Title
Formal Specification And Verification Of The Entry And Exit Maneuvers

Authors
Sachs, S. R.
Varaiya, P.

Publication Date
1996-02-01

CALIFORNIA PATH PROGRAM
INSTITUTE OF TRANSPORTATION STUDIES
UNIVERSITY OF CALIFORNIA, BERKELEY

Formal Specification and Verification of the Entry and Exit Maneuvers

Sonia R. Sachs
Pravin Varaiya

California PATH Research Report
UCB-ITS-PRR-96-3

This work was performed as part of the California PATH Program of the University of California, in cooperation with the State of California Business, Transportation, and Housing Agency, Department of Transportation; and the United States Department of Transportation, Federal Highway Administration.

February 1996

ISSN 1055-1425

Formal Specification and Verification of the Entry and Exit Maneuvers

Sonia R. Sachs and Pravin Varaiya
Department of Electrical Engineering and Computer Sciences
University of California
Berkeley CA 94720

January 30, 1996

Abstract

This is the final report of a study of key questions relating to the interface between the Automated Highway System or AHS and Urban Arteries or UA. Those questions are formulated in terms of four tasks:

1. Specify physical arrangement, operational procedures for entry/exit;
2. Conceptualize functions of transfer zone between the AHS and UA;
3. Characterize interaction between AHS and UA;
4. Propose ways of controlling the interaction.

This report documents how these tasks were completed. Five different physical arrangements and associated operational procedures are proposed, differing in cost, land requirements, and sophistication of coordination and control. The transfer zone functions as buffer between the traffic on the AHS and UA, as a place for check-in and check-out, and as a controller of flow into the AHS. It also provides the infrastructural elements needed for entry and exit. The interaction between the AHS and UA is characterized by the queues that develop in the transfer zone, the waiting times to which vehicles must submit before they gain AHS entry, and the disruption those vehicles inflict on AHS traffic. Lastly, control of the interaction between AHS and UA is exercised through the coordination of entry and exit with AHS traffic and the vehicle feedback control laws that govern the actual trajectory of the vehicle as it enters and leaves the AHS. **This** study suggests that most issues relating to the AHS/UA interface can be resolved, although for some configurations, sophisticated coordination and control will be necessary for smooth and safe operation.

*Work supported by the PATH program, Institute of Transportation Studies, University of California, Berkeley, the Federal Highway Administration Contract **DTFH-61-93-C-001-99,** by National Science Foundation Grant **ECS9417370** and Army Research Office Contract DAAH04-94-G-0026. **The work reported here** is largely the work of Datta Godbole, Tony Hitchcock, Sonia Sachs and Pravin Varaiya. The simulations were carried out by Farokh Eskafi, Delnaz Khorramabadi and Ekta Singh. We also acknowledge the help of Mireille Broucke.

1 Executive Summary

An automated highway system or AHS will be initially deployed as an "implant" on the existing roadway network. That is, the AHS will most likely be deployed by converting one or more lanes of existing highways, or by building a new network link. Like any implant, the success of the AHS will depend on how well the "host" accepts the implant, i.e., how well the AHS interfaces with the urban arterials or UA. Past AHS research at PATH has understandably been concentrated on the design and operation of the AHS in isolation. That research has now led to a reasonably complete understanding about AHS architecture, the design options available, and a (not yet complete) set of tools to simulate the performance of different designs. Although much work remains to be done, our understanding has reached a stage where we can meaningfully ask how the AHS might be deployed. (We are concerned with deployment in an engineering sense, not with the institutional and public policy prerequisites to deployment.)

AHS entrances are the narrow veins that feed the wide arteries of the automated lanes. If those veins get constricted, the arteries will be starved and the AHS capacity will remain underutilized. The stream of vehicles leaving the automated lanes debouch into narrow AHS exits. If those exits are blocked, traffic can spill back into the automated lanes, disrupting traffic. The design of AHS entry and exit, the management of the processes by which vehicles negotiate their passage through them, and the coordination of that passage with the stream on the automated lanes thus have a determining effect on the achievable traffic flows of the AHS.

Thus key issues relating to deployment concern the AHS/UA interface. Those issues are the subject of this report. The work presented here was funded in part by Caltrans under MOU 134; greater support came from the Federal Highway Administration Precursor Analysis Program and from the National Science Foundation.

Those key issues were formulated in MOU 134 as four tasks:

1. Specify physical arrangement, operational procedures for entry/exit;
2. Conceptualize functions of transfer zone between the AHS and UA;
3. Characterize interaction between AHS and UA;
4. Propose ways of controlling the interaction.

This report documents how these tasks were carried out. The work strongly supports the conclusion that the key issues of AHS/UA interface can be satisfactorily resolved, although for some configurations, the interaction must be controlled in a sophisticated manner to ensure smooth and safe entry and exit.

We now summarize our findings for each task.

Task 1

We propose five different physical configurations for entry and exit. The configurations differ in terms of the physical layout of the "transition zone" between AHS and UA, the relative

cost, and the amount of land that is taken up by the transition zone. The operational procedures concern the coordination between roadside infrastructure and vehicle controller, the treatment of check-in and check-out, the role of barriers between automated and manual lanes. (Our study does not concern check-in and check-out mechanisms.)

Task 2

The transfer zone acts as a buffer between the AHS and the UA, provides for check-in and check-out, controls the flow from UA into the AHS (similar to "ramp-metering"), and contains the infrastructure (sensors and communication devices) needed for the coordination of entry and exit.

These configurations and the operational procedures are reported in P. Varaiya, Precursor *Systems Analysis of Automated Highway Systems Activity Area J-Entry/Exit Implementation Final Report,* also available as a PATH Report. A revised version of the configurations appears in S. Sachs and P. Varaiya, Formal *Specification and Verification of the Entry and Exit Maneuvers,* included herein. The operational procedures are also described in that report.

Task 3

The interface between AHS and UA can be measured in terms of the queues that develop in the transfer zone and the disruption that entering vehicles inflict upon AHS traffic. A queuing model is proposed in *Precursor Systems . . . .* A more elaborate study is conducted in M. Broucke and P. Varaiya, "A theory of traffic flow in automated highway systems," Seventy-Fifth Annual Meeting of the Transportation Research Board, Washington, D.C. January 7-11, 1996. These studies provide a quantitative measure of the disruption.

Task 4

The interaction between AHS and UA vehicles is determined by two sets of controllers located in the coordination and regulation layers of the AHS architecture that PATH has developed. (See, P. Varaiya, "Smart cars on smart roads," *IEEE Trans. Auto. Contr.*, vol. 38(2), 195-207, Feb. 1993.) The coordination layer controller synchronizes the movement of the vehicles entering the AHS from the transfer zone with the vehicles on the AHS in a way that minimizes disruption of the AHS flow. It also is used to permit safe exit from the AHS into the transition zone. The regulation layer controller determines the feedback laws on-board the vehicles that calculate the throttle, braking and steering inputs which govern the trajectory of the vehicle.

The coordination layer design is specified in S. Sachs and P. Varaiya, op *cit.* That report also provides a limited verification of correctness of the design. The verification is conducted using Cospan. It is limited because the continuous behavior is abstracted away.

The design of the feedback control laws is presented in Chapter 6 of D. Godbole, *Hierarchical Hybrid Control of Automated Highway Systems,* PhD Thesis, Department of Electrical Engineering and Computer Science, U.C., Berkeley. Simulations of those procedures are summarized in D.N. Godbole, F. Eskafi, E. Singh and P. Varaiya, "Design of entry and exit maneuvers of IVHS," *Proc. American Control Conference,* June 1994, pp. 3566-3570.

Formal Specification and Verification of the Entry and Exit Maneuvers

Sonia R. Sachs and Pravin Varaiya
Department of Electrical Engineering and Computer Sciences
University of California
Berkeley CA 94720

January 30, 1996

1 Entry and Exit maneuvers

Entry and exit maneuvers depend on the physical configuration of automated highways. Five configurations were studied in [1]. They are illustrated in Figures 1, 2, 3, 4, and 5. An automated lane (AL) is a lane of the highway where vehicles are under automatic control.' A transition lane (TL) is a lane where vehicles make the transition from manual control to full automated control. The other lanes are called manual lanes (ML). The "stop sign" in these Figures refers to a location where entering or exiting vehicles must stop. They proceed only after permission is received from the roadside controller.

For the different configurations, different entry/exit maneuvers may need to be designed. For the arrangement shown in Figure 1, for example, there may be multiple gates between the TL and the AL. In order to take advantage of this, multiple entry and exit opportunities may be offered to vehicles in the TL. In the configuration shown in figure 2, the entry/exit maneuver does not require any processing at the coordination layer because the AL, which follows the TL, does not have vehicles other than the ones arriving from the TL (i.e., there is no conflict between vehicles already in the AL and vehicles arriving from the TL). In the configuration shown in Figure 3, only a single entry/exit gate may exist. Notice that in this configuration, the TL leads only to an AL, and thus a failed entry maneuver requires that vehicles, which do not enter the AL, stop at the gate. This is different from the arrangement shown in Figure 1 with one entry gate, because there a vehicle that fails is not required to stop; it simply proceeds along the TL until a new entrance is encountered. The configuration shown in Figure 4 is very similar to the arrangement in Figure 1. The differences are that, in the former, there may be fewer entry/exit gates (since they all have to fit within a distance of approximately one mile), and vehicles which fail entry must exit the automated highway system.

The TL may be an entire lane alongside the AL, as in Figure 1; it may also be a very short stretch as in Figures2–5.

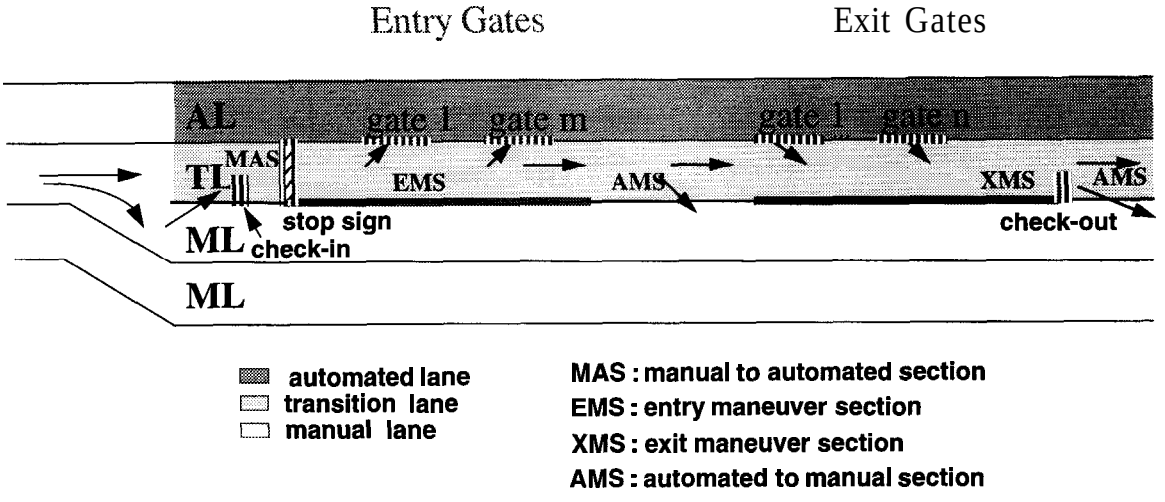


Figure 1: Entry/exit configuration 1

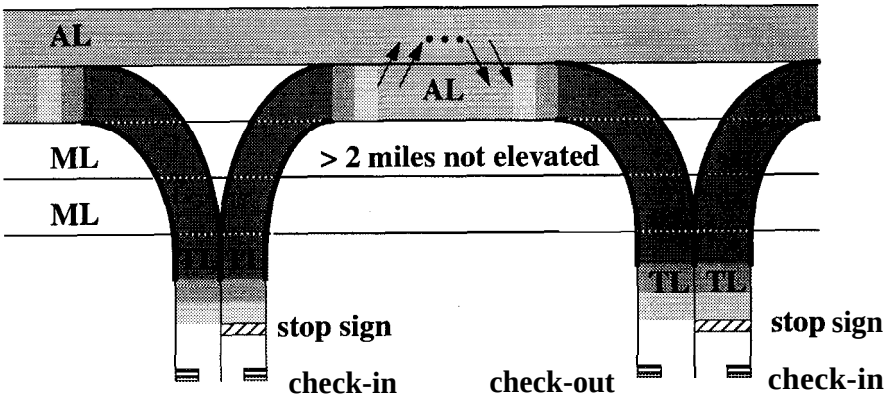


Figure 2: Entry/exit configuration 2

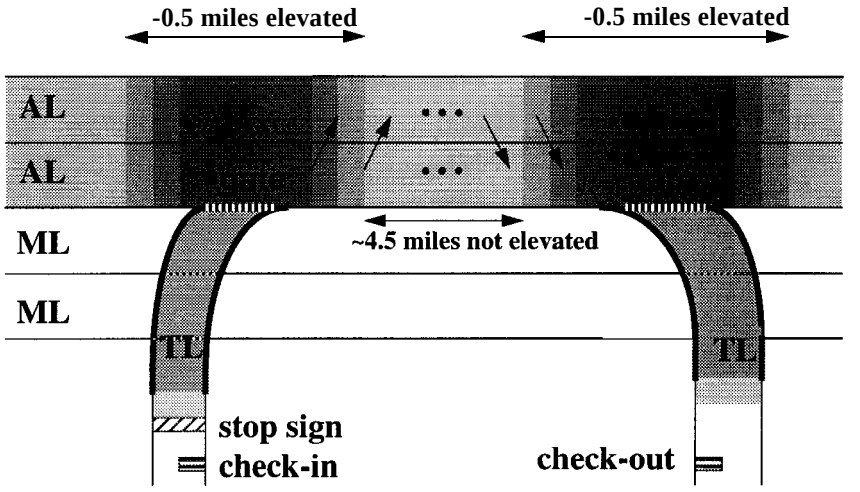


Figure 3: Entry/exit configuration 3

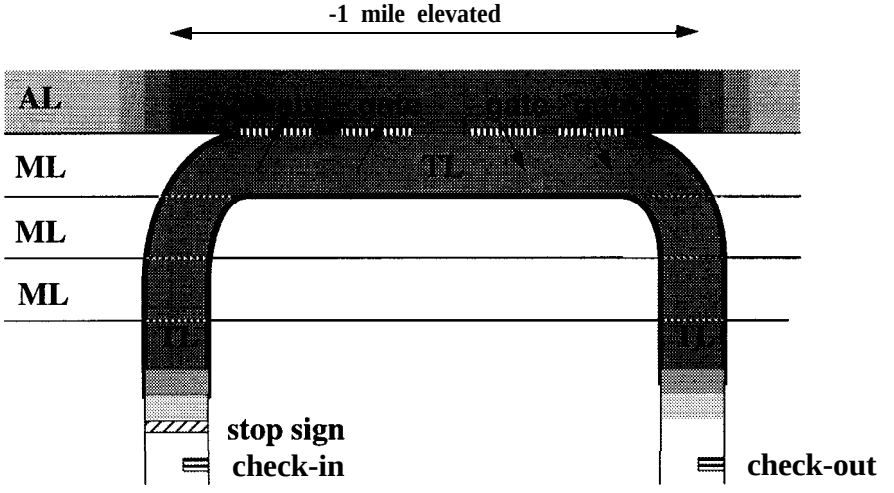


Figure 4: Entry/exit configuration 4a

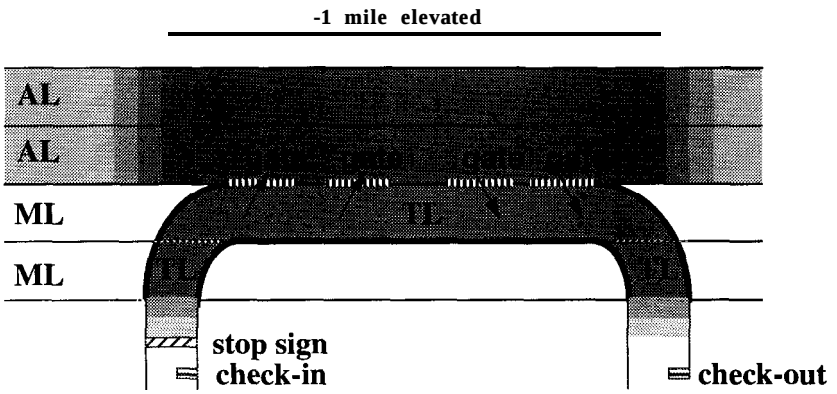


Figure 5: Entry/exit configuration 4b

The configuration of Figure 1 is likely to require the least additional construction costs. A vehicle will enter this highway in the rightmost ML using a currently available entrance ramp. It will continue its journey on the fast (leftmost) manual lane. At the beginning of the highway section named Manual-to-Automated Section (MAS), it will change to TL manually. After passing through a check-in point, the control of the vehicle will be given to the automated control system. All vehicles at the beginning of the highway section named Entry Maneuver Section (EMS) will be automatically controlled. On the EMS, the vehicles will accelerate to match the speed of the AL, and eventually change lanes onto the AL. Those vehicles which fail to enter the AL will be slowed down in the highway section named Automated-to-Manual Section (AMS), and their drivers will be asked to resume manual control.

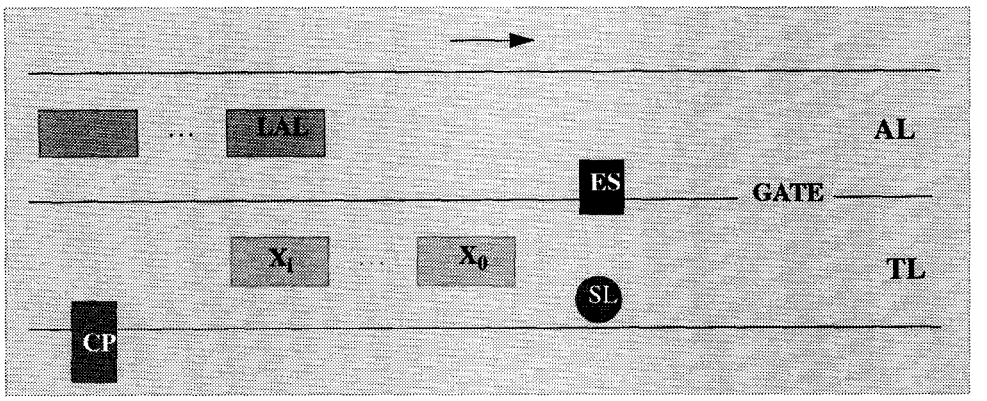
Exiting vehicles will change lanes from AL onto TL at the Exit Maneuver Section (XMS). They will be slowed down and control will be transferred to manual drivers on the AMS. These manually driven vehicles will have to change lane from TL to the leftmost (fast) ML. Exit from the highway is via the rightmost ML using the current exit ramp.

We have modeled the arrangement in Figure 1, assuming one entry and multiple exit gates, as was suggested in [2]. The system considered in [2] does not include a stop sign, where the vehicles stop before entering the automated section of the transition lane (EMS). However, for reasons explained in [1], a complete stop before entering the EMS section of the TL is preferable, since it reduces the length of the EMS section.

1.1 Entry Maneuver System

Figure 6 illustrates the system components of the entry maneuver for the configuration in Figure 1.

A closely spaced train of vehicles in the transition lane is called a pre-platoon. Vehicle X_0 is the leader of a pre-platoon in the TL. Such a pre-platoon may be formed as vehicles enter the TL. A pre-platoon with only one vehicle is called a free agent. Vehicle X_i represents any vehicle in the TL which becomes a follower within a pre-platoon. The leader vehicle X_0 communicates directly



LAL: Leader Vehicle in AL
X_0: Leader Vehicle in TL
X_i: Follower Vehicle in TL
CP: Check-Point
SL: Stop Light
ES: Sensor

Figure 6: Entry maneuver: system's components

with each vehicle X_i, and thus it can be seen as playing the role of the "hub" of a star network. Vehicle LAL is the leader of a platoon in the AL, which communicates with the entering pre-platoon. The check-point component Cl' executes the "check-in" procedure: it inquires whether a vehicle entering the transition lane is properly equipped, and either allows or stops its entry into the automated highway system. The stop light component SL wakes up a roadside sensor, and becomes "green" when the sensor indicates that there is space in the AL for the maneuver. The sensor determines when there is an inter-platoon gap in the AL which is at least as large as the minimum space required for the entry maneuver. If a platoon is in the AL within the sensor's range, then the sensor communicates with it, requesting its identification number. Having sensed a space in the AL for the maneuver, the sensor communicates this fact to the stop light. In addition, the sensor communicates with the X_0 vehicle, indicating that space was found, the size of the space, and the identification number of the leader in the AL, if one was identified. The sensor also gives information to the entering vehicles about the distance to the turn markers at the gate.

The logical steps needed for the entry maneuver are given in Figures 7 and 8. These figures show under which conditions vehicle X_0 forms a pre-platoon, and under which conditions X_0's pre-platoon enters or fails to enter the automated lane. They also show the communication among the platoon leader (in the AL), the pre-platoon leader (in the TL), the check post, the stop light, and the sensor components.

One of our goals is to prove that the entry maneuver satisfies the following properties with respect to system behavior:

- A vehicle which requires entry into the automated highway, eventually enters;
- Entering vehicles do not collide with vehicles already in the AL.

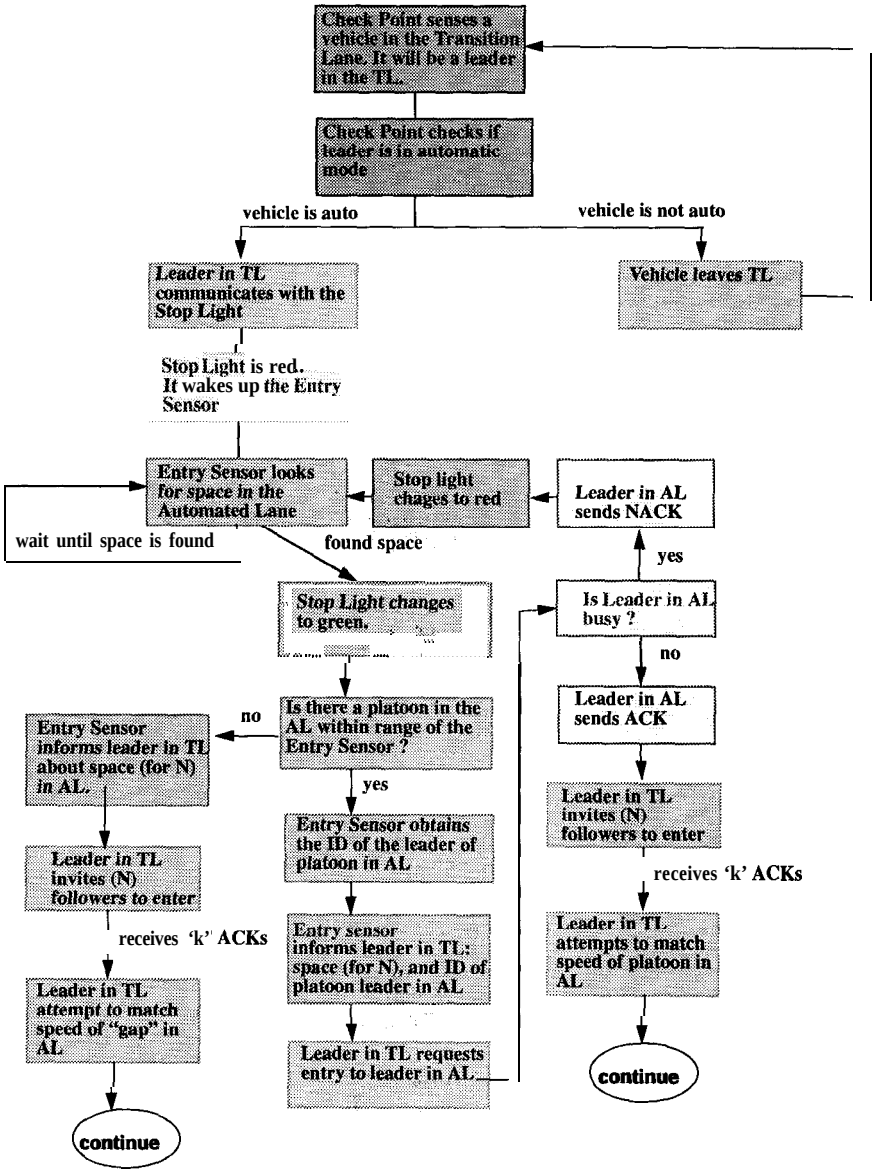


Figure 7: Logical steps for the entry maneuver

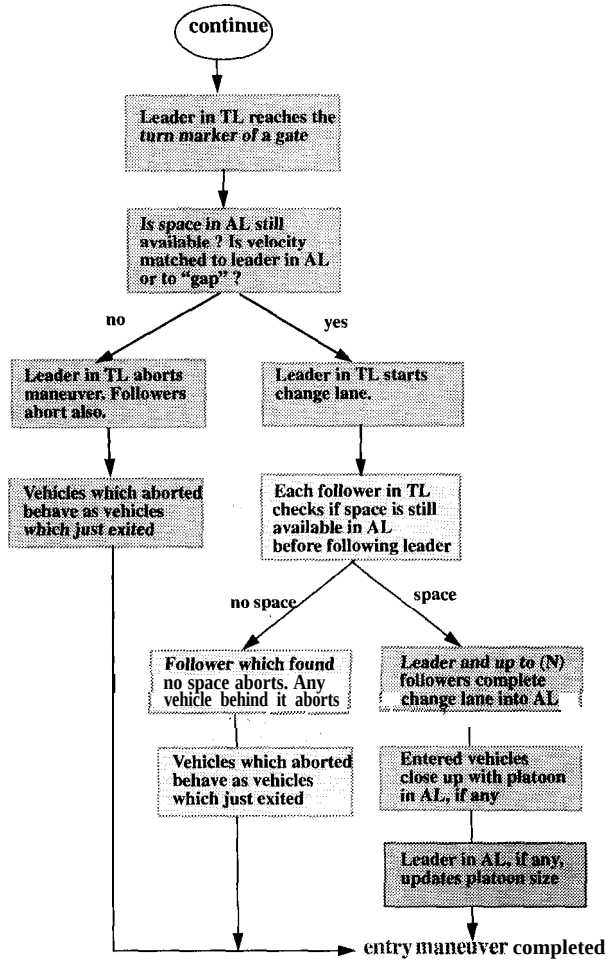
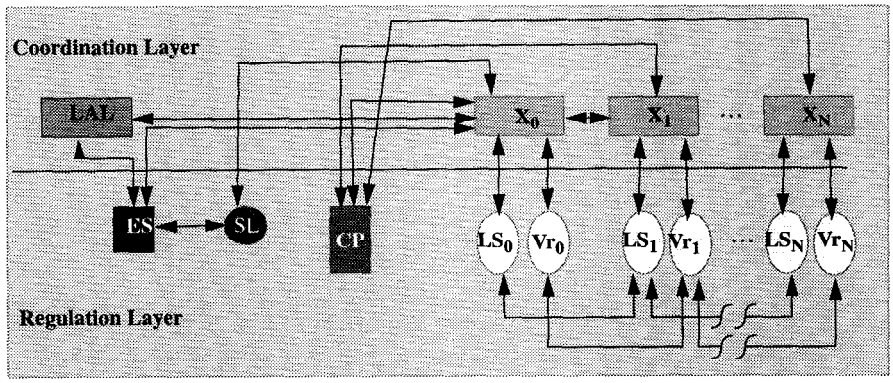


Figure 8: Logical steps for the entry maneuver (continuation)



LAL: Leader Vehicle in AL
X_0: Leader Vehicle in TL
X_1: First Follower Vehicle in TL
X_N: N^{th} Follower Vehicle in TL

CP: Check-Point
SL: Stop Light
Sensor
LS_i: Lateral Sensor of vehicle i
Vr_i: Velocity Response of vehicle i

Figure 9: Model of the entry maneuver system

To that end, we have represented this system within the L-automata/L-processes framework (see Chapter 2 of [3]). Figure 9 shows the model of the described system. Multiple layers of control have been proposed for the automatic control of vehicles. The performance of a maneuver involves only the first two layers-namely, the coordination and regulation layers. At the coordination layer of control, we have specified the following L-processes: vehicles X_0, X_i, and LAL, the stop light SL, the check-post CP, and the entry sensor ES. Components are shown to communicate as required by the entry maneuver. The regulation layer of control is modeled via the following L-processes.

LS : lateral sensor response of vehicles in TL. Every 0.1 seconds this sensor updates the sensed distance and velocity of the leader in the AL with which the maneuver is being coordinated. It also senses when the vehicle is at the stop light, and when it reaches a turn marker. The model of the lateral sensor for the followers $(LS_i, i = 1, \ldots, N)$ is slightly different from that of the leader(LS_0) to represent the fact that a follower can only sense an object on the highway after all vehicles in the platoon ahead of it have already done so.

Vr : velocity response of the vehicles in TL. Similarly, the model of the velocity response for the followers $(Vr_i, i = 1, \ldots, N)$ is different from that of the leader(Vr_0). The modeling of events indicating that a follower approached or reached a stop light, achieved full velocity, started or completed the change lane maneuver depends on the modeling of these events for the leader and for all followers ahead of it.

Note that this is a closed system model because every component requires only those inputs which are produced by another component within the system.

1.2 Formal Specification and Verification of the Entry System

The formal specification of these components in the L-automata framework is given in Appendix A. The desirable behavior of this system is modeled by two L-automata: *no-collision* and *eventually_enters*, also detailed in Appendix A. Verification that the specified system presents the desirable behavior represented by both *no-collision* and *eventually-enters* was successful.

One source of error in protocol design is improper termination of functions. When a distributed function, executed by asynchronous components, is not properly terminated, the distributed components may lose "function synchrony" ; i.e., while one component exchanges messages required by the execution of a function f_i, other coordinating components may be exchanging messages required by the execution of a function $f_j, f_i \neq f_j$. This lack of synchrony at the function level leads to very complex sequences of events, eventually resulting in deadlocks. A common mistake is to design a protocol such that only one component checks for the proper termination of a function.

In the process of verifying this protocol, several deadlocks were found. The error traces produced by the unsuccessful attempts, which preceded the final successful attempt, were very useful in determining how to modify the entry protocol. Some of the detected deadlocks were the consequence of a very complex sequencing of events, which could be missed in a design that was tested only by simulation.

In order to guarantee that the specification of the entry maneuver is amenable to analysis, verification of the above-mentioned properties were restricted to the following:

- The number of follower vehicles is a parameter N of the specification (specified at run time). The verification was done with N = 2.
- The sensor may find that there is space for 0, 1, 2, or 3 vehicles to enter. The size of the space is not a parameter because it is not possible to assign an arbitrary range to a variable in the tool used for formal verification (Cospan).

By using the tool Cospan, we have verified that the specified system presents the desirable behavior. Documentation of this verification is also found in Appendix A.

1.3 Induction Proof for the Entry System

The formal verification of the entry maneuver was successful, as shown in Appendix A. However, because of the simplifications made to the specification, the correctness proof was limited to a pre-platoon of at most three vehicles-i.e., one leader and N = 2 follower vehicles. We would like to extend the proof to an arbitrary number of follower vehicles. Induction methods [4] can be applied to extend the proof in question.

The induction method proposed in [4] consists of finding a reduction for the system representation which can be used in its place, for an arbitrary number of components. Such a reduction is referred to *as* an *invariant* because it preserves the structure of the system's representation. A system

with N coordinating processes is represented by the composition of its processes. The composition of L-automata/L-process $P_1, P_2, \ldots P_N$, denoted by $\bigotimes_{i=1}^{N} P_i$, is explained in Chapter 2 of [3]. A system S $= \bigotimes_{i=1}^{N} P_i$ can be shown to perform a property *T* by applying induction methods, also explained in Chapter 2 of [3].

Applying induction methods to the system in question, we have to show that the entry maneuver with an arbitrary number of followers performs the same tasks as the one with only two followers.

The entry system, parametrized by N, the number of followers in the transition lane, is given by:

$$S(N) = P_0 \bigotimes (\bigotimes_{i=1}^{N} P_i),$$

$$P_0 = X_0 \bigotimes LAL \bigotimes LS_0 \bigotimes Vr_0 \bigotimes CP \bigotimes ES \bigotimes SL,$$

$$P_i = X_i \bigotimes LS_i \bigotimes Vr_i, i > 0 .$$

We now briefly describe the induction method we will use. Let a parametrized system of L-Processes S(N) $= \{P_0, \ldots, P_N\}$ be given. The output of an L-Process P_i may be input to one or more L-Processes $P_j, j \neq i, j = 1, \cdots,$ N. A "variable" of a L-Process can be its output or input. An L-Process may use both input and output variables in order to determine its "next" state, and consequently, its output at the next state.

Definition 1.1 *[5] A variable x is of bounded computation if the number of variables upon which its computation depends, is bounded. In particular, a variable of bounded computation cannot have dependencies upon the number of processes in a system with unbounded number of components.*

Definition 1.2 *[5] Let D denote the domain of the variables of L-Processes $P_1, \ldots, P_n$. A function $f : D^n \rightarrow D$ is associative if there exists $f_i : D^2 \rightarrow D$, for $i = 1, \cdots, n-1$, such that for $z_i = f_i(x_i, z_{i+1}), (z_n = x_n), f(x) = z_1$.*

Definition 1.3 *Let a parametrized system of L-Processes S(N) = $\{P_0, \cdots, P_N\}$ be given. If there exists a number C such that for* **any** *finite N, the number of variables associated with S(N) is bounded by C, then we say that the number of variables in the system is of order one, denoted by* O(1). *We also say that the number of variables in the system is bounded. If the number of variables in the system is bounded by C x N, then we say that the number of variables in S(N) is of order N, denoted by $\mathcal{O}(N)$.*

A result from [5] is mentioned here without proof:

Theorem 1.1 *A parametrized system of processes S(N) = $\{P_0, \cdots, P_N\}$ is linearizable if:*

(a,) For all i > 0, every variable of P_i is of bounded computation, and the number of variables of P_i is $\mathcal{O}(1)$;

(b) The number of variables of P_0 is O(N);

(c) Every variable of P_0 is either of bounded computation, or is associative;

(d) The number of variables of P_0 not of bounded computation, is bounded.

A discussion of these conditions, an algorithm for the linearization of a system, and the proof that a system which satisfies *(a)* – *(d)*above is linearizable is given in [5]. The reader will also find in [5] a detailed example of the use of this theorem in the context of distributed fault-tolerant memory systems.

1.3.1 Linearization of the Entry System

The entry system is linearizable because:

(a) the variables of P_i, for i > 0 are of bounded computation(i.e., the variables of the L-Processes $X_i, LS_i, V_{r_i}, i > 0$ do not depend on the number of processes in the system), and the number of variables of P_i, for $i > 0$ is of $\mathcal{O}(1)$, because no variable of P_i is a vector which depends upon the number of follower vehicles N in the system, thus the bound to the number of variables of P_i is given by the number of local scalar variables of P_i;

(b) the number of variables of P_0 depends upon the number N of follower vehicles in the system, hence of $\mathcal{O}(N)$;

(c) the variables of P_0 are either of bounded computation (i.e., they are independent of the number of followers N), or associative, i.e., each variable x of P_0 can be "distributed" into variables $x_1, \cdots, x_n$ of $P_1, \cdots, P_N$, by redefining the assignment of x_i at P_i, and by associating to each P_i a new internal variable z_i, for $i = 0, \cdots$, n, called "propagation" variable, such that $z_n = x_n$, $z_i = f_i(x_i, z_{i+1})$, and $z_0 = f_1(x_0, z_1)$ assigns the same value to x as the original P_0 does. An example of associative variable is the output "no_space_now" generated by the leader X_0. This output is generated when an entry sensor indicates the amount of space available in the automated lane. The leader checks the number of followers ready to join a transition lane pre-platoon. If there is space for m vehicles, and $k >$ m followers are ready, than the leader sends "no-space-now" to followers $m + 1, \ldots, k$. This output is associative because it can be distributed among the followers by allowing the first follower to read the entry sensor, decide if there is space for it, decrement the sensor information if space was found, and "propagate" the updated information to the next follower. When a follower reads the propagated sensor information and finds it to be zero, it knows that there is no space for entry;

(d) the number of variables in P_0 which depend upon the number of follower vehicles in the system is bounded.

By applying the construction suggested in the proof of [5, Theorem 3.11, we have that the linearized entry system is given by

$$\tilde{S}(N) = \tilde{P}_0 \bigotimes (\bigotimes_{i=1}^{N} \tilde{P}_i).$$

The process $\tilde{P}_0$ has a modified leader, $\tilde{X}_0$, as follows:

- The variable invitation indicates to each follower whether they have been invited to join the entering pre-platoon. In the linear case, this variable models a broadcast message to all followers, without the value "no_space_now". In the non-linear case, we had instead an array invitation[N] that models point-to-point messages between the leader and each follower;
- The variable ppsize indicates the size of the entering pre-platoon. In the linear system, this variable is equal to a new internal variable of P_n, denoted by number-of-acks. This variable is is recursively assigned to the number of processes in the range 1, . . ·, i, which responds with "ack_come_with_me";
- Every transition predicate which in P_0 depends upon a range of processes, is modified in $\tilde{P}_0$ to depend upon new internal variables of P_n, which are recursively assigned. The new internal variables of P_i, for ⁎ = 1, . . ·, N, are: number-of-aborts, number-entered-AL, number-entry-complete.

The process $\tilde{P}_i$ has a modified follower $\tilde{X}_i$, as follows:

- The new internal variables mentioned above are assigned in the following way:

number-ofacks = (X_{i+1}.number_of_acks + 1) if output of X_i is come-withme, else is X_{i+1}.number_of_acks.

number-of-aborts = (X_{i+1}.number-of-aborts + 1) if output of X_i is abort, else is X_{i+1} .number-of-aborts.

number-entered41 = (X_{i+1}.number_entered_AL + 1) if output of X_i is entered-AL, else is X_{i+1} .number-of-aborts.

number-entered-AL = (X_{i+1}.number_entered_AL + 1) if output of X_i is entered-AL, else is X_{i+1}.number_of_aborts.

number-entry-complete = (X_{i+1}.number_entry_complete + 1) if output of X_i is entry-complete, else is X_{i+1} .number-entry-complete.

- The predicate of the state transition at-stop-light → go-withleader is modified to:

$$(X_0.\text{invitation} = \text{come-with-me}) * (i < ES.\text{how\_much\_space})$$

- The predicate of the state transition at-stop-light → entry-complete is modified to:

$$(X_0.\text{invitation} = \text{come-withme}) * \text{☎⁎} \geq ES.\text{how\_much\_space}) + (X_0.\text{invitation} = \text{too-late-to-join})$$

- The predicate of the state transition entry-complete → idle is modified to:

$$(X_0.\# = \text{entry-complete}) * (X_{i+1}.\text{number-entry-complete} = N - (i + 1))$$

- Any dependence on X_{i-1} is modified to a. dependence upon X_{i+1};
- Any comparison $i = 0$ is changed to i = N, and $i > 0$ to $i <$ N.

We anticipate that an induction invariant q_i might be made of i system processes, such that $\tilde{P}_0$ always communicates with the end of a string (or cascade) of processes, i.e., for a system with N processes, $\tilde{P}_0$ communicates with $\tilde{P}_N$. We thus view the leader in the transition lane followed by the vehicle which has the highest index. The tail of the pre-platoon is the vehicle with the lowest index $i > 0$.

The "next" operator for $\tilde{P}_0$ is given by $\phi(\tilde{P}_0) = \tilde{P}_0$, and for $\tilde{P}_i$, i > 0, is as follows:

$$\begin{aligned} \phi(\tilde{X}_i) &= \tilde{X}_{i+1}, \\ \phi(LS_i) &= LS_{i+1}, \\ \phi(Vr_i) &= Vr_{i+1}. \end{aligned}$$

We propose the following invariant:

$$\begin{aligned} Q_i &= \tilde{P}_0 \bigotimes \tilde{P}_i \bigotimes \hat{P}_{i-1}, \text{ where} \\ \hat{P}_{i-1} &= FREE(\tilde{X}_{i-1}) \bigotimes LS_{i-1} \bigotimes Vr_{i-1}. \end{aligned}$$

Recall that the operation $FREE$ applied to an L-process gives its trivial homomorphic reduction. The idea behind the invariant is that all the possible behaviors of a pre-platoon with N > 2 followers is representable by a pre-platoon with only two followers. For our induction base we choose $i = 2$. We thus have as our induction base a system with one leader and two followers. In order to satisfy the induction base, we need to verify that

$$\mathcal{L}\left(\tilde{P}_0 \bigotimes \tilde{P}_2 \bigotimes \tilde{P}_1\right) \subset \mathcal{L}(Q_2) = \mathcal{L}\left(\tilde{P}_0 \bigotimes \tilde{P}_2 \bigotimes \hat{P}_1\right).$$

The language containment of the induction base is trivially satisfied because the language of $\hat{P}_1$ is larger than the language of $\tilde{P}_1$.

The "next" operator for $\hat{P}_i$ is given by: $\phi(\hat{P}_i) = \hat{P}_{i+1}$. We thus have that the induction step is given by

$$\mathcal{L}\left(\tilde{P}_0 \bigotimes \tilde{P}_3 \bigotimes \tilde{P}_2 \bigotimes \hat{P}_1\right) \subset \mathcal{L}(Q_3) = \mathcal{L}\left(\tilde{P}_0 \bigotimes \tilde{P}_3 \bigotimes \hat{P}_2\right).$$

Although one may want to submit the induction step to verification by the tool Cospan, it is clearly satisfied because:

1. The language of $\hat{P}_2$ is larger than the language of $\tilde{P}_2$;
2. The other processes in the right hand side of the equation are the same as the ones in the left hand side. The additional process in the left hand side can only constrain the language of the product in the left hand side.

With the induction step satisfied, we can conclude that, for an arbitrary number of followers iv ≥ 2,

$$\mathcal{L}\left(\tilde{P}_0 \bigotimes (\bigotimes_{i=N}^{1} \tilde{P}_i)\right) \mathbf{C}\ \mathcal{L}(Q_N).$$

Appendix *A* documents the verification that Q_2 performs the task *T* defined in section 1.2. For any i > 2, Q_i performs the task *T* because the processes of Q_i are the same as the processes of Q_2, i.e., $\tilde{P}_i$ is equal to $\tilde{P}_2$ of Q_2, $\hat{P_{i-1}}$ is equal to $\hat{P}_1$ of Q_2, and both share $\tilde{P}_0$. Thus we may conclude that the entry maneuver is correct for any number of follower vehicles, i.e.,

$$\mathcal{L}\left(\tilde{P}_0 \bigotimes (\bigotimes_{i=k}^{1} \tilde{P}_i)\right) \subset \mathcal{L}(T), \forall 2 \leq k \leq N.$$

1.3.2 Timed Formal Specification and Verification

Many details were abstracted in the untimed system specification via non-deterministic outputs or delays. Because lower and upper bounds of these delays are known, we can make use of timed specification and verification to achieve a more detailed and accurate model. In [1], we find a lower bound of 16.1 seconds for the time a vehicle takes to accelerate from a stopped position to a speed compatible with the platoon in the AL at the first gate. There, we also find a lower bound of 19.1 seconds for the time a stopped vehicle takes to accelerate to the second gate. While the upper bounds of acceleration remain to be estimated, the wait at a stop sign is estimated to vary in the interval [0, 10] seconds. Other timing information concerns lower and upper bounds for:

- The time a leader takes to reach the turn marker;
- The delay of vehicles arriving at the stop light after passing the check-in point;
- The delay of vehicles arriving at the check-in point (this needs to reflect some choice of interarrival distribution for the vehicles);
- The delay between vehicles in the entering pre-platoon:
  - How long after vehicle ⁎ achieves a correct velocity for entry, should vehicle *i* + 1 achieve the same velocity;
  - How long after vehicle i sees a turn marker, should vehicle i + 1 see the turn marker;
  - How long after vehicle *i* completes a lane change, should vehicle i + 1 complete its change lane maneuver.
- Changing lanes;
- Reaching the next EMS;
- Timeouts for the reception of responses.

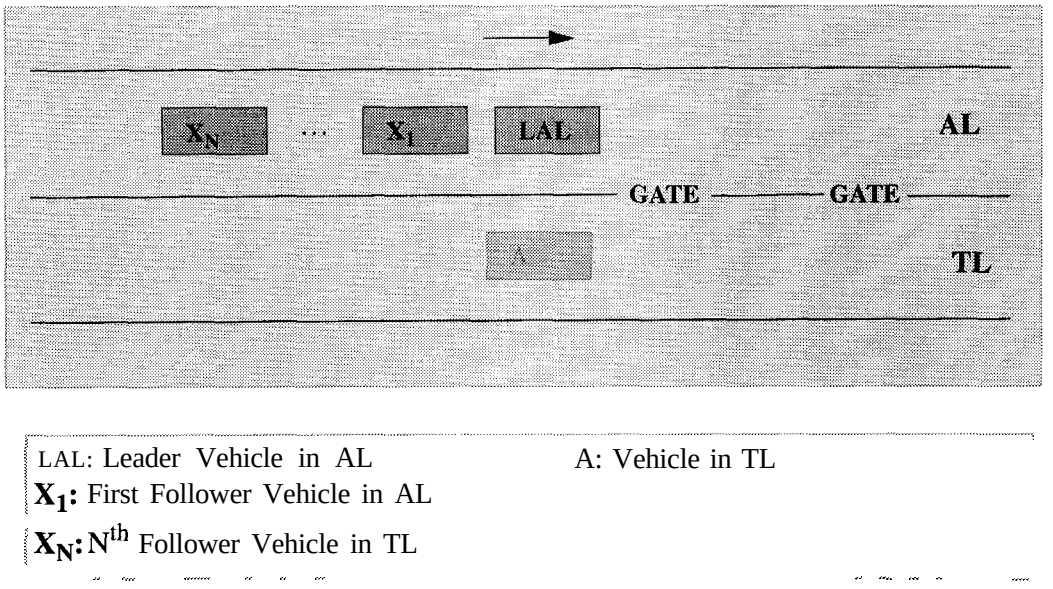


Figure 10: Exit maneuver system

Let S be a given system of L-processes. In order to add timing information to a system, one may construct an automaton S_T, which removes the timing inconsistent sequences from the language of the system S, i.e., the system given by $S \otimes S_T$ is timing consistent. Such an approach is taken, for instance, in [6]. Because timing information constrains the behavior of a system, the same properties verified for the untimed situation hold for the timed system.

The desirable behavior of the timed system can be modified to incorporate bounds on delays. For example, one desirable behavior is that a vehicle in the TL must either enter the AL within one minute, or proceed in the TL in search of a new entry section. Another desirable behavior is that a vehicle admitted to the automated highway system must enter the automated lane within k minutes, or it must exit the system. The new desirable behavior needs to be properly expressed as timed L-automata, and the timed system verified against them.

At the time of this writing, many of these delays are not yet known, thus the specification of the entry maneuver with timing information will be the subject of further research.

1.4 Exit Maneuver System

Figure 10 shows the exit maneuver system with multiple gates.

In the automated lane, vehicle *LAL* is the platoon leader; vehicles $X_1, \ldots, X_n$ are the followers. They all may request exit. In the transition lane, vehicle *A* represents a free agent that missed its entry in the previous entry maneuver section (EMS), or a vehicle which exited recently and which failed to leave the TL.

As the vehicles *LAL,* $X_1, \ldots, X_n$ exit, their order in the transition lane is not necessarily the same

as in the automated lane. Both the leader in the automated lane and the leader in the transition lane need to maintain information about the vehicles participating in the exit maneuver. Any vehicle in the automated lane, which is not a leader, may become one if a vehicle ahead of it exits. Also, a follower in the automated lane may become a leader in the transition lane, if no other vehicle of its platoon exits before it does, or if it is required to exit in front of a vehicle (or platoon).

Different exiting configurations are possible. Consider, for instance, a system with three potentially exiting vehicles, ***LAL,*** X_1, and X_2. The possible configurations for exit are: *(i)* only ***LAL*** wants to exit, *(ii)* ***LAL*** and X_1 want to exit, *(iii)* ***LAL*** and X_2 want to exit, (iv) ***LAL,*** X_1 and X_2 want to exit, *(v)* only X_1 wants to exit, *(vi)* only X_2 wants to exit, *(vii)* X_1 and X_2 want to exit, *(viii)* none want to exit.

The logical steps needed for this maneuver are shown in Figure 11.

One of our goals is to prove that the exit maneuver is correct with respect to the following properties:

- At most ***M*** out of N > ***M*** vehicles take an exit with ***M*** gates;
- Non-requesting vehicles do not exit;
- Intra-platoon spacing in the automated lane is closed up after a vehicle exits;
- The exit maneuver completes for all participating vehicles;
- A vehicle which requests exit does eventually exit;
- Only one vehicle exits via an exit gate at one time.

In order to prove these properties, we have modeled the system in the L-automata framework as shown in Figure 13. At the coordination layer of control, we have specified identical L-processes for the vehicles ***LAL,*** $X_0, \cdots, X_n$. In order to model the fact that any vehicle can be a leader, we have created an L-process which performs all the required data base update functions (shown in Figure 13 as DB), and which is dynamically associated with the leader vehicles on the AL and the TL.

The regulation layer of control is modeled via the following L-processes:

Vr : vehicle velocity response for exit. The model of the velocity response for the followers (Vr_i) is different from that of the leader (Vr_0), because the response of each follower has to be consistent with the response of the leader, and of all followers ahead of it.

Sr : vehicle lateral sensor response. The model of the lateral sensor for the followers (Sr_i) is different from that of the leader (Sr_0), because it is necessary to represent the fact that each follower can only sense an object on the highway after a leader, and all followers ahead of it have already sensed it.

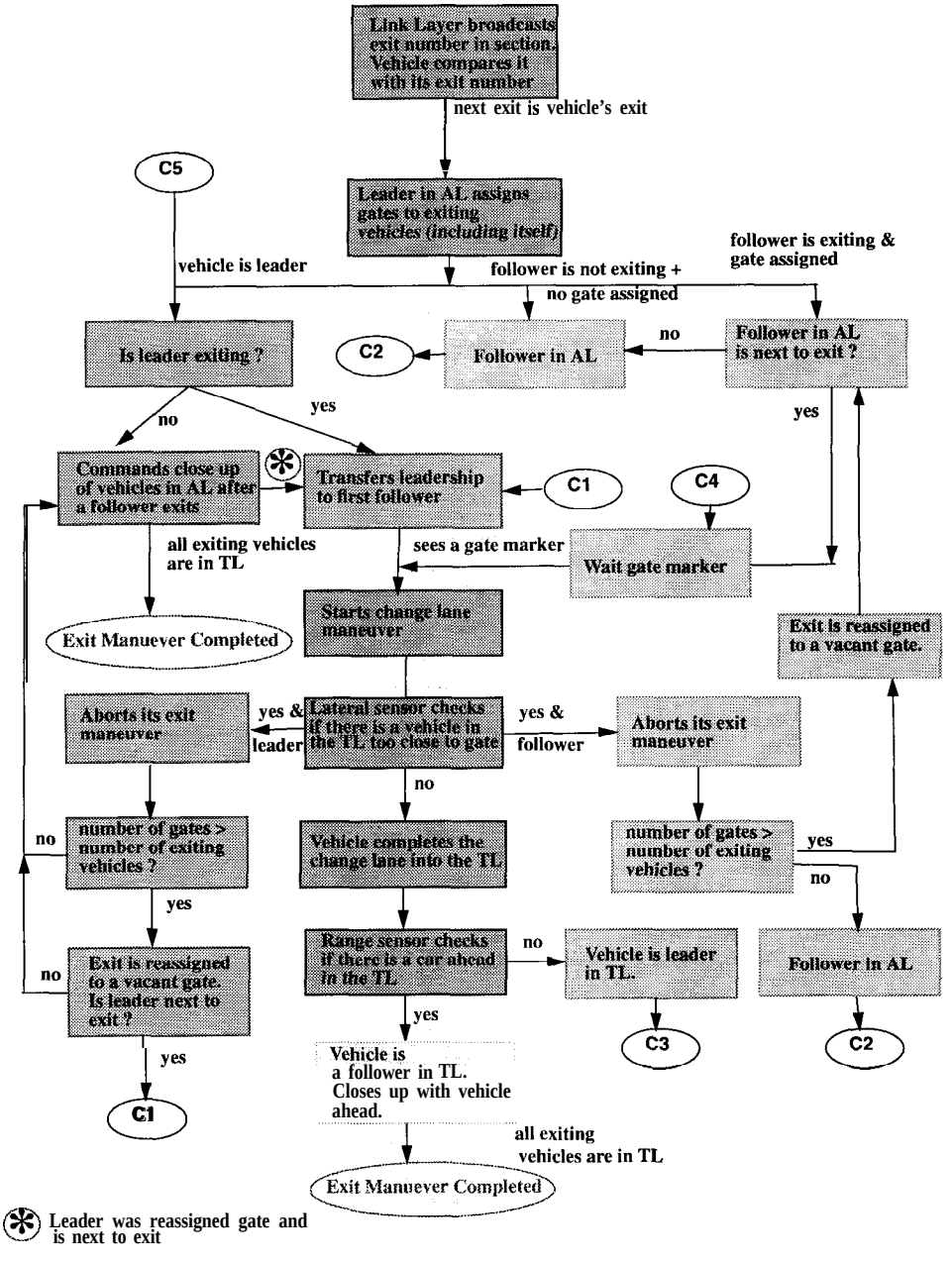


Figure 11: Logical steps for exit maneuver

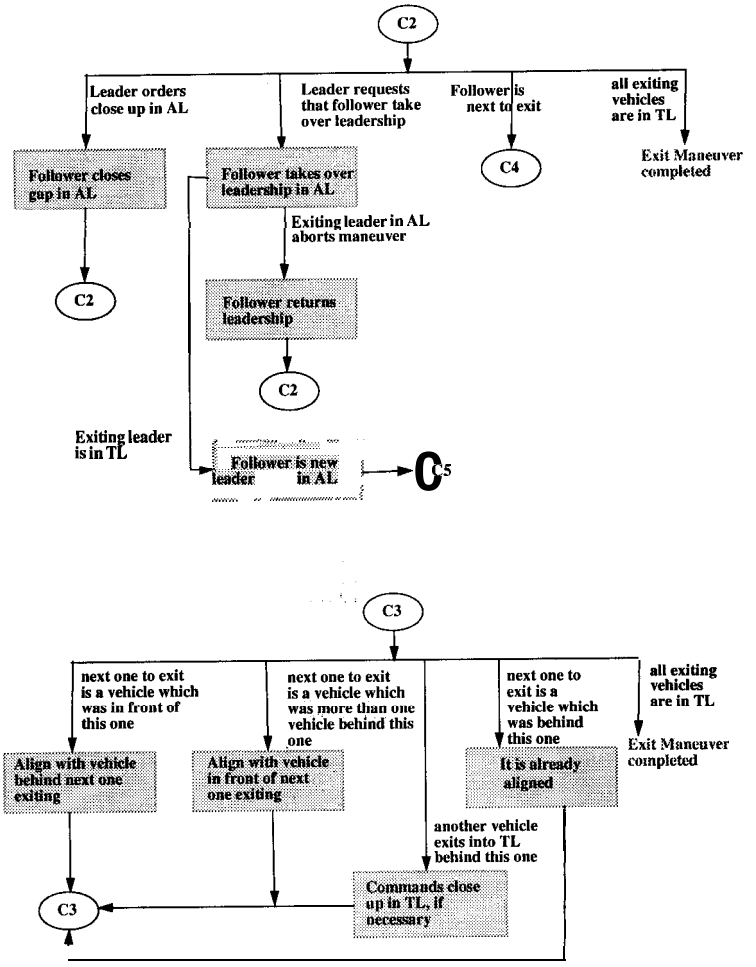


Figure 12: Logical steps for the exit maneuver (continuation)

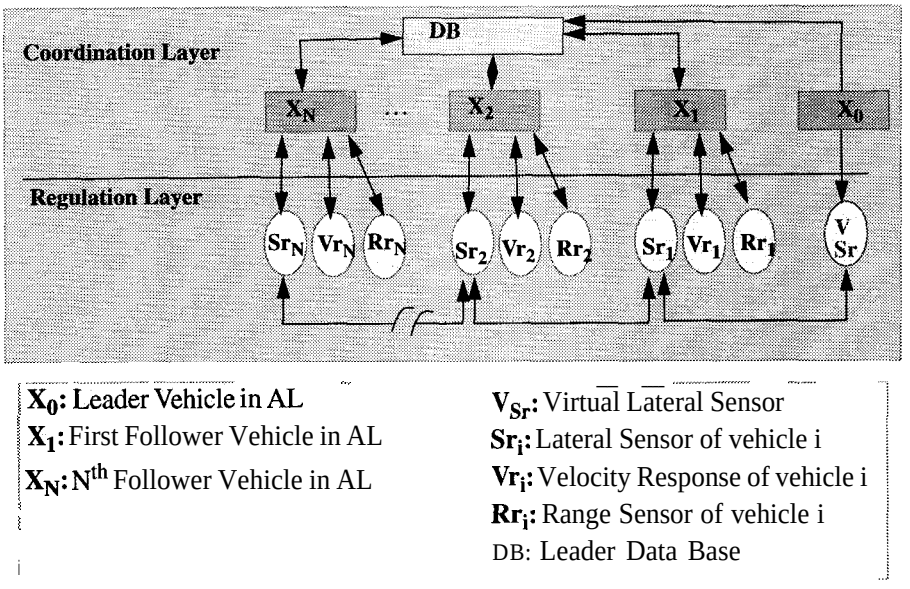


Figure 13: Exit maneuver: system model

V_{Sr} : "virtual" lateral sensor response. This component is used to maintain a constant leader sensing component in the AL, even though platoon leadership may dynamically change.

Rr : vehicle range sensor response. This component senses whether an exiting vehicle is a leader in the TL. The range sensors of the various vehicles also need to be modeled in a way that no two vehicle responses are contradictory.

In the transition lane, vehicles whose distance to the gate might interfere with the exit maneuver are modeled as a variable within the lateral sensor response of the exiting vehicles. This variable is non-deterministically chosen to be "vehicle too close" or "vehicle OK distance".

1.5 Formal Specification and Verification of the Exit System

The untimed formal specification of the modified system is given in Appendix B. The desirable behavior of this system (without timing constraints) is modeled by the L-automaton $Tusk\ A$, which checks for the behaviors 1 through 4 listed above.

In order to guarantee that this system's specification is amenable to analysis, we have made the following simplifications:

- Only three vehicles are modeled on the AL, and
- only four gates at most are modeled.

1.6 Induction Proof for the Exit System

Similar to the entry maneuver, we will prove that the exit maneuver presents the desirable properties listed in section 1.4. In order to extend the proof to an arbitrary number of X_i vehicles, once again we apply induction methods explained in section 1.3. In order to proceed with the induction proof, we fix the number of gates to three.

The exit system, parametrized by the number of follower vehicles N, is given by:

$$\begin{aligned} S(N) &= P_0 \bigotimes (\bigotimes_{i=1}^{N} P_i), \\ P_0 &= DB \bigotimes V_{Sr}, \\ P_i &= X_i \bigotimes Sr_i \bigotimes Vr_i \bigotimes Rr_i, i > 0. \end{aligned}$$

The processes of the exit system, interconnected in a star topology, communicate with the hub of the star, the L-process *DB*. Similar to the entry system, the hub of the star has variables which depend upon a parametrized number of processes N. Because the number of processes grow without bound, and because a hub of a star accumulates information from other N components in the system, it is possible that no invariant independent of N exists. In order to apply induction methods to the exit system, one needs first to determine whether this system is linearizable.

1.6.1 Linearization of the Exit System

By inspection, one can see that the exit system is linearizable because:

(a) the variables of P_i, for $i > 0$, are of bounded computation, and the number of variables does not depend upon the number of follower vehicles N in the system, hence of $\mathcal{O}(1)$ as required by theorem 1 .1;

(b) the number of variables of P_0 depends upon the number N of follower vehicles in the system, hence of 0(N);

(c) the variables of P_0 are either of bounded computation, or associative. One example of an associative variable is the output "exit-info", which the leader *LAL* maintains in order to know whether the followers have requested exit. This variable is associative, because the number of vehicles which request exit can be computed recursively in the following way. Each follower vehicle maintains its own information about exit request. The last follower propagates its exit request information to the vehicle ahead of it, which propagates this information, plus its own exit request information, to the next vehicle. The propagated number of exit requests that reaches the leader *LAL* is precisely the same number that the leader would be able to determine;

(d) the number of variables in P_0 that depend upon the number of follower vehicles in the system is bounded.

By applying the construction suggested in the proof of [5, Theorem 3.11, we have that the linearized exit system is given by:

$$\begin{aligned}\tilde{S}(N) &= \tilde{P}_0 \bigotimes (\bigotimes_{i=1}^{N} \tilde{P}_i),\\ \tilde{P}_0 &= \tilde{DB} \bigotimes V_{Sr},\\ \tilde{P}_i &= \tilde{X}_i \bigotimes Sr_i \bigotimes Vr_i \bigotimes Rr_i, i > 0 .\end{aligned}$$

The process $\tilde{P}_0$ has *a* modified *DB* process, as follows:

- All variables which were arrays in P_0 are changed into a scalar variable. These new variables are assigned the value of the variable with the same name in $\tilde{P}_N$, which is the propagated value in range N, · . . , 1.
- The array msg_from_X[cars] needs to be split into several variables. To each value in the domain of this array corresponds a variable which represent the number of vehicles in the range that have sent a message with the same value. These variables are assigned to the propagated value of $\tilde{P}_N$;
- The variables msg-to-X, to_X, gate-DB, assign-DB are removed, because the vehicles $\tilde{P}_i$ assign themselves to the gates.
- Variable msg-to-C is removed, because vehicles request the neighbor downstream to close up in *AL* or in *TL,* or to take over leadership.
- The variable exiting-cars is removed, because it becomes redundant (with exit-DB) in the linearized system;
- The variables l_AL and l_TL are changed to leader-index41 and leader-index-TL, and assigned to the value of the variables of same name in P_N.

Each process $\tilde{P}_i$, i > 0, has several new state and selection variables:

- A new state variable exitinfo that remembers whether this vehicle has selected itself for exit, and *a* new selection variable exit-DB that copies the values of the new state. Also, another new variable is used to propagate the number of vehicles in the range N, . . . , i + 1 that have selected themselves for exit.
- New propagation variables:
  - waiting-exit, which adds to the propagated value from the i + 1^{th} vehicle, whether the i^{th} vehicle is waiting exit;
  - index_in_TL, which discovers the most recent vehicle that exited to the TL;
  - index-abort, which discovers the index of the vehicle that aborted last;

– index-next-vehicle, which discovers the index of the vehicle next to exit.

- A new local variable gate-assignment that assigns a gate based on the assignment of the previous vehicle.
- A new local variable msg_to_C that indicates to vehicle P_{i-1} when it needs to close up in AL, or in TL, or to take over leadership in AL. Note that if vehicle P_{i-1} is not in the same lane as P_i, P_{i-1} is required to propagate this request downstream.
- A new local variable msg_from_C, that indicates that vehicle P_{i-1} did complete the action requested via msg_to_C.

We have made the assumption that P_0 always communicates with the end of a string of processes, i.e., for a system with N processes, it always communicates with $\tilde{P}_N$. We thus view the leader in the automated lane followed by the vehicle which has the highest index. The tail of the platoon is the vehicle with the lowest index.

The "next" operator applied to the new components in $\tilde{P}_i$, and $\tilde{P}_0$ is defined as follows:

$$\begin{aligned} \phi(\tilde{X}_i) &= \tilde{X}_{i+1}, \\ \phi(\tilde{DB}) &= \tilde{DB}. \end{aligned}$$

We propose an invariant given by:

$$\begin{aligned} Q_i &= \tilde{P}_0 \bigotimes \tilde{P}_i \bigotimes \tilde{P}_{i-1} \bigotimes \hat{P}_{i-2}, \text{ where} \\ \hat{P}_{i-2} &= FREE(\tilde{X}_{i-2}) \bigotimes Sr_{i-2} \bigotimes Vr_{i-2} \bigotimes Rr_{i-2}. \end{aligned}$$

The idea behind the invariant is that all the possible behaviors of a platoon with N > 3 vehicles, for a fixed number of gates, is representable by a platoon with three vehicles, the N, N – 1, and N – 2 vehicles.

The induction base is given by:

$$\mathcal{L}\left(\tilde{P}_0 \bigotimes \tilde{P}_3 \bigotimes \tilde{P}_2 \bigotimes \tilde{P}_1\right) \subset \mathcal{L}\left(\tilde{P}_0 \bigotimes \tilde{P}_3 \bigotimes \tilde{P}_2 \bigotimes \hat{P}_1\right).$$

The language containment of the induction base is trivially satisfied because the language of $\hat{P}_1$ is larger than the language of $\tilde{P}_1$.

The "next" operator for $\hat{P}_i$ is given by: $\phi(\hat{P}_i) = \hat{P}_{i+1}$. We thus have that the inductive step is given by:

$$\mathcal{L}\left(\tilde{P}_0 \bigotimes \tilde{P}_4 \bigotimes \tilde{P}_3 \bigotimes \tilde{P}_2 \bigotimes \tilde{P}_1\right) \subset \mathcal{L}\left(\tilde{P}_0 \bigotimes \tilde{P}_4 \bigotimes \tilde{P}_3 \bigotimes \hat{P}_2\right).$$

Although one may submit the inductive step to verification by the tool Cospan, it is clearly satisfied because:

- the language of $\hat{P}_2$ is larger than the language of $\tilde{P}_2$;
- the other processes in the right hand side of the equation are the same as the ones in the left hand side. The additional process in the left hand side can only constrain the language of the left hand side, thus guaranteeing language containment.

With the induction step satisfied, we can conclude that, for an arbitrary N $\geq$ 3:

$$\mathcal{L}\left(\bigotimes_{i=0}^{N} P_i\right) \subset \mathcal{L}(Q_N).$$

Appendix *B* documents the verification that Q_3 performs the task *T* defined in section 1.4. for any $i > 3$, Q_i performs the task *T* because the processes of Q_i are the same as the processes of Q_3, i.e., $\tilde{P}_i$ is equal to $\tilde{P}_3$ of $Q_3, \tilde{P_{i-1}}$ is equal to $\tilde{P}_2$ of $Q_3, \hat{P_{i-2}}$ is equal to $\hat{P}_1$ of Q_3, and both share $\tilde{P}_0$. Thus we may conclude that the exit maneuver is correct for any number of follower vehicles, i.e.,

$$\mathcal{L}\left(\bigotimes_{i=0}^{k} \tilde{P}_i\right) \subset \mathcal{L}(T), \forall 3 \leq k \leq N.$$

1.6.2 Timed Formal Specification and Verification

Timing information for the exit maneuver includes lower and upper bounds for:

- the time interval for the change lane maneuver;
- the time interval for acceleration or deceleration of exiting vehicles;
- the time interval for the leader to reach each turn marker;
- the time interval for followers to reach the turn markers;
- the time interval for the close-up after a vehicle leaves;
- the ime interval for vehicles to reach the next XML.

Similar to the timed specification of the entry maneuver, the desirable behavior of the timed exit maneuver system can be modified to incorporate bounds on delays. At the time of this writing, timing constraints were largely unknown, and thus the specification of the exit maneuver with timing information will be the subject of future research.

References

[1] P. Varaiya. AHS entry/exit implementation. AHS Precursor Systems Analysis contract report. Submitted to Federal Highway Administration, December 1994.

[2] A. Hitchcock. Organization of exit and entry on an automated freeway. Technical Report 8, PATH MOU 19, Occasional Papers, Institute of Transportation Studies, University of California, Berkeley, 1993.

[3] S. R. Sachs. *Formal Verification of Discrete Event and Hybrid Systems.* PhD thesis, University of California, Berkeley, Berkeley, California, June 1995.

[4] R. P. Kurshan and K. McMillan. A structural induction theorem for processes. In *Proceedings of 8th ACM Symp. on Principles of Distributed* ***Computing,*** pages 239-247, 1989.

[5] R. P. Kurshan, M. Merritt, A. Orda, and S. R. Sachs. A structural linearization principle for processes. In *Proceedings of Computer-Aided Verification, 5th International C onference, CAV 93, Elounda, Greece,* pages 491-504, June 1993.

[6] R. Alur, R. Itai, R. P. Kurshan, and M. Yannakakis. Timing verification by successive approximations. In *Lecture Notes in* ***Computer Science,*** volume 663, pages 137-150, 1993.

2 Appendix A

This appendix contains the latest run of the verification tests for the entry maneuver, and the COSPAN code of the Entry Maneuver discussed in Section 1.2. Please refer to Figures 6, 7, 8, and 9.

```
Tue Sep 13 18:32:27 PDT 1994
moonlight [SunOS 4.1. 1 sun4c] : /tmp_mnt/net/automount/sur0/varaiya/ssachs/private/research.
cpn : cospan -DN=2 -DOP=1 -DTeventually entrylast.sr

cospan: Version 8.8.2 (AT&T-BL) 22 Apr 1994
+ sr -DN=2 -DOP=1 -DTeventually entrylast.sr -o entrylast.c
entrylast.sr: Tue Sep 13 18:31:59 1994
./1.h: Sun Sep  4 14:50:17 1994
27 selection/local variables
26 bounded state variables: 8.84e18 states
0 unbounded state variables
0 boolean cysets
2 free selection/local variables: 6 selections/state
8 pausing processes
4 non-deterministic (non-free) selection/local variables
16 selections/state (maximum)
96 total selections/state (maximum)

sr: 8 pausing processes
+ cc -o entrylast.an -I/home/sur0/varaiya/kurshan/include entrylast.c /home/sur0/varaiya/ku
+ ./entrylast.an
./entrylast.an: Synchronous model
entrylast.an: Initialization complete.
1 initial state.
entrylast.an: Search complete.
47323 states reached.
47323 states searched.
4281 DFS trees generated.
4528 nontrivial SCC's generated.
133944 edges transversed:
10766 plus, 43042 tree, 1014 self, 58 forward,
3739 back, 1785 cross-intra, 73540 cross-inter.
778602 resolutions made.
2+12 boundary frames allocated.
290.133 cpu seconds
entrylast.an: Task performed!
```

```
proctype Check-post (LTL, X:proc)   /* check post machine */      Check_post()

    import LTL, X
    selvar #:(see_vehicle, see-no-vehicle, are_you_auto, allow-entry,
          stop_entry)
    stvar $:(idle, active, allow-entry)
    cyset {idle}
    init  idle
    trans

    idle                {see_vehicle, see_no_vehicle}
        ->active        : # = see_vehicle
        ->$             : else;

        /* All vehicles which are to enter will see at the same
           time the message 'are you auto' */
    active              {are_you_auto}
        ->allow_entry : (LTL.#=yes_auto)+((+[i in 0..range](
                          1 ? X[i].#=yes_auto | 0))>0)
        ->$             : else;

    allow-entry           {allow-entry)
        ->idle          : LTL.#=idle
        ->$             : else;


    end   /* Check-post */
```

```
proctype Sensor (LAL, SL, LTL:proc)   /* Entry sensor*/          Sensor()

    import LAL, SL, LTL
    selvar #:(no_space, space, I-found-space, what_is_your_id)
    stvar $:(idle, no-space, space, found-space)
    stvar L_id      : (nul, id, no-id)
    asgn    L-id    ->    id      ? (LAL.#=id) |
                          no_id  ? (LAL.#=empty_space) |
                          nul     ? ($=idle) |
                          L_id
    selvar leader_id: (nul, id, no-i-d)
    asgn  leader_id:= L_id
    selvar x:(0..2)
    asgn   x:={0,1,2}
    stvar  space-size: (0..3)
    asgn   space-size -> 1       ? (x=0)*(#=space) |
                         2       ? (x=1)*(#=space) |
                         3       ? (x=2)*(#=space) |
                         0       ?   ($=no_space) |
                         space-size
    selvar how-much-space: (0..3)
    asgn how-much-space:= space-size
    cyset  {no_space}
    init  L_id:=nul, space_size:=0
    init   idle
    trans

    idle                  {no-space}
         ->no_space       : SL.#= wake-up-sensor
         ->$              : else;

    no-space               {no-space, space)
         ->space    :     #=space
         ->$              : else;

    space                  {what-is-your-id)
         ->found_space : (LAL.#=id) + (LAL.#=empty_space)
         ->$              : else;

    found-space            {I-found-space)
         ->idle           : LTL.#=idle
         ->$              : else;

    end    /* Entry Sensor*/
```

```
proctype Light (LTL, ES:proc)   /* light stop machine */            Light()

    import LTL, ES
    selvar #:(red,green, wake-up-sensor)
    stvar $:(red, active, green)
    stvar leader id: (nul, id, no_id)
    asgn   leader_id-> ES.leader_id ?(ES.#=I_found_space)*(leader_id = nul)
                     | nul          ?($=red)
                     | leader-id
    selvar how-much-space: (0..3)
    asgn how-much-space:= ES. how_much_space
    init leader_id:=nul
    init   red
    trans

    red                  {red}
        ->active          : LTL.# =I_am_here
        ->$               : else;

    active               {wake-up-sensor)
        ->green           : ES.#=I_found_space
        ->$               : else;

    green                {green}
        ->red             : LTL.#=accelerate to enter
        ->$               : else;
    end    /* Light */
```

```
proctype  Leader-Lateral-Sensor(LTL,  LAL,  Vr:proc)        Leader_Lateral_Sensor()
          /* leader's lateral sensor machine: it senses velocity
          of the platoon in AL whith which it is coordinating the
          entry maneuver, and the turn markers */
     import LTL, LAL, Vr
     selvar #: (%, turn-marker, nothing)
     selvar   velocity-comparison:   (velocity-correct.
                              velocity-notcorrcct)
     asgn    velocity_comparison:=
                  velocity-correct  ?(Vr.#=full_velocity)|
                  velocity-not-correct

     stvar $: (idle, in_eml, turn)
     cyset  {in_eml@}
     init idle
     trans

     idle                {%}
          ->in_eml         : (LTL.#=accelerate to enter)
          ->$              : else:

     in_eml               {%: turn-marker}
          ->turn           : (# = turn-marker)
          ->$              : else:

     turn                 {turn-marker}
          ->idle           : (LTL.#=idle)
          ->$              : else;

     end    /* Leader Lateral Sensor'/
```

```
proctype Follower_Lateral_Sensor(i: integer; LTL, X, Xs:proc)    Follower_Lateral_Sensor()
        /* followers lateral sensor machine: it senses
           only the turn markers. Velocity of followers as they
           approach the turn marker is assumed to be correct
           because their regulation layer is simply performing
           follower law */
    import i, LTL, X, Xs
    selvar #: (%, turn_marker, nothing)
    stvar $: (idle, turn)
    cyset  {turn@}
    init idle
    trans

    idle                {%}
        ->turn           : (Xs[i].#=ack_come_with_me)*
                           (((i=0)*(LTL.#=reach_gate)) +
                           ((i>0)*(X[i-1].#=reach_gate)) +
                           ((i>0)*(X[i-1].#=abort)) )
        ->$              : else;

    turn                {%: turn-marker}
        ->idle           : (X[i].#=idle)
        ->$              : else;

    end    /* Follower Lateral Sensor*/
```

```
proctypc    Leader-Velocity-Rasponsc(LTL:proc)          Leader-V&city-Response0

      import LTL
      selvar #:(%, cruise, closing, closed-up, approaching,
               reached-stop-light, accelerating, full-velocity,
               changing, change_lane_complete, abort)
      stvar $:(cruising, close_up_in_AL, approach-light, waiting,
               accelerate, ready, change, abort)
      cyset {close_up_in_AL@}, {approach-light@}, {accelerate@),
            (change@}
      init   cruising
      trans

      cruising                    {cruise}
            ->approach_light       : LTL.# = approach stop-light
            ->close_up_in_AL       : LTL.# = close-up-in AL
            ->$                   : else;

      closeup in AL                 {closing: closed_up}
            ->cruising             : (#=closed_up)
            ->$                   : else:

      approach-light                {approaching:    reached-stop-light)
            ->waiting              : (#=reached_stop_light)
            ->$                   : else:

      waiting                     {reached-stop-light}
            ->accelerate           : LTL.# = accelerate_to_enter
            ->$                   : else;

      accelerate                  {accelerating: full-velocity)
            ->ready                : (#=full_velocity)*(LTL.#≠abort)
            ->abort                : (LTL.#=abort)
            ->$                   : else;

      ready                       {%}
            ->change               : (LTL.#=ok_to_change)
            ->abort                : (LTL.#=abort)
            ->$                   : else;

      change                      {changing: change_lane_complete}
            ->cruising             : # = change-lane-complete
            ->$                   : else;

      abort                       {abort)
            ->cruising             : true;

      end    /* Leader Velocity Response */
```

```
proctype Follower_Velocity_Response                Follower_Velocity_Response()
        (i: integer; X,Vr, Vr_X:proc)
    import i, X, Vr, Vr_X
    selvar #:(%, cruise, approaching, reached_stop_light,
          following_leader, changing, change_lane_complete, abort)
    stvar $:(cruising, approach-light, get_there, waiting,
          follow-leader, change, abort)
    cyset {change@), {get-there@}
    init  cruising
    trans
    cruising                    {cruise}
        ->approach_light          : X[i].# = approach_stop_light
        ->$                     : else;
    approach-light                {approaching }
        ->get there               :((i>0)*
                                  (Vr_X[i-1].#=reached_stop_light))
                                 +((i>0)*
                                  (X[i-1].#=entry_complete))
                                 +((i=0)*
                                  (Vr.#=reached_stop_light))
        ->$                     : else;
    gel-there                     {approaching: reached-stoplight}
        ->waiting                 : (#=reached_stop_light)
        ->$                     : else:
    waiting                     {reached_stop_light}
        ->follow_leader           : (X[i].# = follow_leader)
        ->cruising                : (X[i].#= entry_complete)
        ->$                     : else;
    follow_leader                 {following_leader}
        ->change                  : (X[i].#=ok_to_change)
        ->abort                   : (X[i].#=abort)
        ->$                     : else;
    change                        {changing: change_lanc_complete}
        ->cruising                : # = change-lane_complete
        ->$                     : else;
    abort                       {abort}
        ->cruising                : true;
    end   /* Follower Velocity Response */
```

```
proctype Leader-in-AL(LTL, ES:proc)                    Leader_in_AL()
    import LTL, ES
    selvar AL-option: (platoon, space)
    asgn AL-option:=    platoon ? (option = 1) |space

    /* update the platoon size after receiving entry complete message
      from the leader in the TL */
    stvar nps:(0..2*N+1)
    asgn   nps-> pps + LTL.new_size      ? (LTL.#=entry_complete) |
             pps

    selvar #: (%, id, empty-space, ack_entry, nack_entry, entry-complete)
    stvar $: (idle, send-id, empty-space, check-busy,
          wait-complete,   entry-complete)
    cyset {check-busy}
    init nps:=pps
    init idle
    trans
    idle                {%}
        ->send_id      : (ES.#=what_is_your_id)*(AL_option=platoon)
        ->empty_space  : (ES.#=what_is_your_id)*(AL_option=space)
        ->$            : else;

    send_id             {id}
        ->check_busy   : LTL.#=are_you_busy
        ->$            : else;

    empty-space         {empty_space}
        ->idle         : true;

    /* non-deterministically it decides that it is busy or not */
    check-busy          {ackentry, nack_entry}
        ->wait_complete : #=ack_cntry
        ->$            : else;

    wait_complete       {%}
    ->entry_complete     : LTL.#=entry_complete
    ->$                : else;

    entry-complete      {entry-complete)
    ->idle             : true;

    end /* Leader in AL */
```

```
proctype Leader_in_TL(CP, Vr, ES, SL, LAL, LSr, X, Xs, Y:proc)  Leader_in_TL()

    import CP, Vr, ES, SL, LAL, LSr, X, Xs, Y
        /* If a follower is stopped at the stop light, then it
          will respond with an ack to the 'come with me request'.
          If it didn't reach the stop light when the request
          was issued, then no response will be produced */
    selvar invitation[N]:(come_with_me, no-space-now, too_late_to_join,%)
    asgn [i in 0..range]{invitation[i]:=
              come-with-me ? (((($=check_busy)*(LAL.#=ack_entry)) +
                        ($=entering) )
                        *(i< ES.how_much_space) |
              no-space-now  ? ((($=check_busy)*(LAL.#=ack_entry))
                        *(i>_ ES.how_much_space) |
              too_late_to_join ? ($=enter_AL)+($=check_followers)+
                             ($=check_adjustment)+
                             ($=adjust space)+($=entry_complete)|
              %}

    stvar   ppsize: (0  cars)
    asgn    ppsize ->
          ppsize + (+[i in 0..range](
                        1 ? Xs[i].#=ack_comc_with_me |
                        0 ))
                    ? ($=check_busy)*(LAL.#=ack_entry)*
                      ((+[i in 0..range](
                        1 ? Xs[i].#=ack_comc_with_me |
                        0 )) > 0)    |
          ppsize
    selvar new-size: (0 .. cars)
    asgn    new_size:= ppsize

    stvar gate-sensor: (space-still-available, no-space-available, %)
    asgn    gate-sensor->
              space_still_available  ? (Y.position=OK_distance)*
                                ($=cntering)*
                                (LSr.#=turn_marker) |
              no-space-available      ? (Y.position=too_close)*
                                ($=entering)*
                                (LSr.#=turn_marker) |
              gate-sensor
    selvar #: (%, idle, yes-auto, approach-stop-light, I_am_here,
            are_you_busy, accelerate to_enter,
            reach-gate, ok_to _change, abort,
            closeup_in _AL,  entry-complete)
    stvar $: (idle, check-auto, active, at-stop-light. check_busy,
            entering, proceed, reach-gate, enter_AL, check_followers,
            check-adjustment, adjust-space, abort, wait-completion,
            entry-complete)

    cyset {check-auto}
    recur reach_gate->abort
    init   ppsize:=0, gate-sensor:=%
```

```
init idle
trans

idle                    (idle}
      ->check_auto        : CP.#=are_you_auto
      ->$                 : else;

      /* The possibility that the check returns a no-auto is
         not modeled although possible. In the real system, a
         vehicle checking no-auto will not be allowed in the TL */
check auto              {%, yes_auto}
      ->active            : #=yes_auto
      ->$                 : else;

      /* Commands regulation layer (velocity response) to
         approach the stop light */
active                  {approach-stop-light)
      ->at_stop_light     : Vr.#=reached_stop_light
      ->$                 : else;

      /* Sends the message 'I am here' to the Stop Light */
at-stoplight            {I_am_here}
      ->check_busy        :(ES.#=I_found_space)*
                          (ES.leader_id=id)
      ->entering          : (ES.#=I_found_space)*
                          (ES.leader_id=no_id)
      ->$                 : else;

check-busy              {are_you_busy}
      ->entering          : LAL.#=ack_entry
      ->$                 : else;

      /* Commands the regulation layer to accelerate for entry */
entering                {accelerate-to-enter)
      ->reach_gate        : LSr.# = turn-marker
      ->$                 : else;

reach_gate              {reach-gate}
      ->enter_AL          : (gate-sensor =space_ still-available)
                  *(LSr.velocity_comparison=velocity_correct)
      ->abort             : (gate-sensor =no_space_available)
                  +(LSr.velocity_comparison=velocity_not_correct)
      ->$                 : else;

abort                   {abort)
      ->entry_complete    : (+[i in 0..range]
                          (1 ? Xs[i].#=abort ! 0))
                          = N
      ->$                 : else;
```

```
        /* Commands the regulation layer to change lane */
enter_AL                    {ok_to_change}
        ->check_followers     : Vr.#=change_lane_complete
        ->$                 : else;

check-followers             {%}
        ->check_adjustment    : (+[i in 0..range]
                            (1 ? (Xs[i].# = entered_AL)+
                              (Xs[i].# = abort,) | 0))
                            = N
        ->$                 : else;

check-adjustment            {%}
        ->adjust_space        : ES.leader_id=id
        ->wait_completion     : else;

        /* Commands regulation layer to close up just entered
           platoon with the platoon in AL, if any */
adjust-space                {close_up_in_AL}
->wait_completion             : Vr.#= closed-up
->$                         : else;

wait-completion             {%}
        ->entry_complete      : (+[i in 0..range]
                            (1 ? (Xs[i].#=entered_AL)+
                              (Xs[i].#=abort)|0))
                              = N
        ->$                 :else;

entry-complete              {entry_complete}
        ->idle              : (+[i in 0..range]
                            (1 ? (X[i].#=entry_complete)|0)) = N
        ->$                 :else;


end     /* Leader in TL */
```

```
proctype Follower-in-TL(i: integer; LTL, CP,X,Vr_X,LSr_X,Y:proc)      Follower_in_TL()
    import i, LTL, CP, X, Vr_X,LSr_X, Y
    stvar gate-sensor: (space_still_available,no_space_available, %)
    asgn  gate_sensor->
            space-still-available ?(Y.position=OK_distance)*
                                  ($=go_with_leader)*
                                  (LSr_X[i].#=turn_marker) |
            no-space_available     ? (Y.position=too_close)*
                                  ($=go_with_leader)*
                                  (LSr_X[i].#=turn_marker) |
            gate-sensor
    selvar #: (%, idle, yes-auto, approach-slop-light. at_stop_light,
          follow_leader, abort., reach_gate,
          ok_to_change, entry complete)
    stvar $:(idle, check_auto, active, at-stop-light., go_with_leader,
          abort, reach-gate, enter_AL,entry_complete)
    cyset {check_auto@}
    recur reach_gate->abort
    init  gate-sensor:=%
    init  idle
    trans
    idle                        {idle}
        ->check_auto              : CP.#=are_you_auto
        ->$                       : else;
        /* The possibility that the check returns a no auto is
           not modeled,although possible. In the real system, a
           vehicle checking no-auto will not, be allowed in the TL */
    check_auto                    {%: yes-auto}
        ->active                  : (#=yes_auto) +
                                  ((i=0)*(LTL.#=yes_auto)) +
                                  ((i>0)*(X[i-1].#=yes_auto))
        ->$                       : else;
        /* Commands regulation layer(velocity response) to
           approach the stop light */
    active                        {approach_stop_light}
        ->at_stop_light           : Vr_X[i].#=reached stop light,
        ->$                       : else:
```

```
        /* The follower is prevented from entry at the same
           time that a leader in two cases:
           1. It is at the stop light, but it does not receive an
              invitation to enter with the leader for lack of
              space in the AL (given by the 'ES.how much_space');
           2. It is not at the stop light when the leader is
              entering the Entry_Maneuver Section.
           The first follower which is prevented entry becomes
           a leader of a new round. In this model we view this
           vehicle as a new round of the process 'LTL', thus
           is is possible to 'finish' this process under these
           two conditions. */
at_stop_light                {at_stop_light}
        ->go_with_leader       : (LTL.invitation[i]=come_with_me)
        ->entry_complete       : (LTL.invitation[i]=no_space_now)
                             + (LTL.invitation[i]=too_late_to_join)
        ->$                    : else;

        /* Commands regulation layer (velocity response) to
           follow the leader as it accelerates to enter */
go_with_leader               {follow-leader}
        ->abort              : (LTL.#=abort)
        ->reach_gate         : (LTL.#=/abort)*
                             (LSr_X[i].#= torn-marker)
        ->$                  : else;

abort                        (abort)
        ->entry_complete       : true;

reach_gate                   {reach_gate}
        ->enter_AL           : (gate-sensor = space-still_available)
        ->abort              : (gate_sensor = no_space_available)
        ->$                  : else;

        /* Commands the regulation layer to change lane */
enter_AL                     {ok-to-change)
        ->entry_complete       : Vr_X[i].#=change_lanc_complete
        ->$                  : else;

entry-complete               {entry_complete}
        ->idle               : (LTL.#=entry_complete)*
                             ((+[k in 0..range]
                             (1 ? X[k].#=entry_complete|0))=N)
        ->$                  : else;

end     /* Follower in TL */
```

```
proctype status(i: integer; Vr_X, X, LTL:proc)                    status()

    import i, Vr_X, X, LTL
    stvar $:(no_response, at-stop-light, ack_come_with_me, abort,
            entered_AL)
    selvar #:(no_response, at-stop-light, ack_come_with_me, abort,
            entered-AL)
    init no-response
    trans
    no-response              {no-response)
          ->at_stop_light : (Vr_X[i].#=reached_stop_light)
          ->abort         : (X[i].#=abort)
          ->$             : else;

    at-stop-light            (at-stop-light}
          ->ack_come_with_me
                          : (LTL.invitation[i]=come_with_me)
          ->no_response   : (X[i].#=idle)
          ->$             : else;

    ack  come-with-me        {ack_come_with_me}
          ->entered_AL    : (X[i].#=ok_to change)
          ->abort         : (X[i].#=abort)
          ->$             : else;

    abort                    {abort}
          ->no_response   : (X[i].#=idle)
          ->$             : else;

    entered-AL               {entered_AL}
          ->no_response   : (X[i].#=idle)
          ->$             : else;

    end /*_status_()*/
```

```
    /* Entry sensor machine */
proc ES: Sensor (LAL, SL, LTL)                                  ES
    /* Check post machine */
proc CP: Check_post (LTL, X)                                    CP
    /* Stop Light machine */
proc SL: Light (LTL, ES)                                        SL
    /* TL Leader lateral sensor for entry */
proc LSr: Leader_Lateral_Sensor(LTL, LAL, Vr)                   LSr
    /* TL Leader velocity response for entry */
proc Vr: Leader-Velocity-Response(LTL)                          Vr
    /* TL Followers lateral sensor for entry */
proc LSr_X[i<N]: Follower_Lateral_Sensor(i, LTL, X, Xs)         LSr_X
    /* TL Followers velocity response for entry */
proc Vr_X[i<N]: Follower_Velocity_Response(i,X, Vr, Vr_X)       Vr_X
    /* TL Leader protocol machine */
proc LTL: Leader_in_TL (CP, Vr, ES, SL, LAL, LSr, X, Xs, Y)     LTL
    /* AL Leader protocol machine */
proc LAL: Leader-in-AL (LTL, ES)                                LAL
    /* AL random vehicle protocol machine */
proc Y: Vehicle in AL (LSr, Vr, LSr_X,Vr_X,LTL)                 Y
    /* TL Followers (X[i]) protocol machines */
proc X[i<N]: Follower_in_TL(i, LTL, Cl', X, Vr_X,LSr_X, Y)      X
proc Xs[i<N]: status(i,Vr_X, X, LTL)                            XS
```

```
#if Teventually monitor eventually-enter                    eventually-enter
      import S_LTL, SLAL, S-X
      stvar $: (zero, one, two, three)
      cyset {zero}, {zero, one, two, three}
      init zero
      trans
      zero
      ->one  : (S_LTL.#=approach_stop_light)
      ->$    : else;

      one
      ->two  : (S_LTL.#=entry_complete)
      ->$    : else:

      two
      ->three : (+[i in 0..range]
              (1 ?S_X[i].#=entry_complete| 0))
              =N
      ->$  :  clse;

      three
      ->zero : (S_LAL.#=entry_complete)
      ->$    : else;

end /* eventually_enter */#endif

monitor collision: STOP(((S_LTL.$=enter_AL)*                collision
                   (S_LTL.gate_sensor=no_space_available))+
                   ((+[i in 0..range]
                         (1 ? (S_X[i].$=enter_AL)*
                              (S_X[i].gate_sensor=no_space_available)
                         |0)) >0) )
```

3 Appendix B

This appendix contains the latest verification run for the exit maneuver, and the COSPAN code of the Exit Maneuver discussed in section 1.4. Refer to figures 10, 11, 12, and 13.

```
/* This run specifies that all followers request exit */
/* Runtime parameter x=3 */

Fri Aug 19 01:07:46 PDT 1994
moonlight [SunOS 4.1.1 sun4c]: /tmp_mnt/net/automount/sur0/varaiya/ssachs/private/research.
cpn : cospan -Dfollowers=2 -Dgates=2 -Dx=3 exit8.sr

cospan: Version 8.8.2 (AT&T-BL) 22 Apr 1994
+ sr -Dfollowers=2 -Dgates=2 -Dx=3 exit8.sr -o exit8.c
exit8.sr: Fri Aug 19 01:07:21 1994
./1.h: Thu Aug 4 15:19:31 1994
64 selection/local variables
45 bounded state variables: 6.54e28 states
0 unbounded state variables
0 boolean cysets
3 free selection/local variables: 8 selections/state
4 pausing processes
0 non-deterministic (non-free) selection/local variables
1 selections/state(maximum)
8 total selections/state (maximum)

sr: 4 pausing processes
+ cc -o exit8.an -I/home/sur0/varaiya/kurshan/include exit8.c /home/sur0/varaiya/kurshan/li
+ ./exit8.an
./exit8.an: Synchronous model
exit8.an: Initialization complete.
1 initial state.
exit8.an: Deadlock at 37(26).
exit8.an: Search complete.
194 states reached.
194 states searched.
2 deadlock states reached.
10 DFS trees generated.
19 nontrivial SCC's generated.
348 edges transversed:
9 plus, 184 tree, 26 self, 8 forward,
7 back, 0 cross-intra, 114 cross-inter.
2112 resolutions made.
1+1 boundary frames allocated.
1.23333 cpu seconds
```

```
exit8.an: Task performed!

/* This run specifies that only the first follower requests exit */
/* Runtime parameter x=2 */

Fri Aug 19 00:53:19 PDT 1994
moonlight [SunOS 4.1.1 sun4c]: /tmp_mnt/net/automount/sur0/varaiya/ssachs/private/research.
cpn : cospan -d -Dfollowers=2 -Dgates=2 -Dx=2 exit7.sr

cospan: Version 8.8.2 (AT&T-BL) 22 Apr 1994
+ sr -Dfollowers=2 -Dgates=2 -Dx=2 exit7.sr -o exit7.c
exit7.sr: Fri Aug 19 00:53:10 1994
./1.h: Thu Aug 4 15:19:31 1994
64 selection/local variables
45 bounded state variables: 6.54e28 states
0 unbounded state variables
0 boolean cysets
3 free selection/local variables: 8 selections/state
4 pausing processes
0 non-deterministic (non-free) selection/local variables
1 selections/state (maximum)
8 total selections/state (maximum)

sr: 4 pausing processes
+ cc -o exit7.an -I/home/sur0/varaiya/kurshan/include exit7.c /home/sur0/varaiya/kurshan/li
+ ./exit7.an -d
./exit7.an: Synchronous model
exit7.an: Initialization complete.
1 initial state.
exit7.an: Search complete.
489 states reached.
489 states searched.
27 DFS trees generated.
29 nontrivial SCC's generated.
859 edges transversed:
34 plus, 462 tree, 37 self, 8 forward,
16 back, 8 cross-intra, 294 cross-inter.
5368 resolutions made.
1+1 boundary frames allocated.
3.03333 cpu seconds
exit7.an: Task performed!

/* This run specifies that the leader and the second follower request exit */
/* Runtime parameter x=1 */
```

```
Thu Aug 18 19:09:43 PDT 1994
moonlight [SunOS 4.1.1 sun4c]: /tmp_mnt/net/automount/sur0/varaiya/ssachs/private/research.
cpn : cospan -Dfollowers=2 -Dgates=2 -Dx=1 exit6.sr

cospan: Version 8.8.2 (AT&T-BL) 22 Apr 1994
+ sr -Dfollowers=2 -Dgates=2 -Dx=1 exit6.sr -o exit6.c
exit6.sr: Thu Aug 18 19:09:10 1994
./1.h: Thu Aug 4 15:19:31 1994
63 selection/local variables
45 bounded state variables: 6.54e28 states
0 unbounded state variables
0 boolean cysets
3 free selection/local variables: 8 selections/state
4 pausing processes
0 non-deterministic (non-free) selection/local variables
1 selections/state (maximum)
8 total selections/state (maximum)

sr: 4 pausing processes
+ cc -o exit6.an -I/home/sur0/varaiya/kurshan/include exit6.c /home/sur0/varaiya/kurshan/li
+ ./exit6.an
./exit6.an: Synchronous model
exit6.an: Initialization complete.
1 initial state.
exit6.an: Search complete.
281 states reached.
281 states searched.
16 DFS trees generated.
20 nontrivial SCC's generated.
504 edges transversed:
15 plus, 265 tree, 28 self, 6 forward,
8 back, 0 cross-intra, 182 cross-inter.
3128 resolutions made.
1+1 boundary frames allocated.
1.91667 cpu seconds
exit6.an: Task performed!
```

```
proctype Vr(i:integer; V: proc)        /* velocity response machine for exit Vr */

    import i, V
    selvar #:(%, changing, CL-complete, CL-abort, accclerate, decelerate,
              gap-closed, aligned-with-b-minus-l, aligned_with_b_plus_1)
    stvar $:(cruising, close-up, check-conditions. change,
             abort, align-with_b_minus-l,
             align_with_b_plus_1)
    selvar monitor_TL: (vehicle-too-close, vehicle_OK_distance)
    asgn   monitor_TL:= {vehicle_too_close, vehicle_OK_distance}

    init   cruising
    cyset  {close-up@), {change@},{align_with_b_minus_1},
           {align-with-b-plus-l)
    trans

    cruising                    {%}
           ->close_up             : (V.#=close_up)
           ->check_conditions     : (V.#=ok_to_change)
           ->align_with_b_minus_1 : (V.#=align_with_b_minus_1)
           ->align_with_b_plus_1  : (V.#=align_with_b_plus_1)
           ->$                    : else;

    close_up                    {accelerate: gap_closed}
           ->cruising             : (#=gap_closed)
           ->$                    : else;

    check-conditions              {%}
           ->change               : (monitor_TL = vehicle_OK_distance)
           ->abort                : (monitor_TL=  vehicle-too-close);

    change                      (changing:  CL_ complctc)
           ->cruising             : # = CL_complete
           ->$                    : else;

    abort                       {CL_abort}
           ->cruising             : true;

    align-with_b _minus-l             {decelerate: aligned_with_b_minus_1}
           ->cruising             : (# = aligned_with_b_minus_1)
           ->$                    : else:

    align_with_b_plus_1             {accelerate: aligned_with_b _plus-l}
           ->cruising             : : (#=  aligned-with-b-plus-l)
           ->$                      else:

    end    /* Vr */
```

```
proctype Sr_X(i:integer)                                    Sr_X()
        /* X's lateral sensor machine: it senses when a X vehicle
        reaches the gate marker. Sensing for X[1] depends on sensing
        for A, and sensing for X[2] depends on sensing for X[1]. */
    import i, Virtual-Sr, XSr, X, Info, ME
    selvar #: (%, gate-marker, first-turn-marker, second-turn-marker)
    stvar  $: (nothing, see-gate, wait_gate, gate)
    cyset {gate@)
    recur  gate->nothing
    init nothing
    trans

    /* Only expects to sense a gate after the car in front already
       sensed it */
    nothing         {%}
    ->see_gate      : (Virtual_Sr.#=see_gate)
    ->$             : else;

    see-gate        {%}
    ->wait_gate     :
                    ((Virtual_Sr.#=gate_marker)*(i=0)) +
                    ((XSr[0].#=gate_marker)*(i=1)) +
                    ((XSr[1].#=gate_marker)*(i=2))
    ->$             : else;

    /* Gate cannot be sensed before X[i] is ready to see it */
    wait_gate       {%}
    ->gate          : ((X[i].#=wait_GM)*(Info.next_to_exit=i)) +
                    (Info.msg_to_X[i]=no_exit_gate) +
                    ((Info.assign_DB[i]=exit_assigned)*
                     (Info.next_to_exit+))
    ->$             :    else;

    gate            {gate-marker}
    ->see_gate      : (Virtual_Sr.#=see_gate)*(ME.#≠exit_complete)
    ->nothing       : (ME.#=exit_complete)
    ->$             : else;

    end    /* Sr_X */
```

```
proctype Sr()/* Virtual lateral sensor machine: it, senses when a vehicle        Sr()
        reaches the XMS section and the gate markers */
    import Info., XSr, ME
    selvar #: (%,see_gate, gate marker, XMS_section_reached)
    stvar $: (nothing, wait_exit_initiate, wait_gate, gate,
          wait-completion)
    selvar gates-seen: (0..4)
    asgn gates-seen:= seen

    stvar seen:          (0..4)
    asgn seen-> (seen + 1) ? ($=wait_gate)*(#=see_gate)|
              0        ? ($=nothing)                  |
              seen
    cyset {nothing@}, {wait_gate@}
    recur wait-completion ->nothing, gate->nothing,
          wait_gate->nothing
    init seen:=0
    init nothing
    trans
    nothing                {%: XMS_section_reached}
    ->wait_exit_initiate   : (#= XMS_section_reached)
    ->$                    : else:

    wait_exit_initiate     {%}
    ->wait_gate            : (ME.#=exit_initiate)
    ->$                    : else;

    wait_gate              {%:see_gate}
    ->nothing              : (ME.#=exit_complete)
    ->gate                 : (# = see_gate)*(ME.#=/exit_complete)
    ->$                    : else:

    /* it can only see another gate marker, if the last, vehicle
    gate has already seen this one, and if there is another gate */
                           {gate_marker}
    ->wait_gate
                           (XSr[2].#=gate_marker)*
                           (number_gates > gates-seen)*
                           (ME.#=/exit_complete)
    ->wait_completion      : (gates_seen = number_gates)*
                           (ME.#=/exit_complete)
    ->nothing              : (ME.#=exit_complete)
    ->$                    : else;

    wait_completion        {gate-marker}
    ->nothing              : (ME.#=exit_complete)
    ->$                    : else;
    end    /* Sr */
```


Figure 14: Exit Maneuver: Lateral Sensor of the Vehicles

```
proctype Rr(i: integer) /* Range sensor response in the TL */          Rr()
    import i, Info, ME
    selvar sensing:(car_ahead, no-car-ahead, nothing)
    stvar $:(0)
    asgn  sensing:= /* NTM: this will do for now, but it is not correct */
          car-ahead      ?
                (Info.msg_from_X[i]=I_am_in_TL)*
                ((+[k in 0..N]
                (1?(Info.msg_from_X[k]=I_am_in_TL)|0)) > 1) |
          no_car-ahead   ?
                (Info.msg_from_X[i]=I_am_in_TL)*
                /* is the only vehicle in the TL */
                (((+[k in 0..N]
                ( 1 ? Info.msg_from_X[k]=I_am_in_TL| 0)) = 1 ) +
                /* or it exited through gate 1 */
                (Info.msg_to_X[i]=exit_gate_1) +
                /* or it exited through gate 2 but no one exited
                   through gate 1 */
                ((Info.msg_to_X[i]=exit_gate_2)*
                ((+[k in 0..N]
                ( 1 ? (Info.msg_from_X[k]=I_am_in_TL)*
                (Info.msg_to_X[k]=exit_gate_1)| 0)) = 0 )) +
                /* or it exited through gate 3 but no one exited
                   through gate 1 or 2 */
                ((Info.msg_to_X[i]=exit_gate_3)*
                ((+[k in O..N]
                ( 1 ? (Info.msg_from_X[k]=I_am_in_TL)*
                (Info.msg_to_X[k]=exit_gate_1)| 0)) = 0 )*
                ((+[k in 0..N]
                ( 1 ? (Info.msg_from_X[k]=I_am_in_TL)*
                (Info.msg_to_X[k]=exit_gate_2)| 0)) = 0 )) +
                ((Info.msg_to_X[i]=exit_gate_4)*
                ((+[k in 0..N]
                ( 1 ? (Info.msg_from_X[k]=I_am_in_TL)*
                (Info.msg_to_X[k]=exit_gate_1)| 0)) = 0 )*
                ((+[k in 0..N]
                ( 1 ? (Info.msg_from_X[k]=I_am_in_TL)*
                (Info.msg_to_X[k]=exit_gate_2)| 0)) = 0 )*
                ((+[k in O..N]
                ( 1 ? (Info.msg_from_X[k]=I_am_in_TL)*
                (Info.msg_to_X[k]=exit_gate_3)| 0)) = 0 )) )|
          nothing

    init 0
    trans
    0
    ->$           : else;
    end    /* Rr */
```

```
proctype Vehicle(i: integer)                                    Vehicle()
    import i, Info, X, XVr,XRr,XSr, Virtual-Sr, ME

    selvar #: (%, idle, wait assignment, wait_GM, ok-to-change, close-up,
          align-with-b-&us-l, align_with_b_plus_1, exit-complete)

    stvar $: (idle, wait-assignment, wait-gate-marker, in-AL,
          monitor-followers-in-TL, monitor_followers_in_AL,
          transfer-leadership, take-over, close_up_in_AL,
          return-lead, authorize-CL, abort,
          wait-reassignment, in_TL,leader_TL,follower_TL,
          wait-completion, command-close-up, update_fail,
          closeup_in_TL, align-with b plus-l, align_with_b_minus_1,
          exit_complete)

    /* A platoon has always a leader (vehicle X[0]) and two followers
       X[1], X[2], which may request exit. As the exit maneuverproceeds,
       a follower in AL may become a leader in AL, and the leader in AL
       may become a follower in TL.
       By induction we can show that whatever is true for this platoon is
       true for a platoon of an arbitrary number of X-vehicles.
    */

    selvar next to exit          :(-1, 0, 1, 2)
    asgn   next-to-exit:=         Info.next_to_exit

    selvar msg_from_X            :(I_am_in_TL, abort,
                                  nothing,
                                  wait-assignment,    wail-reassignment,
                                  I-am-next)
    asgn   msg_from_X:=           from_X

    stvar from_x                 :(I_am_in_TL, abort,
                                 nothing,   wait-assignment,
                                 wait-reassignment, I_am_next)
    asgn from_x ->
        I_am_in_TL               ? ($=in_TL)
        abort                    ? ($=abort)
        nothing                  ? ($=idle)
        wait-assignment          ? ($=wait_assignment)
        wait-reassignment        ? ($=wait_reassignment)
        I-am-next                ?
                        ((from_X = wait_assignment)+
                        (from_X = wait-reassignment)) *
                        (((Virtual_Sr.gates_seen=1)*
                            (Info.msg_to_X[i]=exit_gate_1)) +
                        ((Virtual Sr.gates seen=2)*
                            (Info.msg_to_X[i]=exit_gate_2)) +
                        ((Virtual_Sr.gates_seen=3)*
                            (Info.msg_to_X[i]=exit_gate_3)) +
                        ((Virtual Sr.gates seen=4)*
                            (Info.msg_to_X[i]=exit_gate_4)))|
        from_X
```

```
stvar  to_C[cars]     :(close_up_in_AL,close_up_in_TL, takeover,
                       take-back-leadership,    nothing)
asgn   [k in 0..N]{to_C[k]->
                   /* The leader in AL requests vehicles behind
                     the exiting one in AL to close up */
                   close_up in AL ?
                        ($=monitor_followers_in_AL)*
                        (k=/i)*(k>0)*
                        (Info.msg_from_X[k-1]=I_am_in_TL)    |
                   /* The leader in TL requests exiling vehicles
                     to close up */
                   close-up in TL ?
                        ($=close_up_in_TL)*(k=next to exit)*
                        (Info.msg_from_X[k]=I_am_in_TL) |
                   /* For N > 2, the condition for take over
                     needs to search k>=i+1 until find one k
                     which can take over leadership */
                   take-over      ?
                        ($=transfer_leadership)*(k=i+1)*
                        (Info.msg_from_X[k]=/I_am_in_TL)
                   take-back-leadership  ?
                        ($=return_lead)*
                        (k=Info.leader_in_AL)              |
                   nothing ? ($=idle)                      |
                   to_C[i] }

selvar msg_to_C[cars]:(close_up_in_AL,close_up_in_TL, take-over,
                       take-back-leadership,    nothing)
asgn   [k in 0..N]{msg_to_C[k]:=to_C[k]}

stvar  from_C         :(closed_up_in_AL,closed_up_in_TL,  took-over,  nothing)
asgn   from-C->        closed-up in AL ?
                   ($=close_up_in_AL)*(XVr[i].#=gap_closed)|
                   closed-up in TL ?
                   ($=close_up_in_TL)*(XVr[i].#=gap_closed)|
                   took_over    ?
                   ($=take_over)                           |
                   nothing?
                   ($=idle)                                |
                   from_C

selvar msg_from_C     :(closed_up_in_AL, closed_up_in_TL,
                       took-over,  nothing)
asgn   msg_from_C:=    from_C

init to_C[0]:=nothing, to_C[1]:=nothing, to_C[2]:=nothing,
    from_C:=nothing, from_X:=nothing
```

```
init idle
trans

idle                          {idle]
->wait_assignment                 : (ME.#=exit_initiate)
->$                           : else;

wait_assignment                   {wait_assignment}
->wait_gate_marker                  : (i≠Info.leader in AL)*
                              (Info.msg_to_X[i]≠nothing)*
                              (Info.msg_to_X[i]≠no_exit_gate)
->in_AL                           : (i≠Info.leader_in _AL)*
                              ((Info.msg_to_X[i]=no_exit_gate) +
                              (Info.exit_DB[i]=no_exit)) *
                              /* assignment completed */
                              ((+[k in 0..N]
                              (1?Info.msg_to_X[k]≠nothing| 0))
                              =cars)
->monitor_followers_in_AL         : (i=Info.leader_in_AL)
->$                           : else;

wait-gate-marker                  {wait_GM}
->authorize_CL                    : (XSr[i].#=gate_marker)*
                              (Info.msg_from_X[i]=I_am_next)
->$                           : else;

authorize_CL                      {ok-to-change)
->in_TL                         : XVr[i].#=CL_complete
->abort                         : XVr[i].#=CL_abort
->$                           : else;

abort                         {%}
->wait_reassignment                 : number_gates > Info.exiting_cars
->monitor_followers_in _AL          : (number_gates <_ Info.exiting_cars)*
                              (i=Info.leader_in_AL)
->in_AL                           : (number_gates <_ Info.exiting_cars)*
                              (i≠Info.leader_in_AL);

wait_ reassignment                {%}
->wait_gate_marker                  : (i≠Info.leader in AL)*
                              (Info.msg_to_X[i]≠nothing)*
                              (Info.msg_to_X[i]≠no_exit_gate)
->in_AL                           : (i≠Info.leader_in_AL)*
                              (Info.msg_to_X[i]=no_exit_gate)
->monitor_followers_in_AL         : (i=Info.leader_in_AL)*
                              (Info.msg_to_X[i]≠nothing)*
                              (Info.msg_to_X[i]≠no_exit_gate)
->$                           : else;

in AL                             {%}
```

```
in_AL                          {%}
->close_up_in_AL                 : (Info.msg_to_C[i]=close_up_in_AL)*
                                (from_C≠closed_up_in_AL)
->take_over                      : (Info.msg_to_C[i]=take_over)*
                                (from_C≠took_over)
->exit_complete                  : Info.waiting_exit=0
->$                             : else;

close_up_in_AL                   {close_up}
->in_AL                          : (XVr[i].#=gap_closed)*
                                (Info.msg_to_X[i]=no_exit_gate)
->wait_gate_marker               : (XVr[i].#=gap_closed)*
                                (Info.msg_to_X[i]≠no_exit_gate)*
                                (Info.msg_to_X[i]≠nothing)
->$                             : else;

take_over                        {%}
->monitor_followers_in_AL        :((i=1)*(Info.msg_from_X[0]=I_am_in_TL)) +
                                ((i=2)*(Info.msg_from_X[1]=I_am_in_TL))
->return_lead                    :((i=1)*(Info.msg_from_X[0]=abort)) +
                                ((i=2)*(Info.msg_from_X[1]=abort))
->$                             : else;

return-lead                      {%}
->in_AL                          : (i≠Info.leader_in_AL)*
                                (Info.msg_to_X[i]=no_exit_gate)
->wait_assignment                : (i≠Info.leader_in_AL)*
                                (Info.msg_to_X[i]≠no_exit_gate)*
                                (Info.msg_to_X[i]≠nothing)
->$                             : else;

monitorfollowers_in_AL           {%}
->transfer_leadership            : (Info.msg_from_X[i]=I_am_next)
                                /* next to exit is actually last
                                   to exit at this point, since
                                   there was no time to update it
                                   yet. Saving on variables ... */
->command_close_up               : (Info.msg_from_X[next_to_exit]=
                                 I-am_in_TL)*
                                (true ?
                                ((next-to-exit < cars-1)*
                                (Info.msg_from_C[next_to_exit+1]≠
                                 closed_up_in_AL)) | false)
->exit_complete                  : (Info. waiting_exit = 0)
->$                             : else:
```

```
update-fail                    {%}
->wait_reassignment            : number_gates > N
->monitor_followers in AL      : else;


/* This commands the followers in AL to close up gap */

command-close-up               {%}
->monitor_followers_in_AL      : (Info.msg_from_C[next_to_exit+1]=
                                  closed_up_in_AL)
->$                            : else;

transfer_leadership            {wait_GM}
->authorize_CL                 : (XSr[i].# = gatcmarker)
->$                            : else;

in_TL                          {%}
->leader_TL                    : (XRr[i].sensing= no-car-ahead)
->follower_TL                  : (XRr[i].sensing= car-ahead)
->$                            : else:

/* If it is a follower in TL it orders the regulation layer to
   close-up with the vehicle in front. */
follower_TL                    {close_up}
->wait_completion              : (XVr[i].#=gap_closed)
->$                            : else;

wait_completion                {%}
->exit_complete                : (Info.waiting_exit=0)
->$                            : else;

leader_TL                      {%}
->&t-complete                  : (Info.waiting_exit = 0)
  /* align with the vehicle in front if (i=0) and (i=2) is the next
     exiting vehicle */
->align_with_b_minus_1         : (i=0)*(Info.msg_from_X[2]=I_am_next)
  /* If (i=2) and either (i=0) or (i=1) are next to exit, then
     the next one to exit is a vehicle which was in front of this
     one in AL. Align with the vehicle behind the next one exiting */
->align_with_b_plus_1          : (i=2)*
                                 ((Info.msg_from_X[0]=I_am_next)+
                                  (Info.msg_from_X[1]=I_am_next))
  /* With only two followers we cannot model:
     1. cases which requires deceleration because the next one to
        exit is further than 2 vehicles behind it
     2. cases which require alignment with b-2 because the next one
        to exit is a vehicle which was behind this one in AL */
->monitor_followers_in_TL       : else;
```

```
align_with_b_minus_1          (align-with-b-minus-l)
->monitor_followers in T L    : (XVr[i].#=aligned_with_b_minus_1)
->$                           : else;

align_with_b_plus_1           (align-withb-plus-l)
->monitor_followers in TL     : (XVr[i].#=aligned_with_b_plus_1)
->$                           : else;

exit-complete                 {exit_complete}
->idle                        : (ME.#=exit_complete)
->$                           : else;

monitor-followers _in _TL     {%1
/* Only the following configurations are possible:
  (2 1 0),(0 2 1),(0 1 2),(1 0),(0 1),(2 0),(0 2),(2 1),
  (1 2),(0),(1),(2). Each triple (x y z) indicates the car number
  exiting in the order: first x, then y, then z. */
->align_with_b_plus_1         : ((i=1)*(Info.msg_from_X[2]=abort)*
                                (Info.msg_from_X[0]=I_am_next))+
                              ((i=2)*(Info.msg_from_X[1]=abort)*
                                (Info.msg_from_X[0]=I_am_next))
/* all other combinations of aborted maneuversand assignment leads to
  no action by the leader in TL */

/* Request a close-up to the new vehicle in TL if this vehicle
  continues to be the leader in TL */
->close_up_in_TL              : (next_to_exit=2)*
                              (Info.msg_from_X[next_to_exit]=I_am_in_TL)*
                              (Info.msg_from_C[next_to_exit]=/
                               closed-up-in-TL)
     /* checks if it became a follower */
->wait_completion-  -         : (XRr[i].sensing=car_ahead)
->exit_complete               : (Info.waiting_exit = 0)
->$                           : else;

close-up_in_TL                {%}
->monitor_followers_in_TL     : (Info.msg_from_C[next_to_exit]=
                              closed-up-in-TL)
->$                           : clsc;


end /*_Vehicle */
```

```
proctype Leader_info()                                    Leader_info()
    import X, XSr, XRr, Virtual_Sr, ME
    stvar $: (idle, exiting, exit_complete)

    /* The following information is maintained by the leader in the
      AL and communicated to the leader in the TL. Since leadership
      on both lanes may dynamically change, but a proctype cannot be
      dynamically assigned to a process, we model this info as part of
      a proctype which is seen by all vehicles, but only used by the
      current leaders of AL and TL */

    /* Vehicles which entered the automated highway were assigned an
      exit number by the Network Layer. As a vehicle pass a Link Layer
      section, it asks the Link Layer:
      " What is the XMS number of this section my exit?". Obtaining
      the response from thq Link Layer, they compare it with their
      assigned exit number. If it matches, then the vehicle informs its
      platoon leader that it requires exit in the next XMS.
      We model this process by a vehicle data base which is
      non-deterministically set by the Link Layer, and which is seen by
      the platoon leader and the vehicles themselves. */

    stvar exit_info[cars]: (yes-exit, no-exit, nothing)
    selvar exit_DB[cars]: (ycsexit, no-exit, nothing)
    asgn [i in 0..N]{exit_info[i]->
        yes_exit ?    (i=0)*(arbiter=1)*(exit_info[i]=nothing) +
                      (i=1)*(arbiter=2)*(exit_info[i]=nothing) +
                      (i=2)*(arbiter=1)*(exit_info[i]=nothing) |
        no-exit ?     (i=0)*(arbiter=2)*(exit_info[i]=nothing) +
                      (i=1)*(arbiter=1)*(exit_info[i]=nothing) +
                      (i=2)*(arbiter=2)*(exit_info[i]=nothing) |
        exit_info[i] }
    asgn  [i in 0..N]{exit_DB[i]:=exit_info[i]}

    selvar waiting-exit: (0..1)
    asgn  waiting-exit:=
        0 ? (+[i in O..N] (1 ?
                      (exit_DB[i]=no_exit) +
                      (((msg_from_X[i]=wait_assignment)+
                        (msg_from_X[i]=wait_reassignment)+
                        (msg_from_X[i]=abort))*
                        (to_X[i]=no_exit_gate)) +
                      ((assign_DB[i]=exit_assigned) *
                      (msg_from_X[i]=I_am in TL) *
                      (true ? ((i < cars-l) *
                      (msg_from_C[i+1]=closed_up_in_AL)) +
                      (i >= cars-l) | false)) |0))
            = cars |
        1
```

```
selvar exiting-cars: (0..3)
asgn   exiting-cars:=
           0 ? (+[i in 0..N](1 ? exit_DB[i]=yes_exit|0))= 0 |
           1 ? (+[i in 0..N](1 ? exit_DB[i]=yes_exit|0))= 1 |
           2 ? (+[i in 0..N](1 ? exit_DB[i]=yes_exit|0))= 2 |
           3
/* When a gate marker is seen, we know which one is the next
   one to exit */
selvar next_to_exit :(0, 1, 2)
asgn   next-to-exit:=  next-exiting

stvar next-exiting :(0, 1, 2)
asgn   next-exiting-> 0     ? (Virtual_Sr.#=gate_marker)*
                        (msg_from_X[0]=I_am_ next)  |
                  1     ? (Virtual_Sr.#=gate_marker)*
                        (msg_from_X[1]=I_am_next)   |
                  2     ? (Virtual_Sr.#=gate_marker)*
                        (msg_from_X[2]=I_am_next)   |
                  next-exiting

/* The messages from all vehicles is recorded here */

selvar msg_from_X[cars]:(I_am_in_TL, abort,
                         nothing, wait_assignment,
                         wait-reassignment, I_am_next)
asgn   [i in 0..N]{msg_from_X[i]:=X[i].msg_from_X}

/* The leader also maintains a data base about the gate assignment
   to himself and to the followers. */

selvar assign-DB[cars]: (exit-assigned, exit_not_assigned)
asgn   [i in 0..N] {assign-DB[i]:=
       exit-assigned ?
       (to_X[i]=exit_gate_1) + (to_X[i]=exit_gate_2) +
       (to_X[i]=exit_gate_3) + (to_X[i]=exit_gate_4) |
       exit-not-assigned }
```

```
/* This keeps track of the vehicle assigned to which gate*/

selvar gate_DB[5]:(-1, 0, 1, 2)
asgn [i in 0..4]
      { gate_DB[i]:= 0 ?
      ((to_X[0]=exit_gate_1)*(i=1) +
       (to_X[0]=exit_gate_2)*(i=2) +
       (to_X[0]=exit_gate_3)*(i=3) +
       (to_X[0]=exit_gate_4)*(i=4))  |
                  1 ?
      ((to_X[1]=exit_gate_1)*(i=1) +
       (to_X[1]=exit_gate_2)*(i=2) +
       (to_X[1]=exit_gate_3)*(i=3) +
       (to_X[1]=exit_gate_4)*(i=4))  |
                  2 ?
      ((to_X[2]=exit_gate 1)*(i=1)  +
       (to_X[2]=exit_gate 2)*(i=2)  +
       (to_X[2]=exit_gate_3)*(i=3) +
       (to_X[2]=exit_gate_4)*(i=4))  |
                  -1 }
```

```
/* the gate assignment is informed to vehicles via the array msg to X,
   and remembered by using the statc variable to X. */

selvar msg_to_X[cars]: (exit-gate-l, cxit_gate_2,exit_gate_3,
                        exit-gate-4, no_exit_gatc, nothing)
asgn   [i in 0..N]{msg_to_X[i]:= to_X[i]}
stvar  to_X[cars]:    (exit-gate-l, cxit_gate2. exit_gate-3,
                       exit-gate-4, no_exit_gate,nothing)
asgn   to_X[0]->
       exit_gate-l     ? (exit_DB[0]=yes_exit)*
                       (msg_from_X[0]=wait_assignment)      |
       exit_gate_2     ? (msg_from_X[0]=wait_reassignment)*
                       (gate_DB[2]=-1)*(number_gates >_ 2 ) |
       exit-gate-3     ? (msg_from_X[0]=wait_reassignment)*
                       (gate_DB[3]=-1)*(number_gates>_ 3)   |
       exit_gate_4     ? (msg_from_X[0]=wait_reassignment)*
                       (gate_DB[4]=-1)*(number_gates = 4  ) |
       no-exit-gate ? (exit_DB[0]=no_exit) +
                       ((msg_from_X[0]=abort)*
                       (number_gates <_ exiting_cars))      |
       nothing         ? ($=idle) + ((msg_from_X[0]=abort)*
                       (number_gates>exiting_cars))         |
       to_X[0]

asgn to_X[1]->
       exit-gate-l ? ((exit_DB[1]=yes_exit)*(exit_DB[0]=no_exit)*
                     (msg_from_X[1]=wait_assignment)) +
                     ((msg_from_X[1]=wait_reassignment)*
                     (gate-DB[l]=-1))                       |
       exit_gate_2 ? (number_gates >_ 2)*
                     (((exit_DB[1]=yes_exit)*(exit_DB[0]=yes_exit)*
                      (msg_from_X[1]=wait_assignment)) +
                      ((msg_from_X[1]=wait_reassignment)*
                      (gate_DB[2]=-1)))                     |
       exit_gatc_3 ? (number-gates >_ 3)*
                     (msg_from_X[1]=wait_reassignment)*
                     (gatc_DB[3]=-1)                        |
       exit_gate_4 ? (number_gates = 4)*
                     (msg_from_X[1]=wait_reassignment)*
                     (gatc_DB[4]=-1)                        |
       no_exit_gate ? ((number-gates < 2)*
                     (exit_DB[I]=yes_exit)*
                     (exit_DB[0]=yes_exit)*
                     (msg_from_X[1]=wait_assignment)) +
                     ((msg_from_X[1]=wait_reassignment)*
                     ((+[i in 1..4](1 ? gate-DB[i]j-1 | 0))
                     = number-gates)) +
                     ((msg_from_X[1]=abort)*
                     (number_gates<_ exiting-cars)) +
                     (exit_DB[1]=no_exit)                   |
       nothing ?       ($=idle) + ((msg_from_X[1]=abort)*
                     (number_gatcs> exiting-cars))          |
       to_X[1]
```

```
/* the gate assignment is informed to vehicles via the array msg to X,
   and remembered by using the state variable to_X. */

selvar msg_to_X[cars]:(exit_gate_1, exit-gate-2, exit_gate_3,
                exit_gate_4, no-exit-gate, nothing)
asgn    [i in 0..N]{msg_to_X[i]:= to_X[i] }
stvar   to_X[cars]:    (exit-gate-l, exit_gale-2, exit gate-3,
                exit_gate_4,no_exit_gate, nothing)
asgn    to_X[0]->
      exit-gate-l     ? (exit_DB[0]=yes_exit)*
                      (msg_from_X[0]=wait_assignment)       |
      exit-gate-2     ? (msg_from_X[0]=wait_reassignment)*
                      (gate_DB[2]=-1)*(number_gates >_ 2 ) |
      exit-gate-3     ? (msg_from_X[0]=wait_reassignment)*
                      (gate_DB[3]=-1)*(number_gates>_ 3)    |
      exit_gate_4     ? (msg_from_X[0]=wait_reassignment)*
                      (gate_DB[4]=-1)*(number_gates = 4 ) |
      no-exit-gate  ? (exit_DB[0]=no_exit) +
                      ((msg_from_X[0]=abort)*
                      (number_gates (exiting_cars))         |
      nothing         ? (&idle) + ((msg_from_X[0]=abort)*
                      (number_gates>exiting_cars))          |
      to_X[0]

asgn to-X[l]->
      exit-gate-1  ? ((exit_DB[1]=yes_exit)*(exit_DB[0]=no_exit)*
                     (msg_from_X[1]=wait_assignment)) +
                     ((msg_from_X[1]=wait_reassignment)*
                     (gate-DB[l]=-1))                       |
      exit_gate_2 ? (number_gates >_ 2)*
                     (((exit_DB[1]=yes_exit)*(exit_DB[0]=yes_exit)*
                      (msg_from_X[1]=wait_assignment)) +
                      ((msg_from_X[1]=wait_reassignment)*
                      (gate_DB[2]=-1)))                     |
      exit-gate-3 ? (number-gates >_ 3)*
                     (msg_from_X[1]=wait_reassignment)*
                     (gate_DB[3]=-1)                        |
      exit_gate_4  ? (number_gates = 4)*
                     (msg_from_X[1]=wait_reassignment)*
                     (gate_DB[4]=-1)                        |
      no_exit_gate ? ((number_gates < 2)*
                     (exit_DB[1]=yes_exit)*
                     (exit-DB[O]=yes-exit)*
                     (msg_from_X[1]=wait_assignment)) +
                     ((msg_from_X[1]=wait_reassignment)*
                     ((+[i in 1..4](1 ? gate-DB[i]$-1 | 0))
                     = number-gates)) +
                     ((msg_from_X[1]=abort)*
                     (number_gates <_ exiting-cars)) +
                     (exit_DB[1]=no_exit)                   |
      nothing ?       ($=idle) + ((msg_from_X[1]=abort)*
                     (number_gates > exiting-cars))         |
      to_X[1]
```

```
asgn   to_X[2]->
       exit-gate-1 ? ( (exit_DB[2]=yes exit)*(exit_DB[1]=no_exit)*
                     (exit_DB[0]=no_exit)*
                     (msg_from_X[2]=wait assignment) )      |
       exit_gate_2 ? ((exit_DB[2]=yes_exit)*
                     ((exit_DB[1]=yes_exit)+(exit_DB[0]=yes_exit))*
                     ((exit_DB[1]=no_exit)+(exit_DB[0]=no_exit))*
                     (number_gates >_ 2)*
                     (msg_from_X[2]=wait_assignment)) +
                     ((msg_from_X[2]=wait_reassignment)*
                     (gate_DB[2]=-1)*
                     (number_gates > 2))                    |
       exit gate 3 ? ((exit DB[2]=yes_exit)*
                     (exit_DB[1]=yes_exit)*(exit_DB[0]=yes_exit)*
                     (number-gates >_ 3)*
                     (msg_from_X[2]=wait_assignmcnl)) +
                     ((msg_from_X[2]=wait_reassignment)*
                     (gate_DB[3]=-1)*
                     (number_gates>_ 3))                    |
       exit_gate_4 ? ((msg_from_X[2]=wait_reassignment)*
                     (gate_DB[4]=-1)*
                     (number-gates = 4))                    |
       no-exit-gate ? ((exit_DB[2]=yes_exit)*
                     (exit_DB[1]=yes_exit)*(exit_DB[0]=yes_exit)*
                     (number-gates < 3)*
                     (msg_from_X[2]=wait_assignmcnt)) +
                     ((msg_from_X[2]=wait_reassignment)*
                     ((+[i in 1..4](1 ? gate-DB[i]j-1 | 0))
                     = number-gates)) +
                     ((msg_from_X[2]=abort)*
                     (number-gales <_ exiting_cars)) +
                     (exit_DB[2]=no_exit)                   |
       nothing ?       ($=idle) + ((msg_from_X[2]=abort)*
                     (number_gates > exiting_cars))         |
       to_X[2]

selvar msg_from_C[cars]:(closed_up_in_AL, closed_up_in_TL,
                         took-over,  nothing)
asgn   [i in 0..N]{msg_from_C[i]:=X[i].msg_from_C}

selvar msg_to_C[cars]:(close_up_in_AL, close-up-in-TL, takeover, nothing)
asgn   [i in 0..N] {msg to C[i]:=
           close-up_in _AL ?
                 X[leader_in_AL].msg_to_C[i]=close_up_in_AL |
           close-up_ in _TL ?
                 X[leader_ i n _TL].msg to C[i]=close up in TL |
           take _  over   ?
                 X[le ader_in_AL].msg_to_C[i]=take_over     |
                 nothing )
```

```
selvar leader in_AL:(0,1,2)
asgn leader_in_AL:= l_AL
stvar l_AL          :(0,1,2)
asgn  l_AL->          0 ? (msg_from_X[0]=/I_am_in_TL)
                    1 ? (msg_from_X[0]=I_am_in_TL)
                    2 ? (msg_from_X[0]=I_am_in_TL)*
                       (msg_from_X[1]=I_am_in_TL)
                    l_AL

selvar leader in_TL :(0,1,2)
asgn leader_in_TL:=l_TL
stvar  l_TL         :(0,1,2)
asgn  l_TL->          0 ? XRr[0].sensing=no_car_ahead
                    1 ? XRr[1].sensing=no_car_ahead
                    2 ? XRr[2].sensing=no_car_ahead
                    l_TL

init to_X[0]:=nothing, to_X[1]:=nothing, to_X[2]:=nothing,
    l_AL:=0,l_TL:=0,
    exit_info[0]:=nothing, exit_info[1]:=nothing,
    exit_info[2]:=nothing, next_exiting:=0
recur exit_complete->idle
init idle
trans

idle
->exiting      : ME.#=exit_initiate
->$            : else;

exiting
->exit_complete : ME.#=exit_complete
->$            : else;

exit-complete
->idle         : true;

end /*_Leader_info */
```

```
proctype Monitors_Exit()                                    Monitors Exit()
    /* monitors the start and completion of the exit maneuver. It is a
       function of the leader in AL. If alter all vehicles
       exit, no leader in AL remains the function is switched
       to the leader in the TL. */
    import X, Virtual Sr
    selvar #:(%, e&initiate, exit_complete)
    stvar $:(idle,monitoring, exit-complete)
    recur exit-complete->idle
    init idle
    trans

    idle                {%}
    ->monitoring          : (Virtual-Sr.# =XMS_section_reached)
    ->$                  : else;

    monitoring            {exit-initiate)
    ->exit_complete        : (+[i in 0..N](1? X[i].#=exit_complete| 0)) =N+1
    ->$                  : else;

    exit-complete         {exitcomplcte}
    ->idlc               : true;

    end    /* Monitors Exit */
```

```
    /* This is used to order the sensing off the gates by the
       vehicles, since it is not possible to dynamically change
       the association of sensors to vehicles (what would be
       necessary if a follower becomes a leader in AL */

proc Virtual-Sr: Sr()                                          Virtual_Sr

    /* Vehicles lateral sensor machine*/

proc XSr[i<cars]: Sr_X (i)                                     XSr

    /* removes from the behavior of the lateral sensor machines
       the possibility of forever sensing "vehicle too close" in
       the transition lane */

procSr_finally[i<cars]: FINALLY((XVr[i].monitor_TL=vehicle_too_close)*   Sr finally
                        (X[i].#=ok_to change))

    /* vehicles velocity response */

proc XVr[i<cars]: Vr(i, X[i])                                  XVr

    /* range sensors      */

proc XRr[i<cars]: Rr(i)                                        XRr

    /* The leaders' information data bases*/

proc Info: Leader_info()                                       Info

    /* vehicles' protocol machine */

proc X[i<cars]: Vehicle (i)                                    X

    /* monitor exit */

proc ME: Monitors_Exit()                                       ME
```

```
/*    Task A: at most N vehicles take an exit with M=N gates).*/

monitor Task-A: STOP(                                          Task_A
          (+[i in 0..N](1 ? Info.msg_from_X[i]=I_am_in_TL| 0)) >
          number_gates )

/*    Task B: Non-requesting vehicles do not exit.*/

monitor Task-B: STOP( (+[i in 0..N](1 ? (Info.exit_DB[i]=no_exit)*      Task_B
                      (Info.msg_from_X[i]=I_am_in_TL)| 0))
                > 0 ) /*    Task C: Inter-Platoon spacing is closed-up after a vehicle
exits. */

/*    Task D: Any vehicle which requests exit, will eventually exit */

#if   Leader monitor Task-DL                                    Task-DL
      import A
      stvar$: (idle, request)
      cyset {idle}, {idle, request}
      init idle

      trans
      idle
      ->request      : (A.Leader_exit_request = yes)
      ->$            : else;

      request
      ->idle         : (A.$ = Acxitcompletc)
      ->$            : else;

end /* task_DL */ #endif
```

```
#if   F1 monitor Task_DF1                                   Task_DF1
      import X, A
      stvar $: (idle, request)
      cyset {idle), {idle, request)
      init idle

      trans
      idle
      ->request      : (X[0].# = exit-request)
      ->$            : else:

      request
      ->idle         : (X[0].$ = exit-complete)*
                       (A.$ = exit-complete)
      ->$            : else;

end /* task DF1 */ #endif
```

```
#if   F2 monitor Task_DF2                                   Task-DF2
      import X, A
      stvar 8: (idle, request)
      cyset {idle], {idle, request)
      init idle

      trans
      idle
      ->request      : (X[1].#= exit-request)
      ->$            : else;

      request
      ->idle         : (X[1].$= exit-complete)*
                       (A.$ = exit-complete)
      ->$            : else;

end /* task DF2 */ #endif
```