How to Pass an Intersection

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

April 15, 1996

Abstract

An intelligent co-pilot system must possess an extensive knowledge about possible traffic situations and scenarios. It must also be able to evaluate and predict driver's behavior in those situations, in order to provide support, either in form of useful help messages, or by producing meaningful and accurately timed warnings.

This paper presents in this context an analysis of the *intersection passing* scenario. To formulate it, and to verify the initial conceptualization, we use the Process Transition Networks (PTNs, [Mal91d]). The PTNs can be then translated into another formal language, namely Restricted Elementary Temporal Feature Logic, which is the programming language for the rule layer of a prototype co-pilot system [MFM+91].

The analysis also presents a method of characterizing the configuration of cars on an intersection. The method exploits a regular language to describe configurations, and finite automata to recognize them. This introduces the possibility of incremental growth of configuration description, and incremental recognition, which are an advantage for real-time characterization.

1 Introduction

An intelligent co-pilot system must possess an extensive knowledge about possible traffic situations and scenarios. It must also be able to evaluate and predict driver's behavior in those situations, in order to provide support, either in form of useful help messages, or by producing meaningful and accurately timed warnings.

This paper presents in this context an analysis of the *intersection passing* scenario. The purpose of this analysis is twofold:

1. to get insight into the complexity of possible situations that might occur on an intersection, and to describe the set of decisions which may be taken by the driver, and/or by the co-pilot system;

2. to illustrate a methodology of analysis, supported by some of the tools which have been created by our research group.

*This research was supported by the IT4 research program.

To formulate the scenario, and to verify the initial conceptualization, we use the Process Transition Networks (PTNs, [Mal91d, Mal91c]) — a graphical language for describing event-driven systems with cyclic occurrences of processes, and with complex causal dependencies. The PTNs can be then translated into another formal language, namely Restricted Elementary Temporal Feature Logic (RETFL, [Mal92]), which is the programming language for one of the layers of a prototype layered co-pilot system [San90]. This layer is used for implementing decision making procedures associated with various scenarios, and the driver support functions based on discrepancies between the driver's reactions, and the system's suggestions. The autonomous decision making, and the driver support modules can be tested separately, using artificially created streams of input data, before being merged with other modules of the co-pilot system.

The presented analysis also introduces a method of characterizing the configuration of cars approaching an intersection [Mal91b]. The method exploits a regular language to describe configurations, and finite automata to recognize them. This introduces the possibility of incremental growth of configuration description, and incremental recognition (parsing), which are an advantage during real-time characterization.

The paper is divided into five sections. The first one describes the conceptualization process utilizing PTNs as the main representation formalism. The next one presents RETFL. The subsequent section discusses issues related to configuration description and recognition. Finally, the last section shows how one can proceed from PTNs via RETFL formulas to the actual implementation.

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 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 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 exist 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.

Figure 1 presents the most coarse overview of passing through an intersection. 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.*

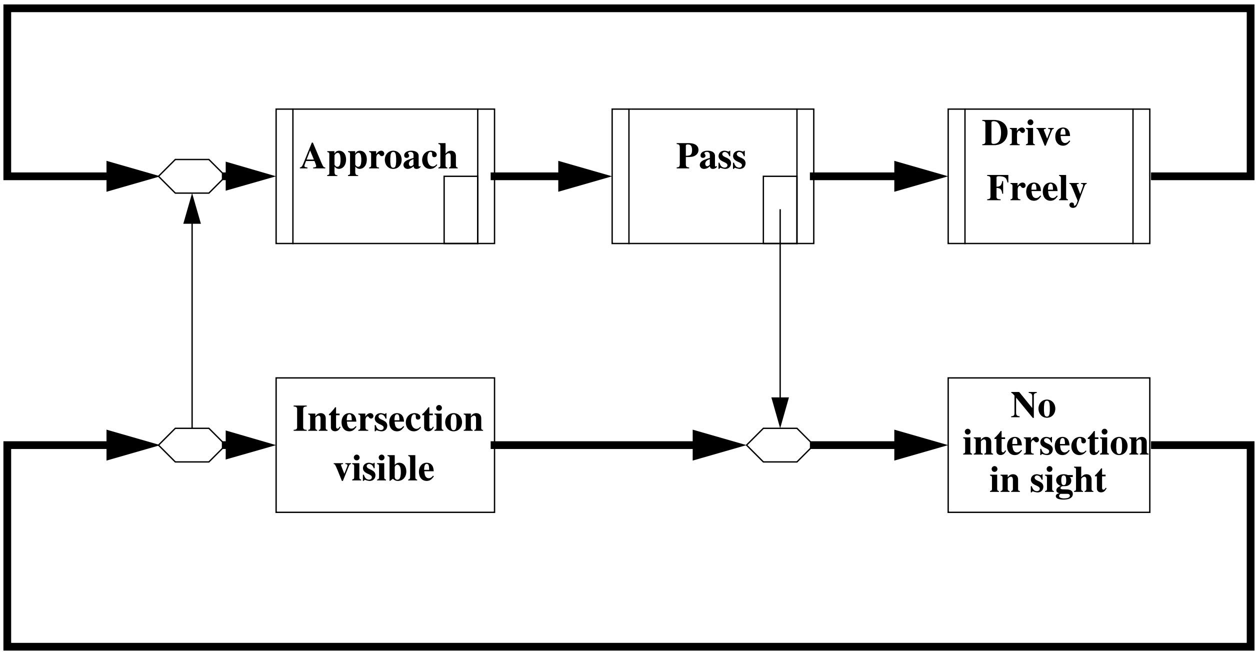


Figure 1: Passing through an intersection.

After PASSing the intersection the system switches to the DRIVE FREELY activity again, until it meets the next intersection.

Figure 2 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 seeing an intersection the system obtains information about its type (either from the visual data — the road signs, etc., or from some electronic information system, eg. the database of road segments). After receiving this information, it switches from APPROACH (which is now an activity) to the 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 to is busy and it has to give way to other cars, PULLUP is changed to STOP activity (which consists of actual stopping the car on the stop line, and continuous observation of the environment in order to determine the conditions for FREE PASS). If the intersection is free or if the own car has the priority, 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 already been done [Mal91d], we omit this path in further diagrams, assuming that its inclusion is easy.

Assumption 3 *Traffic lights do not exist.*

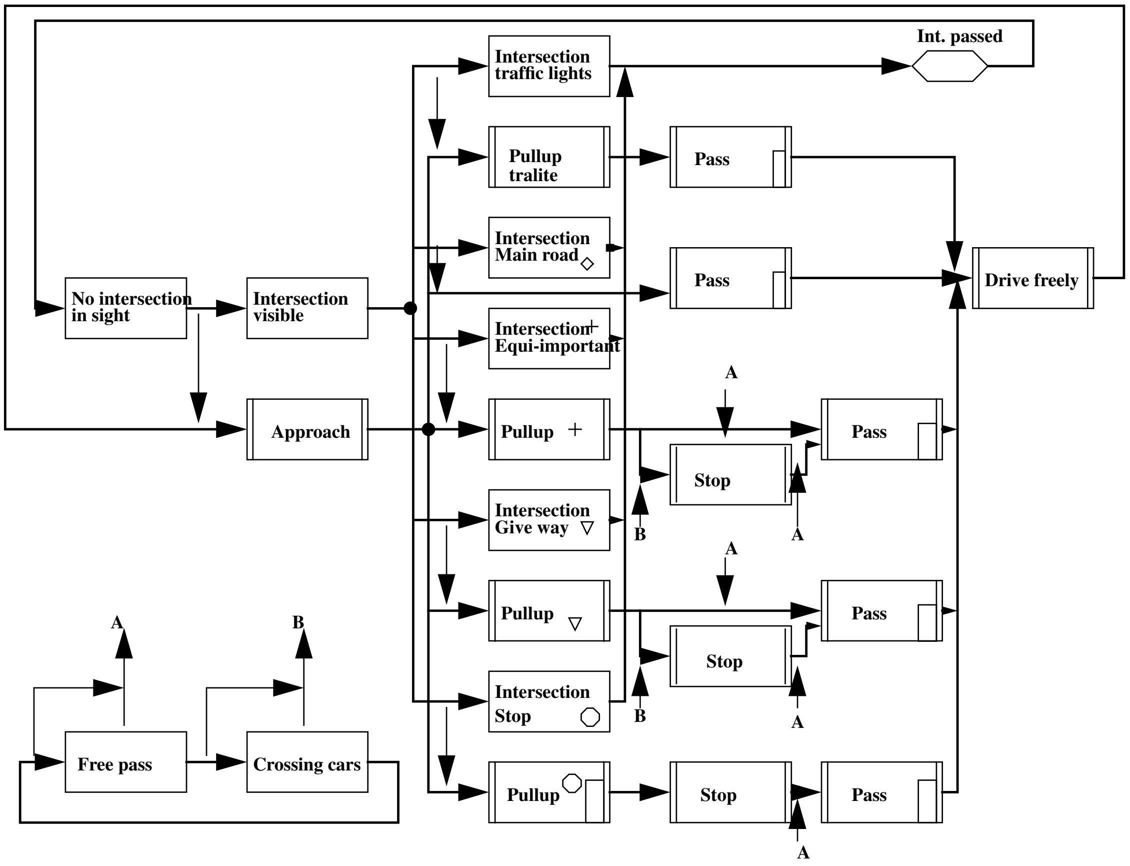


Figure 2: Passing straight.

Until this moment the crossing traffic has been conceptualized as a two-state loop: Free Pass and Crossing Cars. Figure 3 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.

Figure 4 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, eg. 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 nor bikers crossing, etc.*

Figures 5, 6, and 7 are the result of allowing the own car to leave the intersection by whichever exit has been chosen. This reminds us about

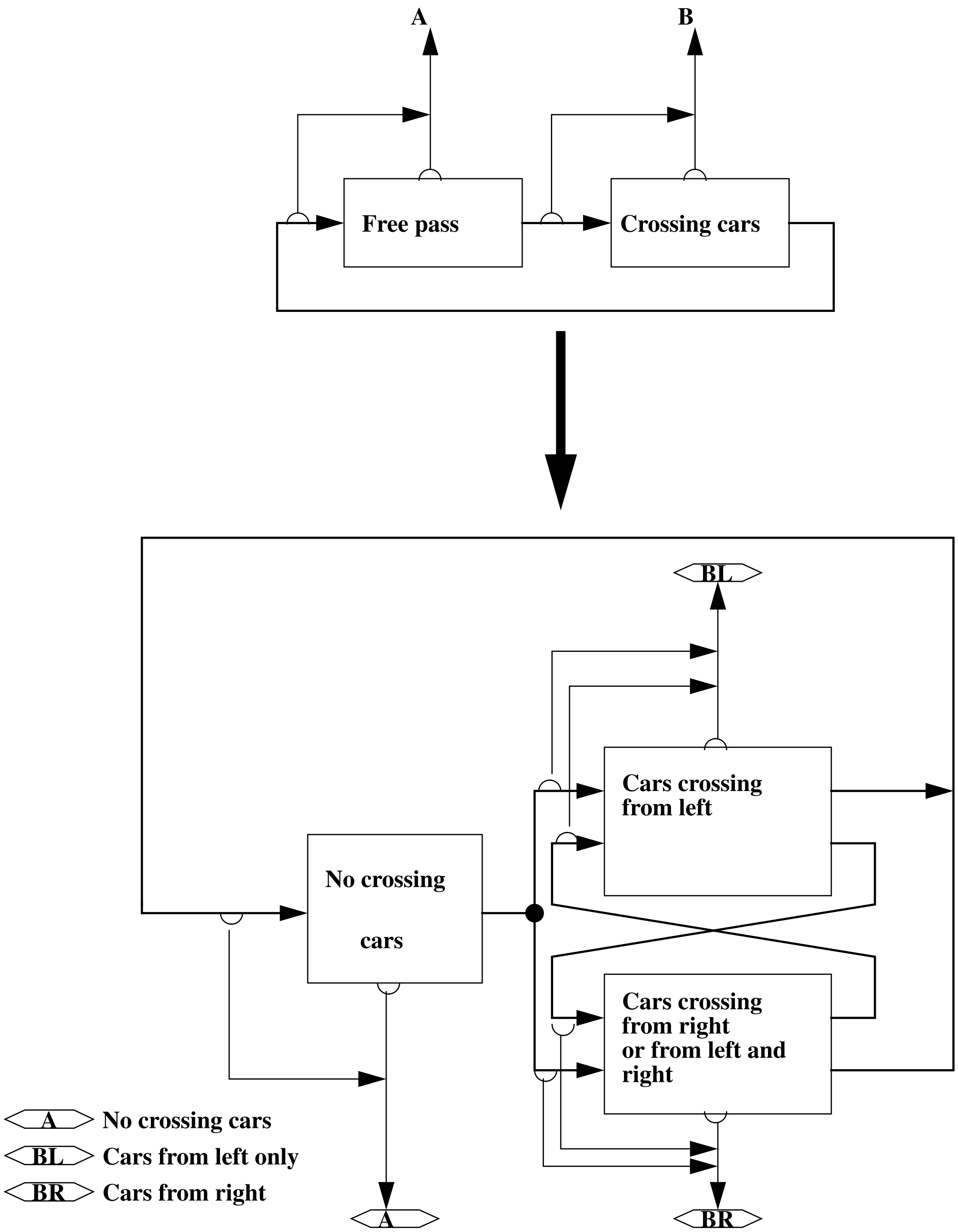


Figure 3: Crossing cars configurations (version 1).

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.

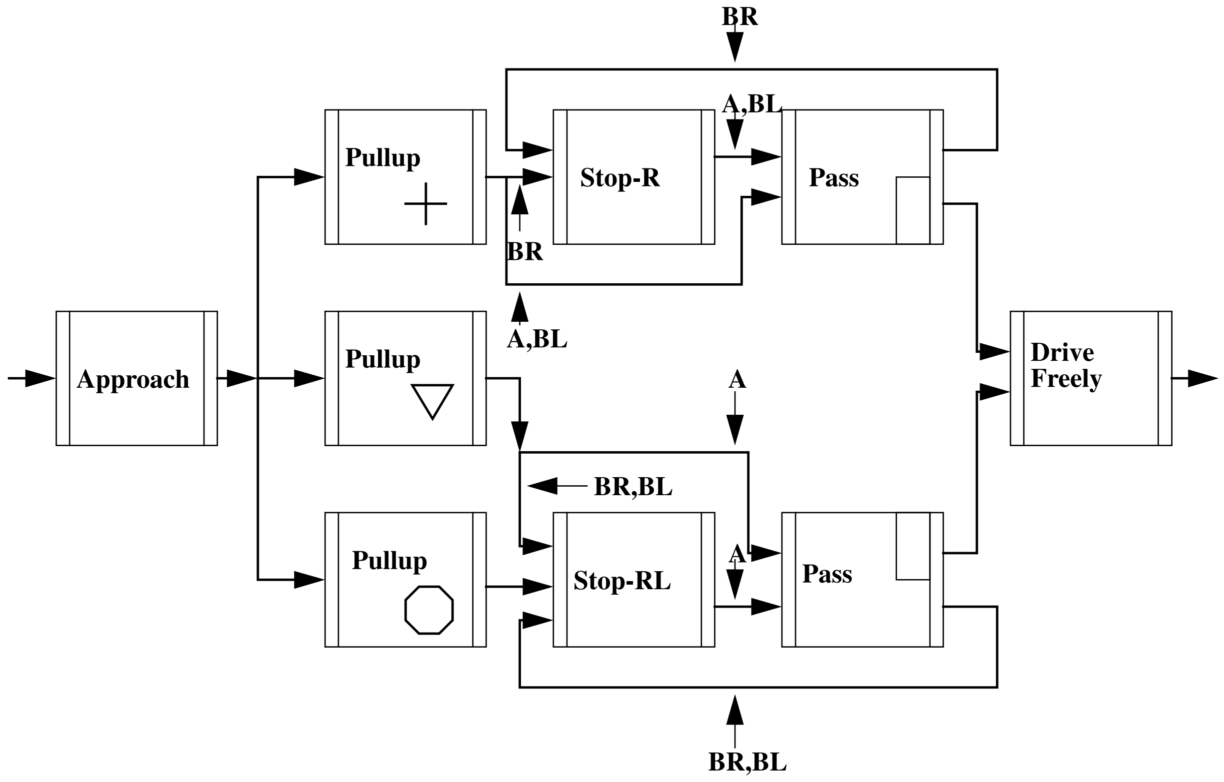


Figure 4: Details of "Stop" and "Pass", driving straight.

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 (eg. +-shaped crossing, with main road turning right or left).

Coming back 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 5. 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; discussion of details will be delayed until later sections.

Figure 6 presents the conceptualization of driver's intentions based on observation of the own car's blinkers.

Finally, Figure 7 presents the most complete version of the APPROACH—PULLUP—STOP—PASS—DRIVEFREELY path of the constructed PTN. We assume that the connections with external processes still hold. The labels of triggering events (configurations) corresponding to different kinds of crossing cars should be read as follows:

A – no crossing cars;

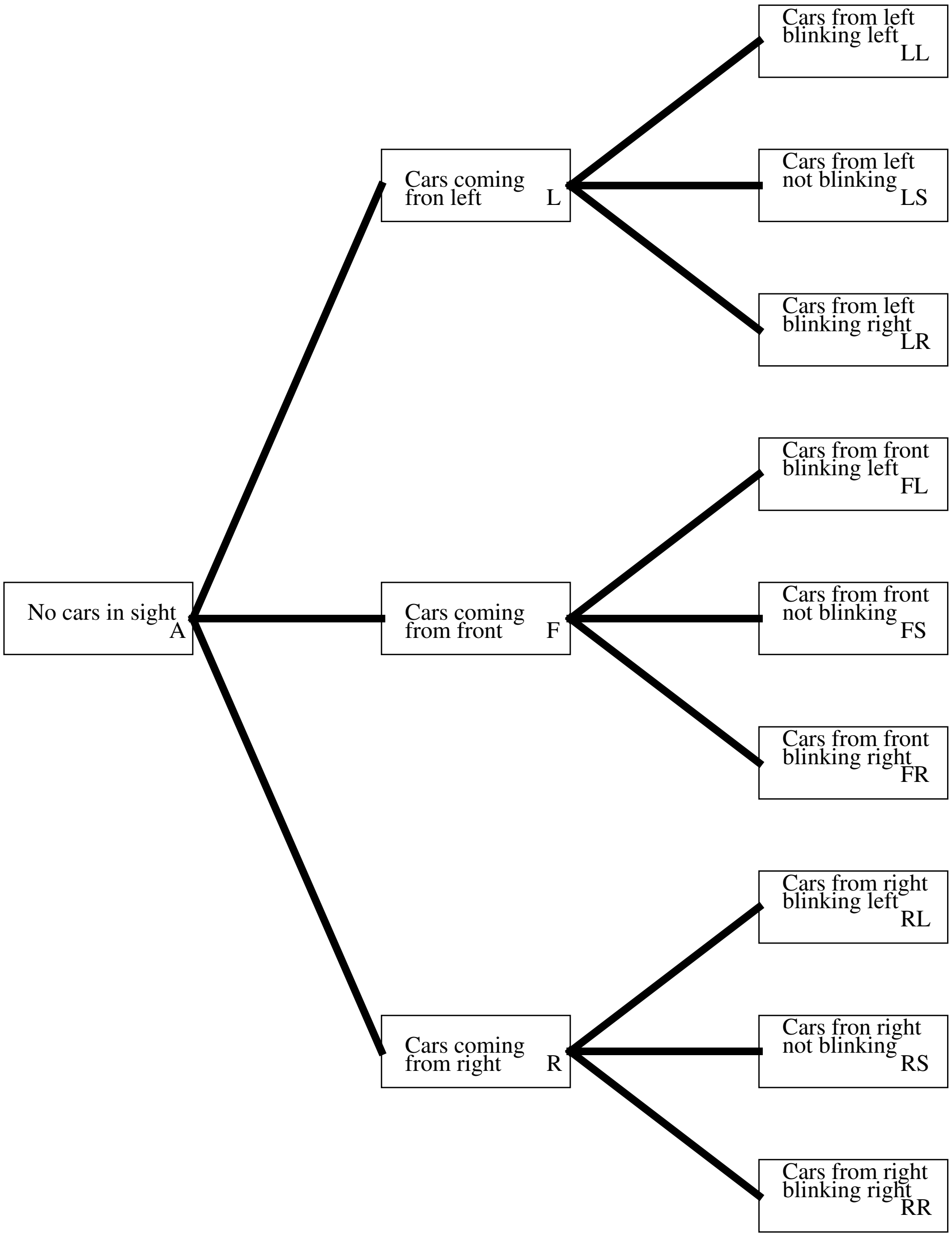


Figure 5: Crossing cars configurations (version 2).

RL (RS, RR, FL, FS, FR, LL, LS, LR) – there is at least one car having the RL (RS, RR, FL, FS, FR, LL, LS, LR) property among the cars approaching the intersection from other directions.

R (F, L) – there is at least one car with either an RL, RS, or RR (either FL, FS, or FR; either LL, LS, or LR) property among the cars approaching the intersection from other directions.

This concludes the construction of the PTN. As the next step we will show how do we

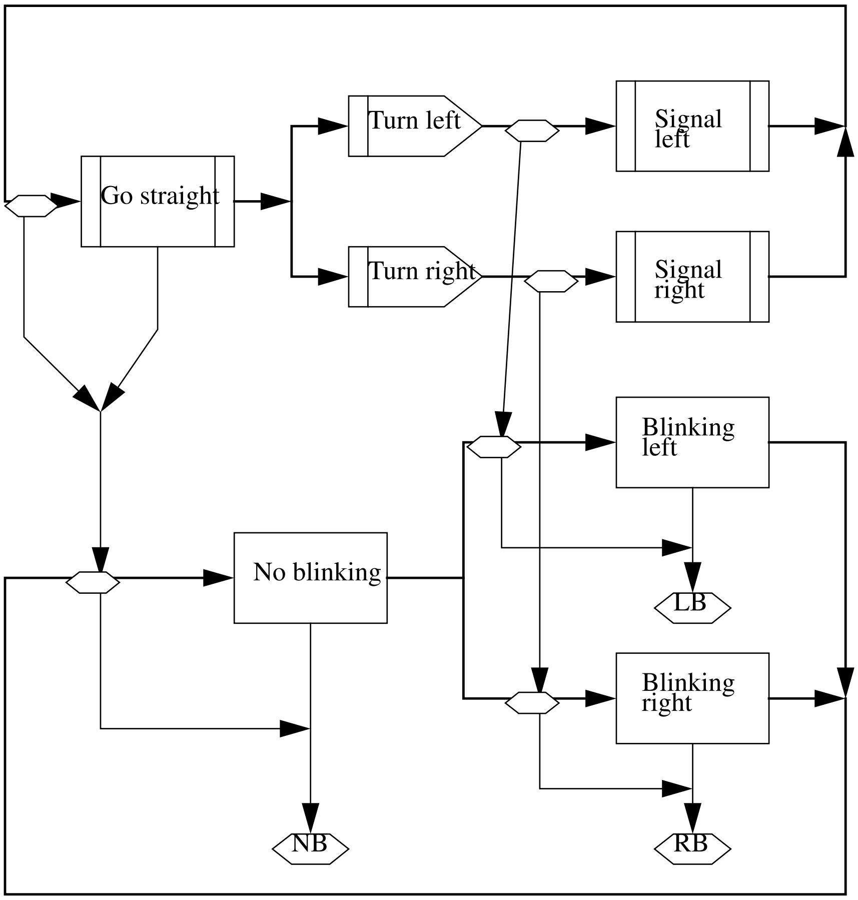


Figure 6: Turning intentions of the own car.

translate it into RETFL formulas, but first we have to define RETFL itself.

3 Restricted Elementary 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. EFL is augmented with a simple temporal operator able to distinguish "now" from "just before now". The restriction removes most of the classical 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 for this language (cf. section 5.1).

3.1 Ontology

The ontology underlying construction of RETFL is precisely the same as that of PTN language. We will briefly summarize it here.

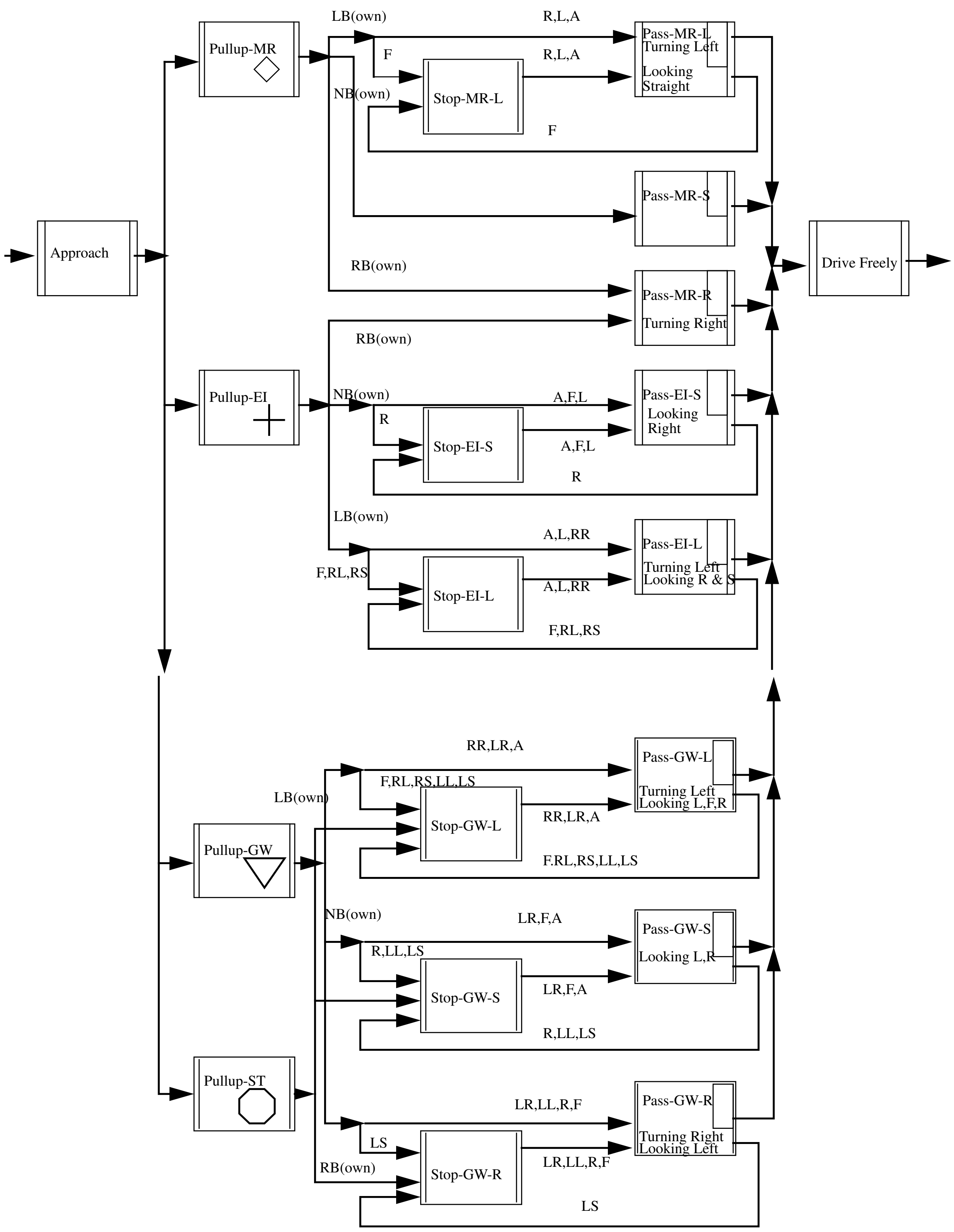


Figure 7: How to pass through an intersection.

The world consists of objects having certain properties, and being involved in different relationships with other objects. Both properties of an object, and relations it is involved in, may change with time. Therefore description of the world in particular time point may consist of data structures describing the object, its properties, and its 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 general case values of different fields of such a data structure may belong to arbitrary value spaces. However lot of the fields characterizing an object will be numerical; moreover their value space will usually be dense (e.g. rational or real numbers, or vectors). In such case we may distinguish such time intervals, in 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, on which the corresponding field value changes in continuous manner. Circumstances may change values only in breakpoints, so they are *piecewise constant*. Obviously, PTN processes correspond to circumstances, and so do features introduced in RETFL. 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.

3.2 Syntax

Similarity type for RETFL is a mapping:

$$\sigma = \{f_1 : \mathcal{D}_1, f_2 : \mathcal{D}_2, ..., f_n : \mathcal{D}_n\}$$

where $n \geq 1$, each $f_1, f_2, ..., f_n$ is a *feature symbol*, and each of $\mathcal{D}_1, \mathcal{D}_2, ..., \mathcal{D}_n$ is a corresponding non-empty, finite *feature domain* consisting of *items*. Items will be written here in lower case; feature names will be capitalized. $\mathcal{X}$ will usually denote a set of feature items, i.e. $\mathcal{X} \subseteq \mathcal{D}_i$, for some $\mathcal{D}_i$.

An *elementary formula* in RETFL with similarity type σ is an expression either of the form

$$f \triangleq \mathcal{X}$$

or of the form

$$f \bullet\!\!= \mathcal{X}$$

(to be read "value of f belongs to $\mathcal{X}$", or "value of f becomes the member of $\mathcal{X}$", respectively), where f is a feature symbol defined in σ and $\mathcal{X}$ is a subset of the domain assigned to f by σ.

An *and-type formula* in RETFL with similarity type σ (or *and-formula* for short) is either:

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

An *or-type constituent* in RETFL with similarity type σ (or *or-constituent* for short) is either:

- an elementary formula of the form $f \triangleq \mathcal{X}$, or
- is composed from two or-constituents using the logical connective $\vee$ (in standard way).

A *arrow-free formula* in RETFL with similarity type σ (or *af-formula* for short) is either:

- an elementary formula $f \bullet\!= \mathcal{X}$, or
- an and-formula, as defined above, or
- is composed from af-formulae using the logical connective $\wedge$ (in standard way).

A *well formed formula* in RETFL with similarity type σ (*logic formula*, or simply *formula*) is either:

- an af-formula, as defined above, or
- if F is an af-formula and G is an elementary formula, then $F \Rightarrow G$ is a logic formula.
- if F is an elementary formula of the form $f \bullet\!= \mathcal{X}$ and G is an or-constituent, then $F \Rightarrow \lambda(G)$ is a logic formula.

3.3 Remarks

The semantics of the language has been defined in [Mal92]. Speaking very informally, the meaning of the basic constituents of the language are as follows:

- if F is an elementary formula of the form $f \hat{=} \mathcal{X}$, then it means "feature f has *now* a value belonging to the set $\mathcal{X}$";
- if F is an elementary formula of the form $f \bullet\!= \mathcal{X}$, then it means "feature f *obtains now* a value belonging to the set $\mathcal{X}$", so in addition to the previous case, we know that the value of f in previous moment of time (and what that means depends on the interpretation of time axis) 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 PTNs can be transformed to RETFL formulae in two phases [Mal92]. 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 resulting formulae will be given it the last section.

4 Configuration description

One of the very specific problems to solve was choosing an appropriate way of describing a configuration of cars approaching an intersection. Such description is the value of some feature, say current configuration. This value should inform about the current configuration of cars coming from each direction, and intending to make some specific maneuver on the intersection. In order to be able to describe an arbitrary configuration of cars approaching an intersection, we have defined for this purpose a special regular language called *configuration language* ($\mathcal{CL}$).

Assuming that $C = \{\lambda, l_l, l_r, l_s, r_l, r_r, r_s, f_l, f_r, f_s\}$ is the alphabet, the *configuration language* $\mathcal{CL}$ can be defined in the classical recursive way:

1. $\emptyset \in \mathcal{CL}$
2. $a \in \mathcal{CL}$ for every $a \in C$

3. If $a_1, a_2 \in \mathcal{CL}$, then also $(a_1 + a_2) \in \mathcal{CL}$ (called *alternative* or *sum*, and intended to mean "either a_1 or a_2")
4. If $a_1, a_2 \in \mathcal{CL}$, then also $(a_1 a_2) \in \mathcal{CL}$ (called *concatenation*, and intended to mean "a_1 and then a_2")
5. If $a \in \mathcal{CL}$, then also $(a)^* \in \mathcal{CL}$ (called *iteration*, and intended to mean "concatenation of arbitrary number of a-s, including 0")
6. Only the strings constructed according to $1 \div 5$ belong to $\mathcal{CL}$.

We will omit parentheses if it does not lead to misunderstanding.

Moreover, we introduce the usual abbreviation $a^+ = a(a)^*$, plus a number of useful, domain-specific abbreviations: $l = (l_l + l_r + l_s)$, $r = (r_l + r_r + r_s)$, $f = (f_l + f_r + f_s)$, $q = (l + r + f)$.

Having defined syntax of $\mathcal{CL}$, let us exemplify its intended meaning. First, some obvious configurations:

- λ — the roads coming to the intersection are empty (except the own car);
- l — a car is coming from the left side;
- l_l — a car is coming from the left side and intends to turn to its left;
- $l_s, l_r, f_l, f_s, f_r, r_l, r_s, r_r, r, f$ — analogously.

Several slightly more difficult expressions:

- $lfrr$ — there are four cars approaching the intersection: one from the left side, one from the front, and two from the right side;
- $rfrl$ — same as above;
- $frrl$ — same as above;
- $l_r l$ — there are two cars coming from the left side, the first one is going to turn 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 say something interesting about some particular car, for example "The second car from the left side intends to turn right": $(r + f)^* l (r + f)^* l_r q^*$.

Saying something both about the second car from the left side <u>and</u> the second car from the right side becomes a bit complicated. E.g.: "The second car from the left side intends to turn right, and the second car from the right side intends to turn right" has to be written as follows: $f^*rf^*r_rf^*lf^*l_rq^* + f^*rf^*lf^*r_rf^*l_rq^* + f^*rf^*lf^*l_rf^*r_rq^* + f^*lf^*rf^*r_rf^*l_rq^* + f^*lf^*rf^*l_rf^*r_rq^* + f^*lf^*l_rf^*rf^*r_rq^*$. However, in this particular application we need to state the properties of <u>all</u> the cars coming from some direction, so the problems with saying something specific about "the n-th car in a row" needed not bother us too heavily.

We have devised (and built into DFMS) an algorithm constructing a *nondeterministic finite state automaton* accepting 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 rights of way), or not. The next section provides some details of this implementation.

5 From PTNs to Implementation

5.1 Discrete Fluent Management System

The RETFL language described in section 3 is a tool used for expressing sets of rules governing the behavior of one of the rule layer subsystems, namely the *maneuver selector* (which selects the appropriate control algorithm to be applied in the process layer). The implementation of this subsystem, called Discrete Fluent Management System, or DFMS [San90], is being loaded with a set of RETFL rules — a "program". Then for any change of input circumstance this program computes appropriate actions (or changes of output circumstances) to be performed by the process layer.

The intended usage of the tools described above (PTNs, RETFL, and DFMS) is as follows: the user (the system designer) conceptualizes the domain employing PTNs. He can easily consult domain experts, as PTNs are relatively comprehensible. Then PTN is translated into RETFL rules, which in turn are loaded into the DFMS shell. DFMS programmed in this way becomes a direct implementation of the control algorithm expressed by the original PTN.

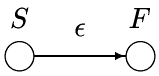
5.2 Implementation of $\mathcal{CL}$

Below we will present an algorithm of constructing a *nondeterministic finite state automaton* accepting given regular language. An alternative approach could be based on *regular expression derivatives*, but assuming the languages we deal with are very simple, the straightforward algorithm presented below probably outweighs the latter approach.

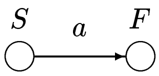
The algorithm is the following:

1. For the language appearing as the right-hand-side argument of the $\triangleq$ (or $\bullet\!=$) relation, build a nondeterministic automaton $\mathcal{A}$ accepting this language. The automaton is constructed recursively, using the following steps appropriately many times:

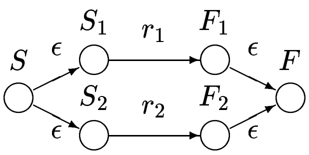
   (a) for the empty expression λ the automaton consists of two states, the initial S, and the final F, linked with a transition caused by the empty letter ϵ:

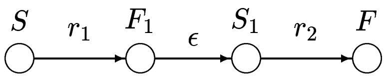
   (b) for the language consisting of one letter of the alphabet, say a, the automaton consists of two states, the initial S, and the final F, linked with a transition caused by the letter a:

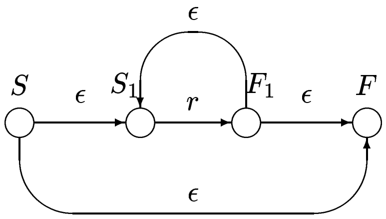
   (c) for the language consisting of the sum of two regular expressions, say $r_1 + r_2$, the automaton is built out of two subautomata, each recognizing one of them:

(d) for the language consisting of the concatenation of two regular expressions, say $r_1 r_2$, the automaton is built out of two automata, each recognizing one of them:



(e) for the language consisting of the iteration of a regular expression, say r^*, the automaton is created from the automaton recognizing r, together with appropriate transitions among surrounding states:



2. Take the left-hand-side argument of the relation $\hat{=}$ (or $\bullet\!=$) and feed it to the input of the automaton $\mathcal{A}$.
3. If the automaton reaches the final state F, then the test succeeds (i.e. the relation is true), else fails.

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.

5.3 Example

We will list here only a few of the RETFL formulae resulting from translation of the PTN shown on Figure 7. The full set of formulae consists of several hundred 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:

  $\lambda(state{-}me \,\hat{=}\, \{Approach\})\ \&\ type{-}intersection \bullet\!= \{MR\} \Rightarrow state{-}me \bullet\!= \{Pullup{-}MR\}$

  or, as the DFMS input:

```
(AddRuleQ type-intersection
          (OwnTransCdl 'MR
                       '(equal (getslot state-me) 'Approach)
                       '(AssignVal state-me 'Pullup-MR)))
```

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

$$\lambda(state - me \,\hat{=}\, \{Pullup - MR\})\ \&\ \lambda(intended - turn - me \,\hat{=}\, \{L\})\ \&$$

$$\&\ dist - to - inters - me \bullet\!= \{Close\}\ \&\ configuration \,\hat{=}\, (r+l)^* f q^* \Rightarrow$$

$$\Rightarrow state - me \bullet\!= \{Stop - MR - L\}$$

```
(AddRuleQ dist-to-inters-me
          (OwnTransCdl 'Close
                       '(and
                          (equal (getslot state-me) 'Pullup-MR)
                          (equal (getslot intended-turn-me) 'L)
                          (TestIn (getslot configuration) '((* (+ r l)) f (* q))))
                       '(AssignVal state-me 'Stop-MR-L)))
```

- If there are no cars coming from the front, then the own car may pass:

$$\lambda(state - me \hat{=} \{Pullup - MR\}) \ \& \ \lambda(intended - turn - me \hat{=} \{L\}) \ \&$$
$$\& \ dist - to - inters - me \bullet\!\!= \{Close\} \ \& \ configuration \hat{=} (l + r)^* \Rightarrow$$
$$\Rightarrow state - me \bullet\!\!= \{Pass - MR - L\}$$

```
(AddRuleQ dist-to-inters-me
          (OwnTransCdl 'Close
                       '(and
                          (equal (getslot state-me) 'Pullup-MR)
                          (equal (getslot intended-turn-me) 'L)
                          (TestIn (getslot configuration) '((* (+ r l)))))
                       '(AssignVal state-me 'Pass-MR-L)))
```

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

6 Conclusions

In this paper we have presented several KR languages used within one particular system, namely the driver support system being designed within the PROMETHEUS project. We have pointed to relationships linking these languages.

The Process Transition Networks are a pictorial language suitable for the conceptualization, and coding of knowledge about complex, often quantitatively intractable environments. The word "coding" is justified by the fact that sentences in this language (i.e. nets) can be translated into a very restricted temporal logic language RETFL, which may be treated as a programming language, because it possesses a working interpreter.

The advantages of the PTN language are simplicity, the possibility of expressing causal dependencies, the ability to describe behavior of several agents acting in the same environment, modularity, and the possibility of using it for subsequent refinements of an initial conceptualization. These properties have been confirmed during the conceptualization of several non-trivial traffic-behavior scenarios.

The Restricted Elementary Temporal Feature Logic is an example of a KR tool based on formal logic. It is however tailored in such way that it is easily computable (up to implementation level). RETFL is the counterpart for PTNs in the sense that specification expressed with a PTN can be easily translated to a program in RETFL.

The last of the presented languages, the Configuration Language, is simply a regular language, but it plays a major role in representing knowledge about the environment our system is to act in. Automata theory provides effective algorithms for any imaginable kind of computations that may be required from such a representation.

References

[Mal91a] Jacek Malec. How to pass an intersection. In *Proc. of the Prometheus Pro-Art Workshop on Intelligent Co-Pilot*, pages 167–182, Grenoble, France, 1991.

[Mal91b] Jacek Malec. Passing an intersection, or automata theory is still useful. In Brian Mayoh, editor, *Proc. of the Scandinavian Conference on Artificial Intelligence – 91*, pages 258–265. IOS Press, 1991.

[Mal91c] Jacek Malec. Process Transition Networks: A formal graphical knowledge representation tool. In Z. W. Ras and M. Zemankova, editors, *Methodologies for Intelligent Systems, 6th International Symposium*, pages 193–202. Springer-Verlag, October 1991. Lecture Notes in Artificial Intelligence 542.

[Mal91d] Jacek Malec. Process Transition Networks: What Are They For? In *Proceedings of the IEEE Systems, Man and Cybernetics Conference, Charlottesville, VA*, pages 1177–1182, October 1991.

[Mal92] Jacek Malec. Process Transition Networks: The Final Report. Research Report LiTH-IDA-R-92-07, Department of Computer Science, Linköping University, 1992.

[MFM+91] Jacek Malec, Reza Farahani, Magnus Morin, Simin Nadjm-Tehrani, Per Österling, and Erik Sandewall. An integrated software environment for development of driver support systems. In *Proc. of the Prometheus Pro-Art Workshop on Intelligent Co-Pilot*, pages 31–39, Grenoble, France, December 1991.

[San90] Erik Sandewall. Discrete Fluent Management System, a program in the HIRW Workbench. Unpublished report, 1990.

[San94] Erik Sandewall. *Features and Fluents.* Oxford University Press, 1994.