UC Berkeley
Research Reports

Title
Design Of Platoon Maneuver Protocols For IVHS

Permalink
https://escholarship.org/uc/item/89c6p0cn

Authors
Hsu, Ann
Eskafi, Farokh
Sachs, Sonia
et al.

Publication Date
1991

eScholarship.org Powered by the California Digital Library
University of California

Program on Advanced Technology for the Highway
INSTITUTE OF TRANSPORTATION STUDIES
UNIVERSITY OF CALIFORNIA AT BERKELEY

The Design of Platoon Maneuver Protocols for IVHS

Ann Hsu
Farokh Eskafi
Sonia Sachs
Pravin Varaiya

PATH Research Report
UCB-ITS-PRR-91-6

This work was performed as part of the Program on Advanced Technology for the Highway (PATH) of the University of California, in cooperation with the State of California, Business and Transportation Agency, Department of Transportation, and the United States Department of Transportation, Federal Highway Administration.

The contents of this report reflect the views of the authors who are responsible for the facts and the accuracy of the data presented herein. The contents do not necessarily reflect the official views or policies of the State of California. This report does not constitute a standard, specification, or regulation.

April 20,199l

ISSN 1055-1425

This paper has been mechanically scanned. Some errors may have been inadvertently introduced.

The Design of Platoon Maneuver Protocols for IVHS

Ann Hsu, Farokh Eskafi Sonia Sachs and Pravin Varaiya
University of California, Berkeley CA 94720

20 April 1991

' Work supported by PATH (Program on Advanced Technology for the Highway), Institute of Transportation Studies, University of California, Berkeley, and &SF Grant ECS-8719779. It is a pleasure to thank Dr. Bob Kurshan for providing COSPAN and explaining its use, Ellen Sentovich for help with COSPAN, Max Holm for computer support, Anuj Puri for comments which improved some parts of the design, and members of the PATH Seminar on AVCS where this design was first presented and discussed.

Contents

1 Introduction 1

2 System Requirements 3

3 Elementary Maneuvers and Path Planning 5

4 Informal Design of Elementary Protocols 8

5 Introduction to COSPAN 16

6 Formal Specification and Verification 18

7 Implications for Regulation Layer 39

8 Conclusions 40

References 42

Appendix 45

List of Figures

1 Control hierarchy 1

2 Definition of platoon 3

3 Lateral sensor range requirements (figure not to scale) 4

4 Inter-vehicle communication link requirements 5

.. The merge maneuver 6

6 The split maneuver 6
7 The change lane maneuver 6
8 A path along highway H 7
9 Flow diagram for merge 9
10 State machines for merge: initiator (top), respondent (bottom) 10
11 Flow diagram for split 11
12 State machines for split: initiator (top), respondent (bottom) 12
13 Flow diagram for change lane 14
14 State machines for change lane: initiator (top), respondent (bottom) 15
15 The COSPAN model 16
16 COSPAN example system 17
17 Monitor for example system 18
18 The four sublayers of the platoon layer 19
19 Merge protocol maneuver processes 21
20 Merge environment processes 22
21 Merge protocol processes 23
22 Merge moni tor 23
23 Split, Protocol maneuver processes 24
24 Split environment processes 25
25 Split protocol processes for 'leader' wishes to split' 25
26 Split protocol processes for ‘follower wishes to split’ 26
27 Split, monitors 27

28 Change lane maneuver processes 28

29 Change lane environment process for vehicle C 28

30 Change lane environment processes for vehicle B_i 28

31 Change lane environment processes for vehicle A 29

32 Cha.ngr lane environment processes for vehicle B 29

33 Change lane protocol process for vehicle A 30

34 Change lane protocol process for vehicle B and B_i 31

35 Change lane protocol process for vehicle C 32

36 Change lane monitor 33

37 Free agent sublayer processes 33

38 Free agent sublaycr monitor 34

39 **SIZE** process for supervisor sublayer environment 35

40 **ALLBUSY** process for supervisor sublayer environment. 35

41 **FREE** process for supervisor sublayer environment 35

42 **SUPR** process for supervisor sublayer 36

43 First monitor for supervisor sublayer 36

44 Second monitor for supervisor sublayer 37

45 Path monitor sublayer processes 38

Abstract

A structured use of control, communication and computing technologies in vehicles and in the highway in the form of an Intelligent Vehicle/Highway System or IVHS can lead to major increases in highway capacity and decreases in travel time without building new roads. Our context is an IVHS system in which traffic on the highway is organized in platoons of up to twenty closely spaced vehicles under automatic control. We consider the design of the controllers for such platoons.

The control tasks are arranged in a three layer hierarchy. At the top or link layer. a centralized controller assigns to each vehicle a path through the highway and the target. size and speed for platoons to reduce congestion. The remaining two layers are distributed among controllers on each vehicle. A vehicle's platoon layer plans its trajectory to conform to its assigned path and to track the target size and speed. The plan consists of a sequence of elementary maneuvers: merge (combines two platoons into one), split (separates one platoon into two), and change lane (enables a single car to change lane). Once the protocol layer determines that a particular maneuver can safely be initiated, it instructs its regulation layer to execute the corresponding pre-computed feedback control law which implements the maneuver.

This paper focuses on the design of the platoon layer. In order to ensure that it is safe to initiate a maneuver, the platoon layer controller enters into a. uegotiation with its neighbors. This negotiation is implemented as a protocol – a, structured sequence of message exchanges. After a protocol terminates successfully, the actions of the vehicles involved become coordinated and the maneuver can be initiated. A protocol is designed in two stages. In the first stage, the protocol is described as an informal state machine, one machine per vehicle. The informal state machine does not distinguish between actions and conditions referring to the vehicle's environment and those referring to the protocol itself. In the second stage this distinction is enforced and the protocol machines are specified in the formal language COSPAN. COSPAN software is then used to show that the protocol indeed works correctly. One can now be reasonably confident in the protocol logic presented here. Much further work remains to be done and some of the more significant problems are outlined.

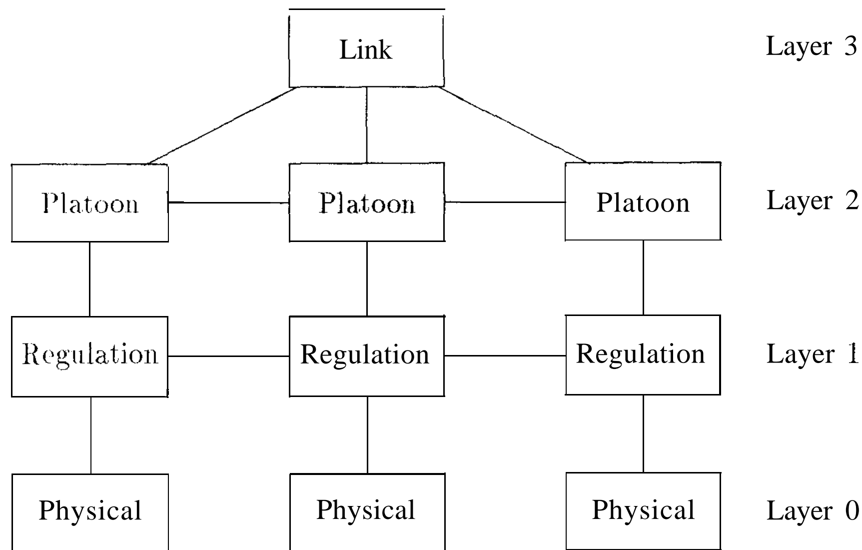


Figure 1: Control hierarchy

1 Introduction

Projections of highway traffic conditions point to the need for an Intelligent Vehicle/Highway System (IVHS) that increases highway capacity and decreases travel time without building new roads. One promising strategy towards achieving this goal is to organize traffic in platoons of closely spaced vehicles. With platoons of 20 vehicles, headways within a platoon of 1m, and headways between platoons of 60m, such an organization can attain a capacity that is four times and travel times that are half the values corresponding to current operating conditions [1]. The design and implementation of the control tasks needed to realize such an IVHS system will require a structured approach that uses control, communication and computing technologies both to maintain within narrow tolerances the position and speed of a vehicle within a platoon and to coordinate platoon maneuvers.

The control tasks are arranged in the three layer hierarchy of Figure 1.[1] There is a single *link* layer for a. long segment of the highway extending several *sections*. Each section may be between 50m and 500m long. The link layer has two functions. It assigns a path to each vehicle entering the highway as explained in §3. And it continuously determines platoon optimal speed (denoted *optspeed*) and optimal size (*optsize*) for each highway section. The values of *optspeed* and *optsize* are selected to maintain smooth traffic flow and to reduce congestion. One approach to computing these values is presented in [1]. The link layer

[1] A more comprehensive discussion of the IVHS control architecture appears in [2].

functions are implemented in a centralized manner: a single computer receives all the information needed to assign vehicle paths and the section target platoon speed and size.

The remaining control tasks are implemented in a distributed manner. There is one *platoon layer,* one regulation *layer* and one *physical* layer per vehicle. Each vehicle's platoon layer plans a sequence of maneuvers and issues corresponding commands to its regulation layer so that the vehicle's trajectory follows closely the path assigned toil, and so that platoon speed and size track *optspeed* and *optsize*. Each vehicle's regulation layer executes the commands issued by its platoon layer by implementing corresponding pre-computed feedback control laws which continuously determine the vehicle's throttle, braking, and steering actions. Finally, the physical layer of a. vehicle is a model of its dynamic behavior against which the feedback control laws arc designed.

This paper presents a design of the platoon layer. It is shown in §3 that the platoon layer's plan can always be constructed as a sequence of three elementary maneuvers called *merge, split* and *change lane*. The merge maneuver joins two platoons into a single platoon; split separates one platoon into two; and a change lane maneuver moves a single vehicle into an adjacent lane. To accomplish these maneuvers safely, the platoon layer controller first coordinates its action with those of its neighbors. This is achieved by exchanging a structured sequence of messages with the platoon layer controllers of neighboring vehicles. This sequence of message exchanges is called a *protocol.* The design of these protocols is carried out in two stages. In the first stage, presented in §4, the merge, split and change lane protocols are described as informal finite state machines. In the second stage, presented in §6, these informal machines are specified in a formal language called COSPAN. COSPAN software is then used to verify that the protocol machines work correctly. Also specified are path planning supervisor machines which invoke the elementary protocols to produce the correct sequence of maneuvers conforming to the assigned path. A very brief introduction to COSPAN is given in §5.

After a vehicle's platoon layer successfully completes an elementary protocol exchange with its neighbors it issues a command to its regulation layer to implement the corresponding pre-computed feedback control law. Although it is not the focus of this paper, the platoon layer design poses some challenging problems for the regulator layer design. These implications are drawn out in $7. Finally, some conclusions reached by this study including suggestions for future work are collected in $8.

The discussion of the platoon and regulation layers presented here presupposes a system of sensors on board vehicles and on the roadside, and a. communication system. These requirements are summarized in §2.

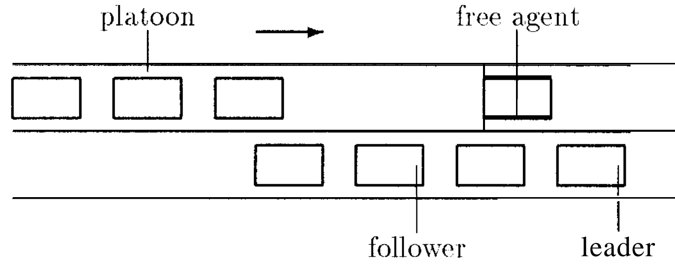


Figure 2: Definition of platoon

2 System Requirements

We introduce certain definitions and background assumptions that will make the discussion more precise. Traffic is organized in ***platoons*** of vehicles. See Figure 2. The size of a platoon is between one and twenty depending on the traffic flow with larger sizes needed to sustain bigger flows [1]. The headway within a platoon is small (about lm); the minimum headway between platoons grows with the platoon size reaching about 60m for platoons of size twenty.[a] The lead vehicle of a. platoon is called its *leader*, the rest are ***followers***. A single vehicle platoon is called a *free agent*. Protocol exchanges are always between leaders (including free agents) of neighboring platoons. If a follower wants to initiate a maneuver it must request its leader to do so. It is also the task of the leader to track *optspeed* and *optsize* announced by the link layer. The follower's task is only to execute a feedback control law which maintains the tight headway with the vehicle in front of it. (This is discussed further in §7.)

Each vehicle's platoon layer maintains its **own** 'state' information:

$$state = (ID\#, hwy\#, ln\#, sect\#, optsize, optspeed, pltn\#, ownsize, pos, busy) \qquad (1)$$

$ID\#$ is the vehicle's ID which is 'hardwired into the controller. $Hwy\#, ln\#, sect\#$ are the highway number, lane number, and section number on which the vehicle is currently traveling; these are either broadcast to the vehicle from the roadside or they are sensed from roadside markers or computed by a navigation system on board the vehicle. *Optsize* and *optspeed* are the targets computed by the link layer and communicated in some way to the vehicle. $Pltn\#$ is the $ID\#$ of the platoon leader. *ownsize* is the size of the platoon, *pos* is the vehicle's position in the platoon (*pos* = 1 is the position of the leader). Lastly, *busy* is a binary flag that is set if the platoon is engaged in a maneuver; it is only used by the leader and its function will become clear when we discuss the protocols. The last four components of the state are updated at the end of each maneuver by communication among vehicles in a platoon. It is assumed that this state is always available to a vehicle's platoon layer controller and updated as needed. The regulation layer controller maintains other state information relating to the vehicle's dynamics such as position, speed and engine rpm.

[a] Our design presupposes a platooning strategy. But it does not depend upon the choice of intra- and inter-platoon headways. Of course, the capacity does depend on these headways.

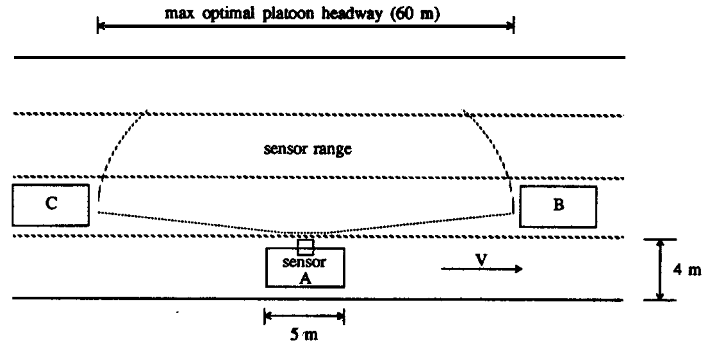


Figure 3: Lateral sensor range requirements (figure not to scale)

In addition to their own states, the platoon and regulator layers need information about their environment chiefly neighboring vehicles and platoons.

Sources of information

A vehicle's controller obtains information about its own state and about its environment from three sources: roadside monitors, sensors mounted on the vehicle, and inter-vehicle communication links. We discuss these briefly in order to indicate in a. little more detail some of the requirement's of an IVHS system and because the formal specification and verification of the protocol design require models of the interfaces between the platoon layer and its environment.

Roadside monitors. They measure traffic conditions based on which the link layer assigns a path to each vehicle and calculates the values of *optsize* and *optspeed*. These are communicated to the vehicles. These monitors would he distributed along the highway.

Sensors. Platoons of size twenty maintain a headway of about 60m. Hence vehicles must be equipped with a longitudinal sensor that measures the distance between itself and the vehicle in front of it up to at least 60m. In order to change lanes the vehicle must be equipped with a. lateral sensor that locates each vehicle within a radius of about 30m as shown in Figure 3. If vehicle A in Figure 3 wishes to change lanes, its sensor should be able to determine that, there is no vehicle within the area marked 'sensor range'. The vehicle's regulation layer needs additional sensors including those that measure its distance and speed relative to the vehicle in front of it within a platoon and its position relative to the center of the lane on which it is traveling."

Inter-vehicle communication. The platoon layer protocols require the capability to exchange messages between one vehicle and other vehicles within the range of its longitudinal

[3]For longitudinal control sensor requirements see [3, 4], for lateral control sensor requirements see [5].

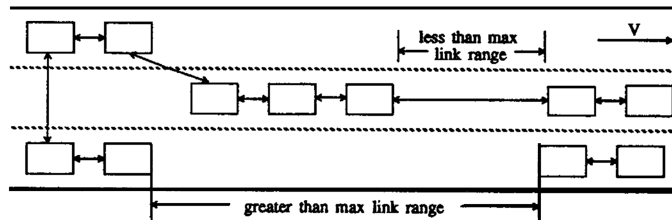


Figure -1: Inter-vehicle communication link requirements

and lateral sensors, i.e. with vehicles within 60m in front of it and within 30m alongside it. In addition, vehicles within the same platoon need to exchange messages among themselves. Thus each vehicle should be capable of setting up communication links with some of its neighbors as illustrated by the double arrows in Figure 4.[4] The traffic on these links consists of messages that convey platoon and regulation layer state information, and messages required by the protocols described below. There are three types of protocol messages: 'broadcast' messages arc sent by a platoon leader to all its followers; 'addressed' messages are exchanged between a platoon leader and a specific follower; and 'direct' messages are exchanged between leaders of neighboring platoons. Of course, depending on the communication technology used, a. message between a pair of vehicles may be forwarded through intermediate vehicles.

The kind of study presented here has an important bearing on the amount of bandwidth that must be available on the various communication links to support the necessary message exchanges. However, we will not address this issue further."

3 Elementary Maneuvers and Path Planning

We describe three elementary maneuvers and then show how these are combined to plan a path conforming to the one assigned by the link layer. The maneuvers are called merge, split, and change lane.

Merge. This maneuver combines two successive platoons in the same lane into a single platoon. See Figure 5. The merge is always initiated by the leader of the rear platoon, vehicle B. If the size of B's platoon, $ownsize(B)$, is smaller than $optsize$, B requests A for permission to merge. If A is not busy, and if

$$ownsize(B) + ownsize(A) \leq optsize$$

this permission is granted. B's platoon layer then requests its regulation layer to accelerate and join A's platoon.

[4]Simple experiments involving radio links is described in [6,7].

[5]An illustrative calculation of the bandwidth needed for longitudinal control is given in [2]; see also [6].

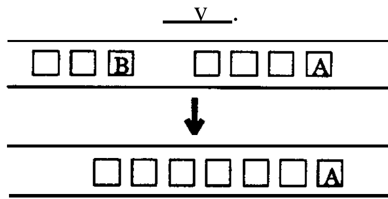


Figure 5: The merge maneuver

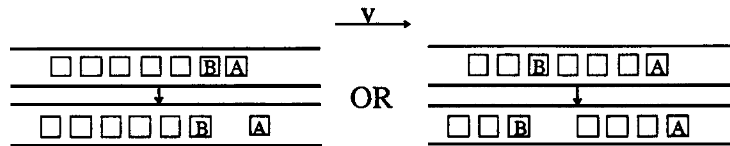


Figure 6: The split maneuver

Split. A split maneuver may be needed because a platoon's size may exceed *optsize*, or because a vehicle in an adjacent lane requests a change lane maneuver (see below), or because a vehicle in a platoon initiates one or two splits in order to become a free agent. As indicated in Figure 6, a split may be initiated by a leader (vehicle A) or by a follower (vehicle B).

Change lane. This maneuver can be initiated only by a free agent, i.e. a single vehicle platoon.[6] See Figure 7. If a vehicle in a multi-vehicle platoon needs to change lanes it must first gain free agent status by executing one split (if it is a leader) or two split maneuvers (if it is a follower).

Suppose the free agent. vehicle A, wishes to move from its current lane 3 to the adjacent lane 2. Before it can do this safely, it must make sure that there is a. vacant space in lane 2, and if there is, it must determine whether any other vehicle (from lane 2 or lane 1) is planning to move into that, space. Thus A must communicate and negotiate with the vehicles within the range of its lateral sensor. There are three mutually exclusive possibilities: the sensor

[6] The restriction to a single vehicle is imposed to simplify the resulting regulation layer feedback control law. However, the additional complexity of multi-vehicle platoon change lane maneuvers may be justified at entrance and exit lanes, especially if this significantly increases capacity.

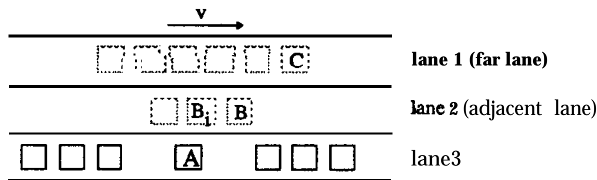


Figure 7: The change lane maneuver

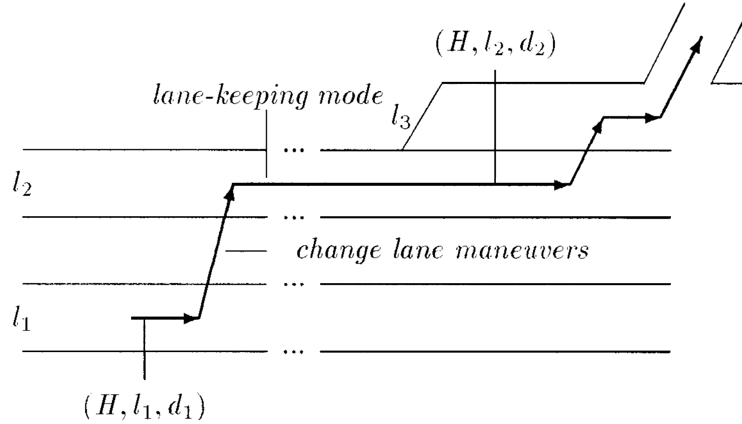


Figure 8: A path along highway H

detects no vehicle in its range (case 0), or the sensor detects a vehicle in lane 2 (case 2), or the sensor detects a vehicle in lane 1 and no vehicle in lane 2 (case 1).

In case 0, A orders its regulation layer to change lane. In case 1, A requests C not to move into lane 2. In case 2, B compares A's speed and position relative to B's platoon and then responds by asking A to decelerate and move into lane 2 behind B's platoon, or B itself decelerates and asks A to move in front of it, or B asks the appropriate follower. B_i, to split and make room for A.

Path planning

We now indicate how a platoon layer controller combines these maneuvers to create a path in conformity to the link layer assignment.

Imagine an automated highway H. Each vehicle continuously senses the section on which it is traveling. A section is a triple (H, l, d) where $hwy\# = H$. $ln\# = l$, and $sect\# = d$. Upon entering the highway, the vehicle declares its destination. In response, the link layer assigns a path (l_2, d_2, l_3). The interpretation is that the vehicle must change to lane l_2 from its current lane l_1; travel along l_2 until section (H, l_2, d_2); and finally change to lane l_3 from which it reaches its exit,. See Figure 8.

Having been assigned its path, the vehicle's platoon layer plans a conforming path as follows:

1. It is initially a free agent in section (H, l_1, d_1). It executes $|l_2 - l_1|$ change lane maneuvers at the end of which it is a free agent in lane l_2.
2. It now enters a lane-keeping mode, staying in lane l_2 until section (H, l_2, d_2). In this

mode it tries to track *optsize* and *optspeed*. To do this, the platoon layer may execute several merge and split maneuvers.

3. Upon reaching (H, l_2, d_2) the platoon layer will execute at most two split maneuvers to become a free agent. It then executes $|l_3 - l_2|$ change lane maneuvers to reach lane l_3.

4 Informal Design of Elementary Protocols

This section presents an informal design for the protocol for each of the three elementary maneuvers. For each maneuver the design involves two steps. in the first step, we express as a flow diagram the coordinated sequence of actions of the platoon and regulation layers of the vehicles engaged in that maneuver. In the flow diagram no attention is paid to the requirement that controllers in different vehicles must be coordinated through explicit message exchanges.

In the second step, this requirement is enforced and the flow diagram is 'distributed' among separate state machines, one for each vehicle. The state machine descriptions are 'informal' since their states and transitions refer to actions and conditions that may depend on the regulation layer, on information from sensors on board the vehicle and on information from roadside monitors and which, therefore, are not part of the protocol machines themselves. In the formal specification, presented in $6, references to these 'environment' actions and conditions are represented as separate state machines.

Merge

The flow diagram that achieves the merge maneuver of Figure 5 is displayed in Figure 9. The sequence of events depicted in the flow diagram requires little comment. A and B refer to the vehicles in Figure 5. B initiates the merge request to which A responds. The condition '*A checks if busy*' refers to the busy flag in A's state (see (I)). That flag is set if and only if A is engaged in a maneuver. Thus if A is busy, it denies B's request represented by the event, 'A sends *nack* to *B*'. After *B* receives permission to merge (*'A sends ack_request_merge to B')*, it orders its regulation layer to accelerate and join *A's* platoon (*'B* accelerates *to merge*'). After *B's* regulation layer task is completed, *B* informs *A* of this (*'B sends confirm_merge to A* '). A then unsets its busy flag, and broadcasts its new updated state to its own followers, and B does the same to its followers.

The paired state machines of Figure 10 achieve the same sequence of events. The machine on top is that of the initiator (B), the other machine is that of the respondent (vehicle *A*). The two machines are coupled together by explicit message exchanges. That is, transitions labeled *'SEND xx'* and *'REC xx'* are supposed to occur simultaneously and synchronize the

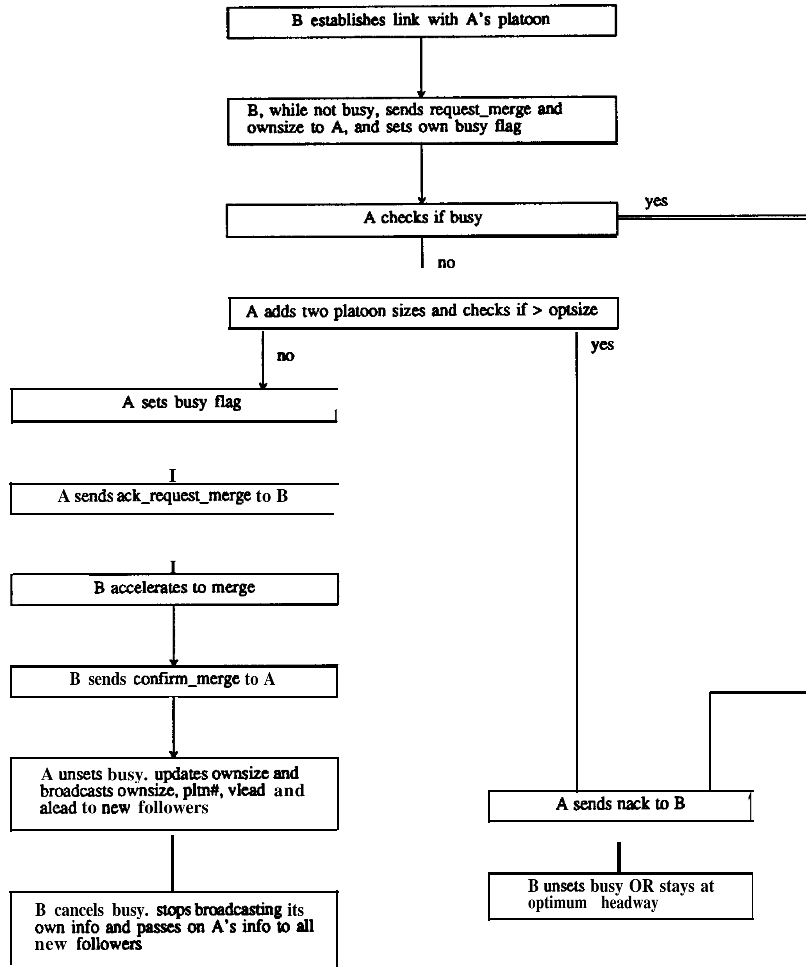
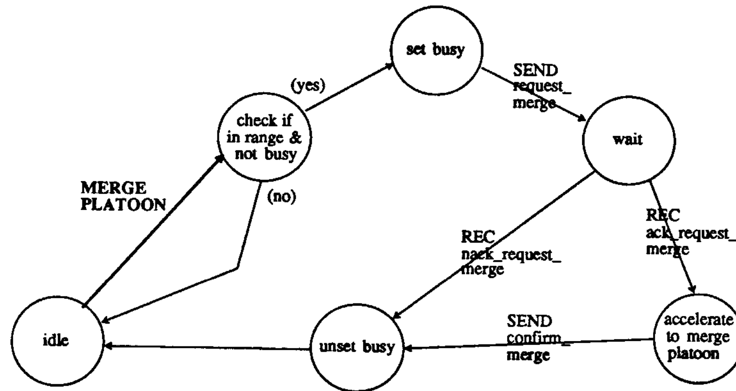


Figure 9: Flow diagram for merge



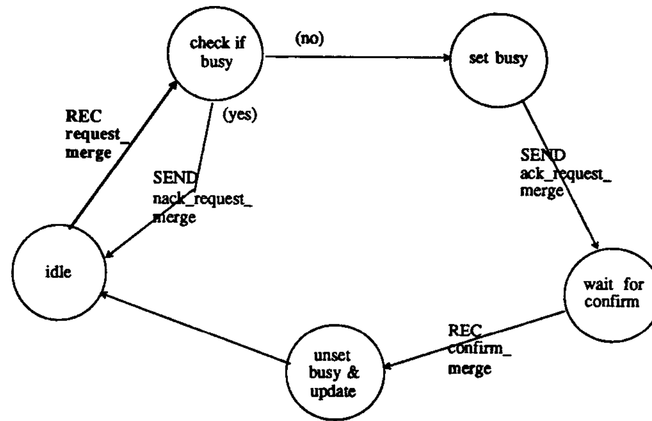


Figure 10: State machines for merge: initiator (top), respondent (bottom)

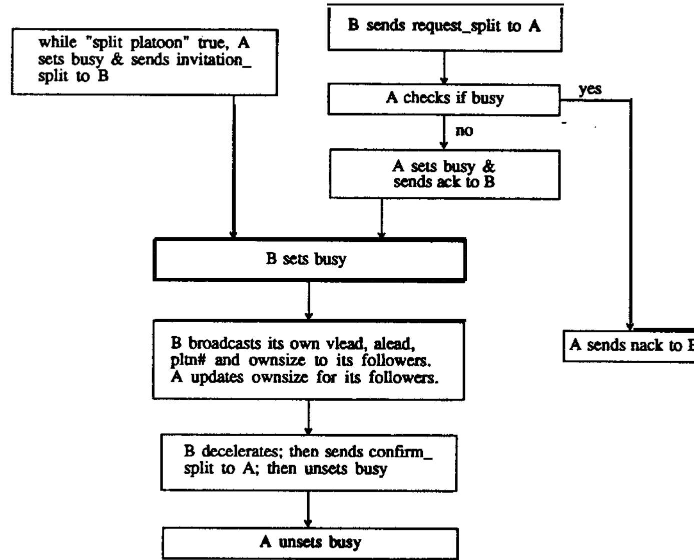


Figure 11: Flow diagram for split

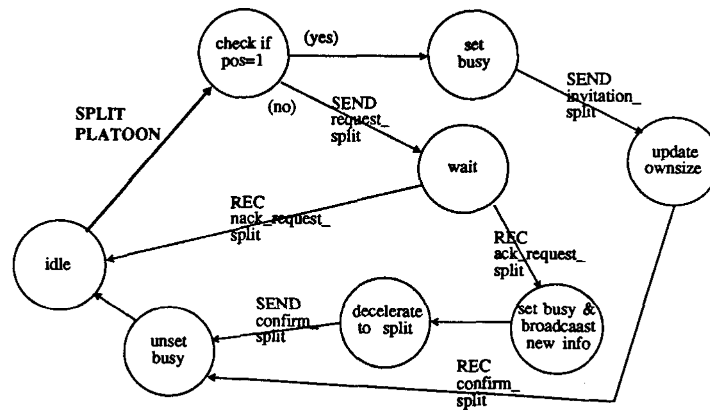
state transitions in the two machines.[7] The state transition labeled *'MERGE PLATOON'* is initiated by a 'supervisor' state machine. That machine, denoted **SUPR,** is defined in §6.

Split

Figure 11 gives the sequence of' events needed to achieve the split maneuver in Figure 6. The split may be initiated by the platoon leader (vehicle A) or a follower (vehicle B). In the latter case, B forwards a request to A; if A is not busy it initiates the split maneuver.

This flow diagram is transcribed into the pair of state machines of Figure 12. The machine on top refers to the initiator (A or B) which initiates the split. The other machine refers to the respondent, (B). B will become the leader of the rear platoon following the split. The two machines are synchronized by matched transitions *'SEND xx'* and *'REC xx'*. Finally, the transition **SPLIT PLATOON'* is initiated by the supervisor machine **SUPR.** Recall that the condition 'check *if* $pos = 1$' returns (yes) if and only if the vehicle is a leader (see definition of state (1)), otherwise it returns (*no*).

'The implementation of synchronous transitions among processes running on two separate computers, one in each vehicle, will require some inter-process communication facility. hlany operating systems provide such facilities.



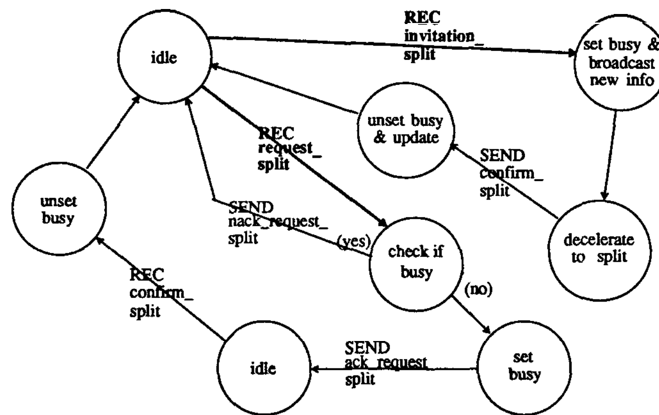


Figure 12: State machines for split: initiator (top): respondent (bottom)

Change lane

This is the most complex of the three maneuvers. see Figure 7. The free agent, vehicle A, in lane 3 initiates the change lane request. Depending on what A's lateral sensor detects in lanes 1 and 2, there follows one of three event sequences of Figure 13. Case 1 occurs when lane 2 is unoccupied and lane 1 is occupied. *A* then requests C not to move. The resulting sequence of events is the leftmost branch in Figure 13.

Case 2 occurs when lane 2 is occupied and the event sequence in the middle branch ensues. If B is not busy, it responds to A's request in one of three ways: (i) B asks A to decelerate, or (ii) B splits its own platoon at a follower B_i, or (iii) B itself decelerates. The maneuver succeeds whether a response is selected arbitrarily or on the basis of traffic conditions.

Case 0 occurs when lanes 1 and 2 are both unoccupied. The rightmost branch of Figure 13 describes the corresponding event sequence. In this case A immediately orders its regulation layer to change lane.

The flow diagram is transcribed into the pair of machines of Figure 14. The machine on top corresponds to the initiator il. A copy of the machine on the bottom is in each of the vehicles B, B_i, and C that may be engaged as a respondent.

Consider *A's* state machine. The transition *'CHANGE LANES'* is initiated by the supervisor machine **SUPR.** If the result of the condition *'check if' adjacent lane clear'* is *'no'*, then Case 2 prevails and A sends *'SEND request_change_lane'* to vehicle B. If the result of the condition is *'yes'*, then the condition 'check if fur *lane clear'* is tested. If the answer is 'yes', Case 0 prevails and A orders **its** regulation layer to change lane. If the answer is *'no'*, it is *Case* 1, and A sends *'SEND request_change_lane'* to vehicle C.

We now discuss the respondent's state machine. Recall that this may be vehicle C, or B, or **both** B and B_i. In Case 1, the respondent is C, and only the loop of transitions labeled 'C' will be activated. In Case 2, either the loop labeled 'B' is involved or those labeled 'B' and 'B_i' both are involved. Only this last situation may need additional comment. If B's machine is in state *'decide'* and the transition *'SEND request_split_chg_ln'* occurs, that message is received by its follower B_i whose machine is in state *'idle'* and undergoes the transition *'REC request_split_chg_ln'* (i.e. loop labeled B_i).

This concludes the design of the informal protocol state machines. We introduce some names for these machines to facilitate relating them to the formal machines presented in §6. The following convention is adopted. Machine names are written in boldface. The first letter is either **A,** B, **Bi** or C corresponding to the vehicle names in Figures 5,6,7. The second letter is **P** for protocol, and to distinguish these machines from those which model the environment. The third letter is either C or **R** depending on whether the machine commands a maneuver or responds to a command. Thus there are two machines for merge, **BPCmerge and APRmerge**. There are four machines for split: **APCsplit** and **BPCsplit** depending

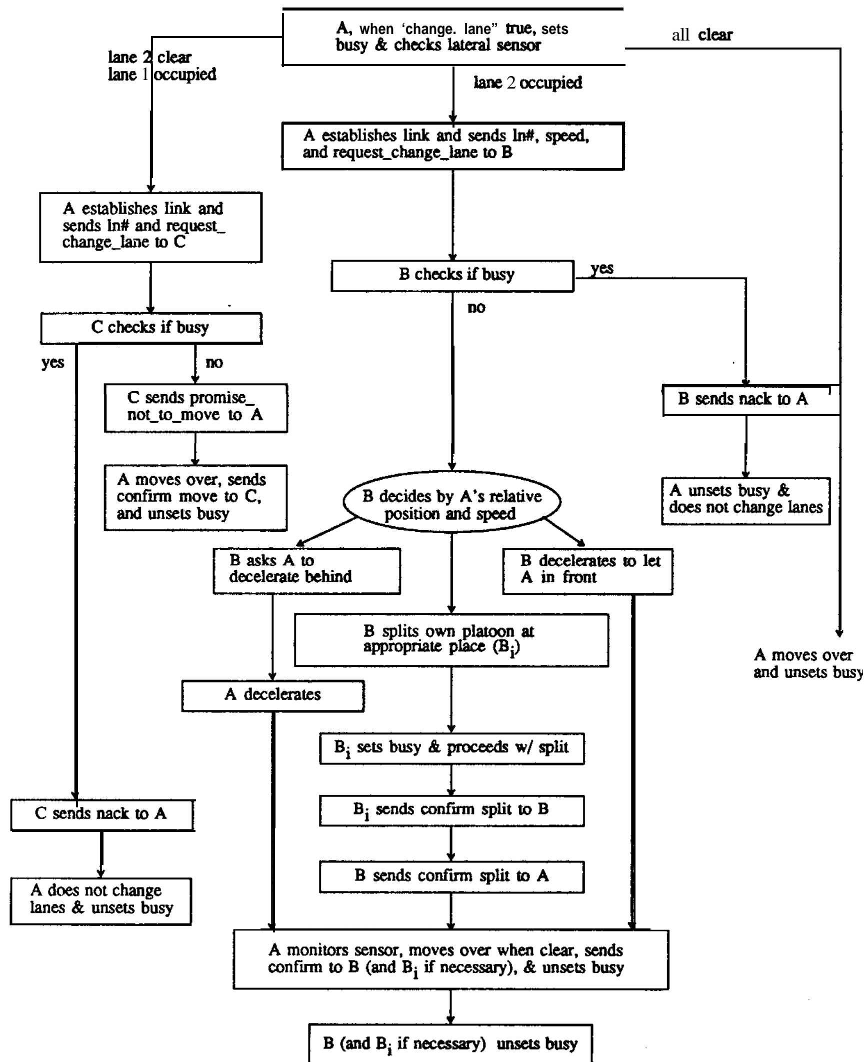
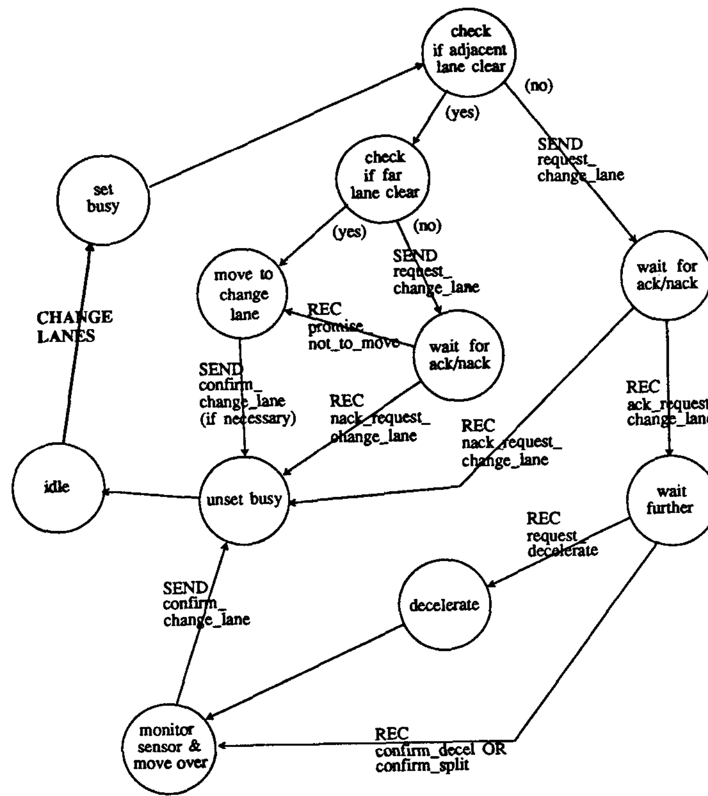


Figure 13: Flow diagram for change lane



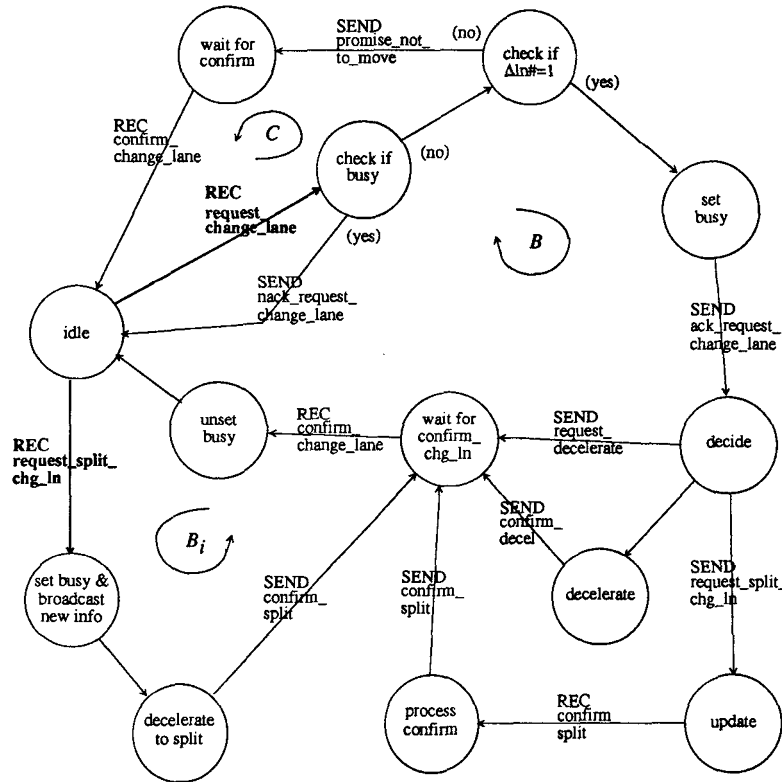


Figure 14: State machines for change lane: initiator (top), respondent (bottom)

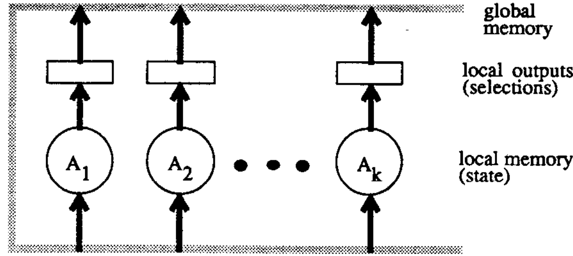


Figure 15: The COSPAN model

on whether **A** or ***B*** in Figure 6 issues the command, and **APRsplit**, **BPRsplit** depending on the respondent. There are four machines for change lane: **APCchg** for the initiator, and **BiPRchg**, **BPRchg**, and **CPRchg** for the three potential respondents. Each protocol layer controller will have a. copy of all these ten machines. Three additional supervisor machines will be introduced in §6.

5 Introduction to COSPAN

We give a very brief introduction to COSPAN. For more details. see [8, 9] and the references therein. COSPAN (coordination-specification analysis) is a software system used to specify a system of interacting finite state machines and to prove or verify that the behavior of the specified system satisfies certain properties.

Specification. COSPAN's model of a system of interacting state machines can be understood with the aid of Figure 15. Each $\mathbf{A_i}$ is a state machine or COSPAN process.[8] A process has internal local memory - its state. Let r;(t) be the state of process $\mathbf{A_i}$ at time $t = 0, 1, 2, \ldots$. With each state is associated one or more *outputs*, and the process selects one of the outputs in a non-deterministic manner. Suppose $\mathbf{A_i}$ selects $y_i(t)$ (associated with $x_i(t)$). The global output $y(t) = (y_1(t), \ldots y_k(t))$ is seen by all the machines.

We now explain state transitions. Associated with each pair of states (x_i, x'_i) of $\mathbf{A_i}$ is a binary predicate on the global output y. It is denoted $P(x_i, x'_i)(y)$. The transition $x_i \rightarrow x'_i$ is *enabled* at a global output y if $P(x_i, x'_i)(y)$ evaluates lo *true*. If for a particular state more than one transition is enabled, the process selects one of these non-deterministically.

Finally, each process starts in some prespecified initial state at time 0. A *behavior* of a system is a. pair of infinite sequences of global or system states and outputs

$$x(t) = (x_1(t), ..., x_k(t)), \quad y(t) = (y_1(t), ..., y_k(t)), t = 0, 1, 2, \ldots$$

such that for every i and t, $x_i(0)$ is the initial state of $\mathbf{A_i}$, $y_i(t)$ is an output associated with

[8] We use 'state machine' and 'process' interchangeably below and in §6.

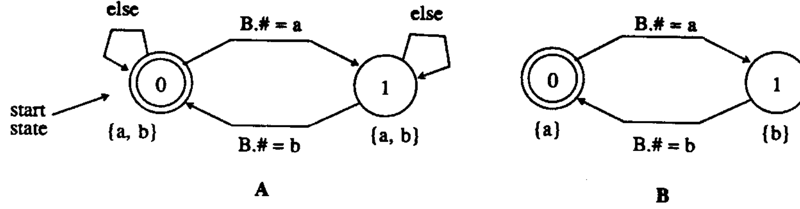


Figure **16:** COSPAN example system

$x_i(t)$, and the transition $x_i(t) \to x_i(t+1)$ is enabled at $y(t)$.

The *language* generated by such a system is denoted $L(\mathbf{A}_1, \ldots, \mathbf{A}_k)$. It consists of the set of all infinite sequences y(t), $t = 0, 1, \ldots$ such that (z(t), $y(t)$), $t = 0, 1, \ldots$ is a behavior for some state sequence $x(t)$, I = $0, 1, \ldots$

We emphasize two features of a COSPAN specification. First, the interacting machines $\mathbf{A}_1, \ldots, \mathbf{A}_k$ (implicitly) define a 'product' machine with state $x(t) = (x_1(t), \ldots, x_k(t))$ and output $y(t) = (y_1(t), \ldots, y_k(t))$. If $\mathbf{A}_i$ has n_i states in all, the product machine has $n = n_1 \times \ldots \times n_k$ states. However, in practice, only a small fraction of these n states is reachable from the initial state $(x_1(0), \ldots, x_k(0))$. The COSPAN compiler generates an internal representation of the product machines including only the reachable states.

Second, the specification gives a 'closed' system in the sense that there is no external input. Therefore, in order to describe the informal machines of §4 in COSPAN, we must specify not only the protocol machines, but also their interfaces to the sensors, monitors, and link and regulation layers.

Example. Figure 16 presents a system consisting of two interacting machines, **A** and B. **A** has two states, 0 and 1. In both states it can select any output from $\{a, b\}$. The transition $0 \to 1$ is enabled if the predicate 'B.# = a' evaluates *true*, otherwise $0 \to 0$ is enabled. (In COSPAN notation, X.# denotes the output of process X.) The transition $1 \to 0$ is selected if B.# $= b$, otherwise $1 \to 1$ is enabled. The initial or start state is 0; it is indicated by a double circle.

B also has two stales. In state 0 only a may be selected, in state 1 only b may be selected. The transition $0 \to 1$ is enabled if B.# = a, $1 \to 0$ is enabled if B.# = b. The transitions $0 \to 0$, $1 \to 1$ arc never enabled (i.e. the corresponding predicates are identically false). The initial state is 0.

This system generates only one state sequence

$$(x_A(t), x_B(t)) = \begin{cases} (0,0) & \text{if } t = 0, 2, \ldots \\ (1,1) & \text{if } t = 1, 3, \ldots \end{cases}$$

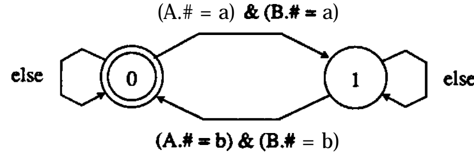


Figure 17: Monitor for example system

But the language $L(\mathbf{A}, B)$ contains infinitely many sequences

$$(y_A(t), y_B(t)) = \begin{cases} (a,a) \text{ or } (b,a) & \text{if } t = 0,2,... \\ (a,b) \text{ or } (b,b) & \text{if } t = 1,3,... \end{cases}$$

Verification. This consists of one or more tests of the form

$$L(\mathbf{A}_1, \ldots \mathbf{A}_k) \subset L(\mathbf{T}) \quad (2)$$

where **L(T)** is the language *accepted* by a task monitor T.[9] A monitor **T** is a state machine coupled to the system **Al, ... A**$_k$ except that **T** has no outputs. Thus the monitor state moves in response to the system output $(y_1, \ldots, y_k)$; however, since it has no outputs of its own, it cannot affect the system behavior. **L(T)** is defined by acceptance conditions involving two sets called cycle *set* and recur. A cycle set is a. collection $C_1, C_2, \ldots$ of subsets of the states of **T,** and recur is a subset of state transitions of **T.** An infinite sequence y(t), $t = 0, 1, \ldots$ is in **L(T)** (it is said to be accepted by **T)** if after some finite time, the state of **T** stays forever in one of the C_i or the transitions in *recur* occur infinitely often. Given **Al, . . .** , A$_k$ and **T,** the COSPAN software can verify if the test (2) succeeds or fails.

Figure 17 gives a monitor for the system of Figure 16. Suppose the acceptance condition is

cyset {0}, recur 0 → 1

Observe that $L(\mathbf{A}, B)$ contains the output sequence

$$(a,a),(a,b),(b,a),(a,b)...$$

which leads to the monitor state sequence 0, 1, 1, 1, ... For this sequence both *cyset* and *recur* conditions fail; so $L(\mathbf{A}, \mathbf{B}) \not\subset$ **L(T).**

6 Formal Specification and Verification

In §4 ten informal state machines were described. They define the protocols for the three elementary maneuvers. Three additional machines are needed for path planning (see §3),

[9] Any $\omega-regular$ language can be expressed in the form L(T); $\omega-regular$ languages are defined by Büchi automata [10, 11].

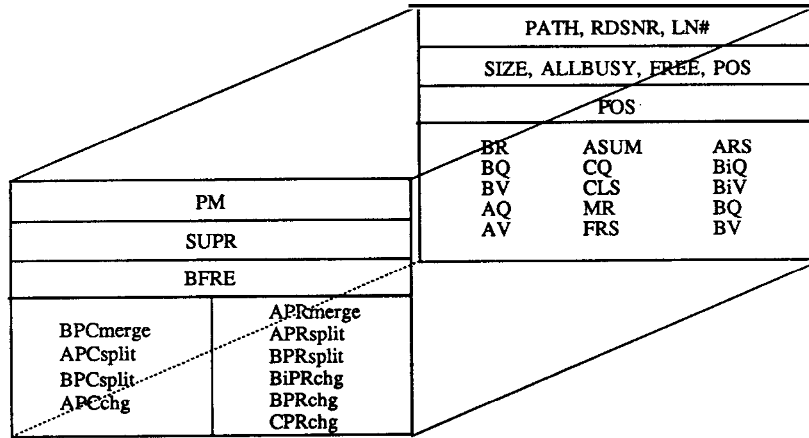


Figure 18: The four sublayers of the platoon layer

PM, SUPR and **BFRE.** The path monitor machine PM compares a vehicle's current *ln#* and *sect#* with the assigned path to determine when to change lane and when to keep in the same lane. Based on the output of **PM SUPR** determines when to invoke **BFRE;** the rest of the time it invokes merge and split in such a way as to track *optsize*. **BFRE** invokes the split maneuver(s) needed to become a free agent. These 13 interacting state machines together constitute the platoon layer design. They can naturally be arranged in the four sublayers displayed in the left panel of Figure 18.

It should be understood that all 13 machines are present in each vehicle's platoon layer. The **PM** and **SUPR** machines are 'running' at all times. The 11 machines in the two bottom layers are 'invoked' as needed depending on the role (command or response) the vehicle plays in a particular maneuver. For example, BPCmerge is invoked by **SUPR** when it issues a command to merge. In fact, all the command protocol machines are invoked by **SUPR.** A response protocol machine on the other hand is invoked by a request (from another vehicle) for that maneuver. A machine which is not invoked remains in its '*IDLE*' state.

We have adopted as a rule of platoon management the restriction that a platoon may engage in at most one maneuver at a time. Thus, for example, a platoon may not merge with another while it is also engaged in a split maneuver. This rule permits an enormous simplification in the design because it can now be modular since the specification and verification of each maneuver protocol can be done separately.

In order to enforce this 'one maneuver at at time' rule the various machines must he coordinated in such a way that when a platoon receives two or more maneuver requests (from vehicles uithin the platoon or from neighboring platoons), one and only one request is granted. A coordination mechanism that grants at least one request is said to be deadlock-

free, and a. mechanism that grants at most one request is said to achieve mutual exclusion.[10] In our design deadlock is prevented by enforcing a priority among requests: among all response machines the change lane maneuver has highest priority. followed by split, followed by merge; and in case of conflict between command and response, command receives priority. Mutual exclusion is achieved by using a, single **ALLBUSY** flag for all the machines: it can he set by any protocol machine (subject to the priority). and a, request is denied if the flag is set. The implementation of mutual exclusion and priority will become clear as we specify each machine.

The panel on the right of Figure 18 lists another collection of machines arranged in four corresponding sublayers. These machines specify the 'environment' within which the platoou layer operates. There are three types of such machines. One type represents interfaces between the platoon layer and the link and regulation layers. The second type represents interfaces to sensors and roadside monitors. The third type represents the results of various tests conducted on the platoon layer state (1). The environment machines serve two purposes. They art needed to 'close' the system around the platoon layer so that verification is possible. And they are essential for future work to develop the IVHS system since they help standardize the interfaces among different components and layers.[11]

The rest of the section is arranged as follows. We first specify and verify in COSPAN the protocols for each elementary maneuver. We then specify and verify path planning.

Protocols for elementary maneuvers

These machines are in the hot tom sublayer of Figure 18. Each maneuver is specified and tested separately. This allows modularity. However, since priority and mutual exclusion are incorporated into these machines, some transitions depend on outputs of machines that are specified much later.

Merge

This involves the seven processes shown in Figure 19. These seven processes correspond to the two informal machines of Figure 10. (The letters **A** and B in the process names correspond to the vehicles A and ***B*** in Figure 5.) The two processes above the dotted line are protocol processes, the other five represent the environment. Double arrows indicate information links. In terms of COSPAN this means, for example, that the state transition predicates of **APRmerge** involves only the outputs of **BPCmerge**, **ASUM** and **AQ**.[12]

[10] These terms are used in oprrating systems.

[11] The role of interface standards in helping to structure IVHS system design is further discussed in [2].

[12] Note that the link between **BPCmerge** and **APRmerge** requires inter-vehicle communication; the other links involve inter-process communication within the same vehicle.

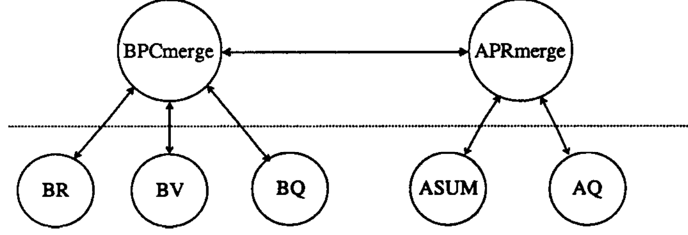


Figure 19: Merge protocol maneuver processes

The five environment processes are specified in Figure 20. **BR** gives the response of vehicle B's longitudinal range sensor, **BV** is the interface with its regulation layer, **BQ** indicates whether its busy flag is set. **ASUM** gives the result of the test '$ownsize(B)+ownsize(A) \leq optsize$', **AQ** refers to A's busy flag. BQ and **AQ** will later be replaced by **ALLBUSY** which refers to a single (global) flag that can be set by all protocol machines on the vehicle; it is used to enforce mutual exclusion.

The two protocol processes are specified in Figure **21. BPCmerge** is normally in the '*IDLE*' state. It initiates the merge protocol when '**SUPR**.# = *merge*' (i.e. when the vehicle's **SUPR** machine selects '*merge*'). In order to test the merge protocol by itself this predicate will be replaced by '*true*' so that this maneuver is repeatedly initiated.

The **APRmerge** protocol machine (in vehicle *A)* responds to the condition '**BPCmerge**.# = *request_merge*'. It responds affirmatively only after checking various conditions in the state '*CHECK STATUS*'. Those conditions enforce the priorities mentioned above. In testing the merge protocol by itself, the conditions involving any process other than those in Figure 19 are removed.

The monitor of Figure 22 defines a test of the system of' seven processes in Figure 19. It has the acceptance conditions

$$cyset\ \{0\},\ recur\ 2-0,1-0 \tag{3}$$

We briefly explain the test. The monitor starts in state 0. The transition into state 1 is enabled only if' '**BPCmerge**.# = *request_merge*', indicating beginning of the maneuver. However, from Figure 21 we see that this transition will not occur unless B's range sensor indicates '**BR**.# = *car_ahead*'. It is possible that BR always makes the selection '*no_car_ahead*'. (Such non-deterministic state machines are frequently used to model the environment.) Therefore, a behavior of the monitor which remains forever in state 0 is correct. Hence the *cyset* condition in (3). However, once the merge request is issued by vehicle B, **APRmerge** should grant or deny the request, causing the monitor to go through state 2 or 3, respectively. In either case, the monitor should return to state 0. This should occur infinitely often, hence the *recur* condition in (3).

The state machines in Figures 20,21,22 can be mechanically transcribed into the COSPAN

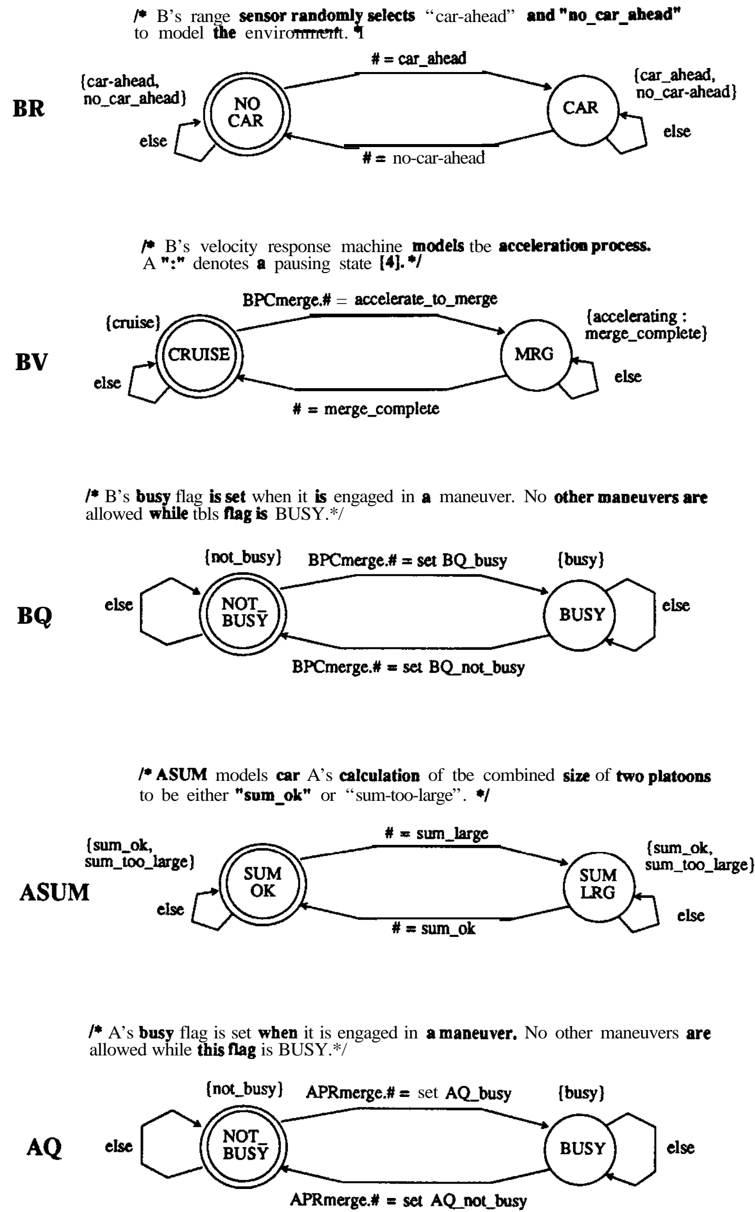


Figure 20: Merge environment processes

Figure 21: Merge protocol processes

Figure 22: Merge monitor

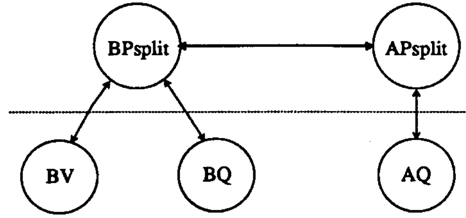


Figure 23: Split protocol maneuver processes

code merge.sr listed in the Appendix. Part of the COSPAN output upon performing this test is shown below:

80 states reached
336 resolutions performed
Task performed!

This means that a. total of 80 states in the system of Figure 19 are reachable from the initial state and 336 state transitions can be enabled. The system behavior is accepted by the monitor.

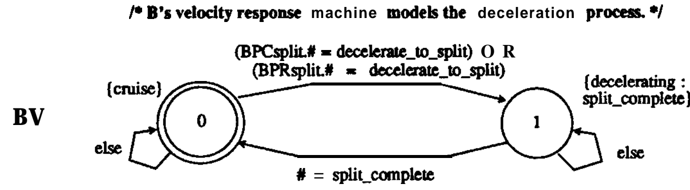
Split

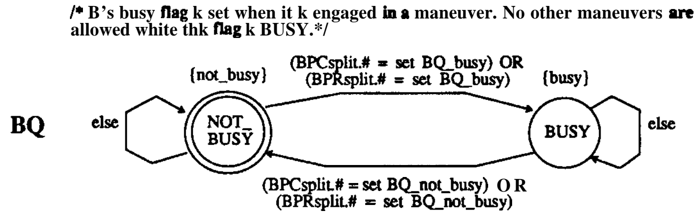
This involves the two protocol processes and three environment processes as in Figure 23. The letters **A** and B correspond to the names of the vehicles in Figure 6. The third letter C or **R** is omitted here since either process can initiate the command or repond to it. **BV** is the interface with the regulation layer; BQ and **AQ** indicate busy flag status. These three processes are specified in Figure 24.

Recall from Figure 6 that. there are two scenarios for the split maneuver. We call these 'leader wishes to split' or 'follower wishes to split'. The protocol machines for the first scenario are specified in Figure 25. In this case **APCsplit** is invoked either by **BFRE** or by **SUPR,** the other conditions enforce mutual exclusion and priorities. For testing the protocol by itself this condition is replaced by '*true*'.

BPCsplit is the follower responding to '**APCsplit**.# = *invite_new_lead*'. A follower normally does not use the busy flag. In this case only, however, it does so since it is asked to become a leader. Before **BPRsplit** commits to the maneuver, it verifies that it is a follower, that its **SUPR** and **PM** are in the appropriate states to allow a positive response, and that the change lane maneuver, which has higher priority, is not invoked. For testing the protocol by itself only the condition on **'APCsplit.#'** is retained.

The protocol machines in the 'follower wishes to split' scenario are specified in Figure 26.





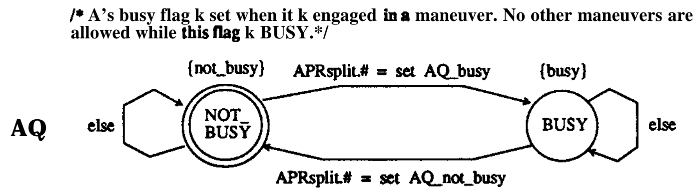


Figure 24: Split environment processes

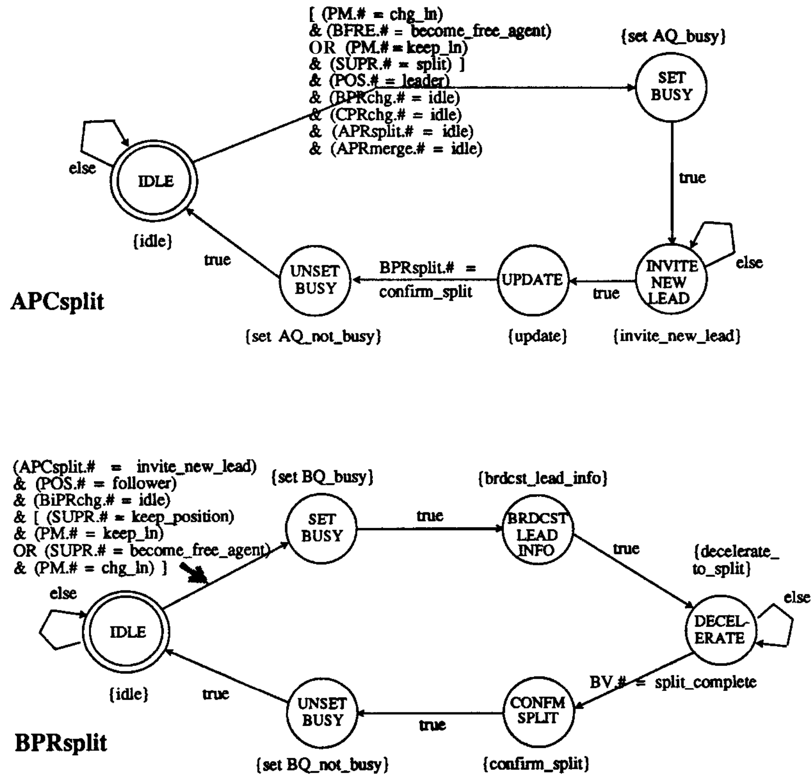


Figure 25: Split protocol processes for 'leader wishes to split'

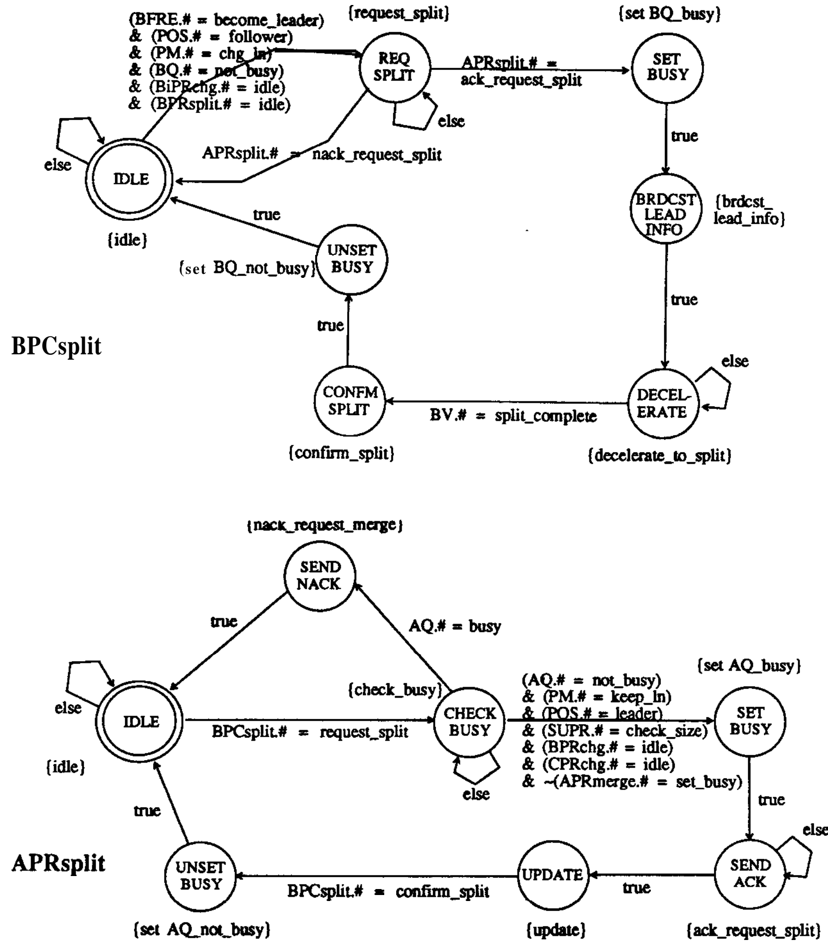


Figure 26: Split protocol processes for 'follower wishes to split'

In this case **BPCsplit** is invoked by the free agent supervisor **BFRE.** The remaining conditions enforce priority and mutual exclusion. For testing the protocol by itself this condition is replaced by `'true'`. **APRsplit** is the process in the lead vehicle that responds to **BPCsplit.** A positive reponse is made only under the following conditions: **PM** and **SUPR** are in appropriate states; higher priority processes are not invoked; and the lower priority machine associated with the merge maneuver, namely **APRmerge**, is not setting the busy flag ($\sim$ denotes '*not*'). For testing the protocol by itself only the busy condition is retained.

The two monitors of Figure 27 test the two scenarios. The 'leader wishes to split' monitor (upper process) involves the condition:

$$recur\ 1 \rightarrow 2$$

The 'follower wishes to split' monitor (lower process) involves the condition:

$$recur\ 1 \rightarrow 2,\ 1 \rightarrow 4$$

Both test were successful; the number of reachable states is 17 and 19, respectively.

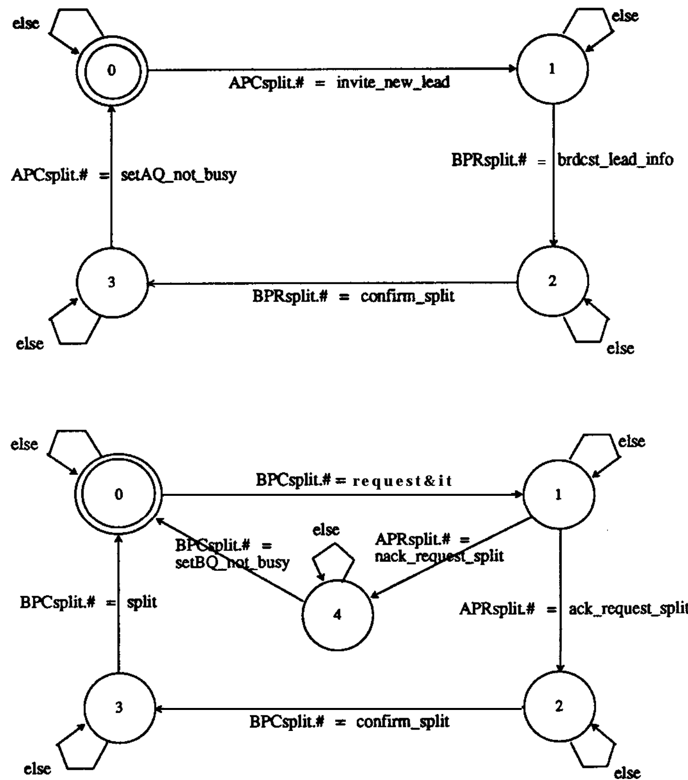


Figure 27: Split monitors

Change lane

This maneuver may involve up to four vehicles as indicated in Figure 7: the free agent initiating the request, the leader of the adjacent lane platoon, a follower in that platoon that may be asked to split, and the leader of the far lane platoon. The specification involves 13 processes as shown in Figure *28.* There are four protocol processes (above the dotted line) with names corresponding to the vehicle names in Figure 7, and nine environment processes. Information links involve message exchanges between the protocol machines in different vehicles and inter-process communication on the same vehicle between a protocol machine and its environment.

Figure 29 indicates the busy flag for vehicle C. Figure 30 specifies the two environment processes for vehicle B_i. Figure 31 specifies the five environment processes for vehicle A. Figure 32 specifics the three environment processes for vehicle *B.* The **CLS** process gives the result of the procedure that *B* uses to decide how to make room for A depending on the latter's position and speed relative to *B's own* platoon.

We now specify the four protocol machines themselves. Figure 33 specifies free agent A's protocol process. It is invoked by the supervisor process **SUPR.** When testing the protocol by itself, the condition '**SUPR.#** = *start_chg_ln*' *is* replaced by '*true*'. The logic of the change lane protocol was explained in §3.

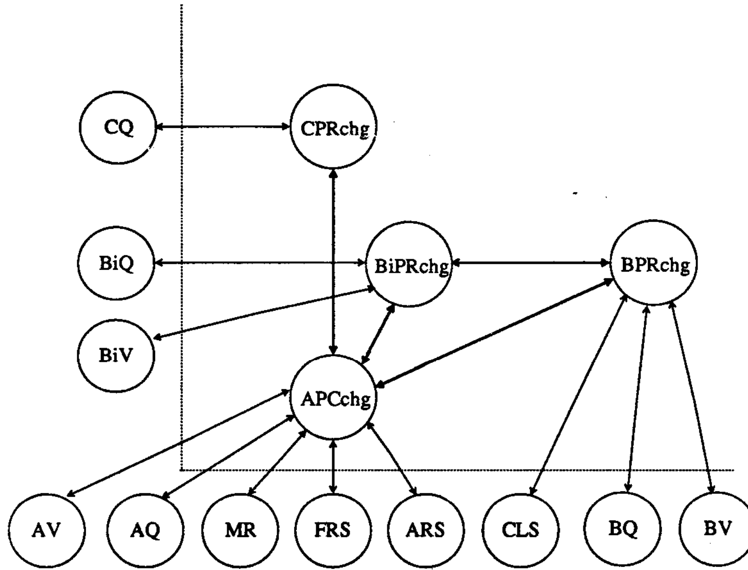


Figure 28: Change lane maneuver processes

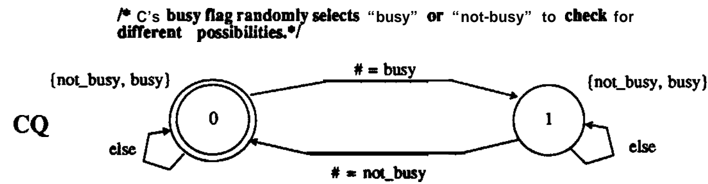
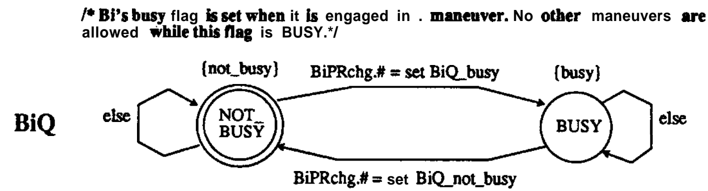


Figure 29: Change lane environment process for vehicle C



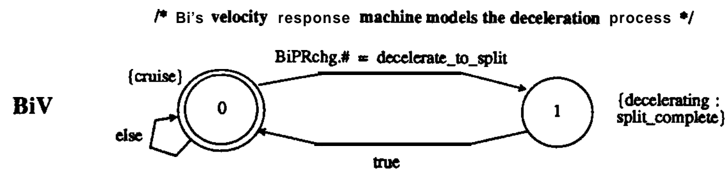
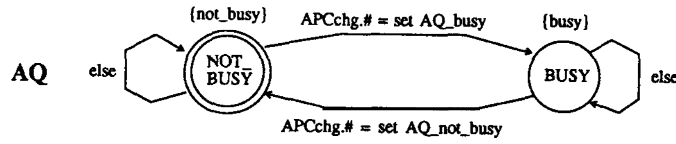


Figure 30: Change lane environment processes for vehicle B_i

/* A's busy flag is set when it is engaged to • maneuver. No otba maneuvers are allowed while this flag is BUSY.*/



/* A's far lane range sensor randomly selects "lateral car" and "no lateral car" to model the environment */

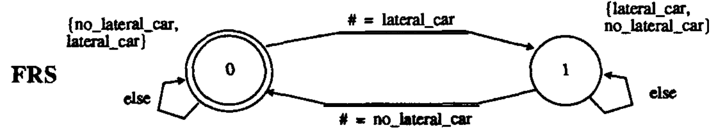
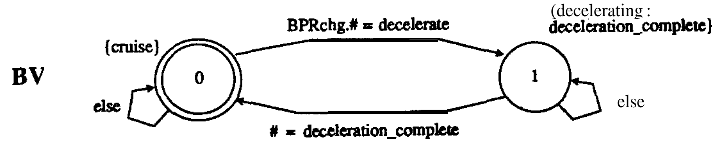
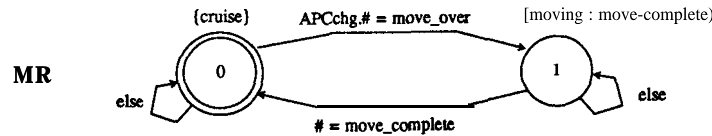


Figure 32: Change lane environment processes for vehicle *A*

/* B's velocity response machine models tbc deceleration process. */



/* A's movement response machine models tbe lateral movement of changing lanes. • !



/* B's change lane supervisor decides how to make space for A according to some function of A's position, speed, and traffic conditions. However, a random selection is used here to model tbe selection process since no actual data is available. */

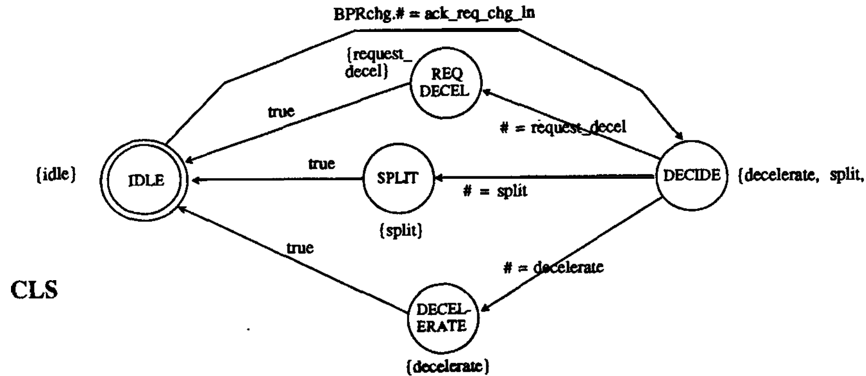


Figure 32: Change lane environment processes for vehicle *B*

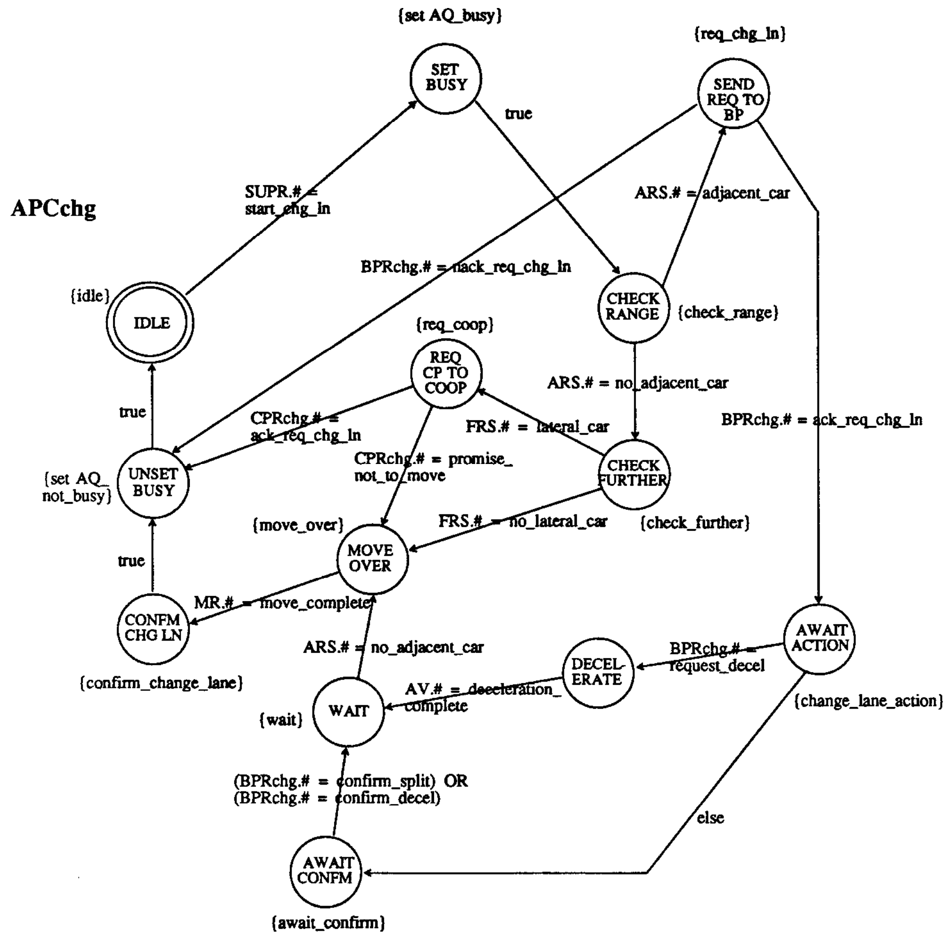


Figure 33: Change lane protocol process for vehicle A

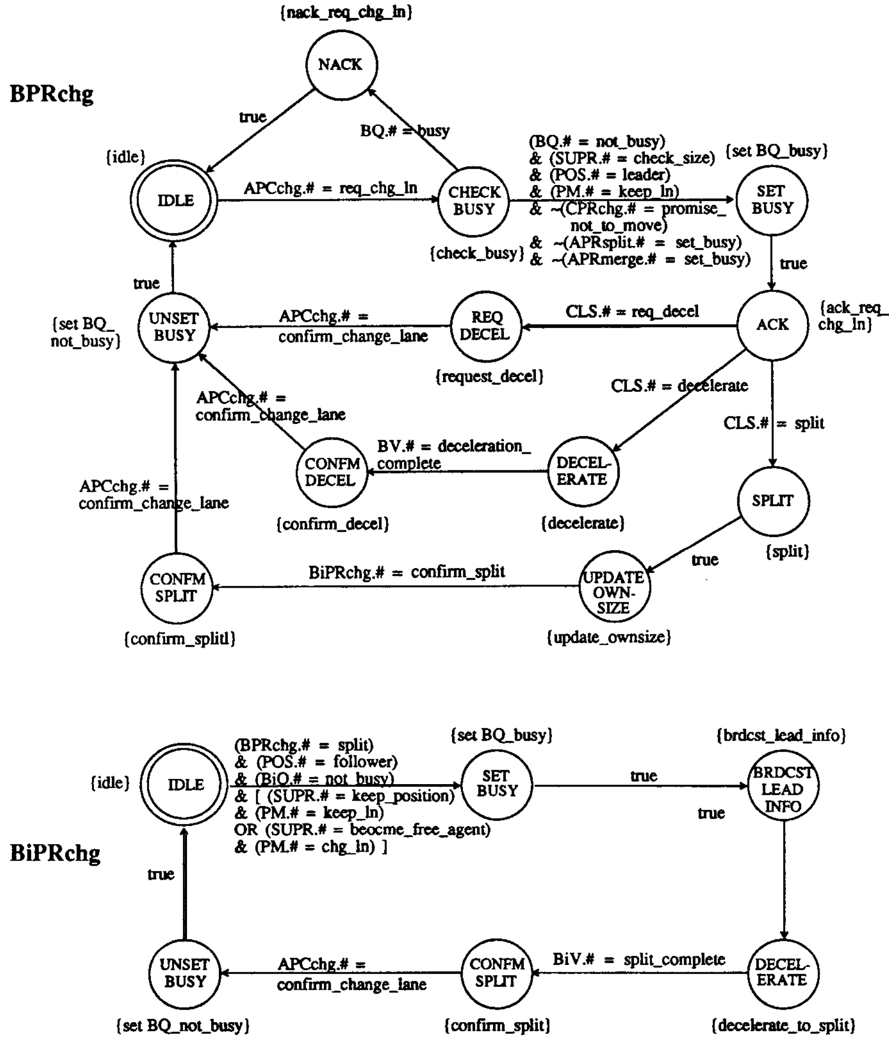


Figure 34: Change lane protocol process for vehicle B and B_i

Figure 34 specifies the protocol processes for the leader B and its follower B_i. If there is a platoon in the adjacent lane, its leader's process **BPRchg** responds to the request from **APCchg.** It responds affirmatively to the request only if its **SUPR** and **PM** are in the appropriate states, it is not responding to a change lane request from another free agent, and other response processes are not setting the busy flag at the same time. If it gives a positive response, **BPRchg** then determines how to make space for A. This is done by the response of its environment machine **CLS.**

B_i is a follower of B. **One** way in which **BPRchg** may decide to make space for A is to ask B_i to split,. The protocol process **BiPRchg** in vehicle B responds to the split request. It initiates the requested split if its **SUPR** and **PM** processes are in the appropriate states; then informs B when the split is complete; and then waits for confirmation from A that the change lane maneuver is complete before unsetting its own busy flag. When testing the change lane protocol by itself the conditions on the **SUPR** and **PM** selections are replaced

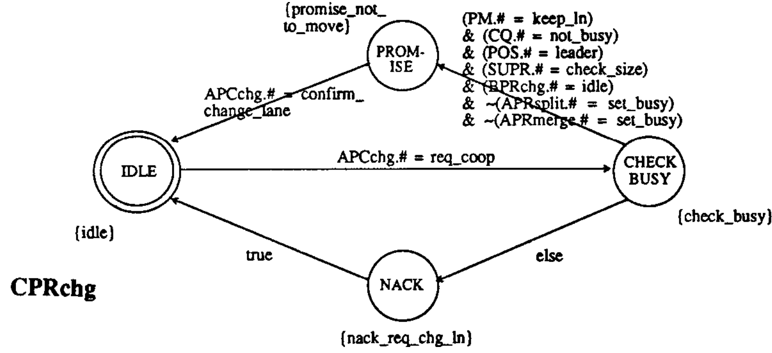


Figure 35: Change lane protocol process for vehicle C

by '*true*'.

Figure 35 specifies the protocol process for vehicle C in the far lane. C is the platoon leader in the far lane. It receives a request to cooperate from A only if there is no vehicle in the near lane (recall Figure 7). The request is denied if C is busy. If it is not busy, it makes sure that its **SUPR** and **PM** are in the appropriate slates, it is not responding to another change lane request, and the lower priority processes are not setting the busy flag at the same time. In testing the change lane protocol by itself only the busy flag is checked.

The monitor of Figure 3G is used to test this protocol. The test condition is

$$\text{recur } 1 \rightarrow 0,\ 3 \rightarrow 0,\ 4 \rightarrow 0$$

This test can be understood along the same lines as the previous monitors. The test is successful; 3,588 states of the system in Figure 28 arc reached from the initial state and 94,176 transitions are enabled.

Free agent supervisor sublayer

This sublayer contains only one process **BFRE;** its environment also contains one process **POS** whose output indicates whether the vehicle is a follower or a leader (this information is in the platoon layer state). The two processes arc specified in Figure 37. **BFRE** is invoked by its supervisor selection '**SUPR.#** $= become_free_agent$'. It then checks its status from **POS.** If it is already a leader ('**POS.#** = lender'), it moves to the state '*BECOME FREE AGENT*' from which it selects '**BFRE.#** = *become_free_agent*'. That selection invokes the **APCsplit** process, see Figure **23.** If it is **a.** follower, it selects '**BFRE.#** $= become_leader$'. That invokes the **BPCsplit** process; when that process completes, the vehicle has become a leader, and the same moves are carried out.

The monitor for this sublayer is specified in Figure 38. To test this sublayer, the split and merge maneuver protocol processes are included. The condition involving **SUPR.#** is

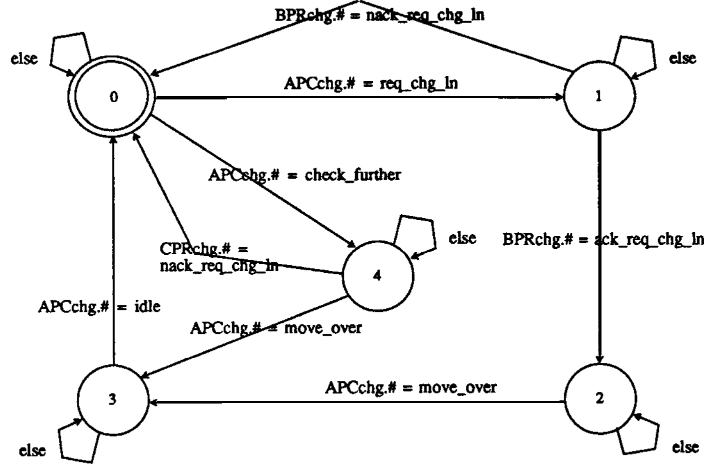


Figure 36: Change lane monitor

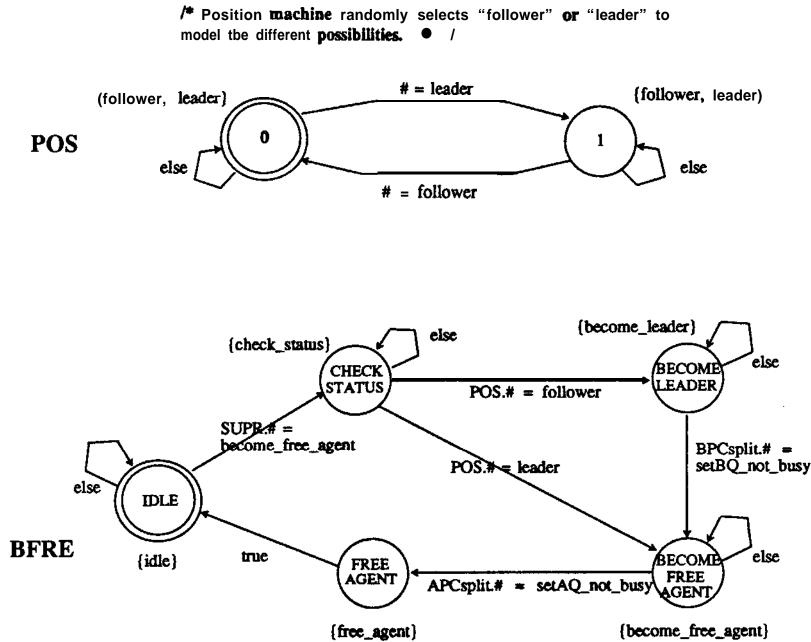


Figure 37: Free agent sublayer processes

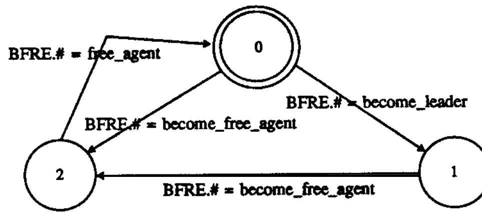


Figure 38: Free agent sublayer monitor

replaced by '*true*'. The acceptance condition is

$$recur\ 0 \rightarrow 2,\ 1 \rightarrow 2$$

The test is successful; 77 states are reached, and 162 transitions are enabled.

Platoon supervisor sublayer

This sublayer contains one process **SUPR** which runs continuously, and four environment processes: **SIZE, ALLBUSY, FREE,** and **POS. POS** is the same process as in Figure 37 and indicates whether the vehicle is a leader or follower. **SIZE** indicates whether the current, platoon size is larger or smaller than *optsize* (see (**I**)). **ALLBUSY** is the overall busy flag for the vehicle. It is set by any of the protocols specified above. Once it is set, no other maneuvers is permitted until the protocol process that set the flag completes the maneuver and unsets the flag. Processes that attempt to request initiation of a maneuver receive a *nack*, so they must try again.[13] **FREE** indicates whether the vehicle is a free agent (also determined from (I)). Figure **39** specifies **SIZE,** Figure **40** specifies **ALLBUSY**, and Figure **41** specifies **FREE.**

The platoon supervisor receives commands from the path monitor **PM** which decides when the vehicle must keep to the same lane and when it must change lane. When 'PM.# = *keep_ln*', **SUPR** tries to track *optsize* by executing split and merge as necessary.[14] When 'PM.# = *chg_ln*', **SUPR** makes sure that the current maneuver is complete before issuing the command '*become_free_agent*' to **BFRE.** Figure 42 specifies the supervisor process.

Figures 43 specifies one monitor for the supervisor sublayer. In order to conduct the test a **PM** process is included. It selects '*keep_ln* or '*chg_ln*' non-deterministically. It is also necessary to include the processes which interact with **SUPR,** namely **BFRE, BPRchg,**

[13] The design does not queue requests since that could lead to very poor performance. To see this suppose platoon 1 is in front of 2 which is in front of 3, Suppose 2 requests merge with 1 and sets itself busy; then 3 requests merge with 2, gets its request queued and then sets itself busy In this wag all platoons become busy. Eventually 2 completes merge; then 3's queued request is considered and may be denied (if the resulting platoon is too large) or accepted; aud so on, one at a time. When queuing is not allowed, several of the merge requests are executed in parallel.

[14] Tracking *optspeed* is a function of the regulation layer, see §7.

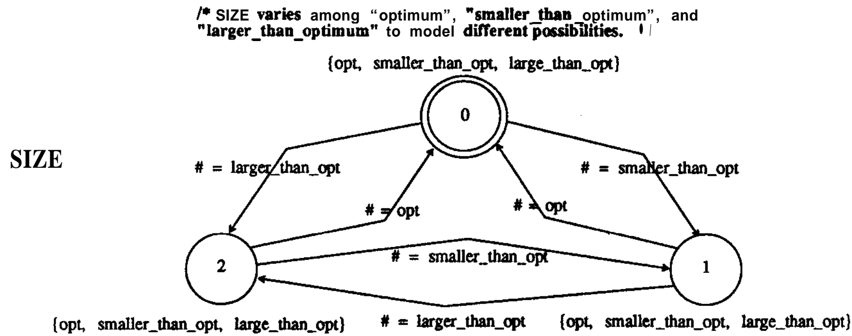


Figure 39: SIZE process for supervisor sublayer environment

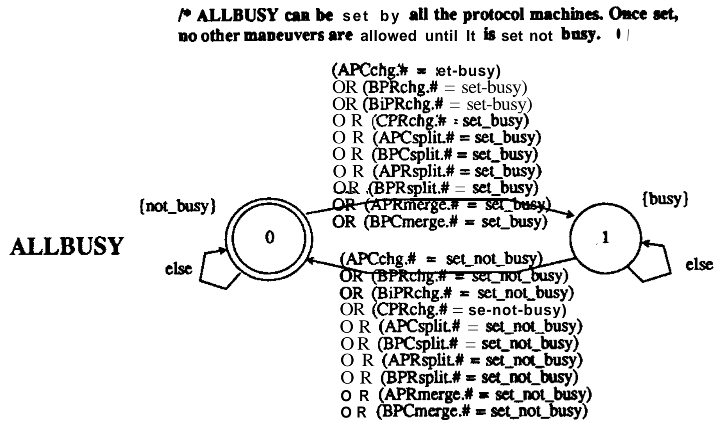


Figure 40: ALLBUSY process for supervisor sublayer environment

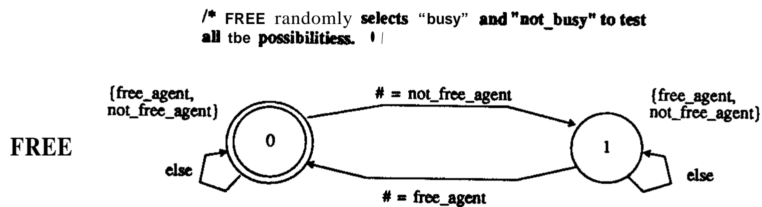


Figure 41: FREE process for supervisor sublayer environment

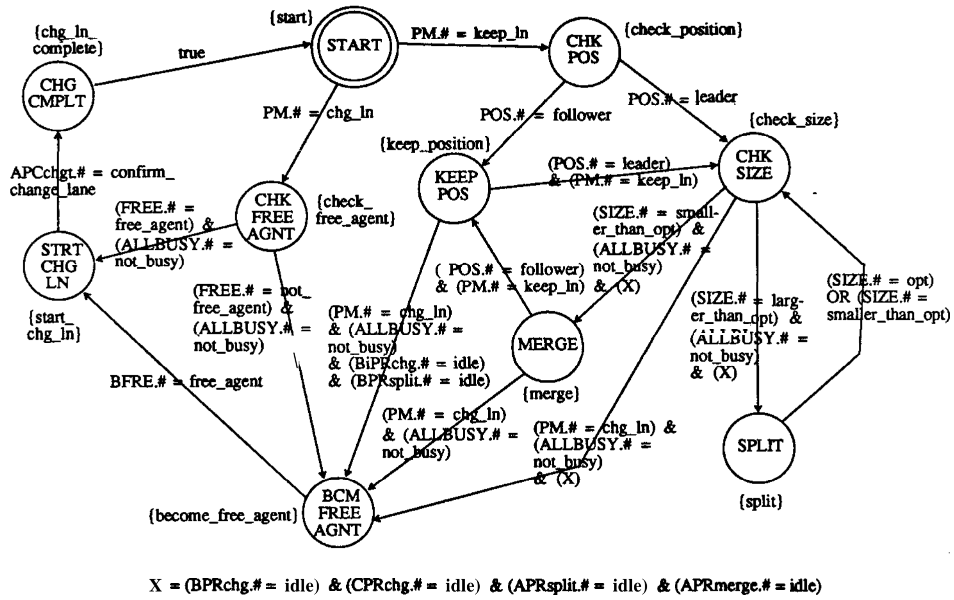


Figure 42: **SUPR process** for supervisor sublayer

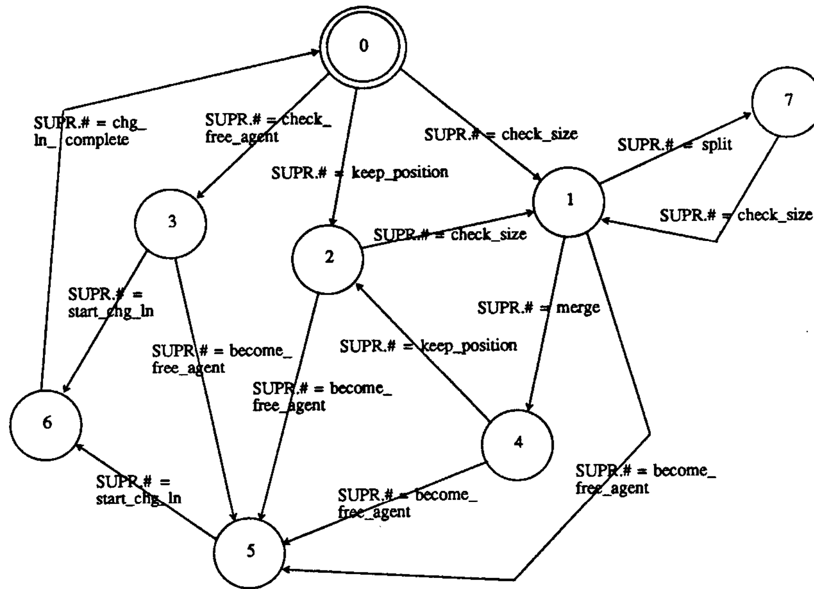


Figure 43: First monitor for supervisor sublayer

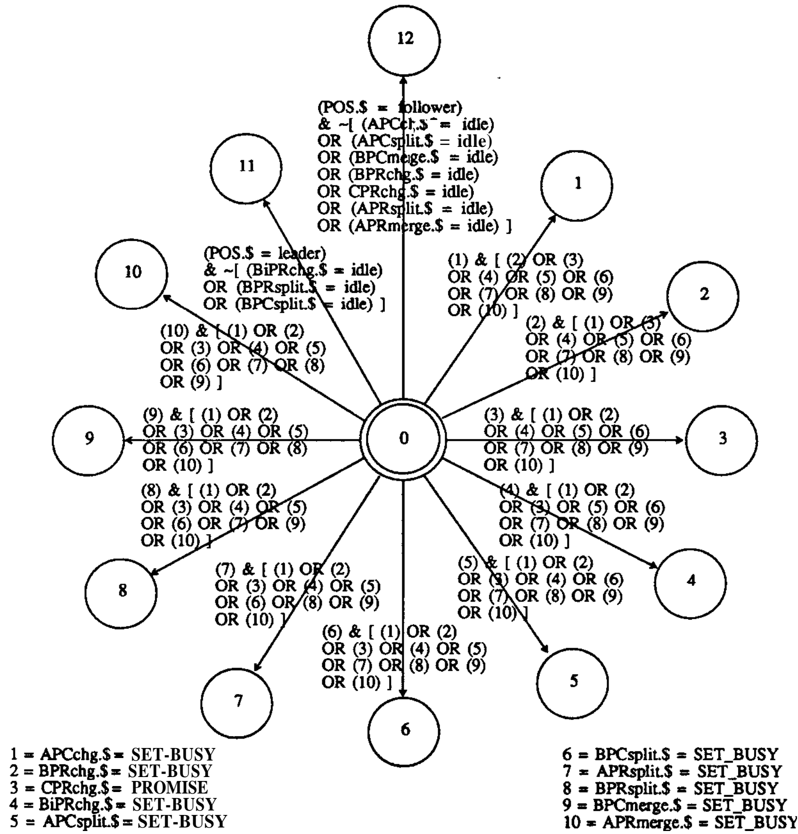


Figure 44: Second monitor for supervisor sublayer

CPRchg, **APRsplit** and **APRmerge** and their corresponding environment processes. [15] The acceptance conditions for this monitor are

$$\begin{array}{ll} \text{cyset} & \{1\},\{2\},\{4\},\{7\} \\ \textit{recur} & 1 \rightarrow 7, 1 \rightarrow 4, 4 \rightarrow 2, 6 \rightarrow 0, 1 \rightarrow 5, 2 \rightarrow 5, 3 \rightarrow 5, 4 \rightarrow 5 \end{array}$$

The *cyset* condition is needed since the vehicle may never be able to merge, split, etc. because the environment conditions may not permit it. The *recur* condition checks that **SUPR** executes the commands in the correct sequence. The test is successful; there are 89,913 reachable states for this system, and 3,179,28X transitions are enabled.

Figure 44 specifies the second monitor for the supervisor sublayer. This monitor tests whether the mutual exclusion scheme functions correctly. The only acceptance condition is

$$\text{cyset } \{0\}$$

[15] Since we have checked that these maneuver processes arc correctly specified, reduced versions of these processes are used; see Appendix. To prove that the verification using the reduced version is correct, one constructs a 'process homomorphism' mapping the original process into the reduced process. The homomorphism guarantees that the language generated by the reduced process (see §5) is larger than that generated by the original process; hence verification using the reduced process implies verification using the orginal process. We do not present the homomorphisms here. For the underlying theory see [12].

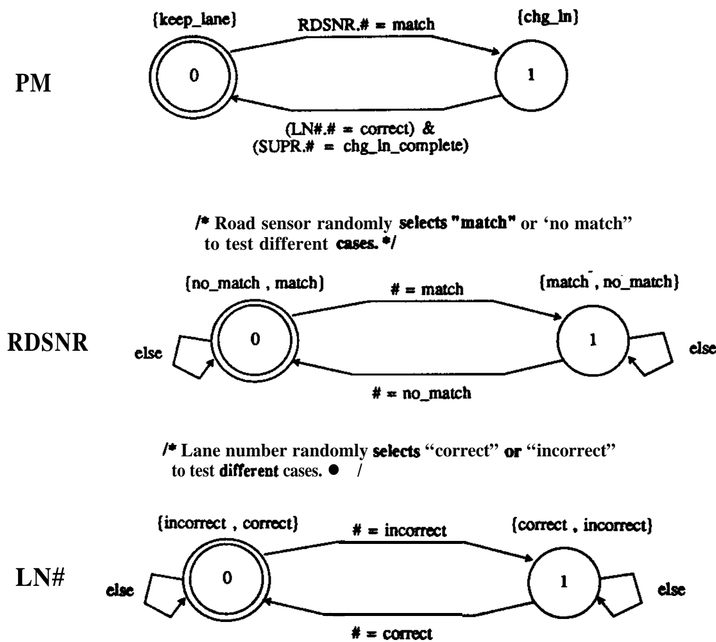


Figure 45: Path monitor sublayer processes

so that if the monitor moves to any other state the test will fail! A transition from 0 to states 1, ... ,10 will occur if two or more processes commit the vehicle to (different) maneuvers at the same time. A transition to states 11 or 12 will occur if the vehicle is a leader and its follower protocol processes are not in their *'idle' states* or *vice versa.* This test is also successful; there are 85,937 reachable states for this system, and 3,037,592 transitions are enabled.

Path monitor sublayer

This contains only the path monitor process, **PM** and two environment processes, RD-**SNSR (road** sensor). aud **LN#** (lane #). They are specified in Figure 45. Recall that the link layer assigns a path (l_2, d_2, l_3) to the vehicle when it enters the highway, see $3. The assigned path is stored in the vehicle. The road sensor determines the section of the highway on which the vehicle is traveling, compares it with the assigned path, and generates a selection '*match*' or '*no_match*'. In the model, this selection is non-deterministic. Similarly, **LN#** compares the assigned path with the current lane number to determine whether the vehicle should change lanes. This selection is also non-deterministic. Depending on the selections made by these two environment processes, **PM** commands **SUPR** to change lane ('*chg_ln*') or keep in the current lane ('*keep_lane*'). The two monitors used for this test are the same as for the platoon supervisor sublayer except that **PM** and its environment' are included. Both tests arc successful; 156,161 states are reached in the first test and 11,072,608 transitions axe enabled; 143,745 states are reached in the second test and 10,188,384 transitions are enabled. The COSPAN code protocol. sr for this system is included in the Appendix.

Concluding remarks

A vehicle's platoon layer controller implements three functions:

- It plans a path as a sequence of elementary maneuvers. The path must conform to the one assigned by the link layer.
- Before executing any maneuver, it must coordinate the vehicle's movement with the movement of neighboring vehicles to make sure that the maneuver can be carried out safely.
- Once coordination has been achieved, it orders the regulation layer to execute the maneuver.

Two features of the control problem make the design complex. First, the tasks require long sequences of decisions. We have limited the resulting complexity by requiring that a platoon engages in only one maneuver at a time. This leads to a modular structure of the platoon layer. The second feature contributing to complexity is that control authority is distributed among vehicles rather than being located in a central controller. We have addressed the resulting coordination problem by organizing the protocols into a hierarchy of four sublayers.

The resulting design may appear complex in terms of size: the platoon layer has about 500,000 reachable states. But the modular and hierarchical structure of the controller made the design specification straightforward and COSPAN made it quite easy to detect and correct, design mistakes.

7 Implications for Regulation Layer

The design of the platoon layer presented above presupposes the capability on the part of the regulation layer to implement five types of feed back control laws to accomplish certain tasks. We describe these tasks briefly.

Follower spacing control. When a vehicle is a follower, it must be controlled by a feedback law that maintains the required tight spacing with the vehicle in front of it in its platoon. This control action is typically decomposed into longitudinal control which determines acceleration and braking, and lateral control which determines the steering action needed to maintain the vehicle in its lane [13, 3, 4, 5].

Leader tracking optspeed. In the lane keeping mode, the leader should try and track the target speed announced by the link layer, while maintaining a safe headway (60m) from

the vehicle in front. This should be similar to current cruise control appropriately modified to account for the headway requirement.

Accelerate to merge. This feedback law is used by a leader to accelerate and merge with the platoon in front. This law could be implemented by first calculating a nominal trajectory given the distance and speed of the vehicle in front (the last vehicle in the preceding platoon), and then embedding the corresponding nominal open loop control in a longitudinal control feedback loop.

Decelerate to split. A follower who has just assumed the role of leader is required to slow down to achieve a. safe headway (60m) from the vehicle in front. This should be similar to, and simpler than. the previous feedback law.

Free agent change lane. This feedback law enables a free agent to move to a vacant space in the adjacent, lane. Again the approach of embedding a nominal open loop control in a feedback loop seems appropriate. This maneuver will require accurate position sensing systems. This task seems to involve the most demanding sensing requirements.

It should be noted that these are *types* of control laws. That is to say each type represents a class of laws indexed by several parameters. For instance, the spacing control law would be parametrized by the required spacing distance; similarly, the change lane law would be parametrized by the location of the vacant space, and the speed of vehicles in the adjacent lane. There may be other parameters as well provided by 'preview' information about, the geometry of the road, road conditions. etc, [14].

The proposed design specifies in a very simple fashion the interface between the platoon and regulation layers: the platoon layer issues a command, and the regulation layer eventually returns a response indicating successful completion. This interface needs to be enriched: the platoon layer may 'pass' several parameters to the regulation layer, and the latter may return 'success' or various kinds of 'errors' and 'exceptions'. The combined platoon and regulation layer together form a hybrid system of the type introduced in [15]. The theory of control of such systems remains to be developed.

8 Conclusions

We have discussed some aspects of the design of a control system for an IVHS system that organizes traffic in platoons of closely spaced vehicles under automatic control. The control tasks are complex. The proposed approach manages this complexity by structuring the design into three layers. The centralized link layer assigns a path to each vehicle entering the highway and targets for the aggregate traffic. The remaining tasks are distributed

among individual vehicles.[16]

The platoon layer in each vehicle is responsible for planning its path as a sequence of three elementary maneuvers, and for coordinating with neighboring vehicles the implementation of each maneuver. The regulation layer is responsible for executing a pre-computed feedback control in response to a command from the platoon layer.

Our main focus is on the design of the platoon layer. The tasks of path planning and negotiating each elementary maneuver are carried out by finite state machines. These machines are themselves structured into four functional sublayers so that each function can be designed and tested separately. The machines are specified and verified to function correctly by the COSPAN software system. The final design involves a system of 40 interacting machines (13 for the platoon layer and 27 for its environment) with about 500,000 states. This is a fairly complex design, but the design process is simplified tremendously by maintaining functional modularity and hierarchy.[17]

It may be worth noting the difference between the way- the vehicle trajectory is determined here (planning a path as a. sequence of elementary maneuvers and coordinating each vehicle's maneuver with its neighbors) and robotics- and artificial intelligence-based approaches to the problem of guiding an autonomous vehicle, see e.g. [21, 22, 23, 24]. The objective of the latter work is to guide one autonomous vehicle in a relatively unstructured environment so there is much emphasis on recognition, learning, and planning moves against diverse 'threats' or 'obstacles'. By contrast,, our concern is on guiding many cooperating vehicles in a relative structured environment, so the emphasis is on communicating information and coordinating plans and movement.

The proposed three layer control hierarchy and the platoon layer design have important implications for the design of the rest of the system. One implication for the regulation layer is that, five types of feedback control law need to be designed. Another implication of our study is the specification of several interfaces: between the platoon and link layers, between the platoon and regulation layers, and between the platoon layer and several sensors on board the vehicle and roadside sensors. The third set of implications concerns requirements for these sensors and monitors: the design indicates the kind of information needed to carry out the platoon layer control functions.

The fourth set of implications concerns the communications capability needed to support the information links. Communications must be established between neighboring vehicles, and between vehicles and the roadside. By figuring out how frequently the protocols must be executed, and the data exchanged during each protocol, one can estimate the data traffic

[16] A design principle that has been followed is to keep control tasks as decentralized as possible in the belief that, the resulting IVHS system would be less vulnerable to failures than one in which tasks are carried out more centrally. A major consequence of this principle is that much of the 'intelligence' resides in the vehicles themselves.

[17] The design process should be of interest also to researchers in the field of control of discrete event. systems [16,17,18]. Modularity and hierarchy are also discussed in [19,20].

that each communication link must support. This will indicate the kind of communication technology that would he viable. A related implication concerns work on standards for IVHS communication message structures.

We now consider some directions for future work. We group our comments under several headings.

Error conditions. The proposed design assumes that the communication system functions perfectly, and the synchronization mechanisms that achieve coordination of processes in the negotiating vehicles function perfectly as well. In any real implementation errors or exceptions will arise, and protocols must be able to handle them. It is likely that the design of these more robust protocols must be based on more detailed assumptions about the underlying communication and computing systems. Related concerns are the measures that need to be designed to counter failures in the communication, control and computing systems themselves.

System failures. The three maneuvers considered here are sufficient for 'normal' operating conditions. Clearly they need to be augmented by other maneuvers that would be invoked when failures of various types occur and are detected. What should be done if a vehicle has a failure, or if an obstacle is detected, or if there is an accident? Serious studies are needed that systematically describe the failure modes, classify them in terms of severity, and propose modifications in the system design to mitigate their impact. The above-mentioned AI-based work may prove valuable in this connection, see also [25].

Real-time behavior. The behavior of the platoon layer is described in terms of sequences of events. These events occur in time only insofar as we can say that one event occurs before another one. There is no notion of 'real' time. i.e. the amount of time between two events. It is important to augment the design to include some real time aspects. For example, in deciding whether to execute a maneuver, the supervisor may use an estimate (if one were available) of the time it would take to complete the maneuver. Such an estimate could be made available in the form of a table or function that summarizes the response of the regulation layer to a particular command. (The table could be built up from simulation experiments or from very simple models of vehicle dynamics.) Recent work on timed finite state systems provide a valuable guide as to how such estimates can be used [26, 27]. Discrete event system formalisms more powerful than finite state systems may also prove valuable [17]; unfortunately, these formalisms do not yet have the kind of software support provided by C'OSPAN.

Performance. Let us suppose that highway automation of the kind described here is technically feasible. Policy makers will then need to decide whether this technology is worth developing. An important element that can inform this decision would be reliable estimates of the increase in highway capacity and decrease in travel time that this technology might bring. Studies such as [1] cannot be relied upon since they are based on unverified assumptions about the automated highway. Reliable estimates, it seems, will have to be based on 'realistic' simulations.

References

[1] U. Karaaslan, P. Varaiya., and J. Walrand, "Two proposals to improve traffic flow," tech. rep., UCB-ITS-PRR-90-6, Institute of Transporation Studies, University of California, Berkeley, CA 94720, December 1990.

[2] P. Varaiya. and S. Shladover, "Sketch of an IVHS systems architecture," tech. rep., UC'B-ITS-PRR-91-3, Institute of Transportation Studies, University of California, Berkeley, CA 94720, 1991.

[3] S. Slreikholeslam and C. A. Desoer, "Longitudinal control of a platoon of vehicles," in *Proceedings of the 1990 American Control Conference, Volume 1*, (San Diego, CA), pp. 291--296, March 1990.

[4] D. McMahon, J. Hedrick. and S. Shladover, "Vehicle modeling and control for automated highway systems," in *Proceedings of 1990 American Control Conference*, (San Diego, CA), pp. 297-303, 1990.

[5] H. Peng and M. Tomizuka., "Lateral control of front-wheel-steering rubber-tire vehicles," tech. rep., IJCB-ITS-PRR-90-5, Institute of Transportation Studies, University of California., Berkeley, CA 94720, 1990.

[6] K. Chang, W. Li, A. Shaikhbahai, and P. Varaiys, "A preliminary implementation for vehicle platoon control system," in Proceedings *of the 1991 American Control Conference*, (Boston, MA), June 26-28 1991. to appear.

[7] M. Aoki and H. Fujii, "An inter-vehicle communication technology and its applications." in *22nd International Symposium on Automotive Technology and Automation, Vol 1*, (Xutomotive Automation Ltd., Croyden, England), pp. 127–134, 1990.

[8] Z. Har'El and R. P. Kurshan, "Software for analytical development of communications protocols," *AT&T Technical Journal*, pp. 45559, January/February 1990.

[9] Z. Har'El and R. P. Kurshan, *COSPAN User's Guide*. AT&T Bell Laboratories, Murray Hill, NJ, 1987.

[10] J. R. Buchi, "On a decision method in restricted second order arithmetic," in *Proceedings of the International Congress on Logic, Methodology and Philosophy of Science*. pp. 1-11, Stanford University Press, 1962.

[11] R. Kurshan, "Complementing deterministic Buchi automata in polynomial time," *Journal of Computer and System Sciences*, vol. 35, pp. 59–71, 1987.

[12] R. Kurshan, "Analysis of discrete event coordination." Preprint, AT&T Bell Laboratories, 1990.

[13] S. Sheikholeslam and C. A. Desoer, "Longitudinal control of a platoon of vehicles," tech. rep., UCB-ITS-PRR-89-3,6, Institute of Transportation Studies, University of California, Berkeley, CA 94720, 1989.

[14] H. Peng and M. Tomizuka, "Preview control for vehicle lateral guidance in highway automation," in *Proceedings of the 1991 American Control Conference*, (Boston, MA), June 26-28 1991. to appear.

[15] A. Gollu and P. Varaiya, "Hybrid dynamical systems," in *Proceedings of the 28th Conference on Decision and Control*, (Tampa, FL), pp. 2708–2712, December 1989.

[16] P. Varaiya and A. Kurzhanski, eds., *Discrete Event Systems: Models und Applications*, vol. IIASA 103. Lecture Notes in Control and Information Sciences. Springer, 1988.

[17] Y. Ho, cd.. *Proceedings of the IEEE: Special Issue on Dynamics of Discrete Event Systems*, vol. 77. 1989.

[18] C. Cassandras and P. Ramadge, eds., *IEEE Control Systems Magazine: Special Section on Discrete Event Systems*, vol. 10. March 1990.

[19] Ii. Inan and P. Varaiya, "Finitely recursive process models for discrete event systems," *IEEE Transactions on Automatic Control*, vol. AC-33, pp. 626-639, July 1988.

[20] C. M. Ozveren, *Analyis and Control of Discrete Event Dynamic Systems: A State Space Approach*. PhD thesis. MIT, 1989.

[21] C. Thorpe, M. Hebert, T. Kanade, and S. Shafer, "Vision and navigation for the Carnegie-Mellon Navlab." *IEEE Transactions on Pattern Analysis and Machine Intelligence*. vol. 10. pp. 362–373, May 1988.

[22] A. M. Waxman, J. LeMoigne, L. Davis. B. Srinivasan, T. Kushner, E. Liung, and T. Siddalingaiah, "A visual navigatiou system for autonomous land vehicles," *IEEE Journal of Robotics and Automation*, vol. 3, pp. 124140, April 1987.

[23] M. Markarinec, *An accident avoidance system for an autonomous highway vehicle*. PhD thesis, Northwestern University, 1989.

[24] E. Freund, Ch. Buhler, U. Judaschke, B. Lammen, and R. Mayr, "A hierarchically structured system for automated vehicle guidance." in *22nd International Symposium on Automotive Technology and Automation, Vol 1*, (Automotive Automation Ltd., Croyden, England), pp. 351–359, 1990.

[25] S. Narain, "A new modeling technique based on the causality relation." Preprint, Rand Corporation, 1989.

[26] D. L. Dill, "Timing assumptions and verification of finte-state concurrent systems," in. *Automatic Verification Methods for Finite State Systems*, vol. 407, Lecture Notes in Computer Sciences, Springer, 1989.

[27] R. Alur, C. Courcoubetis, and D. Dill, "Model-checking for real-time systems," in *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, (Philadelphia, PA), pp. 414–425, June 1990.

Appendix

The numbered lines in the COSPAN code correspond to the following lines that explain the syntax for describing a process. All subsequent processes are to be interpreted similarly.

1> Process name declaration.
2> Names of selection/output variables.
3> Names of state variables.
4> Initial state of the process.
5> Conditions for transition between states follow this.
6> While in the state NO-CAR, the selection/output seen by other processes is either ***no_car_ahead*** or ***car-ahead,*** non-deterministically chosen.
7> The transition from NO-CAR to CAR will occur if the selection, while in the state ***NO-CAR,*** is ***car-ahead.***
8> Otherwise, no transition is enabled, i.e. process stays in the same state.
9> While in the state CAR, the selection/output seen by other processes is either ***no-car-ahead*** or ***car-ahead,*** non-deterministically chosen.
10> The transition from CAR to NO-CAR will occur if the selection, while in the state ***CAR,*** is *no_car_ahead.*
11> Otherwise, no transition is enabled, i.e. process stays in the same state.
12> End of process declaration.
13> Selections from BPCmerge can affect this process' transitions.
14> ***MERGING*** is a *pausing* state, and this is allowed.
15> Upon entering this state, the process selects ***accelerating.*** After some non-deterministic amount of time (pause), the process selects ***merge-complete.***

```
/* merge.sr -- PROTOCOL FOR MERGE MANEUVERS */

1>proc BR               /* B's range sensor machine. It alternates between
                        "car_ahead" and "no_car_ahead" */

2>      selvar  #:(car_ahead, no_car_ahead)
3>      stvar   $:(CAR, NO_CAR)
4>      init    NO_CAR
5>      trans

6>      NO_CAR                  {no_car_ahead, car_ahead}
7>              ->CAR           : # = car_ahead
8>              ->$             : else;

9>      CAR                     {car_ahead, no_car_ahead}
10>             ->NO_CAR        : # = no_car_ahead
11>             ->$             : else;

12>     end     /* BR */

proc BV         /* B's velocity response machine. Upon receiving
                   "accelerate_to_merge" from the BPCmerge machine, it
                   initially selects "accelerating"; after some time
                   it selects "merge_complete" */

13>     import  BPCmerge
        selvar  #:(cruise, accelerating, merge_complete)
        stvar   $:(CRUISING, MERGING)
        init    CRUISING
14>     cyset   {MERGING@}
        trans

        CRUISING                {cruise}
                ->MERGING       : BPCmerge.# = accelerate_to_merge
                ->$             : else;

15>     MERGING                 {accelerating: merge_complete}
                ->CRUISING      : # = merge_complete
                ->$             : else·

        end     /* BV */

proc BQ         /* B's queue busy machine. It initially starts at the
                   NOT_BUSY state. When it receives "setBQ_busy"
                   from BPCmerge machine, it moves to the BUSY state */

        import  BPCmerge
        selvar  #:(busy, not_busy)
        stvar   $:(BUSY, NOT_BUSY)
        init    NOT_BUSY
        trans

        NOT_BUSY                {not_busy}
                ->BUSY          : BPCmerge.# = setBQ_busy
                ->$             : else;

        BUSY                    {busy}
                ->NOT_BUSY      : BPCmerge.# = setBQ_not_busy
                ->$             : else;

        end     /* BQ */

proc BPCmerge   /* B's protocol machine. This machine sends request to merge
                   and confirms merging when the maneuvre is complete */

        import  BR, BQ, BV, APRmerge
```

```
        selvar  I: (Idle, setBQ_busy, setBQ_not_busy, check_range,
                   request-merge, accelerate-to-merge, confirm_merge)
        stvar   S: (IDLE, CHECK-RANGE, REQUEST-MERGE, SET-BUSY. ACCELERATE,
                   CONFIRM-MERGE, UPDATE)
        init    IDLE
        trans

        IDLE                    (idle)
                ->CHECK RANGE   : true;

        CHECK-RANGE             (check_range)
                ->SET_BUSY      : ( i3R.f = car-ahead) .( BQ.# = not-busy )
                ->IDLE          : BR.# = no-car-a head
                ->$             : else;

        SET-BUSY                (setBQ_busy)
                ->REQUEST_MERGE : true;

        REQUEST-MERGE           (request_merge)
                ->ACCELERATE    : APRmerge.# = ack_request_merge
                ->UPDATE        : APRmerge.# = nack_request_merge
                ->$             : else;

        ACCELERATE              (accelerate_to_merge)
                ->CONFIRM_MERGE : BV.# = merge_complete
                ->$             : else;

        CONFIRM-MERGE           (confirm_merge)
                ->UPDATE        : true;

        UPDATE                  (setBQ_not-busy)
                ->IDLE          : true;

        end     /* BPCmerge */

proc AQ         /* A's queue busy machlne. It initially starts at tha
                   NOT-BUSY state. When it receives "setAQ_busy"
                   from APRmerge machine, it moves to the BUSY state ● /

        import APRmerge
        selvar I: (busy, not-busy)
        stvar   $: (BUSY, NOT-BUSY,
        init    NOT_BUSY
        trans

        NOT_BUSY                ( not-busy )
                ->BUSY          : APRmerge.# = setAQ_busy
                ->$             : else:

        BUSY                    (busy)
                ->NOT_BUSY      : APRmerge.# = setAQ_not_busy
                ->$             : else;

        end     /* AQ */

proc ASUM       /* A's size of platoon machlne. */

        selvar  I: (sum-too-large, sum_ok)
        stvar   S: (SIZE-LARGE, SIZE-OK)
        init    SIZE-LARGE
        trans

        SIZE-LARGE              ( sum-too-large, sum-ok )
                ->SIZE_OK       : # = sum_ok
                ->$             : else;

        SIZE-OK                 (sum_ok, sum_too_large)
                ->SIZE_LARGE    : # = sum_too_large
                ->$             : else;

        end     /* ASUM */

proc APRmerge   /* A's protocol machine. This machine is the leader of
                   the platoon which receives request from BPCmerge ● /

        Import  AQ, ASUM, BPCmerge
        selvar I: (Idle, check-status, nack_request_merge, setAQ_busy,
                   setAQ_not_busy, ack_request_merge)
        stvar   $: (IDLE, CHECK-STATUS, SET-BUSY, SEND-ACK, SEND-NACK,
                   UPDATE1
        init    I D L E
        trans

        IDLE                    (idle)
                ->CHECK_STATUS  : BPCmerge.# = request_merge
                ->$             : else;

        CHECK-STATDS            (check_status)
                ->SET_BUSY      : (AQ.# = not_busy) * (ASUM.# = sum-ok)
                ->SEND_NACK     : else;

        SEND_NACK               (nack_request_merge)
                ->IDLE          : true;

        SET-BUSY                ( setAQ_busy t
                ->SEND_ACK      : true:

        SEND-ACK                (ack_request_merge)
                ->UPDATE        : BPCmerge. # =conf irm_merge
                ->$             : else;

        UPDATE                  (setAQ_not_busy)
                ->IDLE          : true;

        end     /* APRmerge */

monitor MERGE_MONITOR   /* Checks for correctness of the Merge Protocol ● /

        import  BPCmerge, APRmerge
        stvar   $: 10. .3)
        cyset   (0)
        recur   2->0, 3->0
        init    0
        trans

        0
                ->1             : BPCmerge.# = request-merge
                ->$             : else;
        1
                ->2             : APRmerge.# = ack_request_merge
                ->3             : APRmerge.# = nack_request_merge
                ->$             : else;
        2
                ->0             : BPCmerge. # = confirm_merge
                ->$             : else;
        3
                ->0             : BPCmerge . # = setBQ_not_busy
                ->$             : else;

        end     /* MERGE-MONITOR */
```

```
/* protocol.sr -- contains reduced command protocol machines and full
                  respond machines as well as all relevant sublayer
                  interface machines */

proc TRIGGER     /* triggers the test machines */

        selvar  #:(1)
        stvar   $:(ONE)
        init    ONE

        trans

        ONE                     (1)
                ->$             : true:

        end     /* TRIGGER */

/*---------------interface machines for PM layer---------------------*/

proc RDSNR       /* Road Sensor */

        selvar  #:(match, no-match)
        stvar   S: (MATCH, NO_MATCH)
        init    NO_MATCH
        trans

        NO-MATCH                (match, no_match)
                ->MATCH         : # = match
                ->$             : else;

        HATCH                   (match, no-match)
                ->NO_MATCH      : # = no-match
                ->$             : else;

        end     /* [illegible] */

proc LN          /* [illegible] */

        selvar  #: (correct, incorrect )
        stvar   $:(CORRECT, INCORRECT1
        init    INCORRECT
        trans

        INCORRECT               (correct, incorrect I
                ->CORRECT       : # = correct
                ->$             : else;

        CORRECT                 (correct, incorrect)
                ->INCORRECT     : # = incorrect
                ->$             : else;

        end     /* LN */

/*-----------------protocol machine for PM layer---------------------*/

proc PM          /* Path Monitor */

        import RDSNR, LN, SUPR
        selvar  #:(chg_ln, keep_ln)
        stvar   $:(CHG_LN, KEEP_LN)
        init    KEEP LN
        trans

        CHG_LN                  (chg_ln)
                ->KEEP_LN       : (LN.# = correct) . (SUPR.# = chg_ln_complete)
                ->$             : else;

        KEEP_LN                 I keep_l n)
                ->CHG_LN        : RDSNR.# = match
                ->$             : else;

        end     /* PM */

/*--------------interface machines for SUPR layer---------------*/

proc FREE        /* free agent status Indicator. */

        selvar #:(not_free_agent, free-agent)
        stvar   $:(FREE_AGENT, NOT-FREE-AGENT)
        init    NOT-FREE-AGENT
        trans

        NOT-FREE-AGENT          (not_free_agent, free_agent)
                ->FREE_AGENT    : # = free_agent
                ->$             : else;

        FREE AGENT              (free_agent, not_free_agent)
                ->NOT FREE_AGENT       : # = not_free_agent
                ->$             : else;

        end     /* FREE */

proc POS         /* Indicates position in platoon */

        import BPCmerge, BPCsplit, BPRsplit, BiPRchg
        selvar  #:(follower, leader)
        stvar   $:(FOLLOWER, LEADER)
        cyset   (FOLLOWER@)
        init    FOLLOWER
        trans

        FOLLOWER                (follower)
                ->LEADER        : (BPCsplit.# = setALLBUSY_not_busy)
                                  + (BPRsplit.# = setALLBUSY_not_busy)
                                  + (BiPRchg.# = setALLBUSY_not_busy)
                ->$             : else;

        LEADER                  (leader)
                ->FOLLOWER      : BPCmerge.# = setALLBUSY_not_busy
                ->$             : else;

        end     /* POS */

proc ALLBUSY        /* over all busy flag */

        import  APCchg, APCsplit, BPCsplit, BPCmerge, BPRchg, BiPRchg, CPRchg,
                APRsplit, BPRsplit, APRmerge
        selvar  #:(busy, not-busy)
        stvar   $:(BUSY, NOT-BUSY,
        init    NOT-BUSY
        trans

        NOT-BUSY                (not_busy)
                ->BUSY          : (APCchg.# = setALLBUSY_busy)
                                  +(BPRchg.# = setALLBUSY_busy)
                                  +(BiPRchg.# = setALLBUSY busy)
                                  +(CPRchg.# = setALLBUSY_busy)
                                  +(APCsplit.# = setALLBUSY_busy)
                                  +(BPCsplit.# = setALLBUSY_busy)
                                  +(APRsplit.# = setALLBUSY_busy)
                                  +(BPRsplit.# = setALLBUSY_busy)
                                  +(APRmerge.# = setALLBUSY_busy)
```

```
                          + (BPCmerge.# = setALLBUSY_busy)
          ->$           : else;

     BUSY                 {busy}
          ->NOT_BUSY    : (APCchg.# = setALLBUSY_not_busy)
                          +(BPRchg.# = setALLBUSY_not_busy)
                          + (BiPRchg.# = setALLBUSY_not_busy)
                          +(CPRchg.# = setALLBUSY_not_busy)
                          +(APCsplit.# = setALLBUSY_not_busy)
                          +(BPCsplit.# = setALLBUSY_not_busy)
                          +(APRsplit.# = setALLBUSY_not_busy)
                          +(BPRsplit.# = setALLBUSY_not_busy)
                          +(APRmerge.# = setALLBUSY_not_busy)
                          +(BPCmerge.# = setALLBUSY_not_busy)
          ->$           : else;

     end      /* ALLBUSY */

proc SIZE     /* Indicates platoon size(will always come back to opt) */

     import   BPCmerge, APCsplit, APRmerge, APRsplit, BPRsplit
     selvar   #:(opt, smaller-than-opt, larger-than-opt)
     stvar    $:(OPT, SMALLER-THAN-OPT, LARGER-THAN-OPT)
     cyset    {OPT@}
     init     OPT@
     trans

     OPT                  (Opt: smaller-than-opt, larger_than_opt}
          ->SMALLER_THAN_OPT      : # = smaller-than-opt
          ->LARGER_THAN_OPT       : # = larger_than_opt
          ->$           : else;

     SMALLER-THAN-OPT     (smaller-than-opt)
          ->OPT         : (BPCmerge.# = setALLBUSY_not_busy)
                          + (APRmerge.# = setALLBUSY_not_busy)
          ->$           : else;

     LARGER_THAN_OPT      {larger_than_opt}
          --OPT         : (APCsplit.# = setALLBUSY_not_busy)
                          + (APRsplit.# = setALLBUSY_not_busy)
                          + (BPRsplit.| = setALLBUSY_not_busy)
          ->$           : else;

     end      /*SIZE ● /

/*------------protocol machine for SUPR layer-------------------------*/

proc SUPR     /* Platoon supervisor, each car has one ● /

     import   PM, FREE, ALLBUSY, APCChQ, BFRE, POS, SIZE, BPRchg,BiPRchg,
              CPRchg, APRsplit, BPRsplit, APRmerge
     selvar   #:(start, check_free_agent, become_free_agent, chg_ln_complete,
                 start_chg_ln, check_position, keep_position, check_size,
                 Split, mergE)
     stvar    $: (START, CHK_FREE_AGNT, BCM_FREE_AGNT, CHG_CMPLT, STRT_CHG_LN,
                 CHK_POS, KEEP-POS, CHK_SIZE, SPLIT, MERGe)
     init     START
     trans

     START                I start I
          ->CHK_FREE_AGNT : PM.# = chg_ln
          ->CHK_POS       : PM.#= keep-ln
          ->$             : else;

     CHK_FREE_ACNT        {check_free_agent}
          ->BCM FREE-AGNT : (FREE.#=not_free_agent)
```

```
                          • (ALLBUSY. = not-busy)
          ->STRT_CHG_LN : (FREE.# = free_agent)
                          • (ALLBUSY.#= not-busy)
          ->$           : else;

     BCM_FREE_AGNT        {become_free_agent}
          ->STRT_CHG_LN : BFRE.# = free_agent
          ->$           : else;

     STRT_CHG_LN          {start_chg_ln}
          ->CHG_CMPLT   : APCchg.# = setALLBUSY_not_busy
          ->$           : else;

     CHG CMPLT            {chg_ln_complete}
          ->START       : true;

     CHK POS              {check_position}
          ->KEEP POS    : POS.# = follower
          ->CHK_SIZE    : POS.# = leader
          ->$           : else;

     KEEP POS             (keep-position)
          ->BCM_FREE_AGNT : (PM.# = chg_ln) • (ALLBUSY.#= not-busy)
                          * (BiPRchg.#= idle) • (BPRsplit.# = idle)
          ->CHK_SIZE    : (POS.# = leader) • (PM.# = keep_ln)
          ->$           : else;

     CHK-SIZE             {check_size}
          ->BCM_FREE_AGNT : (PM.# = chg_ln) • (ALLBUSY.#= not-busy)
                          • (BPRchQ.6 =idle) • (CPRchg.# = Idle)
                          • (APRsplit.# = idle) • (APRmerge.# = idle)
          ->SPLIT       : (SIZE.# = larger_than_opt)
                          • (ALLBUSY.#= not-busy) • (PM.#= keep-ln)
                          • (BPRchg.#= idle) • (CPRchg.#= idle)
                          • (APRsplit.# = idle) • (APRmerge.# = idle)
          ->MERGe       : (SIZE.# = smaller_than_opt) * (POS.# = leader)
                          • (ALLBUSY.#= not-busy) • (PM.# = keep_ln)
                          • (BPRchg.#= idle) • (CPRchg.#= idle)
                          • (APRsplit.# = idle) • (APRmerge.# = idle)
          ->$           : else:

     SPLIT                {split}
          ->CHK_SIZE    : (SIZE.# = opt) + (SIZE.4 = smaller-than-opt)
          ->$           : else;

     MERGe                {mergE}
          ->KEEP_POS    : (PM.# = keep ln) • (POS.# = follower1
          ->BCM_FREE_AGNT :(PM.#= chg ln) • (ALLBUSY.#= not-busy)
          ->$           : else;

     end      /*SUPR ● f

/* --------------protocol machines for change lanes maneuver----------- ● /

proc APCChQ   /* A's change lane protocol machine--reduced ● /

     import   SUPR, BPRchgl
     selvar   #:(idle, setALLBUSY_busy, confirm_change_lane,
                 setALLBUSY_not_busy)
     stvar    $: (IDLE, SET-BUSY, CONFM_CHG_LN, UNSET-BUSY)
     init     IDLE
     trans

     IDLE                 {idle}
          ->SET_BUSY    : SUPR.# = start_chg_ln
          ->$           : else;
```

```
        SET BUSY             {setALLBUSY_busy}
                ->CONFM_CHG_LN  : BPRchg1.# = ack_req_chg_ln
                ->UNSET_BUSY    : else;

        CONFM_CHG_LN         {confirm_change_lane}
                ->UNSET_BUSY    : true:

        UNSET_BUSY           {setALLBUSY_not_busy}
                ->IDLE          : true;

        end     /* APCchg * /

/*-----------------*/
proc BPRchg1    /* test machine for APCchg */

        selvar  #:(ack_req_chg_ln, nack_req_chg_ln)
        stvar   $:(ACK, NACK)
        init    ACK
        trans

        ACK                  lack-req-chg-ln, nack_req_chg_ln}
                ->NACK          : # = nack_req_chg_ln
                ->$             : else;

        NACK                 {ack_req_chg_ln, nack_req_chg_ln}
                ->ACK           : # = ack_req_chg_ln
                ->$             : else;

        end     /* BPCchg1 */
/*-----------------*/

proc BPRchg     /* leader's response machine--full * /

        import APCchg2, PM, SUPR, POS, ALLBUSY, APCsplit, BPCmerge,
               CPRchg, APRsplit, APRmerge
        selvar #:(idle, check-busy, nack_request_chg_ln, setALLBUSY_busy,
                  ack_req_chg_ln, setALLBUSY_not_busy)
        stvar   S:'IDLE, CHECK-BUSY, NACK, SET-BUSY. ACK, UNSET_BUSY)
        init    IDLE
        trans

        IDLE                 'idle)
                ->CHECK_BUSY    : APCchg2.# = req_chg_ln
                ->$             : else;

        CHECK-BUSY           {check_busy}
                ->SET_BUSY      : (ALLBUSY.# = not-busy) * (SUPR.# = check-size)
                                  * (POS.# = leader) * 'PM.+ = keep_ln)
                                  * ~(CPRchg.# = promise_not_to_move)
                                  * ~(APRsplit.# = setALLBUSY_busy)
                                  * ~(APRmerge.# = setALLBUSY_busy)
                ->NACK          : else;

        NACK                 {nack_req_chg_ln}
                ->IDLE          : true;

        SET-BUSY             {setALLBUSY_busy}
                ->ACK           : true;

        ACK                  'ack_req_chg_ln}
                ->UNSET_BUSY    : APCchg2.# = confirm_change_lane
                ->$             : else;

        UNSET_BUSY           {setALLBUSY_not_busy}
                ->IDLE          : true:
```



```
        end     /* BPRchg */

/* -------------------- */
proc APCchg2    /* test machine for BPRchg * /

        import BPRchg, TRIGGER
        selvar  I: 'Idle, req_chg_ln, confirm-change-lane)
        stvar   S:'IDLE, REQ_CHG_LN, CONFM_CHG_LN)
        init    IDLE
        trans

        IDLE                 {idle}
                ->REQ_CHG_LN    : TRIGGER.+ = 1
                ->$             : else;

        REQ_CHG_LN           {req_chg_ln}
                ->CONFM_CHG_LN  : BPRchg.#ck_req_chg_ln
                ->IDLE          : else;

        CONFM_CHG_LN         {confirm_change_lane}
                ->IDLE          : true:

        end     /* APCchg2 */
/* ------------------- */

proc CPRchg     /* far lane leader's machine--full */

        import PM, SUPR, ALLBUSY, POS, APCchg3, BPRchg, APRsplit, APRmerge
        selvar #:(idle, check-busy, promise_not_to_move, nack_req_chg_ln)
        stvar  $:(IDLE, CHECK-BUSY, PROMISE, NACK)
        init   IDLE
        trans

        IDLE                 {idle}
                ->CHECK_BUSY    : APCchg3.# = req_coop
                ->$             : else;

        CHECK-BUSY           {check_busy}
                ->PROMISE       : (PM.# = keep_ln) * (ALLBUSY.# = not-busy)
                                  * (POS.# = leader) * (SUPR.# = check-sir.)
                                  * (BPRchg.# = idle)
                                  * ~(APRsplit.# = setALLBUSY_busy)
                                  * ~(APRmerge.# = setALLBUSY_busy)
                ->NACK          : else;

        NACK                 {nack_req_chg_ln}
                ->IDLE          : true;

        PROMISE              {promise_not_to_move}
                ->IDLE          : APCchg3.# = confirm-change-lane
                ->$             : else;

        end     /* CPRchg * /

/* ------------------ */
proc APCchg3    /* test machine for CPRchg */

        import CPRchg, TRIGGER
        selvar  #:(idle, req_coop, confirm_change_lane)
        stvar   S:'IDLE, REQ_COOP,CONFM_CHG_LN)
        init    IDLE
        trans

        IDLE                 'Idle)
                ->REQ_COOP      : TRIGGER.# = 1
```

```
            ->$               : else;

        REQ_COOP              {req_COOP)
            ->CONFM_CHG_LN    : CPRchg.# = promise_not_to_move
            ->IDLE            : else;

        CONFM_CHG_LN          {confirm_change_lane}
            ->IDLE            : true;

        end     /* APCchg3 */
/* ------------------ */

proc BiPRchg   /* follower's machine in change lanes--left out the interface
                  to BiV ● /

        Import  BPRchg2, PM, SUPR, ALLBUSY, POS
        selvar  #:(idle, setALLBUSY_busy, confirm_split, setALLBUSY_not_busy)
        stvar   $:(IDLE, SET-BUSY, CONFIRM-SPLIT, UNSET-BUSY)
        init    IDLE
        trans

        IDLE                  {idle}
            ->SET BUSY        : (BPRchg2.1 = split) • (POS.# = follower)
                                • (ALLBUSY.# = not_busy)
                                • ((SUPR.# = keep_position) • (PM.# = keep-l")
                                + (SUPR.# = become-free-agent)
                                • (PM.# = chg_ln))
            ->$               : else;

        SET-BUSY              {setALLBUSY_busy}
            ->CONFIRM SPLIT   : true:

        CONFIRM SPLIT         {confirm_split}
            ->UNSET BUSY      : true;

        UNSET-BUSY            {setALLBUSY_not_busy}
            ->IDLE            : true:

        end     /* BiPRchg */

/* ----------------- */
proc BPRchg2   /* test machine for BiPRchg ● /

        import BiPRchg, TRIGGER
        selvar #:(idle, split)
        stvar   $:(IDLE, SPLIT)
        init    IDLE
        trans

        IDLE                  (Idle)
            ->SPLIT           : TRIGGER.# = 1
            ->$               : else;

        SPLIT                 {split}
            ->IDLE            : BiPRchg.# = confirm_split
            ->$               : else;

        end     /* BPRchg2 ● /
/* ----------------- */

/* ------------protocol machine for free agent supervisor------------ ● /

proc BFRE      /* Become a free agent */

        Import  SUPR, POS, APCsplit, BPCsplit
        selvar  #:(idle, check-status, become-leader, become-free-agent,
                   free-agent)
        stvar   $:(IDLE, CHECK-STATUS, BECOME-LEADER, BECOME_FREE_AGENT,
                   FREE-AGENT)
        init    IDLE
        trans

        IDLE                  {idle}
            ->CHECK_STATUS    : SUPR.# = become-free-agent
            ->$               : else;

        CHECK-STATUS          (check-status)
            ->BECOME_LEADER   : POS.# = follower
            ->BECOME_FREE_AGENT   : else;

        BECOME-LEADER         {become_leader}
            ->BECOME_FREE_AGENT   : BPCsplit.# = confirm_split
            ->$               : else;

        BECOME-FREE-AGENT     {become_free_agent}
            ->FREE_AGENT      : APCsplit.# = setALLBUSY_not_busy
            ->$               : else;

        FREE-AGENT            {free_agent}
            ->IDLE            : true;

        end     /* BFRE ● /

/* ---------------protocol machines for split maneuvers--------------- ● /

proc APCsplit  /* sllghtly reduced--left out a state with transition 'true' */

        import PM, SUPR, BFRE, ALLBUSY, BPRchg, CPRchg, APRsplit, APRmerge,
               SIZE, POS
        selvar #:(idle, setALLBUSY_busy, update, setALLBUSY_not_busy)
        stvar   $:(IDLE, SET_BUSY, UPDATE, UNSET-BUSY)
        init    IDLE   .
        trans

        IDLE                  {idle}
            ->SET_BUSY        : ((PH.8 = keep-ln) • (SUPR.# = split)
                                • (SIZE.# = larger-than-opt)
                                + (PM.# = chg_ln)
                                * (SUPR.# = become-free-agent)
                                • (BFRE.# = become_free_agent))
                                • (ALLBUSY.# = not-busy) • (POS.# = leader)
                                • (BPRchg.# = idle) • (CPRchg.# = idle)
                                • (APRsplit.# = Idle) • (APRmerge.# = idle)
            ->$               : else;

        SET-BUSY              {setALLBUSY_busy}
            ->UPDATE          : true;

        UPDATE                (update)
            ->UNSET_BUSY      : true;

        UNSET-BUSY            {setALLBUSY_not_busy}
            ->IDLE            : true;

        end     /* APCsplit *I

proc BPCsplit  /* reduced--left out interface to BV and one 'true' transition
                  state ●

        import PM, POS, BFRE, ALLBUSY, APRsplit2, BiPRchg, BPRsplit
        selvar  #:(idle, setALLBUSY_busy, setALLBUSY_not_busy, request-split,
                   confirm_split)
```

```
        stvar   5: (IDLE, REQ SPLIT, SET BUSY, CONFIRM_SPLIT, UNSET BUSY)
        init    IDLE
        trans

        I D L E                 {idle}
                ->REQ_SPLIT     : (BFRE.# = become-leader) . (POS.#= follower)
                                  . (PM.# = chg_ln) . (ALLBUSY.# =  not-busy)
                                  . (BiPRchg.# = idle) . (BPRsplit.# = idle)
                ->$             : else;

        REQ_SPLIT               (request-split)
                ->SET BUSY      : (APRspllt2.1 = ack_req_split)
                                  * (PM.# = chg_ln) * (ALLBUSY.# = not-busy)
                                  . (BiPRchg.# = Idle) * (BPRsplit.# = idle)
                ->IDLE          : APRsplit2.1 =nack req split
                ->$             : else;

        SET-BUSY                {setALLBUSY_busy}
                ->CONFIRM_SPLIT :  true:

        CONFIRM_ SPLIT          {confirm_split}
                ->UNSET BUSY    : true;

        UNSET-BUSY              {setALLBUSY_not_busy}
                ->IDLE          : true:

        end     /* BPCsplit */

/*--------------------*/
proc APRsplit2 /* test machine for BPCsplit */

        selvar #:(ack_req_split, nack_req_split)
        stvar   $:(ACK, NACK)
        init    ACK
        trans

        ACK                     {ack_req_split, nack_req_split}
                ->NACK          : # = nack_req_split
                ->$             : else;

        NACK                    {ack_req_split, nack_req_split}
                ->ACK           : # = ack_req_split
                ->$             : else;

        end     /* APRsplit2 ● /
/*--------------------*/

proc APRsplit   /* leader response machine--full ● /

        import  BPCsplit2, SUPR, ALLBUSY, BPRchg, CPRchg, APCsplit, BPCmerge,
                POS, PM, APRmerge
        selvar #:(idle, check-busy, "ack-request-split, setALLBUSY_busy,
                     ack_request_split, update, setALLBUSY_not_busy)
        stvar   $:(IDLE, CHECK-BUSY, SEND-NACK, SET-BUSY, SEND-ACK,
                     UPDATE, UNSET-BUSY)
        init    IDLE
        trans

        IDLE                    (Idle)
                ->CHECK_BUSY    : BPCsplit2.# = request_split
                ->$             : else;

        CHECK-BUSY              (check-busy)
                ->SET BUSY      : (PM.# = keep_ln) * (ALLBUSY.# = not busy)
                                  .(POS.# = leader) * (SUPR.# = check_size)
                                  . (BPRchg.# = idle) . (CPRchg.# = idle)
```

```
                                  * ~(APRmerge.# = setALLBUSY_busy)
                ->SEND_NACK     : else;

        SET BUSY                {setALLBUSY_busy)
                ->SEND_ACK      : true:

        SEND_ACK                lack request _split}
                ->UPDATE        :  true;

        SEND_NACK               {nack_request_split}
                ->IDLE          : true:

        UPDATE                  (update)
                ->UNSET_BUSY    : BPCsplit2.1 = confirm split
                ->$             : else;

        UNSET-BUSY              {setALLBUSY not busy)
                ->IDLE          :  true;

        end     /* APRsplit ● /

/* ------------------- ● 
proc BPCsplit2  /* test machine for APRsplit ● /

        import  APRsplit, TRIGGER
        selvar  #:(idle, request-spilt, confirm_split)
        stvar   S:(IDLE, REQ_SPLIT, CONFIRM_SPLIT)
        init    IDLE
        trans

        IDLE                    (idle)
                ->REQ_SPLIT     : TRIGGER.@ = 1
                ->$             : else;

        REQ-SPLIT               {request_split}
                ->CONFIRM_SPLIT : APRsplit.# = ack_request_split
                ->IDLE          : APRsplit.# = nack_request_split
                ->$             : else;

        CONFIRM SPLIT           {confirm_split}
                ->IDLE          : true;

        end     /* BPCsplit2 */
/* ---------------------- ● 

proc BPRsplit   /* follower response to split--left out inface to BV */

        import  APCsplit2, PM, ALLBUSY, SUPR, POS, BiPRchg
        selvar  #:(idle, setALLBUSY_busy, confirm_split, setALLBUSY_not_busy)
        stvar   $:(IDLE, SET-BUSY, CONFIRM-SPLIT, UNSET-BUSY)
        init    IDLE
        trans

        IDLE                    {idle}
                ->SET_BUSY      : (APCspllt2.1 = invite_new_lead)
                                  . (POS.# = follower).
                                  . (ALLBUSY.# = not_ busy) . (BiPRchg.# = idle)
                                  .  ((SUPR.# = keep_position) . (PM.# = keep_ln)
                                  + (SUPR.# = become_free_agent)
                                  . (PM.4 = chg_ln))
                ->$             : else;

        SET_BUSY                {setALLBUSY_busy}
                ->CONFIRM_SPLIT : true;

        CONFIRM_SPLIT           {confirm_split}
```

```
        ->UNSET_ BUSY  :  true:

UNSET_BUSY              {setALLBUSY not busy)
        ->IDLE          : true:

end     /* BPRsplit ● /

/* -------------------- */
proc APCsplit2 /* test machine for BPRsplit */

        impart  BPRsplit, TRIGGER
        selvar  #:(idle, invite new lead)
        stvar   $:(IDLE, INVITE_NEW_LEAD)
        init    IDLE
        t rans

        IDLE                    (idle)
                ->INVITE_NEW-LEAD       : TRIGGER.# = 3
                ->$             : else;

        INVITE-NEW-LEAD         {invite_new_lead}
                ·>IDLE          : BPRsplit.# = confirm_split
                ->$             : else;

        end     /* APCsplit2 ● /
/* -------------------- */

proc BPCmerge   /* slightly reduced machine--left out Interface to BV
                   and a 'true' transition state ● /

        lmport  PM, SIZE, SUPR, POS, APRmerge2
        selvar  #:(idle, setALLBUSY_busy, confirm_merge, setALLBUSY not busy)
        stvar   S:(IDLE, SET-BUSY, CONFIRM-MERGE, UNSET-BUSY)
        init    IDLE
        trans

        IDLE                    {idle}
                ->SET_BUSY      : (SUPR.# = mergE) • (PM.# = keep In)
                                  • (S12.E.I = smaller than opt)
                                  • (POS.# = leader)
                ->$             : else;

        SET BUSY                {setALLBUSY_busy}
                ->CONFIRM_MERGE : APRmerge2.1  = ack_req_merge
                ->UNSET BUSY    : else;

        CONFIRM-MERGE           {[illegible]
                ->UNSET BUSY    : true:

        UNSET-BUSY              {setALLBUSY_not_busy}
                ->IDLE          : true;

        end     /* BPCmerge ● /

/*---------------- */
proc APRmerge2 /* test machlne for BPCmerge ● /

        selvar #:(ack_req_merge,  nack-req-merge)
        stvar   $:(ACK, NACK)
        init    ACK
        trans

        ACK                     lack-req-merge, nack-req-merge]
                 >NACK          ; # = nack-req-merge
                ->$             : else;

        NACK                    {ack_req_merge, nack-req-merge)
                ->ACK           : # = ack_req_merge
                ->$             : else:

        end     /* APRmerge2 ● ,
/*------------------*/

proc APRmerge   /* leader's response machine--full ● /

        import  BPCmerge2, PM, POS, SUPR, ALLBUSY, BPRchg, CPRchg, APRsplit,
                APCsplit, BPCmerge
        selvar  #:(idle, check-status, nack_request_merge, setALLBUSY_busy,
                   setALLBUSY_not_busy, ack_request_merge)
        stvar   $:(IDLE, CHECK-STATUS, SET-BUSY, SEND-ACK, SEND_NACK,
                   UNSET-BUSY)
        init    IDLE
        trans

        IDLE                    {idle}
                ->CHECK_STATUS  : BPCmerge2.1 = request-merge
                ->$             : else;

        CHECK-STATUS            (check-status)
                ->SET_BUSY      : (PM.# = keep_ln) • (POS.# = leader)
                                  • (ALLBUSY.# = not-busy)
                                  • (SUPR.# = check-size) • (BPRchg.# = idle)
                                  • (CPRchg.# = idle) • (APRsplit.# = Idle)
                                  • (SIZE.# = smaller-than-opt)
                ->SEND NACK     : else:

        SEND-NACK               { nack_request_merge}
                ->IDLE          : true;

        SET-BUSY                {setALLBUSY_busy}
                ->SEND_ACK      : true:

        SEND ACK                {ack_request_merge}
                ->UNSET_BUSY    : BPCmerge2.1 = confirm_merge
                ->$             : else;

        UNSET BUSY              {setALLBUSY_not_busy}
                ->IDLE          : true;

        end     /* APRmerge ● /

/* ------------------- */
proc BPCmerge2  /* test machine for APRmerge ● /

        Import  APRmerge, TRIGGER
        selvar  #:(idle, request-merge, confirm_merge)
        stvar   $:(IDLE, REP-MERGE, CONFIRM_MERGE)
        init    IDLE
        trans

        IDLE                    (idle)
                ->REQ_MERGE     : TRIGGER.# = 1
                ->$             : else;

        REQ_MERGE               {request_merge}
                ->CONFIRM_MERGE : APRmerge.# = ack request merge
                ->IDLE          : APRmerge.# = nack_request_merge
                ->$             : else;

        CONFIRM_MERGE           (confirm-merge)
                ->IDLE          : true:
```

```
        end     /* BPCmerge2 */
/* --------------------
/*-------------------------------------------------------------------*/
monitor SUPR_MONITOR1   /* monitors states in SUPR */

        import  SUPR
        stvar   s: (0..7)
        init    0
        cyset   {1}, {2}, {3}, {4}, {5}, {6}, {7}
        recur   1->7, 1->4, 4->2, 6->0, 3->5, 1->5, 2->5, 4->5
        trans

        0
                ->1             : SUPR.# = check size
                ->2             : SUPR.# = keep-position
                ->3             : SUPR.# = check-free-agent
                ->$             : else;
        1
                ->4             : SUPR.# = mergE
                ->5             : SUPR.# = become-free-agent
                ->7             : SUPR.# = split
                ->$             : else;
        2
                ->1               SUPR.# = check-size
                ->5               SUPR.# = become-free-agent
                ->7               SUPR.# = split
                ->$               else;
        3
                ->5             : SUPR.# = become-free-agent
                ->6             : SUPR.# = start-chg-ln
                ->$             : else;
        4
                ->2             : SUPR.# = keep-position
                ->5             : SUPR.# = become-free-agent
                ->$             : else;
        5
                ->6             : SUPR.# = start-chg-ln
                ->$               else;
        6
                ->0               SUPR.# = chg_ln_complete
                ->$               else;
        7
                ->1             : SUPR.# = check-size
                ->$               else:

        end     /* SUPR_MONITOR1 * /
/*------------------------------------------------------*/

monitor SUPR_MONITOR2   /* monitors mutual exclusion * /

        import  APCchg,BPRchg,CPRchg,BiPRchg,APCsplit,BPCsplit,APRsplit,
                BPRsplit,BPCmerge,APRmerge
        stvar   $:(0..12)
        init    0
        cyset   {0}
        trans

        0
                ->1             : (APCchg.$ = SET-BUSY)
                                  * ((BPRchg.$ = SET-BUSY)
                                  + (CPRchg.$ = PROMISE)
                                  + (BiPRchg.$ = SET-BUSY)
                                  + (APCsplit.$ = SET-BUSY)
                                  + (BPCsplit.$ = SET-BUSY)
                                  + (APRsplit.$ = SET_BUSY)
                                  + (BPRsplit.$ = SET-BUSY)
                                  + (BPCmerge.$ = SET-BUSY)
                                  + (APRmerge.$ = SET BUSY))
                ->2             : (BPRchg.$ = SET-BUS?,
                                  • (JAPCcha.$ = SET_BUSY)
                                  + (CPRchg.$ = PROMISE)
                                  + (BiPRchg.$ = SET-BUSY)
                                  + (APCsplit.$ = SET BUSY)
                                  + (BPCsplit.$ = SET_BUSY)
                                  + (APRsplit.$ = SET_BUSY)
                                  + (BPRsplit.$ = SET_BUSY)
                                  + (BPCmerge.$ = SET-BUSY)
                                  + (APRmerge.$ = SET_BUSY))
                ->3             : (CPRchg.$ = PROMISE)
                                  • (JAPCcha.$ = SET_BUSY)
                                  + (BPRchg.$ = SET_BUSY)
                                  + (BiPRchg.$ = SET-BUSY)
                                  + (APCsplit.$ = SET_BUSY)
                                  + (BPCsplit.$ = SET-BUSY)
                                  + (APRsplit.$ = SET_BUSY)
                                  + (BPRsplit.$ = SET_BUSY)
                                  + (BPCmerge.$ = SET-BUSY)
                                  + (APRmerge.$ = SET-BUSY))
                ->4             : (BiPRchg.$ = SET-BUSY,
                                  • ((APCchg.$ = SET-BUSY)
                                  + (BPRchg.$ = SET-BUSY)
                                  + (CPRchg.$ = PROMISE)
                                  + (APCsplit.$ = SET-BUSY)
                                  + (BPCsplit.$ = SET_BUSY)
                                  + (APRsplit.$ = SET_BUSY)
                                  + (BPRsplit.$ = SET-BUSY,
                                  + (BPCmerge.$ = SET-BUSY)
                                  • (APRmerge.$ = SET-BUSY))
                ->5             : (APCsplit.$ = SET-BUSY)
                                  • ((APCchg.S = SET-BUSY)
                                  + (BPRchg.$ = SET_BUSY)
                                  + (CPRchq.S = PROMISE)
                                  + (BiPRchq.$ = SET_BUSY)
                                  + (BPCsplit.$ = SET_BUSY)
                                  + (APRspllt.S = SET-BUSY)
                                  + (BPRsplit.$ = SET-BUSY,
                                  + (BPCmerge.$ = SET-BUSY)
                                  + (APRmerge.$ = SET-BUSY))
                ->6             : (BPCsplit.$ = SET-BUSY)
                                  • ((APCchg.$ = SET-BUSY)
                                  + (BPRchg.$ = SET-BUSY,
                                  + (CPRchq.$ = PROMISE)
                                  + (BiPRchg.$ = SET_BUSY)
                                  + (APCsplit.$ = SET-BUSY,
                                  + (APRsplit.$ = SET-BUSY)
                                  + (BPRsplit.$ = SET-BUSY)
                                  + (BPCmerge.$ = SET-BUSY)
                                  + (APRmerge.$ = SET_BUSY))
                ->7             : (APRsplit.$ = SET-BUSY)
                                  • ((APCchg.$ = SET_BUSY)
                                  + (BPRchg.$ = SET_BUSY)
                                  + (CPRchq.$ = PROMISE)
                                  + (BiPRchg.$ = SET_BUSY)
                                  + (APCsplit.$ = SET_BUSY)
                                  + (BPCsplit.$ = SET_BUSY)
                                  + (BPRsplit.$ = SET_BUSY)
                                  + (BPCmerge.$ = SET_BUSY)
                                    (APRmerge.$ = SET_BUSY))
                ->8             : (BPRsplit.$ = SET-BUSY)
                                  • ((APCchg.$ = SET-BUSY)
```

```
                        + (BPRchg.$   SET_BUSY)
                        +(CPRchg.$ = PROMISE)
                        + (BiPRchg.$ = SET-BUSY)
                        + (APCsplit.$ = SET BUSY)
                        + (BPCsplit.$ = SET-BUSY;
                        + (APRsplit.$ = SET-BUSY)
                        +(BPCmerge.$= SET-BUSY)
                        +(APRmerge.$ = SET-BUSY))
      ->9             :(BPCmerge.$ = SET_BUSY)
                        . ((APCchg.$ = SET_BUSY)
                        + (BPRchg.$= SET_BUSY)
                        + (CPRchg.S = PROMISE)
                        + (BiPRchg.$ = SET BUSY)
                        + (APCsplit.$ = SE?: BUSY)
                        + (BPCsplit.$ = SET-BUSY)
                        + (APRsplit.$ = SET-BUSY)
                        +(BPRsplit.$ = SET-BUSY)
                        + (APRmerge.$ = SET_BUSY))
      ->10            : (APRmerge.$ = SET_BUSY)
                        . ((APCchg.$= SET-BUSY)
                        +(BPRchg.$= SET BUSY)
                        + (CPRchg.$ = PROMISE)
                        + (BiPRchg.$ = SET BUSY)
                        +(APCsplit.$=SET BUSY)
                        + (BPCsplit.$ = SET_BUSY)
                        + (APRsplit.$ = SET BUSY)
                        + (BPRsplit.$ = SET-BUSY,
                        + (BPCmerge.$ = SET_BUSY))
      ->11            : (POS.$ = LEADER) .~((BiPRchg.$ = IDLE)
                        + (BPRsplit.$ = IDLE) + (BPCsplit.$ = IDLE))
      ->12            : (POS.$ = FOLLOWER) * ~((APCchg.$ = IDLE)
                        + (APCsplit.$ = IDLE) + (BPCmerge.$ = IDLE)
                        + (BPRchg.$ = IDLE) + (CPRchg.$ = IDLE)
                        + (APRsplit.$ = IDLE) + (APRmerge.$ = IDLE))
      ->$             : else;

end   /* SUPR_MONITOR2 */
```