process layer executive facilitates the implementation and maintenance of the hard real-time parts of the application which perform the transformations of periodic fluents in the characterizer and effector blocks.
During processing, fluents are represented by a dual state vector. The dual state vector is a global data structure in which every data item is stored in two versions, thus forming two subvectors; one which is read-only and one which is write-only. The values in the read-only vector represent either sensor readings or internal state. The write-only values represent actuator outputs or new internal state.

The use of a dual state vector for representing the fluents in the process layer allow the computation to distinguish unambiguously between the new and the old value of a parameter. If one just has one memory location for each parameter, it becomes necessary to choose the right sequencing of the transformations to obtain the correct results. By having a data representation that distinguishes explicitly between the old and the new value it becomes unimportant in which order the different state-vector components are calculated.

A process layer application defines a sequence of transformations that should be applied to the read-only vector in order to compute the new values of the write-only vector. Since sensor values are read periodically, the transformations have to be applied with the same periodicity. When the transformations have been applied, actuator outputs are flushed to devices and the values of the write-only vector are copied to the read-only vector, thus establishing the new state of the process layer. This operation is referred to as a flip of the state vector.

The Process Layer Executive (PLX) [Mor93] is a software tool which maintains a dual state vector. This task includes performing the flips, and reading sensors and writing to actuators using user defined access functions. In addition, the PLX executes the transformations on the state vector in a periodical fashion. The PLX supports decomposition of the dual state vector into several subvectors, each of which has its own period, forcing transformations to be executed at different rates.

The Process Layer Configuration Language (PLCL) [Mor91b] is used to define the components of the dual state vector and to configure the user supplied library modules which implement the transformations on the fluents. Moreover, the PLCL description forms the basis for a worst case execution timing analysis of the application [Mor91a].

5.1.3 The Rule Layer Executive

The rule layer executive facilitates the rule-based discrete fluent to discrete fluent transformations of the selector block. The fluent is represented there by the time dependent values of a set of slot-records. As is the case in the process layer executive, it is important to be able to unambiguously distinguish between the new and the old value of a slot. The discrete nature of the rule layer fluent suggests that a slot should be updated only when its value changes (in contrast to the periodic updates in the process layer). To still be able to distinguish the old values from the new ones, each slot-record is realized as a data structure containing the new and old value as well as the time for the update.

A more operational view of the rule layer fluent may be described as follows. The fluent is uniquely determined by the initial slot values and a sequence of time-stamped assignments, where each assignment states how the value of a particular slot should be changed. Such assignments are the basic building blocks from which the rules are defined. A rule specifies dependencies between slots, typically of the form that if the value of a particular slot is changed in a certain way, then the value of another slot should also be changed as a result. Internally, each such assignment may trigger consequences, which are new assignments. This forward chaining process may continue in several steps.

We take an object-oriented view of the rule base in the sense that we associate a set of rules with each slot. This view facilitates the flexibility and maintainability needed for complex systems.

The Rule Layer Executive (RLX) [Mor92a] is a software tool which has two major tasks. Firstly, it defines and maintains the set of slots and rules which determines the behaviour of the rule layer. Secondly, it performs the forward chaining of rules. The forward chaining is typically initiated by an assignment “sent” from the process layer. The outcome of the forward chaining process may be an assignment “sent” to the process layer. In the process layer, those update requests may be used as parameters in control algorithms or for choice of control algorithm.

The Rule Layer Configuration Language (RLCL) [Mor92b] is essentially a syntactic variant of RETFL (Section 4.2) and is used for defining behaviour of the rule layer.

5.1.4 Inter-layer Communication Package

The information flow between layers is characterized by a discrete fluent which determines the value of a set of slots (in the other layer).

The nature of the fluent enables us to reduce the communication between the layers to messages indicating that a discontinuity has occurred in one of the slots. Consequently, we have implemented the
fluent as a flow of messages. Each message is a statement representing a change in a slot value, i.e. an assignment.

The fluent exists at three locations of the software system, namely:

- In the process layer;
- In the inter-layer channel;
- In the rule layer.

Different representations of the fluent are needed in different locations as well as transformations from one representation to another.

In the process layer, the fluent is represented by a set of parameters in the dual state vector. When the value of one of the parameters changes, a time-stamped assignment is prepared and sent to the inter-layer channel. The value of the attached time-stamp is determined by the number of flips performed. The inter-layer channel transmits assignments between the layers. The internal representation of an assignment in the inter-layer channel depends on the medium used for communication, e.g., shared memory, serial link, ethernet, CAN-network. In the rule layer the assignment is received and the forward chaining initiated.

Communication from the rule layer to the process layer is handled in a similar fashion. A discontinuity in a slot of the rule layer fluent causes an assignment to be prepared and sent to the process layer through the inter-layer channel.

5.1.5 The Road Segment Data Base Manager

The Road Map Data Base contains all static information about the road network that is needed in the driver support system. A road map is represented as a hierarchical data structure, which on the top level consists of intersections and streets connecting them.

For efficiency reasons, the road map must be manipulated in main memory. However, in examples of realistic size the road map has to be subdivided into partitions that are small enough to be loaded into main memory map cache. The Road Map Data Base Management System is responsible for keeping a partition of the road map which covers the immediate surroundings of the own car in the cache. As the car moves around, the contents of the cache is updated accordingly.

5.2 Implemented prototype of a Driver Support System

In this section we set out the methodology for separate development of application components prior to their integration. As a working example we have chosen the prototype driver support system presented at the Board Members’ Meeting 1991\(^4\). Because at that time we did not have access to sensor data gathered in real traffic situations, we have used a simulated set of sensor outputs for a road environment consisting of several straight road segments, bends, and intersections (cf. Figure 13). In this environment we simulate movements of three cars: the own car, denoted by a rectangle around the mark, and two other road users. The information about positions of all the cars, plus other important information about the own car is then input as simulated sensor data to the process layer of the DSS. This way of providing the input data allows us to adjust to any possible kind of input sensory data, independently of its source. We can imagine using processed vision information, messages sent via radio link either from the environment or by other road users, low-level sensory information about both the car and the environment, etc.

The process layer is equipped with the road-segment database encoding the “static” part of environment description. The “dynamic” part is formed from the preprocessing of input data, and is stored as the state vector consisting of all the state variables required to provide meaningful support to the driver. Some of the state variables, as e.g. the classification of the configuration of cars in the current situation, or the last moment to slow down before some dangerous area such as an icy bend, require non-trivial amount of processing to be performed in real time.

One of the most computationally demanding processes, and in the same time most frequently used one is coordinate transformation. As the input informing about the own car’s position we take the incremental data providing difference between previous and current positions of the own car, with reference to the road segment the car is currently on (in other words, the position is expressed in road-segment coordinates). On the other hand, distances between the trafficants are computed using the Cartesian coordinates (and sometimes even collected in this form). Therefore each time new data about the own car’s position is obtained, it must be recalculated into Cartesian coordinates, and then for all other trafficants back to

road-segment based ones. Due to hard real-time constraints this task must be as efficient as possible, and therefore it has been given much attention during implementation of the DSS prototype.

The configuration of transformations which have to be implemented in the process layer, and the data paths between them, can be specified using the Process Layer Configuration Language. The actual code of the processes is implemented in the C language.

The information relevant to the decision to be made by the DSS (whether to act, and how to act) is sent to the rule layer of the system. Examples of such relevant pieces of information are: lane changes by the own car, changes of traffic condition in each lane in terms of degree of being free during the overtaking manoeuvre, changes of the speed limit.

The information about both how to act in different traffic conditions, and how to support the driver in case he is not acting according to the prescribed rules of behavior, is encoded in the rule layer using a set of rules. The intentions of the driver are currently recognized from the set of observed manoeuvres (e.g. lane changes, accelerating, braking), and some additional data (e.g. blinkers of the own car). If the driver’s behavior apparently differs from the expected one, and if this may lead to dangerous situation, the decision of issuing a warning message is taken, and is sent to the driver. In our implementation these messages appear in fields of the windows presented in Figure 14. (Some of the slots in windows presented in Figure 14 are just reporting current values as seen by the rule system.)

For example, if the current situation with respect to overtaking is classified as unsafe (the driver is trying to overtake the car in front in spite of presence of the meeting car), this information is passed to the driver, with possibility of increasing urgency of the messages (“soon-unsafe”, “unsafe”, “emergency”). The same applies to speed-related warnings. When the DSS has decided that the driver should have started slowing down before a speed limit (due to a bend, or a sign, or some other conditions) it increases the intensity of warning (“soft” meaning you should start slowing down now not to drive uncomfortably, “firm” meaning “brake firm, or you’ll be in big trouble”, and “alarm” meaning “you ARE in a big trouble already” — this might correspond to some intervention taken by the DSS itself, as e.g. braking).

The set of rules encoded in the rule layer has been created in two stages. First, the set of possible
traffic conditions has been specified using Process Transition Networks, as shown in Section 4.1.2. The intended behaviour of the own car has also been specified using the same tool. Finally, the support actions for various dangerous situations the driver may encounter have been encoded. The PTNs have then been translated into RETFL formulae. After checking the consistency of the set of rules, they have finally been compiled and loaded into the system.

5.3 Examples

We will list here only a few of the RETFL formulae (and the corresponding code) resulting from translation of the PTN shown on Figure 11. The full set of formulae consists of a couple hundred of rules. We hope that the example is self-explanatory. The rules given below correspond to the following transitions:

- The type of the intersection becomes determined by external sensory devices:

  \[ \lambda(state - me \in \{Approach\}) \land type - intersection = \{MR\} \Rightarrow \]
  \[ \Rightarrow state - me = \{Pullup - MR\} \]

  or, as the PLX input:

  WHEN type-intersection == MR
  IF state-me /= Approach
  THEN state-me := Pullup-MR

- If there are cars coming from the front, then the own car must stop:

  \[ \lambda(state - me \in \{Pullup - MR\}) \land \lambda(intended - turn - me \in \{L\}) \land \]

Figure 14: The windows of the rule layer.
& dist - to - inters - me • (Close) & configuration = (r + l)∗fq∗ ⇒
⇒ state = me • {Stop - MR - L}

WHEN dist-to-inters-me • (Close)
IF state-me |= Pullup-MR AND intended-turn-me |= L AND
  test(configuration) |= ((* (+ r l)) f (* q))
THEN state-me := Stop-MR-L

• If there are no cars coming from the front, then the own car may pass:
λ(state – me • (Pullup - MR)) & λ(intended - turn – me • (L)) &
& dist - to - inters - me • (Close) & configuration = (l + r)∗ ⇒
⇒ state = me • {Pass - MR - L}

WHEN dist-to-inters-me • (Close)
IF state-me |= Pullup-MR AND intended-turn-me |= L AND
  test(configuration) |= ((* (+ r l)))
THEN state-me := Pass-MR-L

The formulae given above are then directly used by PLX to control the behavior of the implemented prototype Driver Support System.

6 Conclusions

A really useful co-pilot should be able to provide the driver with accurate, reliable, necessary and sufficient, and timely support on the basis of unreliable and limited input information. This information is expected to be generally available in the next-generation cars. The problem is not to acquire enough information, but to process it appropriately and then present to the driver. The characterization of the driver support can be summarized in the following list of requirements. Messages to the driver should be:

• not too frequent;
• not too long;
• not over-informative;
• relevant.

Another dimension of complexity is introduced by the fact that the hardware devices and the software in test cars are changed relatively often. Hence, a DSS has to function in a development environment and under the assumption of incrementality, both in terms of hardware (new modules are added) and software (new functions for the existing modules are implemented).

A driver support system requires distributed computations in real time. The use of different computing resources and the nature of performance constraints on the sub-components dictates solutions where individual components operate semi-independently. But a satisfactory integration of the sub-components is only possible if there is a unifying framework already at the design stage.

The different nature of tasks performed by each component in turn allows the development of the components in an independent fashion. Hence, there is a need for definition of connection interfaces, and a methodology for development which facilitates systematic generation and pre-integration testing of individual components.

We have proposed the use of a layered software architecture for driver support systems. The development of prototypes requires a number of tools each supporting a particular type of computation. In this paper we have described the tools implemented within our group, and indicated the unifying framework for the independent tools. The results of this approach are promising. We have developed a methodology which provides the possibility of systematic validation of individual software components (process layer’s processes, rule layer’s behavior specifications, and possible support activities) before integration into a complete system. This solution seems very suitable since we have to deal with evolving systems whose
structure is likely to change quite often. The main advantages of this approach are also: easy adaptation to new and/or refined functions, and portability.

One of the topics thoroughly studied in the paper was real-time characterization: the process of recognizing substantial changes in the traffic environment and of forming a global description of these changes. We have presented several representation languages supporting systematic specification and validation of knowledge about traffic scenarios. We have also described the first version of a support system, developed within the layered software environment. Initial experiments allow us to expect that the real time constraints can be satisfied. We aim at obtaining a predictable (hard real-time) system, and we plan to concentrate more on this topic in the future.

The operation of the support system within the environment was demonstrated at the Board Members' Meeting in Torino in 1991.

Acknowledgments

The research described here is a result of collaborative work of some of the current and previous members of the Laboratory for Representation of Knowledge in Logic, Department of Computer Science, Linkping University. The research on DSS described in this paper has been done, besides the authors, by Reza Hamidi-Farahani, Magnus Morin, Simin Nadjm-Tehrani and Erik Sandewall. Johan Hultman, Anders Nyberg and Mikael Svensson have contributed to the development of the software architecture.

We are grateful to Ivan Rankin who has helped us with correcting some of the language errors.

This research was supported by the Center for Industrial Information Technology (CENIIT), by the IT4 research program, and by the Swedish National Board for Industrial and Technical Development (NUTEK) grant 92-05768.

References


