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

SmartAHS and SHIFT Enhancements, Persistence and Query Interpretation

Jim Misener
California PATH

California PATH Research Report
UCB-ITS-PRR-2000-6

Report for MOU 258

March 2000

ISSN 1055-1425

SmartAHS and SHIFT Enhancements, Persistence and Query Interpretation

Table of Contents

Abstract. We have enhanced and "tuned" SmartAHS and SHIFT to address a wide variety of functional and user needs. SmartAHS has become an important microsimulation tool for design, analysis and evaluation of AHS – and "pre-AHS" or AHS deployment – concepts and scenarios in dimensions of system performance (i.e., throughput and travel time), safety and comfort. The SmartAHS/Hybrid Systems Tools Interface Format (SHIFT) is the basis for SmartAHS, and it is the general hybrid systems simulator for user-defined AHS architectures.

We have essentially conducted three work thrusts:

1. SmartAHS Enhancements
We added sensor and communication models to enhance functionality, upgrade the internals of the SHIFT system for more efficient computations of large simulations and more accessible platform options, and develop graphical user interfaces for easier use of SmartAHS model libraries.

2. SHIFT Enhancements
We enhanced SHIFT to make it more accessible, available on multiple platforms, improve computational efficiency and enable us to verify and implement the simulation designs.

3. Documentation
We collected the dispersed documentation of the SHIFT and SmartAHS tool sets, and integrates it into a cogent user manual and reference guide. The document was divided into two main parts: SHIFT and SmartAHS.

Key Words. SmartAHS, SHIFT, microsimulation, software, AHS, hybrid systems, models, sensors, communication, verification.

1 Executive Summary

In this project, PATH has fine-tuned the development of SmartAHS, a software system for microsimulation of Automated Highway System (AHS) design and scenarios. SmartAHS has become an indispensable tool for design, analysis and evaluation of AHS – and "pre-AHS" or AHS deployment – concepts and scenarios in dimensions of system performance (i.e., throughput and travel time), safety and comfort.

The underlying basis of SmartAHS is the SmartAHS/Hybrid Systems Tools Interface Format (SHIFT). This is the general hybrid systems simulator for user-defined AHS architectures, and SHIFT also specifies a high-level language invented to specify AHS-specific models for highway layout, vehicle dynamics, actuators and sensors.

The MOU 258 work to date has focused on ease-of-use SmartAHS and SHIFT enhancements, in areas of:

- Design and implementation of simulation checkpointing
- Component tracing
- Design and implementation of a query interpreter

We have built upon this base by focusing our extended work in three areas:

- SmartAHS enhancements
- SHIFT enhancements
- Documentation

The end-result of this extended work was a completed, practical automated highway design and evaluation tool, suitable for use with a wide variety of users and on a wide variety of computer platforms.

This work was complimentary with planned Federally funded efforts within the National Automated Highway System Consortium (NAHSC), particularly with Task B5 (Evaluation, Critical Issues and Tools) but there is no overlap. Activities within this MOU have facilitated completion of ongoing PATH MOU and previous NAHSC efforts. However, this work is not dependent on the progress of these PATH and NAHSC; that that respect, this MOU is stand alone.

2 Background

SmartAHS contains the following features:

- Highway Models: Supports user-defined description, compatible with UC Berkeley/Caltrans SmartPATH; can specify lane, segment, section, block, barrier, weather, source, sink.
- Vehicle Models: Provides simple, 2-D, 3-D and articulated simple vehicle dynamics.
- Controllers: Provides physical layer (steering, throttle, brake and tire burst) controllers; also supports open loop trajectory following controller and cooperating independent vehicle controller.
- Communication Models: Provides spherical, perfect receiver, transmitter and message communications.
- Sensor Models: Provides spherical, perfect closest vehicle sensor.
- Animation: Allows simple, 2D (top view) animation and high fidelity texture-mapped 3D animation (with reused code from SmartPATH)
- Allows full vehicle/environment/roadway processing and interaction.

These features are coded in SHIFT, as is the user-created model of the notional AHS system, then translated to C via a compiler. SHIFT is a programming language developed at PATH for describing dynamic networks of hybrid automata, such as AHS. Such systems consist of components that can be created, interconnected and destroyed as the system evolves. Components exhibit hybrid behavior, consisting of continuous-time phases separated by discrete-event transitions. Components may evolve independently, or they may interact through their inputs, outputs and exported events, and the interaction network itself may evolve.

SmartAHS/SHIFT applications in automated highway evaluations have included:

- Determination of the emissions and energy use of the Houston METRO Katy Freeway scenario. The evaluation used the UCR-developed modal emissions and fuel consumption model within SmartAHS and compared fuel consumption as well as hydrocarbon and nitrous oxide emissions of various AHS concepts versus manual driving.

- Analysis of the merge maneuver with autonomous vehicles, cooperative vehicles, platooning vehicles to define the degree of infrastructure involvement needed in a “merge assist” service. Additionally, this work developed control laws that could be used in other cooperative maneuvers such as lane changing.

SmartAHS analysis of an Intelligent Vehicle Initiative “Gen 1” safety case study is currently underway. The objective of this study is to analyze and compare three progressively more mature “pre-AHS” longitudinal control system concepts:

1. ACC with only engine braking and driver in the loop;
2. ACC with graduated braking authority with driver in the loop;
3. Cooperative longitudinal control system.

The safety case study will utilize and augment (and therefore also test) the SmartAHS simulation tool with various state-of-the-art vehicle, sensor, communication and human driver models.

Potential users from PATH and other research institutions plan on conducting several more analysis studies of specific driver assistance functions, partial automation technologies and traffic control schemes, as well as AHS. These “new users” are the primary beneficiary of the work proposed in this MOU; however, because of the multitude of enhancements in the work scope, it is anticipated that the current PATH analysts will receive significant benefits.

3 Methodology

We have performed the work in three phases in parallel:

2.1 Phase I. SmartAHS Enhancements

We added sensor and communication models to enhance functionality, upgrade the internals of the SHIFT system for more efficient computations of large simulations and more accessible platform options, and develop graphical user interfaces for easier use of SmartAHS model libraries.

2.2 Phase II. SHIFT Enhancements

We enhanced SHIFT to make it more accessible, available on multiple platforms, improve computational efficiency and enable us to verify and implement the simulation designs.

2.3 Phase III. Documentation

We collected the dispersed documentation of the SHIFT and SmartAHS tool sets, and integrates it into a cogent user manual and reference guide. The document was divided into two main parts: SHIFT and SmartAHS.

4 Phase I. SmartAHS Enhancements (Jan – May 98)

The current release of the SmartAHS microsimulator for vehicle-highway systems provides a stable platform with extensive functionality for a wide range of highway safety and throughput analysis studies.

We have further enhanced SmartAHS to make it even more functional, powerful and user-friendly. We will add sensor and communication models to enhance functionality, upgrade the internals of the SHIFT system for more efficient computations of large simulations and more accessible platform options, and develop graphical user interfaces for easier use of SmartAHS model libraries.

We performed four subtasks:

1. Sensor Architecture Development
2. Communication Architecture Integration
3. Human Driver Model Integration
4. Application GUI

4.1 *Sensor Architecture Development*

4.1.1 Introduction

The SmartAHS sensor subsystem architecture has been designed to be expandable and easy to use. The design is given as a set of superclasses that are used as the foundation of various types of sensors. Since sensor technology can vary, it was necessary to abstract the behavior of sensing devices by factoring their common features.

SmartAHS provides the following three-layered sensor architecture of SHIFT types:

- **TargetDetector** . This is the bottom layer, which provides the functionality of perfect sensors, outputting exact range and range rate with respect to the detected vehicle (if any). This means that noise isn't dealt with at this level. The TargetDetector type should be considered private and shouldn't be used by the SmartAHS user. It is intended to be used by sensor designers only.
- **SensorModel** . This is the middle layer, which works as a filter. The perfect results of the lower level are distorted in a way that depends on the model. Noise can be added both to range and range rate and false alarms can also be generated. The SensorModel type should be considered private and shouldn't be used by the SmartAHS user. It is intended to be used by sensor designers only.
- **Sensor** . This is the top layer, which is used to hide the complex details of the lower levels from the user (in fact it is the only public part of the architecture). The main function of this type is that of setting up the lower level mechanisms and of providing the user with the results coming from the middle layer.

New sensors can be written by inheriting from the described superclasses (which preserves the interfaces). Also note that the described root types are to be considered abstract. This means that their implementation isn't complete. They should be used only as supertypes. This structure has been used to implement a realistic radar sensor model, which also takes into account weather conditions.

In order to improve performance, some optimization techniques have been used. The roads are divided in cells and sensors actually check for other vehicles only in the neighboring cells. This avoids checking all the vehicles in the simulated world, which is an n^2 time process. The managment of cells is taken care of by a special type called SEP (Sensor Environment Processor).

4.1.2 Using Sensors

The user only needs to be aware of the SEP type, the Sensor type and their subtypes. The other layers are hidden.

4.1.2.1 The SEP type

Implementing sensors brought up a performance problem. How is a sensor to determine whether a car is within range? The easiest solution is to look for such a car in the set of all the cars in the world. This is obviously very expensive.
To avoid this, road sections have been subdivided in cells. Each cell corresponds to a set of vehicles. Cars move from cell to cell: when they cross a cell border, they add themselves to the new cell and remove themselves from the old one.

The SEP takes care of updating the cells. The means by which this task is performed is hidden from the user. All the user has to know is how to instantiate the SEP and how to connect it to the other components.

If the istantiation happens inside the vehicle, the create statement looks like this (from now on the italic font is used to denote a reference to a specific component):

```
SEP sep := create (
  SEP,
  the_vehicle     := self,
  current_segment := starting_segment,
  current_section := starting_section,
  current_cell    := cells (starting_section)[0]
);
```

The inputs of the SEP can be set with the following connections:

```
rxp (sep)             <- rxp (vrep);
ryp (sep)             <- ryp (vrep);
rzp (sep)             <- rzp (vrep);
current_lane (sep)    <- lane (vrep);
current_segment (sep) <- segment (vrep);
current_section (sep) <- section (vrep);
followLane (sep)      <- followLane (vrep);
the_vehicle (sep)     <- self;

vehicle_sensors (sep) <- {...};
```

The outputs of the SEP are:

- number current_cell_number
- set(Vehicle) current_cell
- set(Vehicle) previous_cell
- set(Vehicle) next_cell
- set(Vehicle) overlap_cell
-

Note that cell specific informations such as cell length are set up in the Section type. The number of cells is obtained from the length of the section and that of the cells.

4.1.2.2 The Sensor Type

A SmartAHS vehicle has one SEP and zero or more sensors. Four sensor links are provided in the Vehicle type: frontSensor, rearSensor, leftSensor and rightSensor. Others may be added in the subtypes of Vehicle.

The Sensor abstract superclass requires the following inputs:

- timer sample_timer
- Vehicle vehicle
- VREP vrep
- SEP sep

The timer type is used to select a sampling time for the sensor.
The outputs of the sensor are the following:

- number range
- number rangeRate
- target detectedTarget

Range is the distance in meters from the detected vehicle. Actually the range is calculated by subtracting half the length of the following car and half the length of the leading car from the distance between the two cars. This method assumes that cars have circular shapes.
Range rate is the difference between the speed of the detected vehicle and the speed of the current vehicle.

The target type encapsulates the detected vehicle. It has the following output:

- Vehicle vehicle_detected

Note that the distance in the target doesn't take into account the length of the vehicles.

4.1.2.3 Currently Implemented Sensors

A few sensors have been implemented on top of the described architecture. Their source code can be studied as an example of implementation.

- **range_sensor** : it detects the nearest vehicle around it (by "around it" we mean the vehicles in the previous, current and next cell). No noise is simulated, so detections are perfect. Two additional parameters can be provided as input:
    - number maxRange: maximum range of detection.
    - symbol direction: this can be assigned the value $front or $rear, depending on the desired direction of detection.
- **RadarSensor** : it detects the nearest vehicle in the area defined by the following parameters, using a radar sensor model. Three additional parameters can be provided as input:
    - number fov: field of vision angle
    - number minRange: minimum range of detection
    - number maxRange: maximum range of detection

4.1.3 Designing New Sensors

4.1.3.1 TargetDetector

The lowest level of the architecture fulfills the purpose of detecting targets if any. It does not simulate any noise. Its function is purely geometric: e.g. to understand wether a vehicle exists, that falls within the sensor range. For this reason TargetDetectors can be categorized according to the shape of their detection field.

The inputs of this type are:

- Vehicle vehicle
- VREP vrep
- SEP sensor_ep
- sensor enclosingSensor
- timer sample_timer
-

The outputs are the same as the sensor class. They are actually processed by the upper level before being passed to the sensor. Three transitions take place. It is important that all of them be present:

idle -> update {sample_timer:timer_tick}

It takes place when the timer ticks. It does the detection work and updates the outputs to reflect the results of the detection.

update -> updateUpperLevels {enclosingSensor:update_values}

This one is needed to synchronize with the upper levels once the computation is finished. On this level it's just a signal to notify that the values in the outputs are ready to be read.

updateUpperLevels -> idle {enclosingSensor:values_updated}

It goes back to the initial state.

4.1.3.2 SensorModel

This is the middle layer and it serves as a noise generator. The perfect results of the Target Detectors are read and distorted according to a specific model. The only input of this type is the fap (false alarm probability) which is optional in some models. It is a value between 0 and 1 and it's used to generate false alarms.

The outputs are:

- number dp
- number range
- number rangeRate
- target detectedTarget

Dp is the detection probability, which is calculated according to the model. The last three values are read from the corresponding targetDetector but can be changed according to some function of fap and of dp.

The state variables are not inherited, so they are included in the root class only as conceptual placeholders. They are:

- sensor enclosingSensor
- TargetDetector targetDetector

- number rE
- number rrE

RE and rrE are the range error and the range rate error.

There must be at least one transition synchronizing on enclosingSensor:update_values. It reads the values in the lower levels and distorts them. Of course there may be more than one such transition (e.g. one for regular detection, one for failed detection, one for false alarms and so on).

4.1.3.3 Sensor

The public interface (input and outputs) of the higher layer has been described already so we'll talk about the private one here.

Two state variables (targetDetector and sensorModel) should be used to link the sensor to its TargetDetector and to its SensorModel. A setup clause should be present to istantiate and initialize correctly these two components.

Two local events are exported:

- update_values
- values_updated

They are used for synchronization with lower levels.

Reading the values from the TargetDetector can be done in two ways. The easiest way is via a flow:

```
flow
  default {

    range     = range(targetDetector);
    rangeRate = rangeRate(targetDetector);
  };
```

The second one is via transitions such as:

```
idle -> update {update_values},
update -> idle {values_updated};
```

Note that two such transitions must be present anyway because they are used by lower level types and also because in this way the sensor signals it is updating its values to other interested types (e.g. automated cruise control types).

4.1.4 Implementation

Some implementation techniques can be studied by reading the sensors/sensor.hs file in the SmartAHS distribution.

4.2 *Communication Architecture Integration*

Work on this aspect is being done under the separately funded MOU 334. In this MOU, we will collaborate with that project to integrate the results into SmartAHS. Communication models for the Physical Layer (non-persistent network, semi-persistent network, persistent network) and the Data Link Layer (with four error situations: frames corrupted, lost, duplicated, misordered, and three error control schemes: acknowledgment, timeout, checksum) will be available to SmartAHS users after system integration of the communication models.

This document describes SHIFT models for Automated Highway System (AHS) communication components. These components must model the following layers of the open systems interconnection (OSI) reference: physical layer, media access control layer, logical link layer, network layer and transport layer. The functionalities of those layers have been adapted to AHS communication needs.

4.2.1 Modeling communication with SHIFT

Communication in the SmartAHS introduces a major problem. We want to interface the communication domain with the vehicle domain, but to simulate communication, the time unit required is around exp(10, -8); using this time unit to simulate vehicle traffic implies an extremely long simulation time. A way to solve this problem is to simulate the system (vehicle + communication) with a ``reasonable'' step in terms of simulation last and to aggregate the communication in this new unit. With this technique, the communication will not be modeled at the bit level, but at the message level. This solution is restrictive in the sense that it does not allow to observe efficiency and protocol delays. Nonetheless knowing the characteristics of a protocol (delay, throughput, etc.), we can model what happens during each time unit.
In this document, we focus on the message level approach, and we describe SHIFT components to simulate communication at the physical layer, at the media access control (MAC) layer and at the Logical Link Control (LLC) Layer. Other layers are not implemented yet. Note that modeling the communication at the bit level has also many interesting features, and should be done in the future.

4.2.2 Communication Components

4.2.2.1 Message

The **Message** type model messages being sent between users. It must be sub-typed to describe each N-Protocol Data Unit (i.e. the data format at the level N).

The **messageType** output specifies whether the message is a supervisory frame ($S) or an information one ($I). The **connectionType** output gives information about the type of the connection: unicast, broadcast, multi-cast. The SHIFT description of the **Message** type is given below:

```
type Message
{
        output symbol messageType;
        output symbol connectionType;
}
```

4.2.2.2 Physical Layer

The physical layer implements an unreliable link on which bits are transmitted. A link consists of a transmitter, a receiver, and a medium over which signals are propagated. The **Transmitter** type is an interface to model the communication transmitter, and the **Receiver** type is an interface to model the communication receiver. Medium properties, such as the capacity and the transmission rate, are modeled in a component called **monitor**. Other medium properties, such as propagation error, co-channel and channel interference, can be described in the **Receiver**. The **monitor** type also models the connection (both for point-to-point and for multi-cast channel) and the medium access control protocol (in the case of multi-cast channel).

4.2.2.2.1 Transmitter interface

The interface for the **Transmitter** type allows to model transmitters for point to point, broadcast and multi-cast communication.

To send a unicast message, the higher layer must synchronize with the **MAC_ready_point** event and it must provide the **receiverIn** and **messageIn** inputs.

In the case of broadcast communication, the higher layer must synchronize with **the MAC_ready_broad** event, and it must provide the **messageIn** input .

In the case of multi-cast communication, the higher layer must synchronize with the **MAC_ready_multi** event, and it must provide the **receiversIn** and **messageIn** inputs.

The **receiver** (or **receivers**) output refers to the destination(s) of the message and the **message** output is the message to send.

When transmitting a unicast message, the transmitter issues the **MAC_data_point** event. The receivers belonging to the same network as the transmitter synchronize with the transmitter's **MAC_data_point** event and they check if they are the destination of the message (i.e. if they are equal to the **receiver** output of the transmitter).

When transmitting a multicast message, the transmitter issues the **MAC_data_multi** event. The receivers belonging to the same network as the transmitter synchronize with the transmitter's **MAC_data_multi** event and they check if they are the destinations of the message (i.e. if they belong to the **receivers** output of the transmitter).

When transmitting a broad-cast message, the transmitter issues the **MAC_data_broad** event.

The receivers belonging to the same network synchronize with the transmitter's **MAC_data_broad** event.

After the message is sent with unicast communication, the transmitter issues the **MAC_confirm_point** event. In the case of multi-cast communication, the transmitter issues the event **MAC_confirm_multi** or, in the case of broadcast communication it issues the **MAC_confirm_broad** event.

The SHIFT description of the interface of the **Transmitter** type is given below:

```
type Transmitter
{
output Receiver receiver;          // Destination of the transmitted message.
       set(Receiver) receivers;    // Destinations of the transmitted message.
       Message message;            // Message being transmitted.

input  Receiver receiverIn;        // One receiver specified for
                                   // unicast communication.
       set(Receiver) receiversIn;  // A set of receivers  specified
                                   // for multi-cast communication.
       Message messageIn;          // Message to be transmitted.

export
closed MAC_confirm_point,      // Issued when a unicast message
                               // has been transmitted.
       MAC_confirm_multi,      // Issued when a multi-cast message
                               // has been transmitted.
       MAC_confirm_broad;      // Issued when a broadcast
                               // message has been transmitted.
open   MAC_ready_point,        // Issued when the transmitter is ready
                               // to transmit a unicast message.
       MAC_ready_multi,        // Issued when the transmitter is  ready
                               // to transmit a multi-cast message.
       MAC_ready_broad,        // Issued when the transmitter is  ready
                               // to transmit a broadcast message.
       MAC_data_point,         // Issued when transmitting a unicast message.
       MAC_data_multi,         // Issued when transmitting a multi-cast message.
       MAC_data_broad,         // Issued when transmitting a broadcast message.
       exiting;

discrete init;
transition
       all - exit {exiting}; // Each subtype of Transmitter must have
}                              // this transition.
```

4.2.2.2.2 Receiver interface for a perfect channel

The interface for the **Receiver** type allows to model receivers for point to point, broadcast and multi-cast communication.

The **Receiver** type describes a perfect channel. It must be sub-typed in order to model errors, due to propagation, co-channel and channel interference.

The **transmitter** output refers to the transmitter that sent out the message and the **message** output contains the message that was sent.

The **TxNetwork** input is the set of transmitters that are involved in the same network. Note that this set can be connected to the set of transmitters which is defined in the **Monitor** type. In this way the user won't have to update this variable when a transmitter leaves or joins the network.

To receive a unicast message, the receiver must synchronize with the **MAC_data_point** event of one of the transmitters in the **TxNetwork** variable.

To receive a broadcast message, the receiver must synchronize with the **MAC_data_broad** event of one of the transmitters in the **TxNetwork** variable.

To receive a multi-cast message, the receiver must synchronize with the **MAC_data_multi** event of one of the transmitters in the **TxNetwork** variable.

If the receiver is the destination of the transmitted message, it issues the **MAC_indication_point** event (for unicast communication), the **MAC_indication_multi** event (for multi-cast communication) or the **MAC_indication_broad** event (for broadcast communication).

The SHIFT description of the interface for the **Receiver** type is given below:

```
type Receiver
{
output Transmitter transmitter;    // transmitter of the received message.
       Message     message;        // received message.

input  set(Transmitter) TxNetwork; // transmitters involved in the same network.

export MAC_indication_point,       // Issued when a unicast message
                                   // has been received.
       MAC_indication_broad,       // Issued when a broadcast message
                                   // has been received.
       MAC_indication_multi,       // Issued when a multi-cast message
                                   // has been received.
       exiting;
discrete
       init;
transition
       init - exit {exiting};
}
```

4.2.2.2.3 Receiver interface for an imperfect channel

A subtype of the **Receiver** type has been implemented to model channel errors (due to propagation and co-channel and channel interference).

For unicast communication, when the receiver issues the **MAC_indication_point** event, it also issues a message status event (**error_free_frame** or **error_frame**).

For multi-cast communication, the receiver issues the event **MAC_indication_multi**, only if the message arrives error free, otherwise, the receiver deletes the message.

For broadcast communication, the receiver issues the event **MAC_indication_broad**, only if the message arrives error free, otherwise, the receiver deletes the message.

The probability to estimate the percentage of wrong messages is stored in a global variable called **ErrorTransmission.** It has to be set when the simulation begins.

The SHIFT description for the **Receiver** interface is given below:

```
type ErrReceiver : Receiver
{
```

```
export error_free_frame,  // Issued when the message is error free.
       error_frame;       // Issued when the message is not error free.
discrete
       init;
transition
       init - exit {exiting};
}
global number ErrorTransmission := 0;
```

4.2.2.2.4 Interface for the Monitor

The **Monitor** works as a centralized component and there is one for each network. It models some physical layer and some MAC layer functionalities.

On one hand, the **Monitor** is a representation of a set of users adopting the same physical medium; it models channel properties and keeps track of the transmitters sharing the channel. The **Monitor** also models the connection type (point to point or broadcast channel).

On the other hand, communication is not simulated in real time, instead we aggregate many transmissions in a time unit, called slot, the length of which is defined at the beginning of the simulation. For each slot, a centralized algorithm (defined in the **Monitor**) decides which transmitters are allowed to transmit. Since point-to-point and multi-access channel are different from each other, **Monitor** has to be subclassed in order to manage the two cases.

For a point-to-point connection, the number of allowed exchanges in the next slot is a function of channel capacity, of transmission rate and of packet length. For a broadcast channel, the number of allowed exchanges is also function of the throughput of the media access function.

The MAC layer functionality of the **Monitor** are discussed, in more details, in the MAC layer section. The SHIFT description of the interface for the **Monitor** type is given below:

```
type Monitor
{
output
       set(Transmitter) transmitters := {};
       number   PacketLenght     := L;
       number   Efficiency       := a;   // Propagation and detection delay.
       number   DataRate         := C;   // Data rate in Kbps.
       number   TransmissionTime := Ts;  // Transmission time in sec.
       number   Throughput;              // Effective throughput of the system.
       number   TheoriticalThroughput;   // Theoretical throughput of the system.
}
global number   slot := 1;               // Unit of the aggregate step
                                         // in sec ( SHIFT step must be smaller).
```

4.2.2.3 Data Link Layer

Networks can be divided into two categories: those using point-to-point connections and those using a broadcast or multi-access channel (that is when the same communication link is shared between several users). In the case of point-to-point connection, the main task of the Data Link Layer is to provide to the higher layer a virtual error free packet link. In the case of broadcast channel, the Data Link Layer is split into two sub-layers, the media access control sub-layer (MAC) and the Logical Link Control sub-layer (LLC). The purpose of the MAC layer is to allocate the multi-access medium among the various users. The functionalities of the LLC are those of the Data Link Layer for a point-to-point connection. The MAC layer and the LLC layer constitute the Data Link Layer for a multi-access channel. Access control is performed by the **Monitor** type and the **LinkLayer** type models the LLC.

4.2.2.3.1 MAC Layer

When the same communication link is shared between several users, we need an additional sub-layer between the LLC and the physical layer. This extra layer is called Medium Access Control layer. Its purpose is to allocate the multi-access medium among the various users.
There are two extremes among the different algorithms designed for this issue. The fist extreme is the "free for all" approach, in which a user sends a message hoping for no interference from other users. The second one is "perfectly scheduled" approach, in which some order is estabilished among the users for channel usage.

The media access function is embedded in the **Monitor**, which must be sub-typed in order to support different media access algorithms.

4.2.2.3.1.1 Features of the Monitor type

As we already mentioned, the monitor is a centralized component, that is assigned to a network. It provides informations about the medium and it monitors all the exchanges (or throughput) allowed in a slot. For point-to-point communication, the throughput depends on slot duration and on channel properties. For a broadcast channel, the throughput depends on slot duration, channel properties and on the chosen MAC protocol. In both cases throughput is the number of successfully delivered packets per packet transmission time. Note that if the used throughput is the number of successfully sent packets, the hidden conflict and the transmission during the critical period (collision due to the use of one channel by several hosts) are part of the model; the propagation errors (due to interference with other networks for example) are modeled in the receiver.

The **Monitor** type must be sub-typed in order to implement specific MAC layer protocols, such as CSMA or Token Ring.

For CSMA, the monitor keeps track of how many transmitters ask to send data, and it uses this load to compute the number of transmitters allowed to transmit in the next slot.

4.2.2.3.1.2 Interface for the Monitor

The **Monitor** type has a **transmitters** output variable. It is the set of transmitters belonging to the network, which is described by that **Monitor**. This set is updated by the transmitters themselves (in their setup action or in the initial transition).

```
type Monitor
{
        output set(Transmitter) transmitters := {};
}
```

4.2.2.3.2 Logical Link Layer

The purpose of the the Logical Link Layer is to transfer messages without error in the case of unicast communication. For broadcast (or multi-cast) communication, the error correction must be done at an upper layer.

The Logical Link Layer interface is described in the **LinkLayer** type and it has to be subtyped in order to implement a specific error control algorithm.

Each **LinkLayer** component belongs to one network, and can deal with several connections at the same time. The following two paragraphs provide a description of the Buffer type and of the Link type. Both of them are used by the **LinkLayer**.

4.2.2.3.2.1 Interface for the Buffer

To write in the buffer, the user must provide the **ItemIn** input and he must synchronize with the buffer's **not_full** event. To read and delete an item in the buffer, the user must synchronize with the buffer&acute;s **not_empty** event. After this the value to be read can be found in the **ItemOut** output.

The SHIFT description of the interface for the **Buffer** type is given below:

```
function modulo(number index;       // The modulo function is a C
                number bufferSize)  // function  used  by the
                - number;          // Buffer type.
type Buffer
{
input  Message     itemIn;          // Message to record.
output Message     itemOut;         // Message to read.
       number      getPlace      := 0; // Index of the next message to read.
       number      putPlace      := 0; // Index of the next message to write.
       number      numberOfItems := 0; // Number of recorded message.
       number      bufferSize;         // Size of the buffer.
       array(Message) buffer;          // Array of recorded messages.
setup do { buffer := [nil : i in [0 .. bufferSize - 1]];};
export open not_empty,
            not_full;
discrete
       empty,           // Buffer accessible in writing only.
       neither,         // Buffer accessible in writing and reading.
       full;            // Buffer accessible in reading only.
transition
       empty - neither {not_full}
       do {
              buffer[putPlace] := itemIn;
              numberOfItems    := numberOfItems + 1;
              putPlace         := modulo((putPlace + 1), bufferSize );
       },
       neither -empty {}
       when numberOfItems <= 0,
       neither - neither {not_full}
       do {
              buffer[putPlace] := itemIn;
              numberOfItems    := numberOfItems + 1;
              putPlace         := modulo((putPlace + 1), bufferSize);
       },
       neither - neither {not_empty}
       do {
              itemOut       := buffer[getPlace];
              numberOfItems := numberOfItems - 1;
              getPlace      := modulo((getPlace + 1), bufferSize);
       },
       neither - full {}
       when numberOfItems = bufferSize,
       full - neither {not_empty}
       do {
              itemOut       := buffer[getPlace];
              numberOfItems := numberOfItems -1;
              getPlace      := modulo((getPlace + 1), bufferSize);
       };
}
```

The **LL_Buffer** subtype doesn't delete the item immediately after reading it. To delete the last read message, the user must explicitly synchronize with the buffer's **cancel** event.

4.2.2.3.2.2 Interface for the Link

Each connection between a local and a remote user is modeled at the Data Link Layer by the **Link** type. There are several informations that must be remembered about a Data Link connection. These informations are stored in a type called **Link**.

Among the outputs in **Link** are the remote and local **LinkLayer** (or a set of **LinkLayer** in the case of multi-cast communication) and a reference to the local user's two buffers (one to read from messages from the upper layer and one to write messages to the upper layer). In addition there are several variables dealing with the send and receive sequence numbers.

When a **LinkLayer** component receives a new message from the local user to a remote user, it creates a **Link** to model this connection. A few parameters must be set in the **Link**: the connection type (unicast, broadcast, multi-cast), the size of the buffers and a reference to the remote **LinkLayer** (or a set of remote **LinkLayer** in case of multi-cast communication).

A **Link** is also created when the **LinkLayer** receives a new message from a remote user. Therefore for each connection, two **Link** instances are created (one at the source and one at the destination).

Note that **Link** must be subtyped according to the error correction algorithm used at the Logical Link Layer.

The SHIFT description of the interface for the **Link** type is given below:

```
type Link
{
output symbol          connectionType;     // $UNI, $BROAD, $MULTI.
       LinkLayer       destination;        // Remote linkLayer(s) involved
       set(LinkLayer)  destinations := {}; // in the connection
       LinkLayer       source;
       LL_Buffer       rBuffer; // Buffer accessible in reading by the linkLayer.
       LL_Buffer       wBuffer; // Buffer accessible in writing by the linkLayer.
       number          bufferSize;
       // The following variables are used by the Stop and Wait
       // protocol implemented at the Logical Link Layer.
       number          sequenceNumber := 0;
       number          requestNumber := 0;
       symbol          LastFrameAck := YES;
       symbol          LastFrameNack := NO;
setup define {
              LL_Buffer t_rBuffer := create(LL_Buffer,
                                            link := self,
                                            bufferSize := bufferSize);
              LL_Buffer t_wBuffer := create(LL_Buffer,
                                            link := self,
                                            bufferSize := bufferSize);
       }
       do {
              rBuffer := t_rBuffer;
              wBuffer := t_wBuffer;
       };
discrete      init,
              establish;
export  open  exiting;
transition
       init - establish {}
       define {
             LinkLayer t_linkLayer := source;
       }
       do {
             Links(t_linkLayer) := Links(t_linkLayer) + {self};
       },
       all - exit {exiting};
}
```

4.2.2.3.2.3 Interface for the Logical Link Layer

The **LinkLayer** type is an interface for the Logical Link Layer. It must be sub-typed in order to implement different error correction algorithms.

At creation time, the **receiver** and **transmitter** outputs must be properly initialized.
For unicast communication, when the network layer wants to send data, it must provide the following inputs: **messageIn** and **destinationIn** (the **LinkLayer** at the remote host). It must also synchronize with the **LL_ready** event.

For multi-cast communication, when the network layer wants to send data, it must provide the following inputs: **messageIn** and **destinationsIn** (which is a set containing the **LinkLayer**s at the remote hosts). It must also synchronize with the **LL_ready** event.

For broadcast communication, when the network layer wants to send data, it must provide the following **messageIn** input and it must synchronize with the **LL_ready** event.

Each connection between local and remote users is modeled through the **Link** type. When an error free frame arrives, the **LinkLayer** stores the frame in the write-buffer corresponding to this connection, and issues an **LL_indication** event.

To receive the frame, the upper layer must synchronize on the **LL_indication** event and it must also synchronize with the link's buffer where the frame was stored. The **itemOut** output of this buffer contains the frame to be read. To delete the frame and free the buffer, the upper layer must synchronize with the **cancel** event issued by the buffer.

The SHIFT description of the interface for the **LinkLayer** type is given below:

```
type LinkLayer
{
input  Message        messageIn;      // Message to be transmitted.
       LinkLayer      destinationIn;  // Message destination.
       set(LinkLayer) destinationsIn; // Message destinations.

output Receiver     receiver;      // Receiver attached to the link layer.
       Transmitter  transmitter;   // Transmitter attached to the link layer.
       set(Link)    Links := {};   // Set of connection recorded at the link layer.
       number       bufferSize;    // Size of the buffers at the link layer.
export open         LL_ready,
                    exiting;
       closed       LL_confirm,
                    LL_indication;
}
```

4.2.2.3.2.4 How Logical Error Control is Modeled

The problem: one protocol entity wants to send another protocol entity a sequence of frames without errors.

The **LinkLayer** type has to model the four following scenarios: corrupted frames, lost frames, mis-ordered frames, duplicated frames. It also has to set up the following three error control mechanisms: acknowledgment, timeout, checksum (see Figure1).

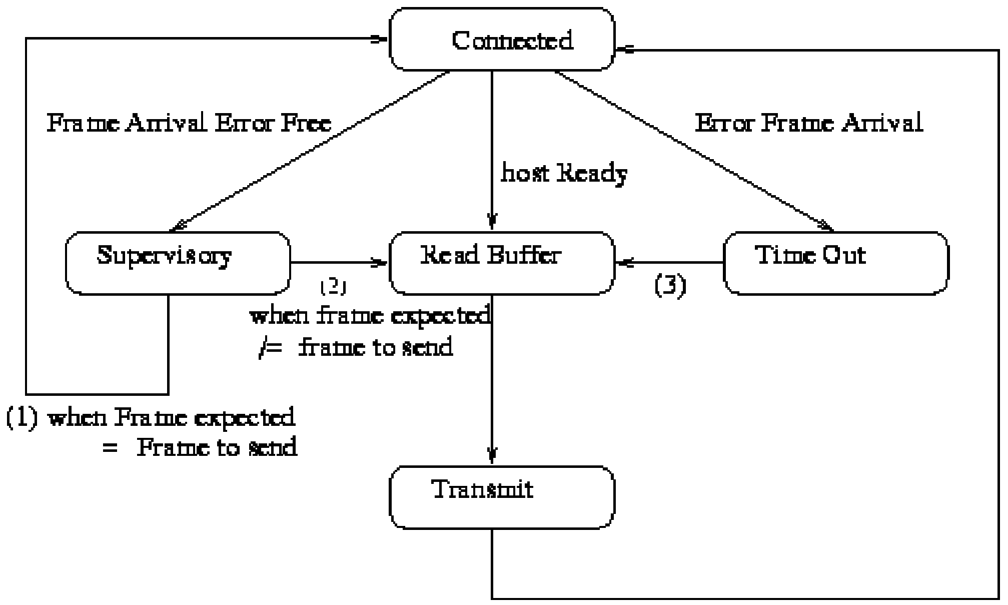
Scenario	**Cause**	**Detection Method**
corrupted msg	bit error on the link	checksum
lost msg	Congestion	ack/timeout
mis-ordered msg	different paths and retransmission	sequence #
duplicated msg	Retransmission	sequence #

Figure 1. Possible scenarios and detection methods

Error control mechanisms:
- acknowledgment: tells the sender what has (not) been received (ACK or NACK)
- timeout: an entity waits a given amount of time before retransmitting (sender timeout) or asking to retransmit (receiver timeout).
- checksum.

At the sender, we model three scenarios:
(1) the receiver issues the **error_free_frame** event; the new frame is a supervisory frame which acknowledges the last sent frame and asks for the next one.
(2) the receiver issues the **error_free_frame** event; the new frame is a supervisory frame but does not acknowledge the last frame sent and asks for a retransmission.
(3) the receiver issues the **error_frame** event; the new frame is a corrupted supervisory frame.

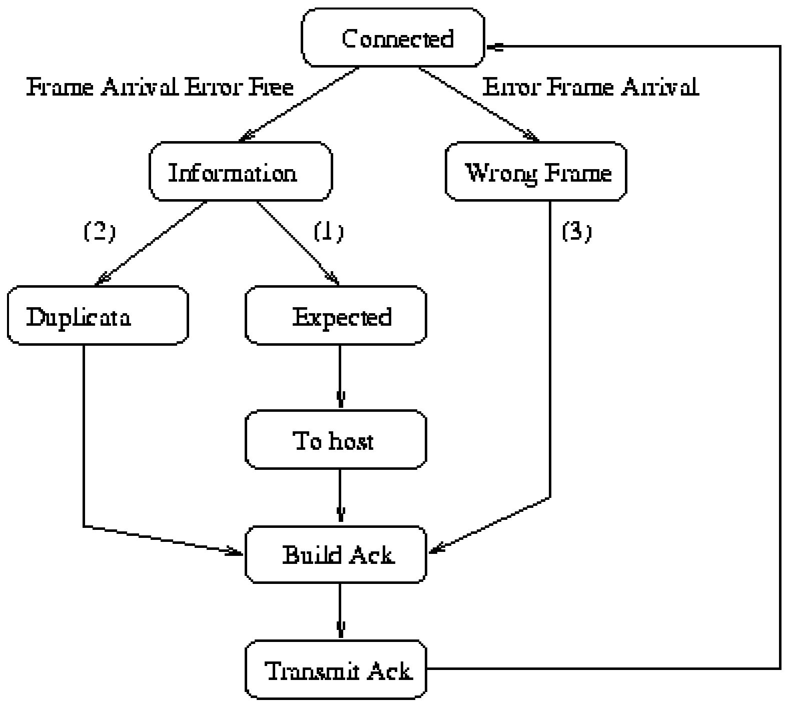
This mechanism allows us to model the timeout. The sender retransmits the last frame.
Figure 2 shows the simplified logical error control mechanism at the transmitter.



(1) Frame received by the receiver + ack correct
(2) Frame was not received correctly by the receiver, the receiver asks for a retransmission
(3) The received ack was wrong or lost (model the time out)

Figure2: Logical error control at the sender.

At the receiver, we model three scenarios:
(1) the receiver issues the **error_free_frame** event; the new frame is a data frame and its sequence number corresponds to the expected sequence number. The remote user sends back an acknowledgment and asks for the next frame.
(2) the receiver issues the **error_free_frame** event; the new frame is a data frame but its sequence number does not correspond to the expected sequence number (it's a duplicate). The remote user asks for the next frame.
(3) the receiver issues the **error_frame** event; the new frame is a corrupted data frame. The remote user asks for the same frame.
Figure 3 shows the simplified logical error control mechanism at the receiver.



(1) Frame received error free, ask for the new one

(2) Duplicated frame (Hypothese: last ack lost or wrong), ask for the next frame

(3) Frame not received correctly, ask for the same frame

Figure3: Logical error control at the receiver.

4.2.3 Examples

In this chapter we describe the subtypes of Transmitter, Receiver, Monitor and LinkLayer, implementing a point-to-point connection using Stop-And-Wait algorithm at the Logical Link Layer. For a broadcast channel, one can find a lot of algorithms to provide the functionalities required by the MAC and the LLC layer. We made a selection among all these algorithms and we decided to model a semi persistent network, and a persistent network. For the semi persistent network, we used a Carrier Sense Multiple Access (CSMA) protocol at the MAC layer and a Stop-And-Wait (SAW) protocol at the Logical Link Control layer.
For the persistent network, we used a Token Ring protocol at the MAC layer and a Stop-and-Wait (SAW) protocol at the Logical Link Control layer.

4.2.3.1 Point-to-point connection

In order to model a point to point connection, **Transmitter**, **Receiver** and **Monitor** are subtyped in **UniPointTransmitter**, **GnrErrReceiver** and **UniPointLink**.

4.2.3.1.1 The UniPointTransmitter type

Figure 4 shows the logical behavior of the **Transmitter** sub-type for a point-to-point connection. The transmitter begins in the *idle* state. When creating the transmitter, the user must initialize the **monitor** output. In its setup action, the transmitter adds itself to the **Transmitters** output variable of the monitor. This variable is a set of **Transmitter** which is used by the monitor to detect ready-to-transmit transmitters.

The higher layer must synchronize with the **MAC_ready_point** event. It must also provide the **messageIn** and **receiverIn** inputs. Then the transmitter moves to the *check_point* state and stores the message and the destination of the message in its **message** and **receiver** outputs. After this it goes in the *get_channel_point* state and issues the **get_channel** event. The **monitor** must synchronize with this event to keep track of all the ready-to-transmit transmitters during a slot. The monitor must synchronize with the **MAC_data_point** event to begin the transmission. To defer the transmisson to the next slot the monitor synchronizes with the **backlogged** event. After transmitting the message, the transmitter issues the **MAC_confirm_point** event.

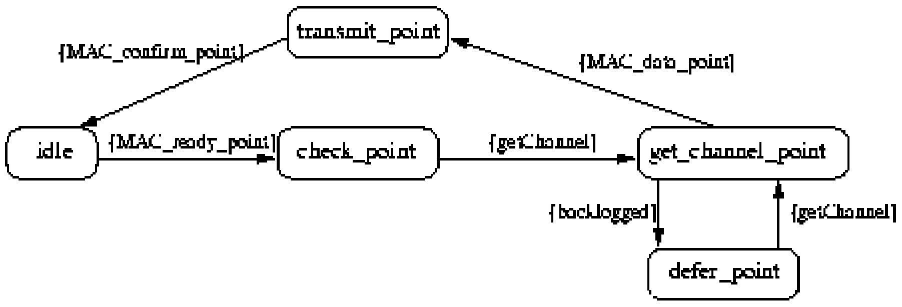


figure 4: State machine for Point-to-Point transmitter.

The SHIFT description of the **UniPointTransmitter** is given below in several fragments.

Output
```
type UniPointTransmitter : Transmitter
{
output  UniPointLink uniPointMonitor;   // Monitor involved in the
...}                                    // point-to-point connection
```
State
```
type UniPointTransmitter : Transmitter
{
state  number timer := 0;

flow   defer_law {timer' = 1;};
...}
```
Exported events
```
type UniPointTransmitter : Transmitter
{
export
open   backlogged,
       getChannel;
...}
```
Transition

1. In this transition, the transmitter adds itself to the monitor's set of transmitters.

```
type UniPointTransmitter : Transmitter
{
transition
       init - idle {}
       do {
       uniPointTransmitters(uniPointMonitor)
                           := uniPointTransmitters(uniPointMonitor)
                              + {self};
        },
...}
```

2. The higher layer must synchronize with the transmitter's **MAC_ready_point** event and it must provide the **messageIn** and **receiverIn** inputs to send a message.

```
type GnrCsmaTransmitter : Transmitter
{
transition
       idle - check_point {MAC_ready_point},
...}
```

3. The transmitter checks the status of the channel. The monitor must synchronize with the **getChannel** event to keep track of all the ready-to-transmit transmitters in a slot.

```
type GnrCsmaTransmitter : Transmitter
{
transition
       check_point - get_channel_point {getChannel}
       do {
              receiver := receiverIn;
              message  := messageIn;
       },
...}
```

1. This transition is taken when the transmitter is not allowed to transmit. The monitor must synchronize with the transmitter's **backlogged** event to defer its transmission.

```
type GnrCsmaTransmitter : Transmitter
{
transition
       get_channel_point - defer_point {backlogged},
...}
```

2. The transmitter senses again the channel, one slot later.

```
type GnrCsmaTransmitter : Transmitter
{
transition
       defer_point - get_channel_point {getChannel}
       when timer = slot
       do {
              timer := 0;
       },
...}
```

3. The monitor must synchronize with the transmitter's **MAC_data_point** event to start the transmission.

type GnrCsmaTransmitter : Transmitter
{
transition

get_channel_point - transmit_point {MAC_data_point},
...}

1. The transmitter issues the **MAC_confirm_point** event when the message is sent.

```
type GnrCsmaTransmitter : Transmitter
{
transition
        transmit_point - idle {MAC_confirm_point},
...}
```

4.2.3.1.2 The GnrReceiver and GnrErrReceiver Types

The **Receiver** type is the same for point-to-point connection and for a broadcast channel. Figure 5 shows the logical behavior of the **Receiver** for unicast communication.

The receiver starts in the *awaiting* state. It moves to the *forwarding_point* state by synchronizing with the **MAC_data_point** event of one of the transmitters in the **TxNetwork** variable. In this state, the receiver checks if it is the destination of the message. If it is not, it goes back to

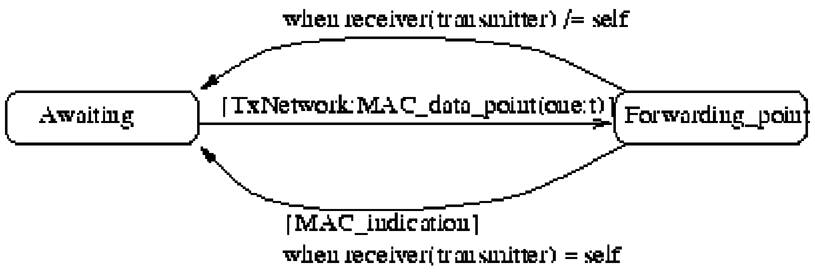

when receiver(transmitter) /= self
Awaiting
[TxNetwork:MAC_data_point(one:t)]
Forwarding_point
[MAC_indication]
when receiver(transmitter) = self

Figure5: State machine for Point-to-Point receiver.

awaiting. If it is, it issues the **MAC_indication_point** event and goes back to *awaiting*.

The SHIFT description of the **GnrReceiver** for unicast, broadcast, and multi-cast communication is given below.

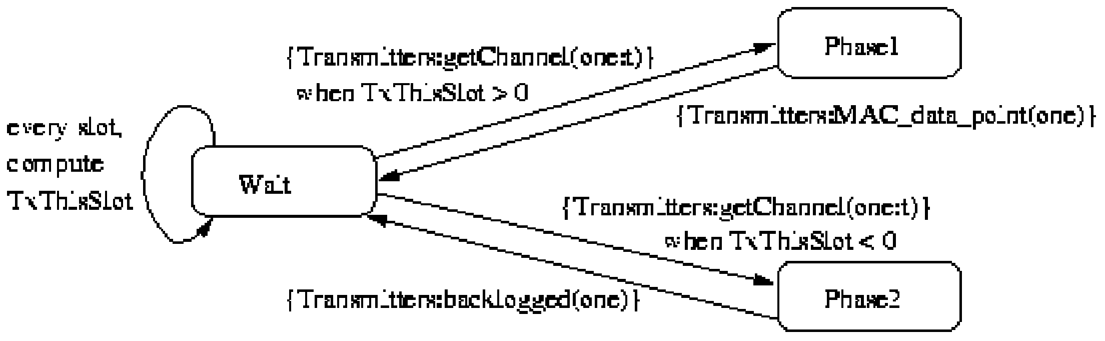
```
type GnrReceiver : Receiver
{
state
       Receiver receiver;
       set(Receiver) receivers;
discrete
       awaiting,
       forwarding_point,
       forwarding_multi,
       forwarding_broad;
transition
       awaiting - forwarding_point {TxNetwork:MAC_data_point(one:t)}
       do {
              transmitter := t;
              receiver    := receiver(t);
              message     := message(t);
       },
       forwarding_point - awaiting {}
       when receiver /= self
       do {
              transmitter := nil;
              message     := nil;
       },
       forwarding_point - awaiting {MAC_indication_point}
       when receiver = self,
       awaiting - forwarding_multi {TxNetwork:MAC_data_multi(one:t)}
       do {
              transmitter := t;
              receivers   := receivers(t);
              message     := message(t);
       },
       forwarding_multi - awaiting {}
       when not (self in receivers)
       do {
              transmitter := nil;
              message     := nil;
       },
       forwarding_multi - awaiting {MAC_indication_multi}
       when (self in receivers),
       awaiting - forwarding_broad {TxNetwork:MAC_data_broad(one:t)}
       do {
              transmitter := t;
              message     := message(t);
       },
       forwarding_broad - awaiting {MAC_indication_broad},
       all - exit {exiting};
}
```

The SHIFT description of the **GnrErrReceiver** for an non-perfect channel is given below.

```
type GnrErrReceiver : GnrReceiver
{
state  number ErrorProbability := 0;
       Receiver receiver;
       set(Receiver) receivers;
export error_free_frame,   // Issued when the message is error free.
       error_frame;        // Issued when the message is not error free.
discrete
       awaiting,
       forwarding_point,
       forwarding_multi,
       forwarding_broad;
transition
       awaiting - forwarding_point {TxNetwork:MAC_data_point(one:t)}
       do {
              transmitter := t;
              receiver    := receiver(t);
              message     := message(t);
              ErrorProbability := random();
       },
       forwarding_point - awaiting {}
       when receiver /= self
       do {
              transmitter := nil;
              message     := nil;
              ErrorProbability := 0;
        },
       forwarding_point - awaiting {MAC_indication_point, error_free_frame}
       when receiver = self
       and ErrorProbability = ErrorTransmission
       do {
              ErrorProbability := 0;
       },
       forwarding_point - awaiting {MAC_indication_point, error_frame}
       when receiver = self
       and ErrorProbability < ErrorTransmission
       do {
              transmitter := nil;
              message     := nil;
              ErrorProbability := 0;
       },
       awaiting - forwarding_multi {TxNetwork:MAC_data_multi(one:t)}
       do {
              transmitter := t;
              receivers   := receivers(t);
              message     := message(t);
              ErrorProbability := random();
       },
       forwarding_multi - awaiting {}
       when not (self in receivers)
       do {
              transmitter := nil;
              message     := nil;
       },
       forwarding_multi - awaiting {MAC_indication_multi}
       when (self in receivers)
       and ErrorProbability = ErrorTransmission
       do {
              ErrorProbability := 0;
       },
       forwarding_multi - awaiting {}
       when (self in receivers)
       and ErrorProbability < ErrorTransmission
       do {
              transmitter := nil;
              message     := nil;
              ErrorProbability := 0;
       },
       awaiting - forwarding_broad {TxNetwork:MAC_data_broad(one:t)}
       do {
              transmitter := t;
              message     := message(t);
              ErrorProbability := random();
       },
       forwarding_broad - awaiting {MAC_indication_broad}
       when ErrorProbability = ErrorTransmission
       do {
              ErrorProbability := 0;
       },
       forwarding_broad - awaiting {}
       when ErrorProbability < ErrorTransmission
       do {
              ErrorProbability := 0;
       },
       all - exit {exiting};
}
```

4.2.3.1.3 The UniPointLink Monitor Type

Figure 6 shows the logical behavior of the **Monitor** subtype for point-to-point connection.



{Transmitters:getChannel(one:t)}
when TxThisSlot > 0
Phase1
{Transmitters:MAC_data_point(one)}
every slot,
compute
TxThisSlot
Wait
{Transmitters:getChannel(one:t)}
when TxThisSlot < 0
{Transmitters:backLogged(one)}
Phase2

Figure 6: State machine for Point-to-Point monitor.

The monitor begins in the *Wait* state. In every slot, the monitor allows a fixed number of transmissions. The number of allowed transmissions is called **TheoreticalThroughput** and it is calculated in the monitor's setup action.

If **TxThisSlot** is greater than 0, the monitor moves to *phase1* by synchronizing with the **getChannel** event of one of the transmitters in the **transmitters** variable. If **TxThisSlot** is less than 0, the monitor moves to state *phase2* by synchronizing with the **getChannel** event of one of the transmitters in the **transmitters** variable.

In *phase1*, the monitor allows the transmission by synchronizing with the specific transmitter's **MAC_data_point** event. In *phase2*, the monitor defers the transmission by synchronizing with the specific transmitter's **backlogged** event.

The SHIFT description of the **Monitor** subtype is given below.

```
type UniPointLink : Monitor
{
output set(UniPointTransmitter) uniPointTransmitters := {};

state
       number timer := 0;
       number TxThisSlot;

       UniPointTransmitter uniPointTransmitter;

flow default { timer' = 1;};

setup define {
       number t_TransmissionTime       := PacketLenght / DataRate;
       number t_TheoriticalThroughput  := ( slot
                                          / t_TransmissionTime);
        }
       do {
       TransmissionTime       := t_TransmissionTime;
       TheoriticalThroughput  := t_TheoriticalThroughput;
        };
discrete
       await,
       phase1,
       phase2;

transition
       await - await {}
       when timer = slot
       do {
              TxThisSlot := TheoriticalThroughput;
              timer      := 0;
        },

       await - phase1 {uniPointTransmitters:getChannel(one:t)}
       when TxThisSlot  0
       do {
              TxThisSlot := TxThisSlot - 1;
              uniPointTransmitter := t;
       },

       await - phase2 {uniPointTransmitters:getChannel(one:t)}
       when TxThisSlot = 0
       do {
              uniPointTransmitter := t;
       },
       phase1 - await {uniPointTransmitter:MAC_data_point},
       phase2 - await {uniPointTransmitter:backlogged};
}
```

4.2.3.2 Broadcast channel: CSMA at the MAC layer

4.2.3.2.1 CSMA algorithm

At the MAC layer we use a Carrier Sense Multiple Access protocol. A description of the algorithm is given below.

```
* The model of a transmitter A is:
    A  has a message to send to B.
    A senses the channel (busy or idle):
    if idle,
      A transmits the message and then deletes it
      A waits for a new message from the higher layer
    else
      A waits a random delay before sensing again the channel.
* The model of a receiver B is:
    B receives a message from A
    B computes the Cyclic Redundancy Code (CRC) and compares it with
    the CRC in the message:
    if the CRCs are equal,
      the message is forwarded to the higher layer
    else
      the message is deleted.
```

In order to model CSMA, **Transmitter**, **Receiver** and **Monitor** are subtyped in **GnrCsmaTransmitter**, **GnrErrReceiver** and **GnrCsma.** These subtypes support unicast, broadcast and multi-cast communication.

4.2.3.2.2 The GnrCsmaTransmitter Type

The SHIFT description of the **GnrCsmaTransmitter** for unicast, broadcast, and multi-cast communication is given below in several fragments.

Output

```
type GnrCsmaTransmitter : Transmitter
{
output  GnrCsma gnrCsmaMonitor; // Monitor involved in the CSMA network.
...}
```

State

```
type GnrCsmaTransmitter : Transmitter
{
state  number timer := 0;

flow   defer_law
       {                     // When a transmitter is backlogged, it
            timer' = 1;      // waits one slot before sensing again
       };                    // the channel. The timer models this
...}                         // delay.
```

Exported events

```
type GnrCsmaTransmitter : Transmitter
{
export
open   backlogged,
       getChannel;
...}
```

Transition

1. In this transition, the transmitter adds itself to the monitor's set of transmitters.

```
type GnrCsmaTransmitter : Transmitter
{
transition
       init - idle {}
       do {
       gnrCsmaTransmitters(gnrCsmaMonitor)
                     := gnrCsmaTransmitters(gnrCsmaMonitor) + {self};
        },
...}
```

2. This transition is for unicast communication. The higher layer must synchronize with the transmitter's **MAC_ready_point** event and it must provide the **messageIn** and **receiverIn** inputs.

```
type GnrCsmaTransmitter : Transmitter
{
transition
       idle - check_point {MAC_ready_point},
...}
```

3. This transition is for unicast communication. It is taken when the transmitter checks the status of the channel. The monitor must synchronize with the **getChannel** event to keep track of all the ready-to-transmit transmitters in a slot.

```
type GnrCsmaTransmitter : Transmitter
{
transition
       check_point - get_channel_point {getChannel}
       do {
              receiver := receiverIn;
              message  := messageIn;
       },
...}
```

4. This transition is for unicast communication. It is taken when the transmitter is not allowed to transmit. The monitor must synchronize with the transmitter's **backlogged** event to defer its transmission.

```
type GnrCsmaTransmitter : Transmitter
{
transition
       get_channel_point - defer_point {backlogged},
...}
```

1. This transition is for unicast communication. it is taken when the transmitter senses again the channel, one slot later.

```
type GnrCsmaTransmitter : Transmitter
{
transition
     defer_point - get_channel_point {getChannel}
     when timer = slot
     do {
          timer := 0;
     },
...}
```

2. This transition is for unicast communication. It is taken when the transmitter sends data. The monitor must synchronize with the transmitter's **MAC_data_point** event to allow it to start its transmission.

   ```
   type GnrCsmaTransmitter : Transmitter
   {
   transition
       get_channel_point - transmit_point {MAC_data_point},
   ...}
   ```
3. The transmitter issues the **MAC_confirm_point** event when the message is sent.

   ```
   type GnrCsmaTransmitter : Transmitter
   {
   transition
        transmit_point - idle {MAC_confirm_point},
   ...}
   ```
4. The following set of transitions are for multi-cast and broadcast communication. The approach is the same than for point to point communication: only event names are different.

   ```
   type GnrCsmaTransmitter : Transmitter
   {
   transition     // Multicast transitions
       idle - check_multi {MAC_ready_multi},
       check_multi - get_channel_multi {getChannel}
       do {
           receivers := receiversIn;    // Set of receivers.
           message   := messageIn;
       },
       get_channel_multi - defer_multi {backlogged},
       defer_multi - get_channel_multi {getChannel}
       when timer = slot
       do {
       timer := 0;
       },
       get_channel_multi - transmit_multi {MAC_data_multi},
       transmit_multi - idle {MAC_confirm_multi},
   ...}
   ```

   ```
   type GnrCsmaTransmitter : Transmitter
   {
   transition     // Broadcast transitions
          idle - check_broad {MAC_ready_broad},

          check_broad - get_channel_broad {getChannel}
          do {
                  message := messageIn;
          },

          get_channel_broad - defer_broad {backlogged},
   ```

```
              defer_broad - get_channel_broad {getChannel}
              when timer = slot
              do {
                     timer := 0;
              },

              get_channel_broad - transmit_broad {MAC_data_broad},

              transmit_broad - idle {MAC_confirm_broad},

              all - exit {exiting};
       }
       ...}
```

4.2.3.2.3 The CSMA Receiver

The **Receiver** subtype for point to point connection can also be used for CSMA.

4.2.3.2.4 The GnrCsma Monitor Type

The monitor begins in the *Wait* state. In every slot, it computes the number of transmitters that will be allowed to transmit in the next slot. This number is stored in **TxThisSlot**.

TxThisSlot is computed as follows.

Let the throughput S be the number of successfully delivered packets per packet transmission time Tp. Let G be the number of transmitters which wanted to transmit in the last slot (offered traffic load in packets per packet time). Let *a* be the propagation and detection delay (in packet transmission unit) that is required for all sources to detect an idle channel after transmission ends.

a is defined as *a = tau * Tp*, where *tau* is the propagation delay. The throughput of the non-persistent CSMA protocol is given by: *S = (G exp(-aG)) / (G*(1+2a) + exp(-aG))* .

Assuming that the **TheoreticalThroughput** is the number of packets that can be transmitted in one slot as a function of capacity *C* and packet length *L*, than we define **TxThisSlot** := floor (**TheoriticalThroughput** * S)

Example:
If *C*=11.2Kb/s, *L*=50Bytes, and *slot*=1sec, then the **TheoreticalThroughput** is 28 packets in one slot. Assuming that *G* is the number of transmitters which wanted to transmit in the last slot, say 4, and *a*=0.01 we get *S*=0.76 and we conclude that **TxThisSlot** = 17 packets can be successfully transmitted during the next slot.

The SHIFT description of the **Monitor** is given below.

```
type GnrCsma : Monitor
{
output set(GnrCsmaTransmitter) gnrCsmaTransmitters := {};

state  number    timer := 0;
       number    TxThisSlot := 0;  // Transmitters allowed to transmit in
                                   // the next slot.
       number    SumTx := 0;       // Transmitters which want to transmit during
```

```
                                    // the slot.

       GnrCsmaTransmitter gnrCsmaTransmitter;
                                   //Current requesting transmitter.
flow
default {
       timer' = 1;
};
setup
define {
       number  t_TransmissionTime      := PacketLenght / DataRate;
       number  t_TheoreticalThroughput := (slot / t_TransmissionTime);
}
do {
       TransmissionTime      := t_TransmissionTime;
       TheoreticalThroughput := t_TheoreticalThroughput;
};
discrete
       await,
       phase1,
       phase2;
transition
       await - await {}
       when timer = slot
       define {
              number t_SumTx      := SumTx;
              number t_Throughput := (t_SumTx * exp(-a*t_SumTx))
                                     / (t_SumTx*(1+2*a) + exp(-a*t_SumTx));
       }
       do {
              Throughput := t_Throughput;
              TxThisSlot := floor(TheoreticalThroughput * t_Throughput);
              SumTx      := 0;
              timer      := 0;
       },

       await - phase1 {gnrCsmaTransmitters:getChannel(one:t)}
       when TxThisSlot  0
       do {
              gnrCsmaTransmitter := t;
              TxThisSlot         := TxThisSlot - 1;
              SumTx              := SumTx + 1;
        },

       await - phase2 {gnrCsmaTransmitters:getChannel(one:t)}
       when TxThisSlot = 0
       do {
              gnrCsmaTransmitter := t;
              SumTx              := SumTx + 1;
       },

       phase1 - await {gnrCsmaTransmitter:MAC_data_point},
       phase1 - await {gnrCsmaTransmitter:MAC_data_multi},
       phase1 - await {gnrCsmaTransmitter:MAC_data_broad},
       phase2 - await {gnrCsmaTransmitter:backlogged};
}
```

4.2.3.3 Broadcast Channel: Stop And Wait at the Logical Link Control (using CSMA at the MAC layer)

4.2.3.3.1 Stop And Wait algorithm

At the Data Link layer we use the simplest retransmission protocol called Stop_And_Wait protocol (SWP). Whenever the receiver gets a correct packet it transmits an acknowledgment back to the sender. The sender automatically sends a copy of the packet if it does not get the acknowledgment within T seconds. The packets and the acknowledgments are numbered. The channel between the sender and the receiver is half-duplex, so the packets and the acknowledgments can not propagate at the same time.

A description of the Stop_And_Wait algorithm is given below.

```
* The algorithm at node A for A-to-B transmission
  1) Set the integer variable SN (Sequence Number) to 0.
  2) Accept a packet from the higher layer; assign SN to this new packet.
  3) Transmit the SNth packet in a frame containing SN in a sequence
     number field.
  4) If an error-free frame is received from B containing a Request
     Number greater than SN, increase SN to RN and go to step 2. If no such
     frame is received go to step 3.

* The algorithm at node B for A-to-B transmission
  1) Set the integer variable RN to 0 and then repeat step 2 and 3
     forever.
  2) Whenever an error-free frame is received from A containing a
     sequence number SN equal to RN. Release the received packet to the
     higher layer and increment RN.
  3) After receiving any error-free data frame from A, transmit a frame
     to A containing RN in the requesting number field.
```

4.2.3.3.2 State machine for Stop And Wait Logical Link Layer

Figure 7 shows the logical behavior of the **LinkLayer** subtype implementing the Stop and Wait algorithm for unicast communication. Broadcast and multi-cast communication are omitted for legible reasons.

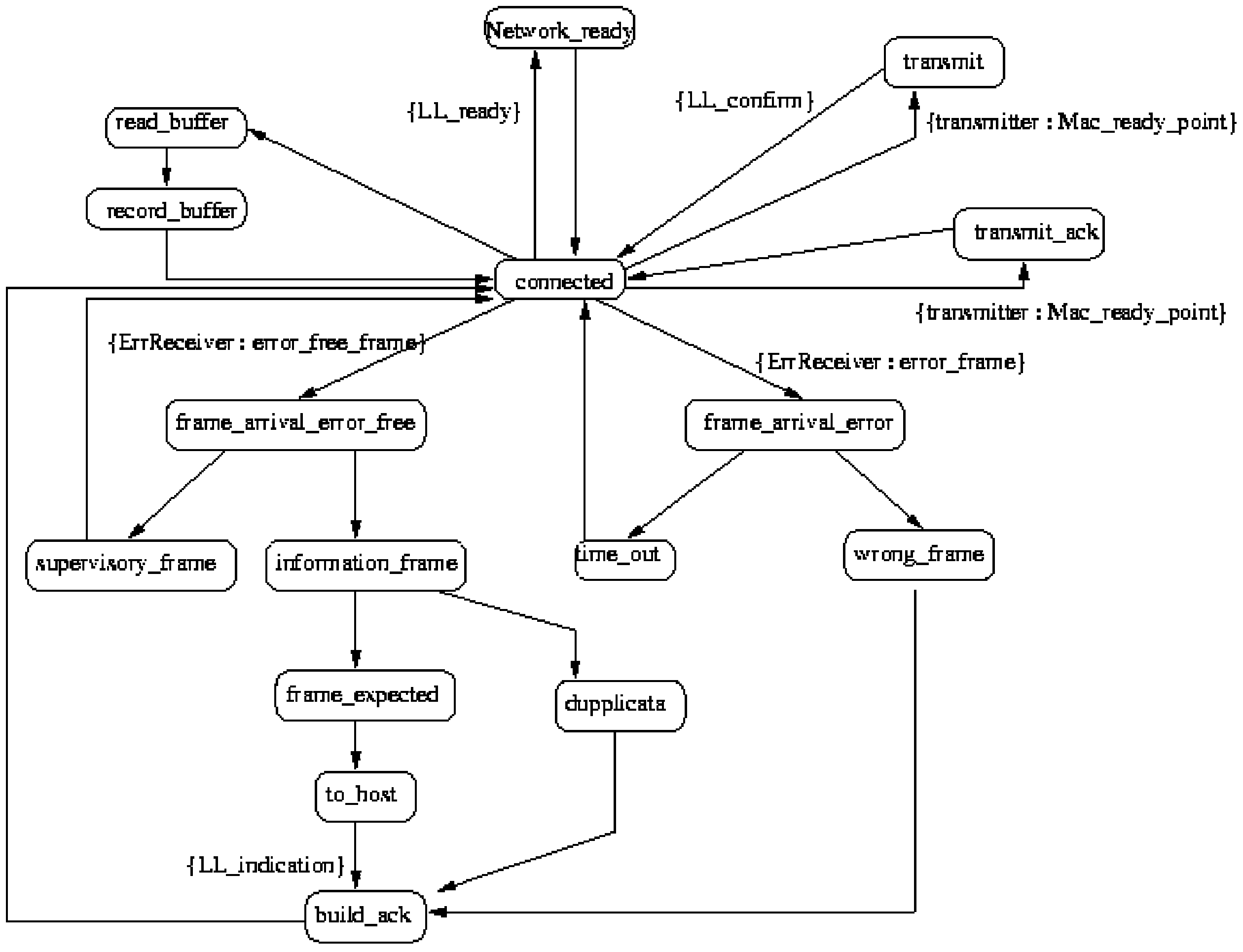


Figure 7: State machine for Logical Link Layer using Stop And Wait algorithm.

4.2.3.3.3 SHIFT description for LL_Message type

Message is subtyped to model the N-Protocol Data Unit at the Logical Link Layer.

```
type LL_Message : Message
{
output symbol          messageType;     // Supervisory or Information.
       number          sequenceNumber;  // Used by the SAW protocol.
       number          requestNumber;   // Used by the SAW protocol.

       LinkLayer       destination;        // linkLayer at the remote user.
       set(LinkLayer)  destinations := {}; // linkLayer at the remote users.
       LinkLayer        source;             // linkLayer at the local user.
       Message         message;            // N-Protocol Data Unit from
                                                   // the Network Layer.
}
```

4.2.3.3.4 SHIFT description for LinkLayer type

The SHIFT description for the Logical Link Layer using Stop_And_Wait protocol for unicast communication is given below in several fragments.

Inputs

```
type UniCsmaLinkLayer : LinkLayer
{
input   Message       messageIn;
        LinkLayer     destinationIn;
} ....
```

Outputs

```
type UniCsmaLinkLayer : LinkLayer
{
output UniErrReceiver      uniErrReceiver;
       UniCsmaTransmitter  uniCsmaTransmitter;

       LL_Message          message_from_host;
       LL_Message          message_to_host;
       set(LL_Message)     Acknowledgments := {};
       Link                link;

flow default {
       uniCsmaTransmitter = narrow (UniCsmaTransmitter, transmitter);
       uniErrReceiver     = narrow (UniErrReceiver, receiver);
       };
...}
```

States

```
type UniCsmaLinkLayer : LinkLayer
{
...
state  LL_Buffer rBuffer;
       LL_Buffer wBuffer;
       symbol    ReadyToSendData := NO;
...}
```

Discrete Transitions

1. The upper layer must synchronize with the **LL_ready** event and it must provide the **destinationIn** and **messageIn** inputs. The first and the third terms in the guard imply that the transition may be taken when the connection to **destinationIn** does not exist: a new **Link** will be created (currently there is no maximum number of connections allowed). The second term of the guard implies that the transition may be taken when the connection to **destinationIn** already exists and its read-buffer is not full. **MessageIn** is assigned to the **itemIn** input of the read-buffer.

```
type UniCsmaLinkLayer : LinkLayer
{
...
transition
       connected - network_ready {LL_ready}
       when size (Links) = 0
            or exists i in Links :
                     (destination(i) = destinationIn
                      and numberOfItems(rBuffer(i))
                                     /= bufferSize(rBuffer(i)))
            or not(exists j in Links :
                     (destination(j) = destinationIn))
define {
       LinkLayer       t_destination := destinationIn;
       Message         t_message     := messageIn;
       Link            t_link        := find{i: i in Links
                                              | (t_destination = destination(i))}
                                        default {create(Link,
                                                 sequenceNumber := 0,
                                                 requestNumber := 0,
                                                 destination := t_destination,
                                                 source := self,
                                                 bufferSize := bufferSize,
                                                 LastFrameAck := YES)};
       LL_Buffer       t_rBuffer     := rBuffer(t_link) ;
       }
do {
       itemIn(t_rBuffer) := t_message;
       rBuffer           := t_rBuffer;
       destinationIn     := nil;
       messageIn         := nil;
       },
....
}
```

2. Synchronization between the LinkLayer and the read-buffer that was chosen in the *network_ready* state. When issuing the **not_full** event, the buffer stores the message that was written in its **itemIn** input.

```
type UniCsmaLinkLayer : LinkLayer
{
...
transition
       network_ready - connected {rBuffer : not_full},
....
}
```

3. The linkLayer is looking for a new message to transmit. The first term of the guard implies that the transition is taken only if the previous **message_from_host** has been transmitted; The second term of the guard implies that at least a connection has its last frame acknowledged and another frame to transmit. In the *read_buffer* state, the linkLayer records the chosen connection in its **link** output.

```
type UniCsmaLinkLayer : LinkLayer
{
...
transition
       connected - read_buffer {}
       when ReadyToSendData = NO
            and (exists i in Links : LastFrameAck(i) = YES
                                     and (numberOfItems(rBuffer(i)) /= 0))
       define {
              Link t_link := find {i : i in Links
                                     | LastFrameAck(i) = YES
                                       and (numberOfItems(rBuffer(i)) /= 0)};
              LL_Buffer t_rBuffer  := rBuffer(t_link) ;
       }
       do {
              rBuffer              := t_rBuffer;
              link                 := t_link;
              ReadyToSendData      := YES;
        },
....
}
```

4. The linkLayer is looking for a new message to transmit. This transition is parallel to the previous one. The first term of the guard implies that the transition is taken only if the previous **message_from_host** has been transmitted; The second term of the guard implies that at least a connection has its last frame **NOT** acknowledged. In the *read_buffer* state, the linkLayer records the chosen connection in its **link** output.

```
type UniCsmaLinkLayer : LinkLayer
{
...
transition
       connected - read_buffer {}
       when ReadyToSendData = NO
            and (exists j in Links : LastFrameNack(j) = YES
                                     and (numberOfItems(rBuffer(j)) /= 0))
       define {
              Link t_link := find {i : i in Links
                                     | LastFrameNack(i) = YES
                                       and (numberOfItems(rBuffer(i)) /= 0)};
              LL_Buffer t_rBuffer  := rBuffer(t_link) ;
       }
       do {
              rBuffer              := t_rBuffer;
              link                 := t_link;
              ReadyToSendData      := YES;
        },
....
}
```

5. Synchronization between the linkLayer and the read-buffer that was chosen in the *read_buffer* state. When issuing the **not_empty** event, the buffer saves the next frame to send in its **itemOut** output.

```
type UniCsmaLinkLayer : LinkLayer
{
...
transition
       read_buffer - record_buffer {rBuffer : not_empty},
....
}
```

6. The linkLayer copies the next message to send from the buffer, and creates a **LL_message** with the informations needed by the remote linkLayer.

```
type UniCsmaLinkLayer : LinkLayer
{
...
transition
       record_buffer - connected {}
       define {
              Message t_message         := itemOut(rBuffer);
              LL_Message t_LL_message   := create(LL_Message,
                                                  messageType := I,
                                                  source := self);
       }
       do {
              message_from_host        := t_LL_message;
              message(t_LL_message)    := t_message;
              source(t_LL_message)     := source(link);
              destination(t_LL_message):= destination(link);
              sequenceNumber(t_LL_message)
                                       := modulo(sequenceNumber(link), 2);
              requestNumber(t_LL_message)
                                       := modulo(requestNumber(link), 2);
              LastFrameAck(link)       := NO;
              LastFrameNack(link)      := NO;
              ReadyToSendData          := YES;
              link                     := nil;
              rBuffer                  := nil;
        },

....
}
```

7. When the linkLayer is ready to send the next frame, it must synchronize with the transmitter's **MAC_ready_point** event.

```
type UniCsmaLinkLayer : LinkLayer
{
...
transition
       connected - transmit {uniCsmaTransmitter : MAC_ready_point}
       when ReadyToSendData = YES
       do {
              messageIn(uniCsmaTransmitter) := message_from_host;
              receiverIn(uniCsmaTransmitter)
                         := receiver(destination(message_from_host));
              ReadyToSendData := NO;
        },
...}
```

8. When the message has been sent the linkLayer issues the **LL_confirm** closed event.

```
type UniCsmaLinkLayer : LinkLayer
{
...
transition
       transmit - connected {LL_confirm}
       do {
              message_from_host := nil;
       },
...}
```

9. A message has been received error free by the receiver. There is synchronization on the **error_free_frame** event between the receiver and the linkLayer. The linkLayer stores the new message in its output variable **message_to_host**.

```
type UniCsmaLinkLayer : LinkLayer
{
...
transition
       connected - frame_arrival_error_free {uniErrReceiver : error_free_frame}
       define {
              Message t_message       := message(uniErrReceiver);
              LL_Message t_ll_message := narrow(LL_Message, t_message);
       }
       do {
              message_to_host         := t_ll_message;
       },
...}
```

10. If the message is a supervisory one the linklayer goes in the *supervisory_frame* state (acknowledgment from the remote linkLayer). The linkLayer copies in its **link** output the link corresponding to the connection.

```
type UniCsmaLinkLayer : LinkLayer
{
...
transition
       frame_arrival_error_free - supervisory_frame {}
       when messageType(message_to_host) = $S
       define {
              LinkLayer t_source := source(message_to_host);
              Link      t_link   := find {i : i in Links
                                         | (destination(i)
                                           = t_source)};
       }
       do {
              link               := t_link;
              rBuffer            := rBuffer(t_link);
       },
...}
```

11. The message acknowledges the last sent frame; the linkLayer goes back to the *connected* state by synchronizing with the specific buffer's **cancel** event, and remembers that the last sent frame (for this connection) was acknowledged.

```
type UniCsmaLinkLayer : LinkLayer
{
...
transition
       supervisory_frame - connected {rBuffer : cancel}
       when (requestNumber(message_to_host) =
             modulo((sequenceNumber(link) + 1), 2))
       define {
              Link t_link            := link;
       }
       do {
              sequenceNumber(t_link) := requestNumber(message_to_host);
              LastFrameAck(t_link)   := YES;
              LastFrameNack(t_link)  := NO;
              link                   := nil;
              rBuffer                := nil;
              message_to_host        := nil;
       },
...}
```

12. The message does not acknowledge the last sent frame; the linkLayer goes back to the *connected* state, and remembers that the last sent frame (for this connection) was not acknowledged. The remote user asks for a retransmission.

```
type UniCsmaLinkLayer : LinkLayer
{
...
transition
       supervisory_frame -  connected {}
       when  (sequenceNumber(link) = requestNumber(message_to_host))
       define {
             Link t_link           := link;
       }
       do {
             LastFrameNack(t_link) := YES;
             link                  := nil;
             rBuffer               := nil;
             message_to_host       := nil;
       },
...}
```

13. The linklayer goes in the *information_frame* state if the message is an information message (data from the remote linkLayer). In this state, the linkLayer determines which link, among those in its set, is the one which corresponds to this connection; if no such link exists, the linkLayer creates one.

```
type UniCsmaLinkLayer : LinkLayer
{
...
transition
       frame_arrival_error_free - information_frame {}
       when messageType(message_to_host) = $I
       define {
              LinkLayer t_source := source(message_to_host);
              Link      t_link   := find {i: i in Links
                                          | t_source = destination(i)}
                                 default {create(Link,
                                                 sequenceNumber := 0,
                                                 requestNumber := 0,
                                                 destination := t_source,
                                                 source := self,
                                                 bufferSize := bufferSize,
                                                 LastFrameAck := YES,
                                                 LastFrameNack := NO)};
       }
       do {
              link := t_link;
              wBuffer := wBuffer(t_link);
       },
...}
```

14. The message was expected; The linkLayer passes it to the write-buffer that was chosen in *information_frame*.

```
type UniCsmaLinkLayer : LinkLayer
{
...
transition
       information_frame - frame_expected {}
       when sequenceNumber(message_to_host)
                                 = requestNumber(link)
       do {
              itemIn(wBuffer)     := message(message_to_host);
              requestNumber(link) := modulo((requestNumber(link) + 1), 2);
       },
...}
```

15. Synchronization between the linkLayer and the specific write-buffer. When issuing the **not_full** event, the buffer saves the message that was written in itemIn.

```
type UniCsmaLinkLayer : LinkLayer
{
...
transition
       frame_expected -  to_host {wBuffer : not_full}
       do {
              message_to_host := nil;
       },
...}
```

16. The **LL_indication** event is issued when a message has arrived error free. In the *build_ack* state, the linkLayer creates a supervisory message to acknowledge the received message and to ask for the next one.

```
type UniCsmaLinkLayer : LinkLayer
{
...
transition
       to_host - build_ack {LL_indication}
       define {
              LL_Message t_acknowledgment
                         := create(LL_Message,
                                   messageType    := S,
                                   source         := self,
                                   destination    := destination(link),
                                   sequenceNumber := sequenceNumber(link),
                                   requestNumber  := requestNumber(link));
       }
       do {
              Acknowledgments := Acknowledgments + {t_acknowledgment};
       },
...
}
```

17. The frame is not expected. It is a duplicate.

```
type UniCsmaLinkLayer : LinkLayer
{
...
transition
       information_frame - duplicated_frame {}
       when sequenceNumber(message_to_host)
                              = modulo((requestNumber(link) + 1), 2)
       do {
              message_to_host := nil;
       },
...
}
```

18. In the *build_ack* state, the linkLayer creates a supervisory message asking for the next frame.

```
type UniCsmaLinkLayer : LinkLayer
{
...
transition
       duplicated_frame - build_ack {}
       define {
              LL_Message t_acknowledgment
                         := create(LL_Message,
                                   messageType    := S,
                                   source         := self,
                                   destination    := destination(link),
                                   sequenceNumber := sequenceNumber(link),
                                   requestNumber  := requestNumber(link));
       }
       do {
              Acknowledgments := Acknowledgments + {t_acknowledgment};
       },
...
}
type UniCsmaLinkLayer : LinkLayer
{
...
transition
       build_ack - connected {}
       do {
              link    := nil;
              rBuffer := nil;
       },

...
}
```

19. The linkLayer synchronizes with the transmitter's **MAC_ready_point** event to send an acknowledgment.

```
type UniCsmaLinkLayer : LinkLayer
{
...
transition
       connected - transmit_ack {uniCsmaTransmitter : MAC_ready_point}
       when size(Acknowledgments) /= 0
       define {
              LL_Message t_acknowledgment  := choose{i : i in Acknowledgments};
              LinkLayer  t_destination     := destination(t_acknowledgment);
       }
       do {
              messageIn(uniCsmaTransmitter)  := t_acknowledgment ;
              receiverIn(uniCsmaTransmitter) := receiver(t_destination);
              Acknowledgments                := Acknowledgments
                                                  - {t_acknowledgment};
       },
...}
type UniCsmaLinkLayer : LinkLayer
{
...
transition
       transmit_ack - connected {},
...}
```

20. A message has been received with an error by the receiver. There is synchronization with the receiver's **error_frame** event. The linkLayer saves the new message on its output **message_to_host**.

```
type UniCsmaLinkLayer : LinkLayer
{
...
transition
       connected - frame_arrival_error {uniErrReceiver : error_frame}
       define {
              Message t_message       := message(uniErrReceiver);
              LL_Message t_ll_message := narrow(LL_Message, t_message);
        }
       do {
              message_to_host         := t_ll_message;
       },
...}
```

21. The wrong message is a supervisory message. The linkLayer seeks the link corresponding to this connection and remembers that the last frame, for this connection, was not acknowledged. This mechanism allows us to model the timeout (the acknowledgment was lost or is wrong). The sender has to resend the last frame.

```
type UniCsmaLinkLayer : LinkLayer
{
...
transition
       frame_arrival_error - time_out {}
       when messageType(message_to_host) = S
```

```
        define {
               LinkLayer t_source := source(message_to_host);
               Link      t_link   := find {i : i in Links
                                          | (destination(i)
                                            = t_source)};
        }
        do {
              link := t_link;
              LastFrameNack(t_link) := YES;
        },
...}
type UniCsmaLinkLayer : LinkLayer
{
...
transition
        time_out - connected {}
        do {
               link            := nil;
               rBuffer         := nil;
               message_to_host := nil;
        },
...}
```

22. The wrong message is an information message. The linkLayer seeks the link corresponding to this connection or creates one. The resulting link is then stored in the link output.

```
type UniCsmaLinkLayer : LinkLayer
{
...
transition
        frame_arrival_error - wrong_frame {}
        when messageType(message_to_host) = I
        define {
               LinkLayer t_source := source(message_to_host);
               Link      t_link
                          := find {i : i in Links
                                    | (destination(i) = t_source)
                             default {create(Link,
                                            sequenceNumber := 0,
                                            requestNumber  := 0,
                                            destination    := t_source,
                                            source         := self,
                                            bufferSize     := bufferSize,
                                            LastFrameAck   := YES,
                                            LastFrameNack  := NO)};
        }
        do {
               link := t_link;
        },
...}
```

23. The frame was not received correctly by the linkLayer. A retransmission is needed.

```
type UniCsmaLinkLayer : LinkLayer
{
...
transition
        wrong_frame - build_ack {}
        define {
               LL_Message t_acknowledgment
                          := create(LL_Message,
                                    messageType    := S,
                                    source         := self,
                                    destination    := destination(link),
                                    sequenceNumber := sequenceNumber(link),
                                    requestNumber  := requestNumber(link));
        }
        do {
               Acknowledgments := Acknowledgments + {t_acknowledgment};
        },
 ...}
 type UniCsmaLinkLayer : LinkLayer
 {
 ...
 transition
        all - exit {exiting};
 ...}
```

4.2.3.4 Broadcast channel: Token Ring at the MAC layer

4.2.3.4.1 Token Ring algorithm

In a Token Ring network the hosts are connected in the shape of a ring. The protocol is based on the use of a small frame called token, that circulates when all users are idle. A ready-to-transmit user has to wait until it detects the next available token as it passes by. When a station seizes a token and begins to transmit a data frame, there is no token in the ring, so other users wishing to transmit must wait. The transmitting user will insert a new token on the ring when both of the following conditions are met: the user has completed the transmission of its frame and an acknowledgment from the destination of the frame has been received.

In order to implement the basic functionalities of the token ring protocol the **Transmitter** type and the **Monitor** type have been subtyped with **UniTokenTransmitter** and **UniToken**.

Currently only unicast communication is implemented. The following features of the Token Ring protocol are not implemented:
1) there is no initialization sequence. When a user joins the network, it should go through an initialization sequence to become part of the ring. This is done to inform the neighbors of its existence.
2) there is no *active monitor*. The active monitor is a host on the network, usually the first recognized station when the LAN comes up. It watches over on the network and looks for problems. The *active monitor* basically makes sure the network works efficiently and without errors.
3) if the *active monitor* should fail, other users should be available to take its place.
To implement these features, the monitor and the transmitter subtypes for Token Ring must be updated. A description of the Token Ring algorithm we used is given below.

```
* The model of a transmitter A is:
  A receives the token.
  A wants to send a message to B
  A sends a unicast message to B
  A waits for an acknowledgment from B
  if A receives an acknowledgment, then
    A deletes the frame
    and sends the token to the next user in the list
  else A sends the token to the next user in the list after the timeout.
* The model of a receiver B is:
  B receives a message from A
  B computes the CRC and compares it with that inside the message
  If the CRCs are equal, B sends back an acknowledgment and forwards the
    message to the upper layer
  else it deletes the message
```

4.2.3.4.2 The UniTokenTransmitter type

Figure 8 shows the logical behavior of the Token Ring subtype of **Transmitter** for unicast communication. When creating the transmitter, the **monitor** output variable must be properly initialized. In its setup action, the transmitter adds itself to the **Transmitters** output variable of the monitor. This variable is a set of **Transmitter** which is used by the monitor to keep track of the transmitters in the network.

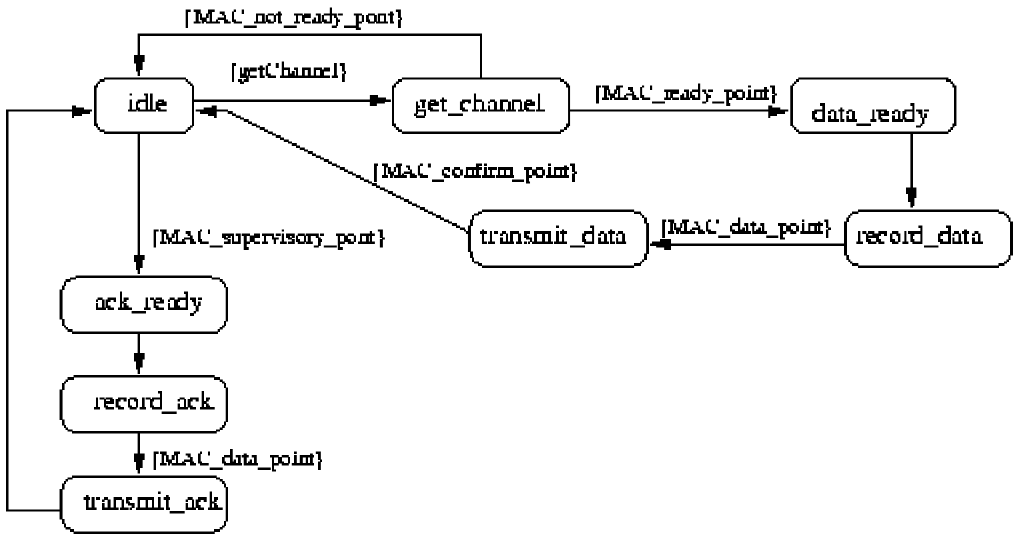


figure 8: State machine for Token Ring transmitter.

The SHIFT description of the **UniTokenTransmitter** for unicast is given below in several fragments.

Output

```
type UniTokenRingTransmitter : Transmitter
{
output UniToken uniTokenMonitor;     // Monitor involved in the network.
       number   priority;            // Priority of transmitter.
...}
```

Exported events

```
type UniTokenRingTransmitter : Transmitter
{
export open
       MAC_not_ready_point,
       MAC_supervisory_point,
       getChannel;
...}
```

Setup action

The transmitter adds itself to the monitor's set of transmitters. This action can also be done in a init - idle transition, like for the CSMA transmitter.

```
type UniTokenRingTransmitter : Transmitter
{
setup do {
       uniTokenTransmitters(uniTokenMonitor) :=
                     uniTokenTransmitters(uniTokenMonitor) + {self};
       };
...}
```

Transition

1. The monitor must synchronize with the **getChannel** event to symbolize the reception of the token by a transmitter.

```
type UniTokenRingTransmitter : Transmitter
{
transition
       idle - get_channel {getChannel}
       when self = nextTransmitter(uniTokenMonitor),
...}
```

2. The transmitter is ready to transmit. The higher layer must synchronize with the **MAC_ready_point** event and provide the **messageIn** and **receiverIn** inputs.

```
type UniTokenRingTransmitter : Transmitter
{
transition
       get_channel - data_ready {MAC_ready_point}
       do {
              channelStatus(uniTokenMonitor) := BUSY;
       },
...}
```

3. The transmitter saves the message to send in its **message** output and the destination in its **receiver** output.

```
type UniTokenRingTransmitter : Transmitter
{
transition
       data_ready - record_data {}
       do {
              receiver := receiverIn;
              message  := messageIn;
        },
...}
```

4. The transmitter is ready to transmit. If there is no message to transmit the higher layer must synchronize with the **MAC_not_ready_point** event.

```
type UniTokenRingTransmitter : Transmitter
{
transition
       get_channel - idle {MAC_not_ready_point},
...}
```

5. The monitor must synchronize with the transmitter's **MAC_data_point** event to start the transmission.

```
type UniTokenRingTransmitter : Transmitter
{
transition
       record_data - transmit_data {MAC_data_point},
...}
```

6. After transmitting the message, the transmitter issues the **MAC_confirm_point** event.

```
type UniTokenRingTransmitter : Transmitter
{
transition
       transmit_data - idle {MAC_confirm_point},
...}
```

7. The transmitter is ready to transmit an acknowledgment. In this case, the transmitter doesn't wait for the token in order to transmit. The higher layer must synchronize with the

transmitter's **MAC_supervisory_point** event and it must provide the **messageIn** and **receiverIn** inputs.

```
type UniTokenRingTransmitter : Transmitter
{
transition
       idle - ack_ready {MAC_supervisory_point},
...}
```

8. The transmitter saves the message to send and the destination in its **message** and **receiver** outputs.

```
type UniTokenRingTransmitter : Transmitter
{
transition
       ack_ready -record_ack  {}
       do {
              receiver := receiverIn;
              message  := messageIn;
       },
...}
```

9. When transmitting the acknowledgment, the transmitter issues the **Mac_data_point** event.

```
type UniTokenRingTransmitter : Transmitter
{
transition
       record_ack - transmit_ack {MAC_data_point}
       do {
              channelStatus(uniTokenMonitor) := IDLE;
       },
...}

type UniTokenRingTransmitter : Transmitter
{
transition
        transmit_ack - idle {},
...}

type UniTokenRingTransmitter : Transmitter
{
transition

        all - exit {exiting};
...}
```

4.2.3.4.3 The UniToken Monitor Type

In every slot, the monitor allows a fixed number of transmissions. This number is called **TheoreticalThroughput** and is calculated in the setup action. The SHIFT description of the **Monitor** is given below in several fragments.

Output

The **channelStatus** flag is set to BUSY when a transmitter begins to transmit a message, and is set back to IDLE when the transmitter receives an acknowledgment to its message.

```
type UniToken : Monitor
{
output set(UniTokenTransmitter) uniTokenTransmitters := {};
       UniTokenTransmitter      nextTransmitter;
       number networkSize    := 0;
       symbol channelStatus  := $IDLE;
...}
```

State

The state variable **nextTx** refers to the next transmitter that is allowed to transmit and the state variable **TxThisSlot** is the number of transmissions available in the current slot.

```
type UniToken : Monitor
{ ...
       number timer  := 0;
       number nextTx := 0;
       number TxThisSlot;

flow   default { timer' = 1;};
...}
```

Setup action

In the setup action the monitor computes the number of transmissions per slot.

```
type UniToken : Monitor
{
setup  define {
              number t_TransmissionTime       := PacketLenght / DataRate;
              number t_TheoriticalThroughput := (slot / t_TransmissionTime);
       }
       do {
              TransmissionTime       := t_TransmissionTime;
              TheoriticalThroughput := t_TheoriticalThroughput;
       };
...}
```

Transition

1. In every slot, we update the variable **TxThisSlot**.

```
type UniToken : Monitor
{
transition
       await - await {}
       when timer = slot
       do {
              TxThisSlot := TheoriticalThroughput;
              timer      := 0;
       },
...}
```

2. When the slot is not full, the monitor seeks the next transmitter that's allowed to transmit.

```
type UniToken : Monitor
{
       await - phase1 {}
       when TxThisSlot  0
            and size(uniTokenTransmitters) /= 0
       do {
              nextTransmitter := find { i : i in uniTokenTransmitters
                                       | priority(i) = nextTx};
              networkSize     := size(uniTokenTransmitters);
       },
...}
```

3. Synchronization with the specific transmitter's **getChannel** event.

```
type UniToken : Monitor
{
transition
       phase1 - phase2 {nextTransmitter : getChannel}
       when channelStatus = IDLE
       do {
              TxThisSlot := TxThisSlot - 1;
              nextTx     := modulo((nextTx + 1), networkSize);
       },
...}
type UniToken : Monitor
{
transition
       phase2 - await {};
...}
```

4.2.3.5 Stop And Wait at the LLC (using Token Ring at the MAC layer)

4.2.3.5.1 SHIFT description for Logical Link Layer

The Token Ring **linkLayer** is very similar to the CSMA **linkLayer**. Only the following two transitions are actually different.

Discrete Transitions

1. To send an acknowledgment the Logical Link Layer synchronizes with the transmitter's **MAC_supervisory_point** event.

```
type UniTokenRingLinkLayer : LinkLayer
{
...
transition
       connected - transmit_ack {uniTokenTransmitter : MAC_supervisory_point}
       when size(Acknowledgments) /= 0
       define {
              LL_Message t_acknowledgment := choose{i : i in Acknowledgments};
              LinkLayer  t_destination    := destination(t_acknowledgment);
       }
       do {
              messageIn(uniTokenTransmitter)  := t_acknowledgment ;
              receiverIn(uniTokenTransmitter) := receiver(t_destination);
              Acknowledgments := Acknowledgments - {t_acknowledgment};
       },

       transmit_ack - connected {},
....
}
```

2. When the transmitter is allowed to transmit (i.e. after receiving the token) but there's no new message to send, the Logical Link Layer synchronizes with the transmitter's **MAC_not_ready_point** event.

```
type UniTokenRingLinkLayer : LinkLayer
{
...
transition
       connected - connected {uniTokenTransmitter : MAC_not_ready_point}
       when ReadyToSendData = NO,
...
}
```

4.3 Human Driver Model Integration

Longitudinal and lateral control models of the human driver at the regulation level (throttle, brake, steering) have been implemented in SmartAHS. In this MOU, we propose to implement strategic decision making models and human perception models developed under the auspices of NAHSC and MOU 284 into SmartAHS to complete the picture of human driving.

This functionality will help us simulate mixed traffic, partial automation and driver assist functions and assess their safety and comfort impacts.

We describe the human driver models which we are incorporating into our vehicle-highway microsimulator tool, SmartAHS. We discuss them in context of the well-known perception-decision making guidance framework, drawing from previous work in each of these areas. In particular, we select from the variety of target acquisition models; implement well-known tracking model; and we apply the R.W. Allen's crossover model for steering and throttle control. Our objective was to implement human diver models balanced with suitable complexity, yet with enough computational simplification, to adequately describe the evolution of driver-assist longitudinal control, beginning with adaptive cruise control. As a result of ongoing review process, the best available models for different aspects of human driving were chosen. These models are being represented in the SHIFT programming language and added to the SmartAHS modeling environment. At this time the family of crossover control models are fully integrated within SmartAHS. Decision making models are being constructed on case by case basis, and some models are available within SmartAHS as a direct result of case studies conducted at PATH. Currently the group is implementing the set of perception models of choice: BR-based perception models.

4.3.1 Introduction

Since its inception, the AHS program has undergone a gradual refocusing toward deployment of intermediate systems. Many of these systems are directed at improving highway safety. Our current efforts reflect this redirection and center around near-to mid-term, deployable "partially automated" or, driver assist and vehicle-highway cooperative systems, the type which is under consideration for the upcoming US DOT Intelligent Vehicle Initiative. The long term goal remains vehicle-highway automation, but to achieve this, a progression of short-term stepping stones must be mapped. The SmartAHS microsimulation tool is being used to assess the dynamics and interaction of emerging driver-assist devices and features such as adaptive cruise control (ACC), forward collision warning (FCW) or avoidance (FCA) and other potential collision warning and avoidance systems. The microsimulation approach - where the detailed driver-vehicle interactions are executed at a fine level of granularity with multiple vehicles - brings the appropriate understanding of the physiology, physics and control laws necessary to determine the efficacy of these new candidate systems. On the implementation side, we consider the SmartAHS simulation framework developed at the California PATH program . A detailed and validated model of a human driver is the foundation of our SmartAHS tool to address the pre-AHS deployment issues of safety, driver assist and even "mature" AHS concepts which include mixed traffic.

4.3.2 Background: SmartAHS Simulation Framework

The SmartAHS simulation framework addresses the needs of several categories of users: model developers who provide detailed vehicle, sensor, control, and communication device libraries; system engineers who will develop automation architectures; control and communication engineers who will design, implement, and test individual control and communication components; system analysts who will test and evaluate automation strategies; and system planners who will select the automation strategy for deployment based on evaluation results. Along with traditional software engineering requirements, the framework was developed to satisfy the following requirements: it must provide time and event driven evolution models; it must allow the designers to use a specification language that fits their domain, in this case differential equations and finite state machines; it must provide a structured specification, simulation, and evaluation environment with formal semantics; it must provide constructs that are object-oriented; and it must represent dynamic interaction dependencies. Additionally, the framework must model a number of entities that are particular to simulating vehicles on the highway. These entities must be able to represent arbitrary highways; incoming and outgoing traffic patterns; vehicles consisting of many components; roadside controllers consisting of many components; different types of vehicles on the highway; inter-vehicle and vehicle-to-roadside communication; accidents; and collection of arbitrary statistics.

The SmartAHS simulation framework is written in SHIFT programming language. SHIFT combines system-theoretic concepts into one consistent and uniform programming language with object-oriented features. It is ideal for the design, specification, simulation, control and evaluation of large dynamical systems that consist of multiple interacting agents whose behaviours are described by a combination of state machines and ordinary differential equations. SmartAHS consists of different building blocks which facilitate quick development of a simulation tailored to particular case scenario.

The highway library provides building blocks to create arbitrary highways. The roadway is represented in terms of components of types Section, Segment, Lane. Sections consist of segments and lanes. Segments represent the geometry of a highway. Highway types reside in the global coordination frame meaning that each point of the highway is referenced by its global coordinates. Each section has its own coordinate frame. Sink and Source types provide the flows of vehicles facilitating the representation of desired Origin-Destination patterns and flow volumes. A basic set of Monitor types collects necessary statistics used for concept evaluation. An arbitrary vehicle consists of several types. It contains VehicleDynamics which models vehicle dynamics at the desired level of detail, a Controller which is to provide the throttle, steering, and brake inputs, which behavior depends on either it is automated or human controller; and a Vehicle-Roadway Environment Processor (VREP).

The VREP is a logical object that maintains a Vehicle's position on the highway. It performs the coordinate translations between the vehicle coordinate frame, and the roadway and global coordinate frames. The Automated Vehicle also contains Sensor and Communication devices and their environment processors. Additional SmartAHS libraries provide a set of foundation classes for sensors and the communication infrastructure. A vehicle type may be augmented by Sensor Environment Processor. Completely different sets of the communication and sensor libraries may be provided by other development groups, and this interchange does not affect other blocks of the SmartAHS framework. In modeling a human driver the sensor libraries are augmented by perception libraries which combined together represent human perception. The SmartAHS simulation framework can be downloaded from the PATH web page: http://www.path.berkeley.edu/SmartAHS.

4.3.3 Human Driver Models

Several taxonomies have been developed describe the sequence of "normal", i.e., non-emergency, driving actions: perception-decision making-control, navigation-guidance-control, and the 3T architecture. The perception-decision making-control taxonomy was derived to describe localized, near-term driver actions; the navigation-guidance control conceptual framework was developed to bridge higher level goal-setting or navigation activities with near-term driver or control actions via monitoring, or guidance. The 3T architecture navigates an automaton via a succession of micro-level skills, then progressing to tactical and strategic levels. Because our application is aimed at assessing the safety of driver-assist devices, our current focus is on the local (or borrowing from the 3T lexicon, tactical) layer. We bypass the guidance or strategic levels, and we therefore base our model structure on the perception-decision making-control hierarchy.

For simplicity, we assume an alerted driver - one who is already vigilant, attentive and monitoring. The specifics of objectively modeling these precursors to perception is a complex undertaking, and one that we defer. An advantage of not addressing this now is that by considering only an already-alerted driver, we sidestep issues in warning and human machine interface design for now; we can therefore initially focus on driver reactions and responses to exogenous disturbances from outside the vehicle. Perception Models Two elements of perception are considered in our human vision- and cognition-based detection models: acquisition (defined for our purposes as proximal obstacle or vehicle detection probabilityat range x) and tracking (defined for our purposes as deceleration relative to the driver). We show currently implemented models and anticipated developments for our SmartAHS microsimulation.

4.3.3.1 Acquisition Models

We aim to provide progressively higher fidelity models in human vision-based target acquisition by following a work plan of implementing a sequence of three detection probability models. We have reviewed the Bailey-Rand (BR) Contrast Model, the Doll-Schmieder (DS) Model, and the National Automotive Center-Visual Performance Model (VPM). These models will sequentially yield higher confidence results as we begin analyzing more specific CAS implementations and scenarios.

4.3.3.1.1 Bailey-Rand Contrast and Similar Models.

The BR model incorporates, in a compact manner, the first- order effects of luminance contrast. To do so in a compact manner, it assumes that targets are static and can be represented by circles with varying contrasts to an appreciably uniform (and therefore uncluttered) background. The BR model also includes a target visibility factor. Limitations of the BR model primarily include the aforementioned assumption of static, circular targets. (Some targets such as stalled vehicles and stationary objects are certainly static, but the driver's vehicle is normally moving, nominally at 30m/s or 10m during the 1/3s glimpse.) Moreover, the surround is not clutter-free, and scenes have considerably more spatial, spectral (such as color) and temporal features that contribute toward target discrimination, e.g., transient glare. These assumptions may be less far-fetched under certain constrained scenarios such as with a limited search within a rural highway and under relatively high but diffuse luminance. However, they have been successfully applied in DoD applications, on non-circular targets and on natural backgrounds with considerably more clutter than many highway scenes. For this reason, we have enough confidence in the BR model to use it under carefully designed scenarios. We plan to improve the BR formulation by incorporating an explicit driver search model. Because of its empirical foundation, we will maintain the independent 1/3s BR glimpses. Head dwell and gaze abduction behavior measurements have been conducted for driving, and at intersections, but there are very few field experiments on which to build driver visual search models.

The BR model differs from the Visibility Index and Visibility Index/Fog (VI/FOG) models used in the NHTSA-sponsored Perception-Decision-Response framework constructed to assess causes of reduced visibility crashes, in that the BR model more rigorously defines meteorological parameters affecting detectability. The VI/FOG models, however, take glare from artificial illumination into consideration, including streetlights. An improvement to the psychophysics embedded within the VI/FOG models is represented by the PCDETECT model, which takes into account driver age and glare. All three competing models (BR, VI/FOG, PCDETECT) are based on data from the classical Blackwell experiments, which relate differences in visual contrast over a wide range of illumination conditions. In these experiments, targets are circular, the background is uniform, and the target-to-background discriminant is luminance (i.e., gray scale) contrast. As these assumptions exist within the other models, the common foundation makes use of any of the three similar classes of models almost equally valid. We prefer the BR model because it can be compactly expressed, and it is acknowledged outside the transportation safety community. However, we believe that a more robust and higher fidelity representation of the human visual system must be employed to reduce systematic errors from the simplifying assumptions of the BR and like models.

4.3.3.1.2 Doll-Schmieder and Other TSD Models.

The DS model overcomes clutter limitations and introduces a theoretical framework-the Theory of Signal Detection (TSD) - for detection decision-making. This framework is implicit with Blackwell's relationships. The use of TSD puts the DS model into class of acquisition models commonly used for in sensor processing.

To make the application of TSD in modeling detectivity by humans complete, the models must be populated with specific background imagery, then human jury tests to determine ROC's must be conducted. Hence, to make this model applicable for highways, appropriate vehicle, obstacle and highway scene data must be gathered and fit. The data set could be large, as variations such as the diurnal cycle, different highway topologies and obstacle types must be considered;

however, given a contained and very specific scenario, a reasonable data set with a high degree of realistic visual cues could be collected, i.e., glint, glare and other spatially or temporally unique features. Although the DS model overcomes the simplified target and background assumptions of the BR model, it still applies to stationary objects. Additionally, vehicle motion and the decision whether the detected object is an obstacle are still not addressed by this model.

4.3.3.1.3 National Automotive Center - Visual Performance Model.

The VPM introduces effects of color and motion with mathematical representations of the early vision process from the receipt of photons on the retina through response by the photoreceptive fields, and concluding with TSD for the target detection decision. The VPM combines aspects of preattentive vision and human reasoning to model the human visual/detection system. As with the human, VPM produces sequential channels for color (obtained by dividing the image into color-opponent channels), motion (obtained by temporal filtering), spatial frequency (obtained by transforming images into two-dimensional frequency space scenes) and orientation (obtained by performing horizontal and vertical filtering). The model produces a signal-to-noise ratio for each channel, then summed over all channels to essentially produce a ROC, and from TSD. Mathematical models such as VPM have only recently been formulated to emulate the human early vision process. Currently, the VPM search model is primarily based on human visual search in cluttered combat backgrounds and to some extent, at roadway intersections; this may not be extendible to highway driving without adjustment.

4.3.3.1.4 Tracking Model

In light of our near term objective of determining the efficacy of longitudinal control and warning, we focus our tracking models to longitudinal acceleration only. To be complete, we would also address tracking for objects moving laterally across the field of view, e.g., deer moving on the roadway, or cars crossing at an intersection. Such a model would be complex, and look up tables of empirically derived eye tracking data might be the most appropriate means to approach the problem. At the present time, we do not anticipate embedding a more complete lateral tracking model in SmartAHS. Time to collision (TTC) estimates are obtained from empirical studies of variations from the population norm. This measure is a standard input to highway design standards. In recent years, however, the question of how drivers gauge TTC has arisen. It is now known that the human visual system is sensitive to the looming angular target size and its rate of increase (or decrease), but it is not certain whether the driver is directly sensitive to, or equivalently, TTC. and d is the forward vehicle or obstacle diameter.

4.3.3.1.5 Decision Making Models

Decision making is a difficult and very situationally dependent process to model. It involves elements of the roadway condition and configuration, vehicle performance, the nature and perceived intent of proximate vehicles, and quite importantly, the mood, inherent aggressiveness and possible impairment of the driver. Various methods to incorporate risk, and complex rule-based approaches to factor the contribution of driver risk have been proposed. However, utilizing a simple risk model or applying a rule-based approach is oftentimes inflexible to a particular driving situation or event. More recently, a promising computer science/artificial intelligence-inspired genetic algorithm to "grow" successful tactical driver behavior traits has been developed, but such a scheme was intended to produce highly successful and even selfish autonomous vehicle controllers, and not necessarily a human driver decision making model. Efforts are currently underway to evolve an algorithm with more typical human driving behaviors, but such an effort will require substantial validation. We consider incorporating this work once it becomes available. We plan on adopting the best features of the rule-based modeling method by creating statistical decision making rules intended for the specific driver-roadway and driver-assist situation. In the common construction of a "decision tree" or "decision graph" various sources of information are input into a hierarchical if-then progression, with subsequent alternatives branching out from the predecessor. Traveling down the tree can be as simple as using Bayesian statistics, or the complexity can increase with more sophisticated fusion techniques such as the Dempster-Shafer approach which uses plausibility and belief values in lieu of single probabilities.

4.3.4 Control Models

There are two major approaches in modeling a control part of a human driver: the crossover approach and the optimal control approach. A specific optimal control approach has been called preview control model. The nature of the crossover model is adjusting parameters in order to get a proper closed-loop response as viewed in the frequency-response domain. Optimal control models, including the preview control model, predict the driver's control output using a state-variable representation of vehicle response. Control parameters are computed automatically, using methods from the optimal control theory, and based on the task description (cost function) and driver limitations (phase constraints). The optimal control approach is twofold: it assumes that estimation and control can be computed sequentially. The estimator could be based on either Kalman filtering (stochastic approach) or a guaranteed state estimate approach. The parameter identification should satisfy an optimality criterion in a sense that the estimate is as good as it could be, and an uncertainty (estimation error variance) assosiated with this estimate lies within some predefined and reasonably bounded range. The optimal control laws then provide the control outputs based on these estimates for each system state variable. The control laws should exhibit a certain level of robustness towards the identification errors. The preview control model is a significantly simplified version of a generic optimal control approach. It considers a single output variable, external disturbances are not being accounted for, and the identification of the parameters is exact (perfect knowledge). Functionally this model is similar to the crossover model except for the frequency domain vs. state-space. A general optimal control model is much more flexible than the crossover model. Driver knowledge base could be modeled by state-variable representation of all system response parameters, and driver limitations could be expressed in terms of phase constraints, whereas the crossover model algorithm is tightly coupled with a strictly predefined limited set of parameters. A set of perceptual inputs could be varied in optimal control model without any significant change of the algorithm. Imperfect knowledge could not be modeled explicitly in the crossover approach. Control outputs could be stochastically varied in the optimal control model to achieve the effect of behavioral variability. However, there is a steep price to be paid for this flexibility. A generic optimal control approach requires much more computation. Moreover, in a number of applications this level of flexibility is not required, since assuming a perfect knowledge and taking only a limited number of state variables into account is sufficient for large-scale microsimulations in order to obtain correct results. At this time we have implemented in SmartAHS the crossover models for longitudinal and lateral control. These controllers model the ability of a human driver to steer in order to follow the desired path and to operate the throttle to keep a desired headway to the front car.

4.3.4.1 Steering Controller

Since the crossover model is linear, it is not suited for a highly congested traffic situations. It represents normal and moderate driving conditions. The driver represented by the model has no special skills for rare and extreme conditions. Hence, the controller could be used for modeling crash avoidance maneuvers and similar studies.

The main component of the controller is a feedback loop which reduces the curvature error. The error calculation is based on the comparison between desired and actual paths. The desired path is constructed by choosing a look-ahead distance and setting an aim point. The controller integrates the curvature error and modifies the steering accordingly. Another feedback is added to maintain the vehicle at the desired y position to keep the vehicle within the lane bounds. It is called lane position trim. This lane position error is integrated and added to the curvature error. The last loop is the motion feedback..

As mentioned above, the driver has perfect knowledge of all input variables which are lane position, road curvature, current velocity and yaw rate of the vehicle.

4.3.4.2 Throttle Controller

The process of calibration and validation of this model was conducted by Ford Company using real vehicles and the number of drivers on a test track and on a highway. The model consists of two feedback loops to regulate the velocity and the headway to the car ahead. A global delay models both neuromuscular and visual delays. Note that while for lateral control a realistic delay for a human driver constitutes only fractions of a second, for longitudinal control it may be as much as 1.5 seconds since human driver tends to perform poorly in perceiving a mild velocity change of the vehicle in front.

4.4 Application GUI

This “application GUI” will allow the relatively new users to correctly mix and match the various complex models available -- sensors, actuators, vehicle dynamic models, human and automatic controllers, and highways – and quickly combine them to create an analysis scenario. We will call it the Java Application Wizard for SmartAHS (JAWS), a tool capable of bringing the large set of objects (types) in the SmartAHS framework to the user in a graphical format. It will make SmartAHS accessible to both the highly and less sophisticated user communities, all integrated into a set of pop-up screens which will look the same on all computer platforms.

Some of these objects are specifically designed to work with others, while many of them can be used in a wide spectrum of scenarios. In order to allow a user to create a custom component using many of the existing structures (and optionally plugging a custom component into the simulation), a user interface that directly gives the information to the user was needed. JAWS is an application wizard insofar as it presents the user with a set of choices at each step of the model construction process. One must choose a specific highway, then a specific weather profile, some sensors, some vehicle models, and so on.

If a particular item is selected at a specific stage of the process, all items that are not compatible with the chosen item will not be presented as choices to the user. This frees the user from having to learn everything about every component in the SmartAHS library and it will drastically reduce the learning curve for using SmartAHS components. This tool needs to be created in a modular, general way, such as to make it easy to add any amount of models to the SmartAHS library and preserve the same look and feel of the user interface. This will keep the users of JAWS from having to learn a new user interface paradigm whenever a large amount of new models are integrated into the SmartAHS libraries.

The use of the Java programming language for this project allows us to leverage now and in the future the large amount of free tools available to the community.

5 Phase II. SHIFT Enhancements (Jan – May 98)

The current release of SHIFT provides a stable and fully functional platform for microsimulation development.

We have enhancee SHIFT to make it more accessible, available on multiple platforms, improve computational efficiency and enable us to verify and implement the simulation designs.

To do so, we have performed four subtasks:

1. Enhance Simulation Data Output
2. Port SHIFT to PC Computers
3. Investigate Parallelization of SHIFT
4. Perform SHIFT Implementation and Verification Extensions

This is described below.

5.1 Enhance Simulation Data Output

For easier analysis and presentation of SmartAHS simulations, we have provideed SHIFT with data output formats compatible with standard relational databases.

5.1.1 Introduction

A SHIFT simulation can generate a significant amount of data. Using the command line debugger, the user can decide to dump the state of discrete and continuous variables to a file as time flows. Unfortunately the format of the output file was not easily understandable by spreadsheets and other analysis tools.

Therefore further analysis on the generated data required writing and using some ad hoc format conversion script.

To reduce this problem a new set of trace commands was added to the SHIFT command line debugger. These commands generate files in table format output. These files can be easily imported in any spreadsheet. Converting the generated file is not mandatory anymore. In those cases where some processing is required it is generally easier to process files in the new format (rows and columns) than in the old one.

5.1.1.1 Data formats

Two kind of files can be generated. The first one is type oriented, that is every line contains a copy of the state of a component of a given type at a some time. The second one is transition oriented, that is each line contains informations on a transition that happened in a component of a given type at some time.

5.1.1.1.1 Type Oriented Format

The first line is the header, which contains the column titles. Its format is the following:

```
time <FS> Instance# <FS> mode <FS> <USER_CHOICE_1> <FS> ... <FS> <USER_CHOICE_N>
```

- `time` is the title of the time stamp column.
- `Instance#` is the title of the instance number column.
- `mode` is the title of the column containing the current state of the component in column 2.
- `<FS>` is the user defined field separator.
- `<USER_CHOICE_`*`X`*`>` is the title of the column corresponding to the *x*-th variable chosen by the user.

The other lines of the file have the same number of entries as the header line, but they contain actual values instead of column names.

Example: this excerpt from a trace file shows informations about four components of the same type between time stamp 552 and 557. They switch from the safe state to the unsafe state and they measure the time spent in each state in the timeSafe and timeUnsafe variables. To convert the time stamp in simulation time we must know the simulation time step (sim_time = time_stamp*time_step). Suppose the time step for this simulation was 0.25 sec., then time step 552 corresponds to time 138.00 sec.

```
time Instance# mode timeSafe timeUnsafe
...
552 0 safe 3.750000 0.770000
552 1 safe 0.720000 3.300000
552 2 unsafe 0.220000 1.800000
552 3 safe 0.620000 0.000000
552 4 unsafe 0.110000 0.010000
553 0 safe 3.760000 0.770000
553 1 safe 0.730000 3.300000
553 2 unsafe 0.220000 1.810000
553 3 safe 0.630000 0.000000
553 4 unsafe 0.110000 0.020000
554 0 safe 3.770000 0.770000
554 1 safe 0.740000 3.300000
554 2 unsafe 0.220000 1.820000
554 3 safe 0.640000 0.000000
554 4 unsafe 0.110000 0.030000
555 0 safe 3.780000 0.770000
555 1 safe 0.750000 3.300000
555 2 unsafe 0.220000 1.830000
555 3 safe 0.650000 0.000000
555 4 unsafe 0.110000 0.040000
556 0 safe 3.790000 0.770000
556 1 safe 0.760000 3.300000
556 2 unsafe 0.220000 1.840000
556 3 safe 0.660000 0.000000
556 4 unsafe 0.110000 0.050000
557 0 safe 3.800000 0.770000
557 1 safe 0.770000 3.300000
557 2 unsafe 0.220000 1.850000
557 3 unsafe 0.660000 0.010000
557 4 unsafe 0.110000 0.060000
...
```

5.1.1.1.2 Transition Oriented Format

The first line is the header and it contains the column titles. Its format is the following:

```
time <FS> Transition# <FS> Type <FS> Instance# <FS> mode1 <FS> mode2 <FS> event
```

- `time` is the title of the time stamp column.
- `Transition#` is title of the transition id column.
- `Type` is the title of the type column.
- `Instance#` is the title of the instance number column.
- `mode1` is the title of the from-state column.
- `mode2` is the title of the to-state column.
- `event` is the title of the event column.
- `<FS>` is the user-defined field separator.

The other lines of the file have the same number of entries as the header line, but they contain actual values instead of column names.

5.1.1.2 Debugger Commands

The SHIFT debugger provides 4 new commands to generate trace files in the described formats.

Type oriented commands:

- db_ctracetype <type> (db_ctt). For all components of type <type>, dump all continuous variables every time click in table format; specific variable names can be indicated with the -v option.
- db_ctracecomp <type> <id> (db_ctc). For component <id> of type <type>, dump all continuous variables every time click in table format; specific variable names can be indicated with the -v option.
-

Transition oriented commands:

- db_dtracetype <type> (db_dtt). For all components of type <type>, dump all transition information before the transition happens; specific synchronization events can be indicated with option -e.
- db_dtracecomp <type> <id> (db_dtc). For component <id> of type <type>, dump all transition information before the transition happens; specific synchronization events can be indicated with option -e.
-

The sfs <FS> command sets the field separator. <FS> can be any character. C escape sequences are recognized (e.g. tab is \t).

The trace files can be found in the LOG*x* directory, which is created in the working directory by the SHIFT run-time system.

5.2 Port SHIFT to PC Computers

The porting of the SHIFT simulation development environment to the PC computing platform is a very important project given the PC's widespread use in the user community. The porting requires the following activities:

- The compiler was ported.

- The run-time and debugging libraries was ported ported in a general way such that any further developments in the UNIX (the main development platform) version will be quickly integrated in the PC version. This involved setting up, documenting, and maintaining a rigorous multi-user software management system.

- The SHIFT graphical user interface (TkSHIFT) was be ported. At this stage, there are a few bugs in the PC version of the TCL/TK language, and this constitutes the bottleneck in the complete port of the system to the PC platform. Solutions for this problem include waiting for the problem in the language to be fixed, or implementing a new mechanism for the interface between the SHIFT runtime environment and the graphical debugger.

The Windows NT version of SHIFT and SmartAHS was developed at PATH. This effort involved not only modifications to the original Unix-based code of the system, but also search and adaptation of the NT-based equivalents of all supporting software: lexer and parser tools, Tcl/Tk, gnu C compiler. This effort yielded a packaged software system along with instructions on getting and installing all the supporting software.

5.2.1 SHIFT Compiler

The compiler code was slightly modified in order to compile with the set of GNU utilities for Windows NT platform (gcc, lex, yacc). The development platform is built around the Windows emacs environment. These tools were downloaded from the following locations:
ftp://ftp.cygnus.com/pub/gnu-win32/latest/README.txt
http://www.cs.washington.edu/homes/voelker/ntemacs.html

Minor changes to the Makefile were required as well. The Windows NT version of the compiler code and Makefile are kept within the Windows-based Visual Source Safe version control system. Note that the set of GNU tools is used for the assembly of SHIFT compiler only. These tools are not required for a regular use of the SHIFT system by end users.

5.2.2 SHIFT runtime environment: runtime, libDebug, and socket libraries

The SHIFT runtime core library, network support and debugger libraries were converted to the NT using the Microsoft Visual C++ development environment. This project includes the <runtime>, <socket>, and <libdebug> libraries. The library with debugging symbols is in the debug subdirectory, the one that is optimized and debugger-symbol free is in the release directory.

5.2.3 TkSHIFT debugger

The TkSHIFT debugger is developed with MS VC++ and also uses Tcl/Tk. The Tcl/Tk programming language is developed by a team at sun. The main homepage for Tcl/Tk is:
http://sunscript.sun.com

One of the main differences between the two versions is the way that the C files are incorporated into the tcl interpreter. In Unix, the files were compiled into object format files, later to be linked into an executable with a tclAppInit.c file. With the NT, the way to do this is to create a dynamic link library (dll) out of the C files. A special function provides hooks that allow the tcl interpreter to load the dll correctly, and have to manually load the dll, as is now done in initialize.tcl. Also it is necessary to load the blt dll for graphical widgets used in the GUI (whereas in Unix the libBLT.a was linked directly as a static library during the compilation of the TkSHIFT executable. There is an additional file, TkSHIFT.tcl that simply sources initialize.tcl (wherever that file may be). Since it is specified in the creation of TkSHIFT that any window belonging to the TkSHIFT application must have a certain look and feel, the main program MUST be called TkSHIFT.tcl, otherwise the GUI windows graphics won't work. The debugging was accomplished by running wish8O.exe, then sourcing initialize.tcl from there. By doing this, there is a console window the pops up that can print out many of the useful debugging messages, which are lost when one starts the program with TkSHIFT.tcl (Note: In many cases, a simple puts in the program will make the TkSHIFT.tcl program halt because the program cannot figure out where to print the puts message. Some of these print messages need to be taken out to make TkSHIFT.tcl work).

5.2.4 The Release Process

The release is packaged and distributed using the Install Shield program. By specifying the location of the files that are to be installed and the path they should be installed to, this program takes care of the presentation of the installation process in the standard Windows way. The current distribution includes all of the compiled libraries, some examples, and two example projects: the shop floor example, and the merge simulation example. The distribution also includes all of the SHIFT source files that constitute the SmartAHS library, with the c source code that goes along with it.The main directories in the distribution are the following:

- SHIFT-dev/lib
- SHIFT-dev/lib/TkSHIFT
- SHIFT-dev/docs
- SHIFT-dev/bin
- SHIFT-dev/include
- shift-dev/projects/shopfloor
- shift-dev/projects/SmartAHS
- SHIFT-dev/SHIFT/examples

The release process for the NT version of SHIFT/SmartAHS goes through the following stages:

1. Make the distribution diskettes using the Install Shield program. (Need to have the complete distribution. The current and latest one is saved in: c:\ProgramFiles\InstallShield\ISExpress\VC\SHIFTRelease1.O.ivz)
2. Copy these distribution diskettes to somewhere on the hard drive (C:\Installation)
3. Zip these files into a single SHIFTRelease.zip file.
4. FTP this file to the PC SHIFT page on the web.

Currently the Windows NT release of SHIFT/SmartAHS is available at:
http://www.path.berkeley.edu/~danielw/pcport.html

5.3 Investigate Parallelization of SHIFT

Our objective was to increase the computational speed of SHIFT simulations.

The hardware platform to run the SHIFT simulations is assumed to be a multiprocessor workstation with shared memory architecture (i.e., UltraSparc-2) which has a small number of processors (1-16).

Every SHIFT simulation contains two major parts that simulate continuous and discrete behavior respectively. Timed runs of the simulations show that modeling the continuous evolution of the system consumes 70% to 95% of the total amount of time spent on a simulation.

The maximum achievable speedup is estimated using the Amdahl law. For example, the 70% share of continuous part on a two-processor machine may yield a 1.54 times faster execution when parallelized, while on a four-processor machine this figure is 2.11. This implies that a major effort should be directed towards parallelizing the numerical integration component of the SHIFT system.

Hence, we undertook the development of a multi-threaded version of the run-time simulation environment to utilize the computational power of a multiprocessor system with shared memory architecture as a feasible first step in order to meet our objective.

Consequently we explored the possibility of extending the SHIFT programming language and simulation environment to allow distributed simulations and real-time extensions.

The hardware platform to run the SHIFT simulations is assumed to be a multiprocessor workstation with shared memory architecture (i.e. UltraSparc-2) which has a small number of processors (1-16). The issue of distributed memory architectures and massively parallel systems was not considered.

Every SHIFT simulation contains two major parts which simulate continuous and discrete behaviour respectively. Timed runs of the SmartAHS simulations show that the continuous part consumes 70% - 95% of the total amount of time spent on a simulation even though the algorithm for discrete part is exponential in its nature and the continuous part is linear. Note that it is possible to construct a problem with a complicated discrete behaviour which will consume most of the resources to compute its discrete part, however the 70%-95% figure is sound for the domain of transportation problems SHIFT is used to model.

The maximum achievable speedup is estimated using the Amdahl law. The speedup is given as a function of a ratio of the part to be parallelized, and the number of CPUs. For example: the 70% share of continuous part on a 2-processor machine may yield a 1.54 times faster execution when parallelized, while on a 4-processor machine this figure is 2.11.

These figures imply that a major effort should be directed towards parallelizing the numerical integration component of the SHIFT system. The implementation of the multithreaded version of

SHIFT was successfully accomplished. The details of this implementation are provided in the next section.

5.3.1 Distributing the Continuous Step.

The tool of choice for making a parallelized simulation is the POSIX threads library. Current implementation of SHIFT is a highly synchronised system. The variable and fixed step Runge-Kutta routines require a synchronisation after each internal step of this integration method. Particularly, there is a point of synchronisation after each iteration over the list of differentible variables. The sequence of operations performed over each variable invloves reading from various variables but writing only to the current one. It is very natural to parallelize this loop. The only place for potential memory-write conflict is the opeartion of advancing the gloabl pointer through the loop. This operation is protected by a mutual exclusion lock in each thread in order to resolve this conflict.

There are many ways to make a program parallel. There are three common paradigms though. Each of the paradigms is characterized by load balancing method, whether each thread executes the same code, the synchronization techniques, and what data is shared and how it is protected to avoid memory-write (data racing) conflicts.

- [Master-slave] The main master thread spawns a set of slave threads and allocates a fixed amount of work to be done (known in advance). Then the master waits for the slaves to reach a synchronization point - barrier.
- [Pipeline] A task is passed to a succession of threads where each of those performs a work required.
- [Workpile] A set of working threads request portions of work to do from the "pile", usually some form of a queue. The pattern terminates when the pile is empty.

The workpile paradigm suites our needs best. Each work assignment from the pile is one set of operations over a list of differentiable variables within one component. Here different threads process different components at the same time. Competing threads picking the tasks from the pile provide the best load balancing possible (semi-optimal load balancing).

Memory locations to be rewritten by different threads do not overlap. Data used for the calculation during each cycle is taken from memory locations which are never rewritten during current loop by the definition of the integration method. Thus the only mutual exclusion lock is necessary and should be put on the pointer advancing through the list of components with differentiable variables.

Working threads compete for work trying to grab the component pointer and set a mutually exclusive lock while copying and advancing it. The total number of working threads is equal to the number of processors in the machine. The upper bound on the number of threads may be set adaptively to optimize performance, or statically not to exceed the number of processors multiplied by two.

In order to have truly competing threads on a multiprocessor machine the threads should be system-bound (attached to the LWPs in case of SUN Solaris).

5.3.2 Benchmark Tests

A number of performance tests were accomplished. These tests reveal the benefit of using the multi-threaded version over regular single-threaded in all test runs on a 2-CPU machine (Sun SPARC Ultra-2) and most of the cases on a 4-CPU machine (Sun SPARC HPC).

The set of benchmark ing programs included a uniform family of programs with different numbers of ODEs and hybrid components of the same type, and a real SmartAHS application simulating a freeway traffic merge scenario with a few hundred cars being modeled.

The results for the following programs are presented:
1. sample SHIFT program with 20 components, 30 ODEs.
2. sample SHIFT program with 200 components, 3 ODEs.
3. sample SHIFT program with 4 components, 300 ODEs.
4. SmartAHS application - merge scenario.

The following table compares the run times in seconds for 4, 2, 1-threaded versions on a 4-CPU Sun SPARC HPC server. Each simulation was allowed to run for 6000 steps.

	4 thr	2 thr	single-thr
(1)	22	25	26.5
(2)	43	35	29
(3)	30	45	63
(4)	120	192	318

Note that in case of very simple continuous behavior, the single-threaded version wins because the overhead for multi-threaded support exceeds the computaional load for integration (case (2)). However all usual simulations contain much more sophisticated systems of ODEs; and as seen in the table, multithreaded versions win in all these cases.

The next table shows a comparison of 2-threaded and single-threaded versions on a 2-CPU machine (Sun SPARC ULTRA-2). The table shows how many steps of a simulation were computed with single-threaded version when 2-threaded version reached 1000 steps.

	2 threads	single-threaded
(1)	1000 steps	900 steps
(2)	1000 steps	970 steps
(3)	1000 steps	660 steps
(4)	1000 steps	780 steps

Note that multithreaded version wins over single-threaded for all examples on this particular hardware platform.

Typical CPU usage shown is:

- 4-threaded - 225%
- 2-threaded - 158%
- single-threaded - 97%

in percents of the run time.

5.4 Perform SHIFT Implementation and Verification Extensions

The current release of the SHIFT programming language provides precise syntax and simulation semantics. Programs written in SHIFT are compiled to an intermediate representation and linked with the SHIFT with run-time library to produce an executable simulation of a dynamic hybrid system.

We developed extensions to the SHIFT syntax and run-time libraries for real-time implementation and off-line verification of SHIFT models.

In the first step, we extended the SHIFT syntax, intermediate representation, compiler and run-time operating support, to implement a reactive, real-time hybrid system.

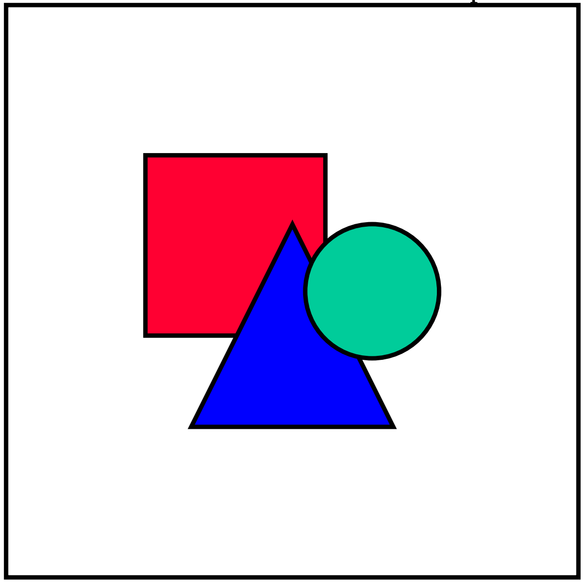
In the second step, we further extended the SHIFT syntax and intermediate representation to produce outputs compatible with existing formal verification tools such as Kronos or HyTech. In this section we report the work carried out within MOU 258 with the aim of both verifying the correctness of SHIFT specifications and generating code to be executed on-board in real-time.

5.4.1 Motivations and Goals

SHIFT is a specification language oriented towards the modeling of so-called *hybrid systems*, i.e., systems comprising of both discrete and continuous behaviors (over time). Discrete behaviors are modeled as finite-state automata: locations correspond to the different discrete modes or phases where the system may be at any time (e.g., high, low), edges model the discrete events that make the system switch from one mode to another (e.g., on, off). The continuous behavior of the system at each phase is described by means of ordinary differential or algebraic equations over a set of real-valued variables (e.g., temperature, gas level, speed). A special case of hybrid systems are the so-called *timed* systems. These systems are only equipped with continuous variables that measure the time elapsed between different discrete events. Such variables are usually called *clocks* and their evolution over time is described as differential equation of the form $x'=1$.

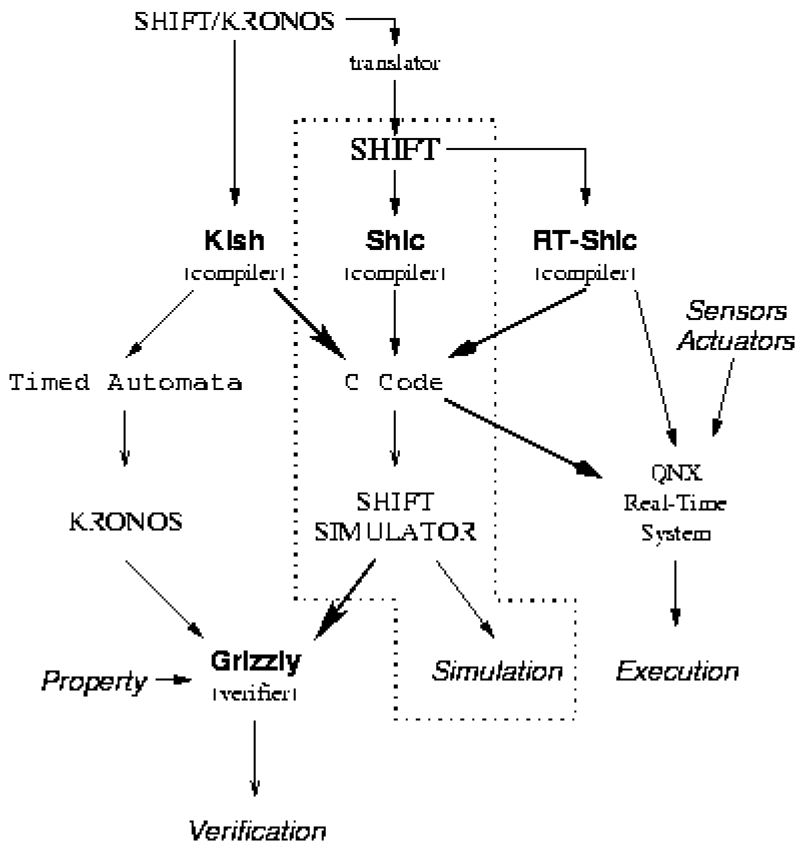
SHIFT is used for instance to specify complex automated maneuvers such as lane changes, car following, and platoon splitting, joining, and merging. All these maneuvers are safety critical. It is clear that it would be desirable to verify that they satisfy the imposed safety requirements. The difference between *simulation* and *verification* is that the former only permits studying a single behavior of the system whereas the latter consists in analyzing all the possible behaviors of it. That is, simulation cannot be, in general, used to prove that the safety requirements are met, but it is very useful to debug the designs and to gain confidence on the control laws. The complexity of these systems make them very difficult, if not impossible, to analyze by hand. Verification should therefore be supported by a computer program. Fully-automatic computer-aided verification of SHIFT programs is in general not possible. That is, it is neither practically nor theoretically possible to construct a program that checks all the possible behaviors of a SHIFT program. Nevertheless, a significant subset of SHIFT programs is indeed verifiable by a computer program. Such programs are those corresponding to timed systems.

Once a SHIFT program has been extensively simulated and, if possible, verified, it would be desirable to be able execute this program on-board. As a matter of fact, there exists today a gap in the chain going from the specification of the maneuvers to their implementation inside the automated cars which needs to be filled out. A solution to this problem is to automatically generate executable code from the SHIFT program. In this manner, the code executed on-board in real-time will behave like the one that has been simulated and verified.
In this context, the contributions made by MOU 258 are depicted in the figure below. The dashed box shows the SHIFT simulation environment whose improvements within MOU 258 are



reported elsewhere.

5.4.1.1 Verification



The work done with the purpose of verifying the correctness of SHIFT programs focused the integration of the SHIFT environment (language and simulator) with the tool KRONOS developed by VERIMAG, a French research center specialized on the development of computer-aided methods and tools for the verification of real-time and hybrid systems

5.4.1.1.1 The syntax

In order to be able to syntactically recognize those SHIFT programs that are indeed timed systems, we have extended the syntax of SHIFT. To avoid confusion, we call this ``new'' language SHIFT/KRONOS. The added features are indeed macro definitions. That is, they do not introduce new concepts into the language as SHIFT/KRONOS can be fully and syntactically translated into SHIFT.

The syntax SHIFT/KRONOS is the one of SHIFT, without any flow declarations and with the following additional statements:

- `kronosclocks x_1, ..., x_n` which corresponds to the SHIFT variable declaration `state continuous number x_1, ..., x_n`, plus the SHIFT flow declaration `x' = 1` associated with every discrete mode.
- `kronoswhen <cond>` where `<cond>` is a condition over the set of variables declared as `kronosclocks`. This statement corresponds to the SHIFT guard `when <cond>`. SHIFT guards

are also allowed, which amounts of taking the conjunction of the conditions stated in the `when` and the `kronoswhen`.

- `kronosinvar <cond>` where `<cond>` is a condition over the set of variables declared as `kronosclocks`. This statement corresponds to the SHIFT invariant `invariant <cond>`. SHIFT invariants are also allowed, which amounts of taking the conjunction of the conditions stated in the `invariant` and the `kronosinvar`.
- `kronosreset { x_1 := v_1; ...; x_n := v_n; }`, where v_i is either `0` or another variable x_j. declared as a `kronosclock`. This statement corresponds to the SHIFT statement `do { x_1 := v_1; ...; x_n := v_n; }`.

5.4.1.1.2 Example of a SHIFT/KRONOS program:

```
type Fischer
{
state
          number id ;
kronosclocks
          x ;
discrete
          idle ,
          wait kronosinvar x<=1 ,
          test ,
          cs ;
export
          closed enter_cs ;
transition
          idle -> wait {}
          when N = 0
          kronosreset { x:=0 ; } ;
transition
          wait -> test {}
          when true
          do { N := id ; }
          kronoswhen x<=1
          kronosreset { x:=0 ; } ;
transition
          test -> cs { enter_cs }
          when N = id
          kronoswhen x>1
          kronosreset { x:=0 ; } ;
}

type MonitorCS
{
state
          number c := 0 ;
discrete
          good,
          bad ;
transition
          good -> good { S:enter_cs(one) }
          when c = 0
          do {
                    c := 1 ;
          } ;
transition
          good -> bad { S:enter_cs(one) }
          when c = 1 ;
}

global Fischer f4 := create(Fischer, id := 4) ;
global Fischer f3 := create(Fischer, id := 3) ;
global Fischer f2 := create(Fischer, id := 2) ;
global Fischer f1 := create(Fischer, id := 1) ;
global number N := 0 ;
global set(Fischer) S := { f1, f2, f3, f4 } ;
global MonitorCS mcs := create(MonitorCS) ;
```

5.4.1.1.3 The equivalent SHIFT specification of **`type Fischer`**

```
type Fischer
{
state
          number id ;
state
          continuous number x ;
flow
          kronosflow { x' = 1 ; } ;
discrete
          idle { kronosflow },
          wait { kronosflow } invariant x<=1 ,
          test { kronosflow },
          cs   { kronosflow };
export
          closed enter_cs ;
transition
          idle -> wait {}
          when N = 0
          do { x := 0 ; } ;
transition
          wait -> test {}
          when x<=1
          do { N := id ;
               x := 0 ;
          } ;
transition
          test -> cs { enter_cs }
          when N = id and x>1
          do { x := 0 ; } ;
}
```

5.4.1.1.4 The compiler

Kish is the compiler that takes as input a SHIFT/KRONOS program and generates as output the files needed for the verification. *Kish* is actually a modification of the SHIFT compiler *Shic*. The additional work carried out by *Kish* with respect to *Shic* is the following.

- Type-checking required by the added syntactical features.
- Generation of the input files, called `Timed Automata`, to the verification tool KRONOS.
- Generation of C code that integrates the code generated by *Shic* for simulation purposes with code needed for verification.

5.4.1.1.5 The verifier

We have developed a verification tool called *Grizzly* that integrates both KRONOS and the SHIFT simulator. *Grizzly* is indeed the tool that explores all the possible behaviors of a SHIFT/KRONOS program. KRONOS provides all data-structures and associated manipulation functions required to store, update and check the consistency of the timing constraints. The run-time library of the SHIFT simulator is the one in charge of manipulating the corresponding SHIFT variables and dealing with the synchronization of transitions.

Grizzly essentially works as follows. Given a state, which is composed of a KRONOS and a SHIFT data-structures, it calls the appropriate functions of the SHIFT run-time library to construct all the possible successors of the state (i.e., the states reachable by all the possible outgoing transitions from the state) only taking into account the constraints imposed by the pure SHIFT statements (i.e., without considering the timing information). Then *Grizzly* calls the appropriate functions of KRONOS to check whether the transitions found in the previous step meet the timing constraints (i.e., the condition imposed by the `kronoswhen` is satisfied), in which case the transition is taken and a new state is created, otherwise the transition is not taken. *Grizzly* keeps repeating this procedure until all the states have been visited.

The algorithm described above generates all the possible states in which the SHIFT/KRONOS program may stay. We can also ask *Grizzly* to check whether some given state is indeed reachable from the initial state.

5.4.1.1.6 Example

We illustrate here a typical verification session. The shell-script called `kSHIFT` should be used to generate the C code, the KRONOS timed automata and to compile and link all the files together with *Grizzly*. The result is an executable binary file named as the input file with the extension `.grz`. This file is the one to be executed to perform the verification.

For instance, we can use `fischer.grz` to check whether the component `mcs` (an instance of the type `MonitorCS`) reaches the discrete mode `bad`. To do so, we use the following command:

```
fischer.grz -REACH mcs_bad -DFS -h
```

The option `-REACH mcs_bad` indicates that we are looking whether the component `mcs` reaches the discrete state `bad`, whereas `-DFS` and `-h` are options provided by KRONOS that correspond respectively to perform the exploration using a depth-first search and to generate a trace if the state is reachable. The output is the following:

```
grizzly: Evaluating reachability: _init AND E<> _reach
grizzly: Using depth-first search with max stack size: 1000.
grizzly: Max symbolic-states set size: 1000.
grizzly: Symbolic states visited: 9
grizzly: reachability successful
   f4   f3   f2   f1   mcs
0: idle idle idle idle good F4_X=F3_X and F4_X=F2_X and F4_X=F1_X
1: idle idle idle wait good F4_X<=10 and F4_X=F3_X and F4_X=F2_X and F1_X<=F4_X
2: idle idle wait wait good F4_X<=10 and F4_X=F3_X and F4_X=F2_X and F1_X<=F4_X
3: idle idle wait test good F4_X<=10 and F4_X=F3_X and F4_X=F2_X and F1_X+1<F4_X
4: idle idle wait cs   good F4_X<=10 and F4_X=F3_X and F1_X<=F4_X and F2_X<=F1_X
5: idle idle test cs   good F4_X<=10 and F4_X=F3_X and F2_X<=F4_X and F1_X<=F2_X
6: idle idle cs   cs   bad  F4_X<=10 and F4_X=F3_X and F2_X<=F4_X and F1_X+1<F2_X
```

5.4.1.1.7 Practical experiments

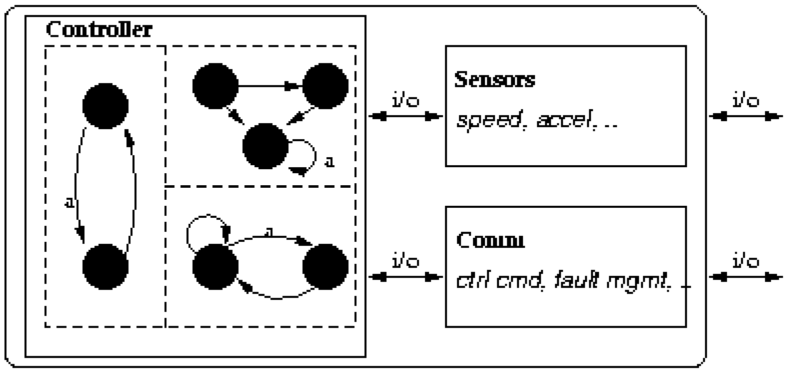
Grizzly is currently being used in the context of MOU 258 to verify a distributed fault-diagnosis protocol. *Grizzly* has demonstrated to be very useful. Several bugs (specially deadlocks due to synchronization problems) have been found by *Grizzly* in early phases of the design.

5.5 Generation of Executable Code

We describe here the work carried out concerning the generation of executable code from SHIFT programs. This is ongoing work.

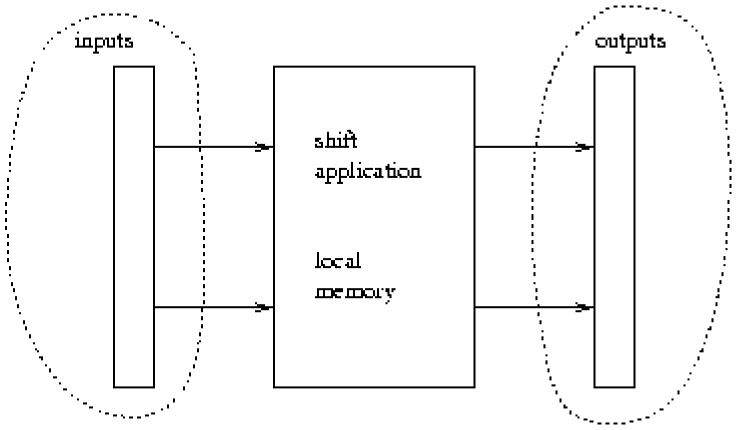
5.5.1 SHIFT Application

We refer to a SHIFT application as a collection of a fixed number of SHIFT components. The external interface of the application is defined to be the set of inputs and outputs of the components. The components are allowed to synchronize with each other through events, but they are only allowed to interact with the external world using the external interface. Such an application is therefore open in the sense that its behavior depends on externally provided inputs. (see the figure below).



5.5.2 SHIFT Executor

Given a SHIFT application, we are interested in generating code to be executed on a real-time operating system (QNX, for instance). We think of a SHIFT application as a single task, i.e., the application itself, rather than as a collection of tasks, i.e., one task per component. With this in mind, generating code for the application consists in implementing a program called ``SHIFT executor'' (see the figure below) which is going to execute the SHIFT application more or less is the same way that the simulator does.



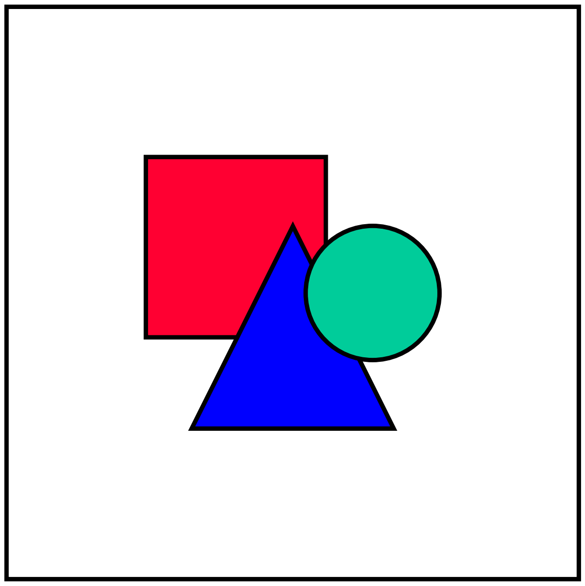


Figure 3: Architecture of the executor.

5.5.3 Benefits of the Approach

This approach allows us to avoid directly implementing a protocol for synchronizing the SHIFT components on top of the operating system. The synchronization between components inside the application is ensured by the SHIFT executor using the same algorithm implemented in the simulator. Another advantage of this approach is that the execution of the SHIFT application on the real-time operating system will be a single thread instead of the interleaving of multiple threads which results when implementing each component as a separate task. Besides, the granularity of the execution time can be set up to be equal to the time increment used for the simulation. Thus, if the executor is scheduled in time, it will reproduce the same behavior observed during the simulation, provided that the time needed to execute the code is smaller than the simulation time-step (in other words, if the code can be simulated in real time). This property can be checked during the simulation.

6 Phase III. Documentation (Jan 98 – Jun 98)

Under this cross-cutting activity, we have collected the dispersed documentation of the SHIFT and SmartAHS tool-set and integrate it into a cogent user manual and reference guide. The document was be divided into two main parts: SHIFT and SmartAHS.

1. SHIFT (documentation available at http://www.path.berkeley.edu/SHIFT/publications.html)

Items and explanations which were included:

- an easy and intuitive example with extensive explanations
- object-oriented features
- hybrid behavior
- advanced topics and all the rest
- description of the associated tools such as the compiler, debugger, document generator, type tree generator, and TkSHIFT

2. SmartAHS (documentation available at http://www.path.berkeley.edu/SmartAHS/sahs-manual/manual.html)

Items and explanations which were included:

- description of the various vehicle and highway models
- description of scenario specification
- description of sample safety and throughput analysis applications
- description of associated tools such as the highway builder, highway compiler, JAWS, SmartAHS canvas in TkSHIFT, and connection to SmartPATH animator.

6.1 ACC Simulation - Example of SmartAHS Applications

6.1.1 Objective of the study

The objective of the study was to understand what benefit can be gained with the deployment of Adaptive Cruise Control (ACC). Another objective, in sync with the MOU 258 objectives, was to test and demonstrate the applicability of SmartAHS to the broader class of Advanced Vehicle Control and Safety System problems outside of fully automated highways.

An ACC is the next step from Conventional Cruise Control, which is currently deployed on vehicles, is ACC without braking authority. This type of ACC maintains a constant headway with the vehicle ahead by controlling the throttle and by using the gear-SHIFT. That is why the maximum possible deceleration this ACC can provide is 0.07g. The responsibility for vehicle safety is on the driver, as he has full braking authority. He also has the authority to switch the ACC on and off. This type of ACC is designed to be comfortable: acceleration and deceleration are very mild. It is also autonomous, i.e., all the needed information for vehicle control is obtained from the devices installed on the vehicle: there is no communication.

The next level of vehicle automation is ACC with full braking authority. In this case longitudinal control of the vehicle is fully handled by the ACC, leaving to the driver only a supervisory role and lateral control. The responsibility for safety with respect to longitudinal control of the vehicle is given to the ACC. This ACC can produce uncomfortable actions, such as hard braking. The driver can still switch the ACC on and off. This type of ACC is also autonomous, there is no communication between vehicles or with the roadside.

What will happen if the ACC has the ability to communicate with other vehicles or with the road? Will this ability bring more safety or more comfort to the driver?

We wanted to investigate each ACC concept from different points of view: how will the ACC affect the driver and what consequences will it have on traffic. How safe and comfortable will the driver be, how much time will he leave the ACC on, how will the ACC affect his travel time. What are the traffic conditions in which ACC can successfully control the vehicle.
To answer these questions the following simulation scenario was designed, and a series of simulation runs was conducted for the ACC without braking authority case.

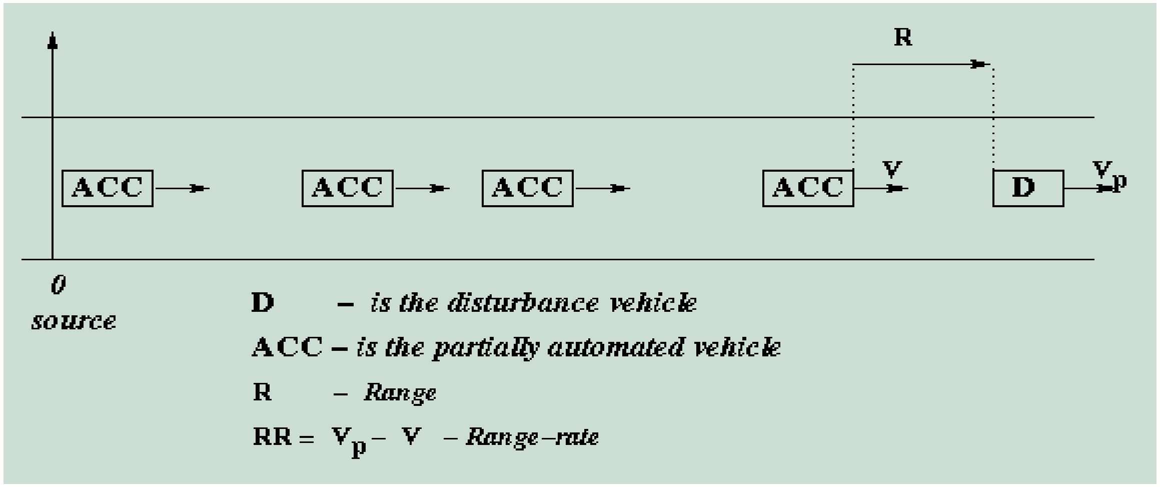
6.1.2 Two Components of the simulation

The simulation was developed in such a way, that all the components can be easily changed without affecting other components of the simulation. Existing SmartAHS components were used: roads, simple and complex vehicle models, sensor models. ACC and driver models were implemented in the SHIFT language and were added to the simulation.

To study the benefits of another ACC controller or to conduct a sensitivity study of ACC parameters, the same scenario and simulation can be used. (The user just need to plug in the new ACC model.) Batch processing facilities are provided to run a series of simulations, which differ from each other only by some parameters. The user can specify, which variables to trace. To facilitate data analysis, post-processing, data filtering, format conversion and display utilities are provided.

6.1.3 Simulation Scenario

The scenario created for the simulation is the following. A few vehicles are driving on a straight one lane road. The road surface is dry, and the vision is clear. The first vehicle is driven manually and it follows a fixed speed-time profile, collected on a real freeway. It is followed by a string of ACC-equipped vehicles. All of them have the same ACC controller. Number of


R
ACC
ACC
ACC
ACC
V
D
Vp
0
source
D – is the disturbance vehicle
ACC – is the partially automated vehicle
R – Range
RR = Vp – V – Range–rate

vehicles and ACC parameters for every vehicle can be varied

The speed-time profile for the disturbance vehicle is predefined and it was collected on highway I-880, California. We chose trajectories which correspond to various traffic conditions: stop-and-go, moderate, and light traffic.

The stop-and-go data was collected during morning hours with heavy traffic. We know that there was an accident at that time, which affected the traffic for about two hours. Traffic density was 220veh/mile. The data collecting vehicle entered the freeway at 45mph speed. When approaching the accident area it began to decelerate. Its velocity dropped almost to 0 in one minute, then it made a few ups and downs, and eventually it began to grow again.
The moderate data was also collected during the morning hours. There was an accident at that time too. But it affected the traffic only for 15 minutes. The trajectory of the subject vehicle was recorded when the traffic began to recover after the incident. Traffic is smoother than the stop-an-go traffic, but it still has some islands of congestion. The density in the accident area reaches up to 220 veh/m, but in general it is around 100-140 veh/h.

The light data was collected during early morning hours, when traffic was free flowing.

6.1.3.1 Vehicle

Two vehicle models are integrated in the simulation: simple point mass kinematic model with lag, and complicated 2D dynamical model. The user can switch between the models. All vehicles are equipped with perfect sensors, which output range and range-rate with respect to the preceding vehicle. The maximum sensor range is a user-defined parameter.

6.1.3.2 Data collection and data processing

Components state can be traced and recorded at every time step. This feature allows to use the output data for further analysis. Special components, called monitors, watch particular aspects of the simulation: when the driver is safe/unsafe, comfortable/uncomfortable, or when the ACC is on/off. The user can specif what information to save for every simulation run.

Typically analysts need to run a set of simulations that differ from each other only in some of the ACC parameters. Often to change the parameters by hand is not feasible because the combinations are too many. For this reason batch processing facilities are provided to automate the compilation process.

On the other hand data analysis requires post-processing, data filtering, format conversion and display utilitiesm, which are also provided.

The whole process can be summarized in the following pseudo-code algorithm:

- loop on a list of ACC parameter values
  - substitute the current values in the source files
  - compile the simulation
  - run the simulation, activating data collection
  - post-process collected data

- end loop
- load data for display/analysis

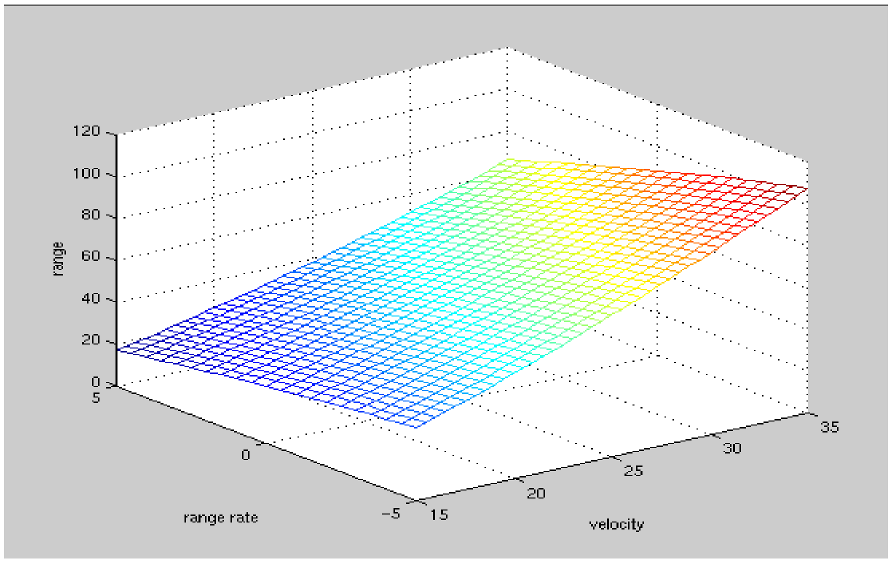
6.1.4 Longitudinal Control

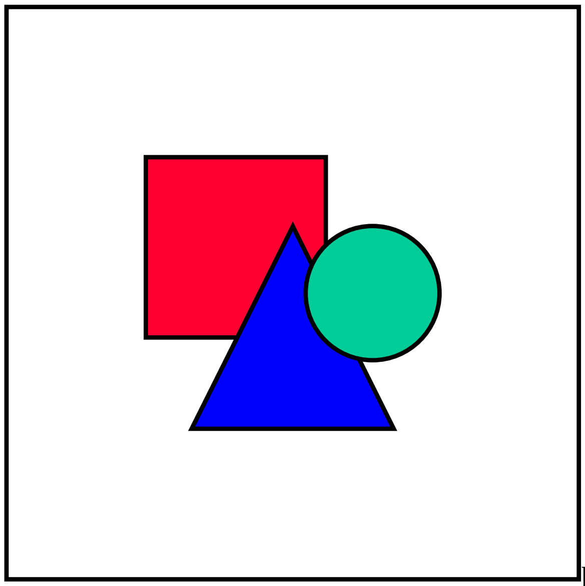
As we already mentioned, for modeling the ACC without braking authority we used a model by P.Fancher, UMTRI. The driver has full braking authority and he is responsible for safety. He decides when to switch the ACC on, and he can intervene or switch the ACC off at any time. We assume, that a driver wants to keep the ACC on as much as possible: he switches it on as soon as he can, and he takes back control when he feels unsafe.

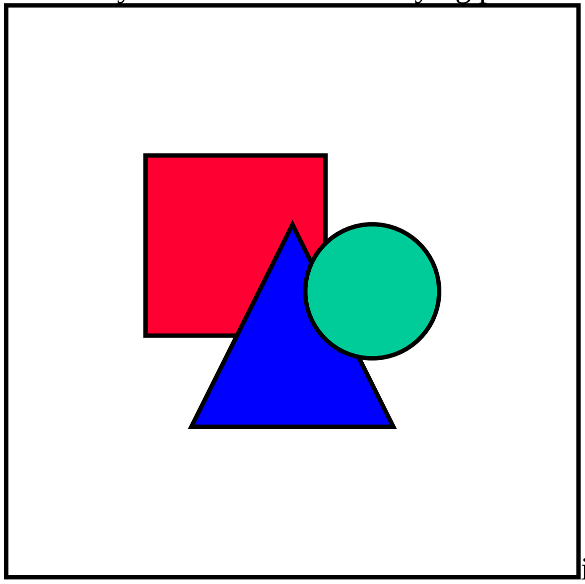
Safety is defined through a surface, which is function of vehicles' accelerations and velocities, current range and range-rate, and driver's reaction time.

When the vehicle is above the surface - it is safe, when it is below - unsafe.

The figure below shows how safety surface divides (*velocity*,*Range*,*Range-Rate*) space into safe and unsafe regions.



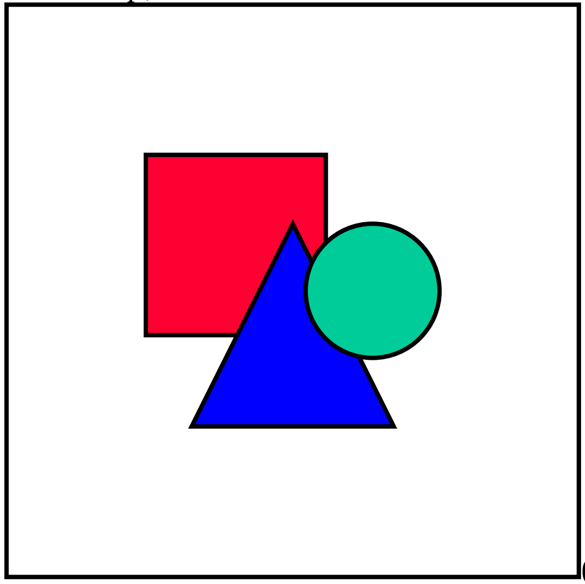
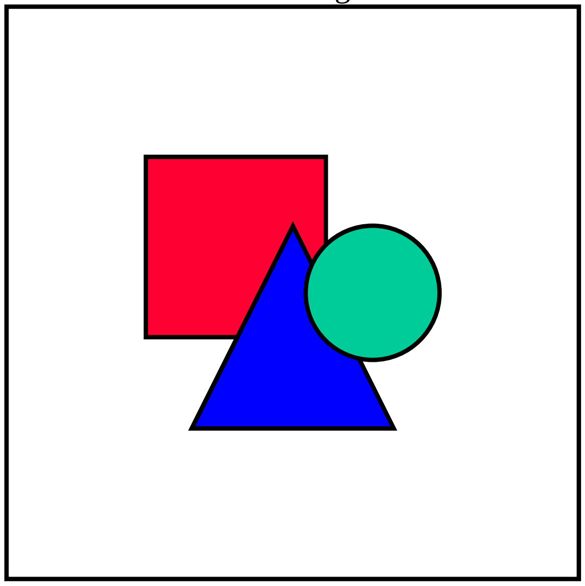
Different people perceive safety differently, so they override ACC at different time in the same situation. To capture this fact we distinguished between risky drivers, who prefer to drive close, from conservative drivers, who prefer to keep longer distance to the preceding vehicle. This risk diversity among the drivers was simulated by varying the braking capability of the preceding vehicle  parameter in the definition of the safety surface.

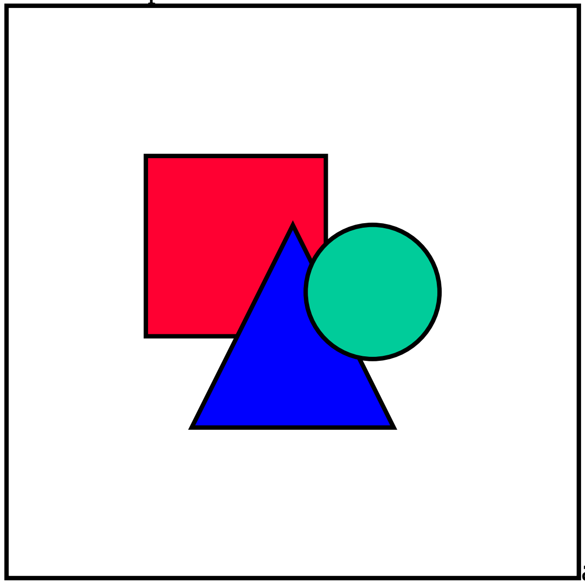
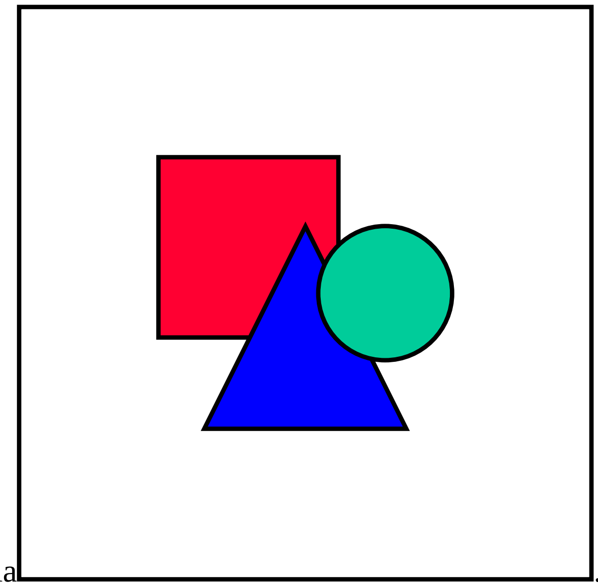
The switching from the unsafe to the safe region is happening when a similar safety surface is crossed back. The only difference from the previous safety surface is that the varying parameter (the braking capability of the preceding vehicle)  is 0.1g more. This gives some kind of gap between the switching lines, which helps reduce switching and make traffic smoother.

6.1.4.1 ACC model

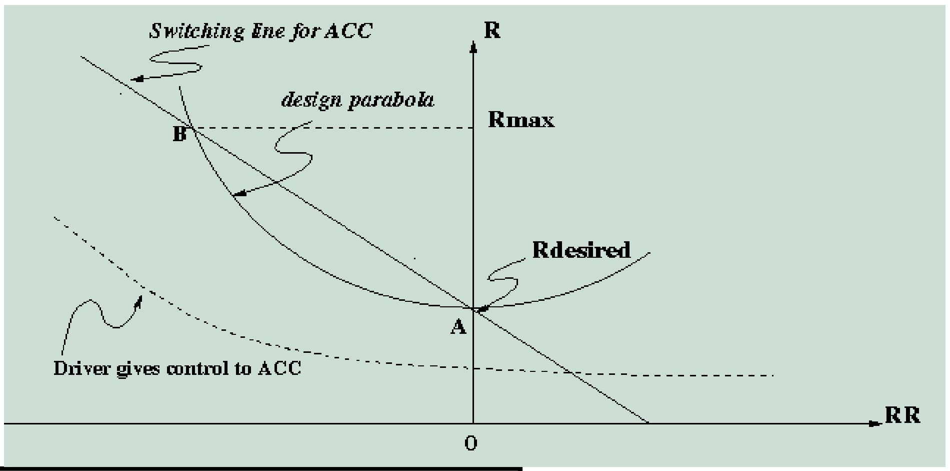
When the ACC is on, it can operate in two modes: velocity control and headway control. It works over the existing cruise control system and it assumes it is functioning properly.
Range and range-rate outputs from the sensor are the inputs for ACC control algorithm. If there is no vehicle within the sensor range then the ACC is in velocity control mode. If there is a vehicle within sensor range then an algorithm decides in what mode the ACC should be: velocity

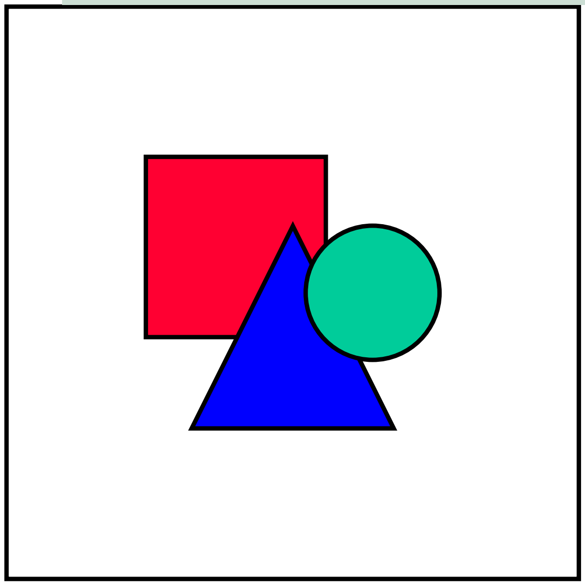
mode or headway mode. This control algorithm can be described using a Range/Range-Rate diagram. It consists of the following steps:

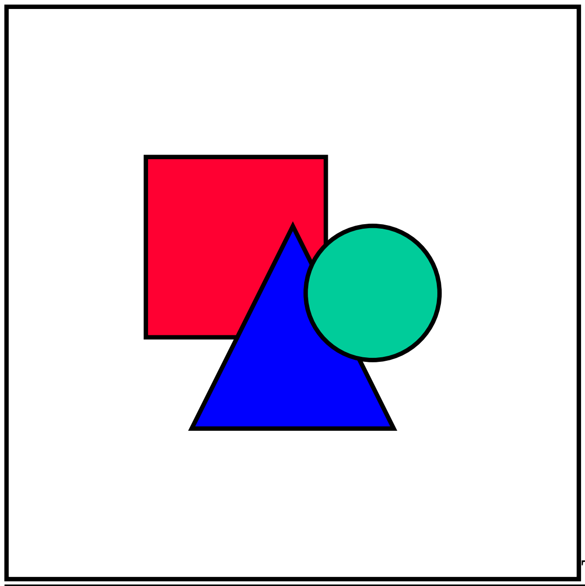
1. Choose the desired headway distance for a selected velocity and headway time.
   *RH=th*Vp;*
   (Desired headway distance = headway time * velocity of the preceding vehicle).
   Select point *A=(0,RH)*
2. Choose the deceleration *D* to be used.
3. Construct (plot) the parabola through *(0,RH)* using deceleration level *D*.
4. Choose the maximum range *Rmax* to consider.
   

5. Determine point *B* from intersection of *R*=*Rmax*  and constructed parabola .

6. Draw the line from point *A* to point *B* to determine its slope *(-T)*.

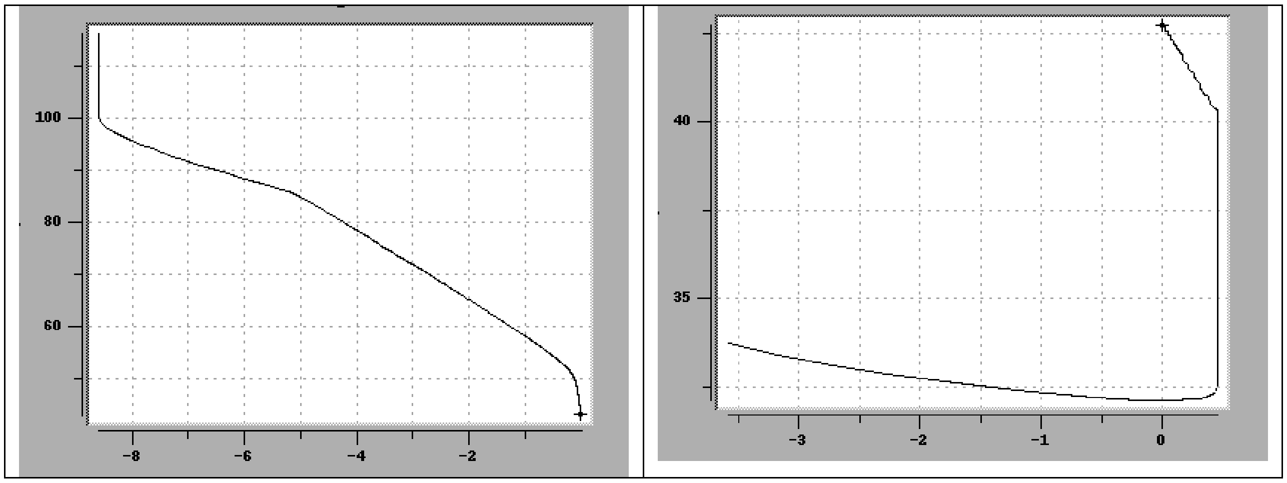






The line (AB) is the switching line between velocity and headway modes. For ranges above this line the system operates as a velocity control. The ACC vehicle tends to follow the speed set by the driver. It accelerates comfortably to the target speed. When the switching line is crossed from above, the system gets to headway control and applies the maximum available deceleration. Eventually trajectory reaches the switching line *(AB)* again. It tries to follow the line by applying the appropriate deceleration, and approaches target point *(0,RH)* exponentially.

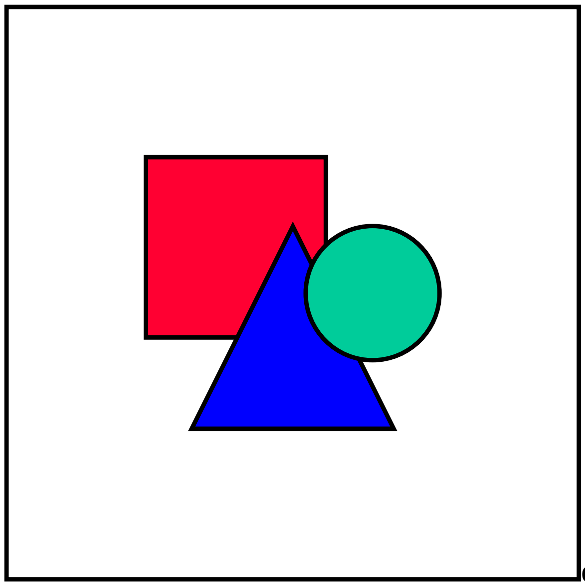
This algorithm can be illustrated in two situations: when approaching a slower moving vehicle, and when a disturbance vehicle suddenly appears within a range that is closer than the switching lane. The trajectories of the vehicle in both situations are portrayed in the figure below.



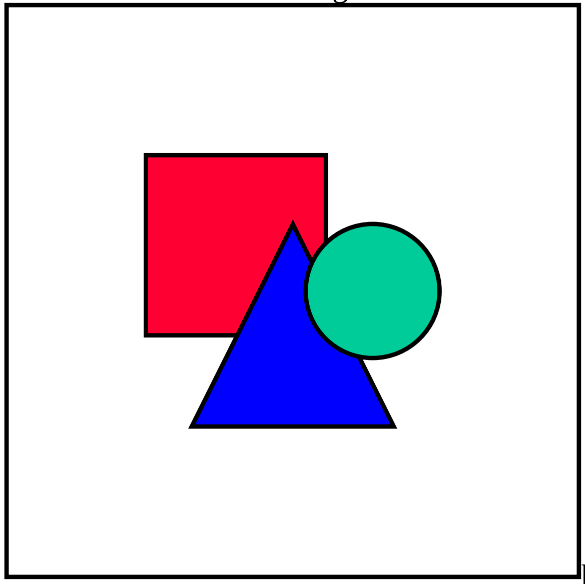
If the vehicle trajectory crosses the safety surface, then driver begins to act.

6.1.4.2 Driver model

We assume, that the driver wants to use the ACC as much as possible. The driver overrides the ACC when he feels unsafe. When the vehicle trajectory crosses the safety surface the driver begins to brake, more than the ACC, but not very hard. As mentioned above various levels of



deceleration of the lead vehicle determine various risks, which the driver accepts. In our simulation this parameter varied from 0.3g to 1.0g. If this level of deceleration is not enough, and at the some point the vehicle trajectory crosses a lower safety surface, then the driver brakes hard. This lower switching surface between hard and soft



braking is the safety surface with the parameter equal to 0.1g.

The driver continues to brake hard until he feels safe enough to give back control to the ACC.

6.1.5 Summary

The simulation is designed and implemented for evaluation and sensitivity studies of ACC. The simulation is designed such a way, that all components are interchangeable without affecting other components. One model of ACC without full braking authority is implemented. Batch facilities are provided to automate compilation and simulation runs. Data processing and display software was developed to facilitate data analysis.