SARNS: The SAOS Road Network Simulator

Kurt T. Rehfuss

Department of Computer Science

Oregon State University

rehfusk@research.cs.orst.edu

Major Professor : Prof. Toshimi Minoura

Contents

List of Figures

SARNS: The SAOS Road Network Simulator

Kurt T. Rehfuss
Department of Computer Science
Oregon State University
Corvallis, Oregon 97331-4602
rehfusk@research.cs.orst.edu

Abstract

The *SAOS Road Network Simulator* (SARNS) is a graphical simulation program for transportation planning, implemented using the *structural active-object system* (SAOS) approach. A SAOS is an *object-oriented concurrent* system that consists of a collection of interacting *structural active objects* (SAOs), whose behaviors are determined by *transition statements* provided in their class definitions. Furthermore, SAOs can be *structurally* and *hierarchically* composed from their component SAOs, allowing various applications to be rapidly developed as SAOS programs. The active components used in transportation planning, such as vehicles and traffic signals, can be naturally modeled as SAOs. The typical composition of these components into a complete road network allows for the rapid prototyping of various road network configurations and vehicle generation scenarios. An interactive graphical user interface displaying the dynamic state of the simulation is an inherent part of a SARNS program. Such a graphical interface can be directly created from a design specification, either by hand or in the future by using a SAOS graphical editor.

Key Words and Phrases: Transportation planning, computerized traffic simulation, object-oriented simulation, active-object systems, structural composition, hierarchical composition, rapid prototyping, graphical user interface

1 Introduction

With current traffic overloading existing facilities, the need for efficient planning to meet this increasing demand has become critical. Unfortunately, before computer models were available, the only way to observe the effectiveness of a given road network configuration was to spend a large amount of money building it. The use of computer simulation in transportation planning has become an accurate and cost effective alternative for dealing with the need to locate new and improved streets to meet the demand. Computerized traffic simulation models have been generally accepted by transportation planners because they provide the opportunity to quickly evaluate multiple network alternatives, produce replicable results, and permit the use of sophisticated models [LEWI86].

The technology available for computerized traffic simulation has advanced substantially since the first models were introduced in the 1950s. Since then, programs have progressed from mainframe computers with no graphic capabilities to large, fast microcomputers and PCs with interactive graphics and menu systems [LEWI90]. Today many computer simulation models are available for analyzing the various operating environments of road networks. These operating environments include signalized intersections, arterial networks, freeway corridors, and rural highways. Both microscopic and macroscopic computer simulation models have been developed for each of these operating environments [DAY90].

Computer simulation of traffic networks has many known benefits [ROSS77, DAY90] but also has several drawbacks. The most significant of these drawbacks is that many current simulation models have extensive data input requirements. In addition, the large amount of data these models require is difficult or impossible to obtain, casting doubt on the validity of simulation results. Despite the inclusion of graphical displays, network input and editing remained very tedious and time-consuming. This difficulty lead to the development of preprocessor programs to simplify the task of preparing and checking data inputs [DAY90].

In this paper, the SAOS Road Network Simulator (SARNS) is introduced. SARNS is a graphical simulation program utilizing an event-driven control mechanism in conjunction with a structural and hierarchical software construction methodology to model traffic on road networks. SARNS takes advantage of the *structural and hierarchical object composition* available through the SAOS approach to create large microscopic arterial traffic simulations from smaller fundamental components. In the field of transportation planning, these components are well defined [HORO87], and their structural interconnectivity is well understood. In addition, the network application schema for a transportation network lends itself to hierarchical construction and composition [HORO87], making it an ideal application for a computer simulation like SARNS.

The primary design goal for SARNS is the ease of simulating various scenarios in transportation planning. SARNS uses triggered callback methods to encapsulate the control of the system within the individual active components. This unique approach, combined with the ability to use structural and hierarchical composition of component objects in forming complete road network configurations, allows for rapid prototyping of network alternatives to meet the SARNS goal.

SARNS provides support for the creation of new road networks or the modification of existing ones. It potentially eliminates much of the tedious data input existing in current traffic simulation systems. A *graphical user interface* (GUI) is utilized to display the network configuration, giving users the ability to analyze traffic flow within the network, observe relative component loads, and detect gridlocks. The SAOS diagram created in the design phase is used as the basis for the simulator's user interface. A SAOS graphical editor can easily be constructed for use in conjunction with the GUI, allowing the network configuration to be edited while the simulator continues to run.

The current implementation of SARNS is a prototype model, permitting users to define an arbitrary street network, including user-defined turning movements and vehicle generation volumes. Simulation components can be interactively selected via a pointing device to view their states as the simulation progresses. Signalized intersection control and mechanisms for automatic evaluation of network effectiveness are not yet supported, but are planned for future versions of the simulator. A robustly developed system will allow transportation engineers the benefits of computer simulation to investigate traffic flow alternatives without the tedium of many existing models.

The rest of this paper is organized as follows: Section 2 describes the SAOS approach that SARNS is based on. Section 3 introduces how vehicles may be constructed as active objects and move within the road segments of the simulation. Section 4 details the basic other components and functionality of SARNS and describes how structural composition, hierarchical composition, and encapsulated control are utilized. Section 5 discusses the features and validity of the SARNS model while showing examples of how SARNS can be utilized to rapidly prototype a set of road network alternatives. Section 6 concludes the paper.

2 The SAOS Approach

Object-oriented programming (OOP) [GOLD80, MEYE88, STRO86] is making fundamental changes in software development. Such features as *encapsulated classes*, *inheritance*, and *polymorphism* provided by OOP allow us to implement highly modular reusable software components. Furthermore, since objects which embody state and behavior resemble real-world objects better than traditional software modules, object-orientation provides a suitable framework for software development [BOOC91, RUMB91].

A *structural active-object system* (SAOS) is an *object-oriented concurrent* system using *transition (production) rules*, *equational assignment statements*, and *event routines* for its behavior description. Production systems have been known to be suitable for various concurrent systems that require flexible synchronization [ZISM78]. The SAOS approach integrates object-orientation and production rules. The key mechanism used by SAOSs is *structural* and *hierarchical* composition of *structural active objects* (SAOs). Structural/hierarchical composition allows SAOs to be constructed from their component SAOs like hardware objects. Note that hardware objects are active autonomous objects. Structural and hierarchical composition is universally used in the design and implementation of such complex electronic and mechanical devices as VLSI chips and automobiles.

However, *passive objects* used in conventional OOP do not provide proper encapsulation or modularization of *control* when they are used to model *active objects* in the real world. A simple example is a (dumb) traffic signal that changes its state according to its own internal timer. When a traffic signal is implemented as a passive object, another object must send the traffic signal a message to let it change its state. When a traffic signal is implemented as an active object, it can completely hide from the outside its capability to change states like a real traffic signal.

The behavior of each SAO is determined in the *transition statements* provided in the *class* definition of that SAO. Each transition statement is a *transition rule*, which is a *condition-action* pair, an *equational assignment statement*, or an *event routine*. Equational assignment statements maintain simple invariant relationships among SAO states. Event routines are activated by messages. Since behaviors of SAOs are determined by user-programmed transition statements, classes for new types of components can be easily added.

The SAOS approach primarily uses structural composition of SAOs whereas conventional OOP uses procedural interfaces provided for passive objects. In order to support structural composition of SAOs, *interface variables,* which resemble terminals of hardware components, are used to interconnect SAOs. These interface variables are crucial for structural composition, through them each SAO can access, besides the state of that object, the states of the other objects known to it through its interface variables, thus realizing inter-object communication.

We can establish interconnections among SAOs by binding objects to their interface variables. This binding is performed when a composite SAO is constructed from their component SAOs. In a sense, binding objects to interface variables is analogous to wiring VLSI chips.

SAOs can be modularized in the same way as hardware objects are modularized. Structural composition of active objects with standardized interfaces provides the basis for realization of software ICs as espoused by Cox [COX87]. Since SAOs are software objects, they are more flexible than hardware objects. For example, such features as inheritance and polymorphism in the standard OOP can be used in defining SAOS classes.

Besides modeling functionalities of components, SAOSs provide dynamic (animated) graphical user-interfaces. A SAOS diagram, which graphically represents the structure of a SAOS as a collection of interconnected SAOs, can be made structurally similar to the system to be simulated. Therefore, a SAOS diagram can be used as the basis of a system design document, actual executable code, and user interface.

SAOS user-interfaces are supported by the Active Object User-Interface Management System (AOUIMS) [CHOI92], which can be used in conjunction with a SAOS graphical editor. AOUIMS allows us to create dynamic (animated) graphical user interfaces by structural and hierarchical composition and to specify the behaviors of graphical objects by transition statements. Low-level SAOs are textually coded, but their graphical representations can be added with small extra effort. High-level SAOs, which are executable, can be created directly as SAOS diagrams by graphical SAOS editors. It is also possible to generate textual representations from SAOS diagrams.

The SAOS approach is a new programming paradigm for object-oriented concurrent systems. The approach is suitable for a system that consists of interconnected components interacting with each other asynchronously. It can be applied to such diverse application areas as discrete simulation [CHOI91], process and manufacturing control [MINO93], hardware logic simulation, graphical user interfaces [CHOI92], graphical editors, and algorithm animation. For these applications, the SAOS approach provides a single framework that can be used throughout a software lifecycle including the analysis, design, implementation, test, operation, and maintenance stages. This paper demonstrates how the SAOS approach can be applied to transportation planning simulation.

3 Traffic Simulation Using The SARNS Approach

SARNS utilizes the four basic component types inherent to transportation planning: `Vehicles`, `Terminals`, (road) `Segments`, and `Intersections`. In this section, we show how a `Vehicle` can be modeled as an SAO. The corresponding movement of `Vehicles` within `Segments` is described by a simple example.

3.1 Vehicles as Structural Active Objects

`Vehicles` are the most significant active object of the SAOS Road Network Simulator. Information regarding the activity and efficiency of a particular simulation configuration can be acquired by observing the `Vehicles` as they appear to physically move within the confines of the `Segments` of the network.

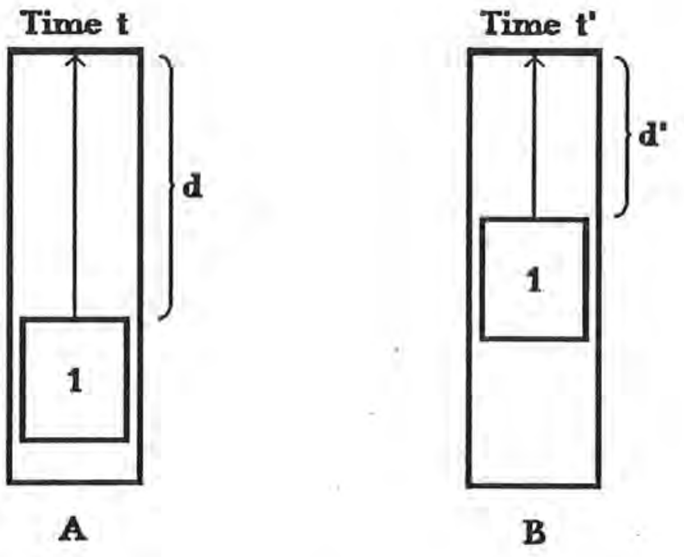


Figure 1: Vehicle Within A Road Segment

Fig. 1 shows the movement of a single vehicle within a road segment. Fig. 1a shows vehicle 1 within a road segment at time `t`. To move properly within a road segment and avoid collisions, the only essential information a vehicle in the real world needs to know outside of its internal state is the vehicle ahead of it and the state of the next component (intersection or terminal) that it will encounter. Since there are no vehicles ahead of vehicle 1, the distance `d` to the next component (the end of the segment) is used in determining how the vehicle will move in the next timestep. Fig. 1b shows the new position of vehicle 1 at time `t'`.

Fig. 2 shows the class definition of a `Vehicle` when modeled as a SAO. Note that the `Vehicle` need only be designed once. Once this component has been created, it can be duplicated to create an entire set of `Vehicles` upon demand [HORO87]. Fig. 3 gives the corresponding class definition for a road `Segment`.

```
class Vehicle : public Arc {
 public:
   ComType          componentType;      // Enumerated Type
   int              vehicleID;          // Unique ID for vehicle instance
   VehState         vehState;           // Enumerated Type
   float            speed;              // speed of vehicle movement
   float            acceleration;       // acceleration coefficient
   Vehicle *        vehAhead;           // Closest vehicle ahead
   Segment *        currentSeg;         // Segment the vehicle is currently in
   Intersection *   nextIntersection;   // Next intersection to stop at
   Terminal *       nextTerminal;       // Next terminal to stop at
   float            ncDistance;         // Distance Remaining to next component
   Vehicle();                           // constructor
   virtual virtual void initialize();
   void         Display_Instance();     // Method to display contents of instance
   void         vehicleSelected();      // Method to display within control panel
 private:
   void         runTimeChanged();
   float        modifyAcceleration(float, float, float);
   float        modifyLocation();
};  /* Vehicle Class */
```

Figure 2: Vehicle Class.

```
class Segment : public Line {
 public:
   ComType     componentType;       // Enumerated Type
   int         segmentID;           // Unique key for a road segment instance
   Float       length;              // Travel length of the road segment
   int         speedLimit;          // Segment speed limit
   void *      source;              // Component that feeds the segment.
   void *      dest;                // Component the segment empties into.
   int         turningPct;          // Percent of vehicles routed to this segment
   Float       travelLength;        // Distance a vehicle travels on the segment
   Vehicle *   lastCar;             // Most recent car to enter the segment
   Segment();                       // Constructor
   void        initialize();
   void        Display_Instance(); // Method is display contents of instance
};   /* Segment Class */
```

Figure 3: Segment Class.

Vehicle Class. When modeling a vehicle as a SAO, only the essential information should be encapsulated into its class definition. Each `Vehicle` in the simulation maintains its own internal state, consisting of the `Vehicle` location, speed, and acceleration. As with real-world vehicles, the only external information a `Vehicle` must keep track of to travel within a road segment is the location/status of the next component and the location/status of the `Vehicle` directly ahead of it (if any). Thus to avoid collisions and properly travel within a `Segment`, a `Vehicle` also maintains a pointer to both to the `Vehicle` directly ahead of it and the component (`Intersection` or `Terminal`) at the end of the segment.

A `Vehicle` proceeds along a `Segment` using these parameters. In each timestep, a `Vehicle` may do one of the following:

1. Be generated by a source `Terminal` and enter the configuration.
2. Be routed to a sink `Terminal` and exit the simulation.
3. Proceed along a `Segment`, optionally accelerating or decelerating while avoiding collisions with other `Vehicles`.
4. Enter an `Intersection` to be routed to the next `Segment`.

Segment Class. The class definition for the `Segment` class is shown in Fig. 3. Instances of this class connect `Intersections` and `Terminals`. `Vehicles` travel within these components throughout the simulation. A `Segment` may be one of three types: *EntrySegs* connect source `Terminals` to `Intersections`, *ExitSegs* connect `Intersections` to sink `Terminals`, and *TravelSegs* connect two `Intersections`. `Segments` are unidirectional, meaning that `Vehicles` travel from the source components to destination components but not the reverse.

A `Segment` must know its type, the source and destination components which it connects, and the length (distance) between those components. In addition, it must know the percentage of `Vehicles` which enter the `Segment` from the source component, the start location for `Vehicles` entering the `Segment`, the travel length to the destination component, and the speed limit for `Vehicles` traveling within it. Lastly, a `Segment` must keep track of the last Vehicle to enter it. When a new Vehicle enters the Segment, this lastCar value can be used to determine the vehAhead value for the new Vehicle.

The functionality for a Segment is encapsulated in it's method definitions. Segments have two external interface variables used for the structural composition of complete network configurations. As might be expected, these interface variables are the pointers to the source and destination components that the Segment instance connects.

3.2 Simulating Vehicle Movement in SARNS

A Vehicle spends a majority of its time moving within the Segments of the simulation. When a Vehicle enters a Segment, its id is used to set lastCar, replacing the previous lastCar. This previous lastCar value is used to set vehAhead for the new Vehicle.

```
if (vehAhead != NULL)
  if ((vehAhead -> myVisible) == FALSE)     vehAhead = NULL;

if (vehAhead != NULL) {
  float xdelta = xRel - (vehAhead->xRel);
  float ydelta = yRel - (vehAhead->yRel);
  float distance = (sqrt((xdelta * xdelta) + (ydelta * ydelta)));

  acceleration = modifyAcceleration(distance, speed, (vehAhead -> speed));
  speed = speed + acceleration;
  . . .
 };
```

Figure 4: Initial Vehicle Movement Code Fragment.

Fig. 4 shows the initial code fragment for a Vehicle moving within a Segment. To prevent collisions, the vehAhead pointer is first checked. If this pointer is not NULL, its value is used to check if the Vehicle ahead has left the Segment in the last timestep. If the Vehicle ahead has left the Segment, vehAhead is reset to NULL.

If vehAhead remains not NULL, there is a Vehicle ahead of the current Vehicle to be considered for collision avoidance. The distance to the Vehicle ahead is used as a parameter in modifying the current Vehicle's speed and acceleration. The value of ncDistance (distance to the next component) is ignored in these computations, since the Vehicle pointed to by vehAhead must be closer to the current Vehicle than the next component.

If vehAhead is NULL, there is no Vehicle ahead of the current one. In this case, the distance to the next component (ncDistance) is checked to see if the Vehicle has reached the end of the Segment. If so, the Vehicle exits the Segment and is routed to next Intersection or Terminal. If the current Vehicle is also the last car to enter the Segment, lastCar is set to NULL. If the Vehicle remains in the Segment, the Vehicle acceleration, speed, and location are modified as before. Fig. 5 shows the code fragment for this case.

```
if (ncDistance <= VehicleGap)     // At Intersection or Terminal
 {
  . . .
  if (((nextIntersection) -> blocked) == FALSE)       // Give to intersection
   {
     ((nextIntersection) -> currentVehicle) = this;
     ((nextIntersection) -> blocked) = TRUE;          // Set Trigger Variable
     if ((currentSeg->lastCar)->vehicleID == vehicleID)        // Is this the
       { currentSeg->lastCar = NULL;   currentSeg = NULL; }   // last car?
   }
  . . .
  return;   // Don't modify acceleration
 }
acceleration = modifyAcceleration(ncDistance, speed, 0);
speed = speed + acceleration;
. . .
```

Figure 5: Vehicle Movement Code Fragment - No Vehicle Ahead.

Figure 6 visually details the process of vehicle movement when more that one Vehicle is within a Segment. Figure 6a shows the Segment when a second Vehicle (2) enters the simulation behind the initial Vehicle (1). In this case, the distance d2 to Vehicle 1 is used in determining how Vehicle 2 will move in the next timestep.

Figure 6b shows the Segment after Vehicle 1 has exited. In this case, Vehicle 1 is determined to no longer be ahead of Vehicle 2. Thus vehAhead for Vehicle 2 becomes NULL and it's associated value is no longer considered in the movement of Vehicle 2. The distance to the end of the segment (ds') is sufficient for this purpose.

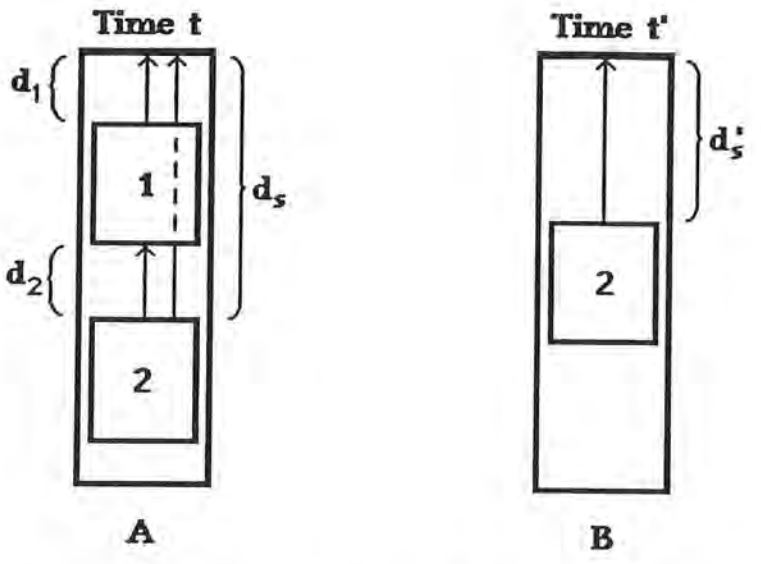


Figure 6: Second Vehicle Within Road Segment

4 Simple Traffic Network Simulation

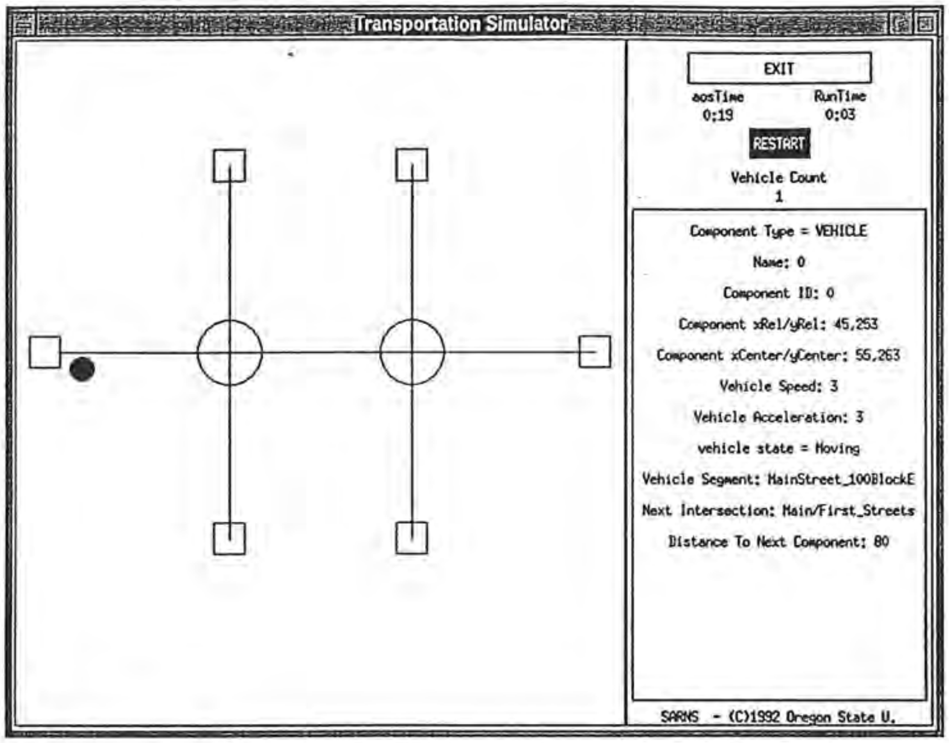
In this section, the remaining components and functionality of the SAOS Road Network Simulator are introduced by another simple example. The key features of the SAOS approach as they apply to SARNS will briefly be discussed here.

4.1 Sample Network Configuration

A SARNS network configuration takes the form of a bidirectional graph consisting of nodes (`Intersections` and `Terminals`) and links (`Segments`). Transportation networks can be entirely represented using only these two fundamental graphical elements [HORO87].

Fig. 7. shows the resulting SARNS simulation derived from the network components defined in Fig. 9. This sample road network configuration consists of six `Terminals` and two `Intersections`, all connected via road `Segments`.

Traffic flow within SARNS proceeds from the source `Terminals` to the sink `Terminals`. Each source `Terminal` waits a defined period of time and then generates a `Vehicle`. It



```
aosTime      RunTime
 0:19         0:03
Vehicle Count
      1
Component Type = VEHICLE
Name: 0
Component ID: 0
Component xRel/yRel: 45,253
Component xCenter/yCenter: 55,263
Vehicle Speed: 3
Vehicle Acceleration: 3
vehicle state = Moving
Vehicle Segment: MainStreet_100BlockE
Next Intersection: Main/First_Streets
Distance To Next Component: 80
SARNS  - (C)1992 Oregon State U.
```

Figure 7: Sample Network Configuration

passes that `Vehicle` to it's adjacent `Intersection` through a connecting road `Segment`. The `Intersection` routes the `Vehicle` to another `Segment` towards either another `Intersection` or a sink `Terminal`. When a `Vehicle` is routed to a road `Segment`, the simulated `Vehicle` appears within the `Segment`. When a `Vehicle` is routed to a sink `Terminal`, it exits the simulation.

```
class TranSim : public TopLevel {
 public:
   NetWindow       netWindow;
   ControlPanel    controlPanel;
   TranSim(char*);
   void            initialize();
   void            setWindowSize();    // callback function to set window size
};  /* TranSim Class - TopLevel Class */
```

Figure 8: Transportation Simulation Main Class.

Fig. 8 shows the class definition for the SARNS simulator main program. As with other SAOS programs, this main program takes the form of a top-level class definition. In the case of SARNS, this top-level component consists of a graphical window encompassing the `netWindow` and `control panel` sub-components.

The netWindow displays the road network configuration currently being simulated. Individual components within the simulation can be selected via a pointing device such as a mouse. When a component is selected in this manner, a set of instance variables for the component is displayed in the control panel. The simulation can be paused or run continuously, allowing the user to view the interaction of traffic within a varied range of road network configurations and vehicle generation scenarios.

```
class NetWindow : public Rectangle {
 public:
   Intersect_List *   intList;
   SourceTerm_List *  sourceList;
   SinkTerm_List *    sinkList;
   Segment_List *     entrySegList;
   Segment_List *     exitSegList;
   Segment_List *     travelSegList;
   NetWindow();
   void               initialize();
   void               Display-Instance();
   void               insertVehicle(Vehicle *);
   Intersect_List *   readIntersectEntries(Intersect_List *);
   SourceTerm_List *  readSourceTermEntries(SourceTerm_List *);
   SinkTerm_List *    readSinkTermEntries(SinkTerm_List *);
   Segment_List *     readEntrySegEntries(Segment_List *,
                                         Intersect_List *, SourceTerm_List *);
   Segment_List *     readExitSegEntries(Segment_List *,
                                        Intersect_List *, SinkTerm_List *);
   Segment_List *     readTravelSegEntries(Segment_List *, Intersect_List *);
};   /* NetWindow Class */
```

Figure 9: Simulator Network Window.

Simulator Network Window Class. Fig. 9. shows the class definition for the SARNS NetWindow class. The netWindow is the generic class which defines the set of components used in the simulation. This class definition is the same for all possible road configurations, including Fig. 7. The methods of this class read a set of auxiliary text files which contain the specification for the configuration currently being simulated.

Once the set of simulator components has been declared through the netWindow methods, structural composition is utilized to connect the component instances through their individual interface variables. These same interface variables are used to set up triggers which encapsulate the control of the system.

The components defined here and their connections defined in the auxiliary text files can be quickly modified to allow the rapid prototyping of multiple road network configurations. The Vehicle class is not included in the netWindow class definition, since Vehicle instances are created dynamically within individual runs of the simulator and thus the number of Vehicles is not known at initialization time.

A discussion of how the components are connected through their interface variables, the use of triggered *callback methods* to encapsulate control, and how the use of SARNS provides the potential for the rapid prototyping of a number of possible road network configurations is deferred to later sections.

To complete the description of network simulator, we must of course define the classes for the remaining component objects used by the system.

```
class Terminal : public Rectangle {
 public:
   ComType     componentType;          // Enumerated Type
   Vehicle *   currentVehicle;         // Most recently created vehicle
   Int         blocked;                // TRUE if occupied by a vehicle
   int         termID;                 // Unique instance identification number
   int         vehicleCount;           // # that have gone through the terminal
   Segment *   port;                   // Segment the terminal is connected to
   int         vehiclesPerHour;        // Number of vehicles generated per hour
   int         minGenerationInterval;  // Minimum Interval between generations
   int         generationRange;        // Range of randomness in interval
   Terminal();                         // Constructor
   virtual virtual void initialize();
   void        terminalSelected();     // Method to display contents of instance
   void        Display_Instance();     // Method to display within control panel
};  /* Terminal Class */
```

Figure 10: Terminal Class.

Terminal Class. The class definition for the Terminal class is shown in Fig. 10. Terminals are the endpoints of the simulation. A Terminal instance may be either a source

```
class Intersection : public Arc {
 public:
   MyTimer     tm;                     // Measures duration of vehicle blockage
   ComType     componentType;          // Enumerated Type
   int         intID;                  // Unique instance identification number
   Int         blocked;                // TRUE if occupied by a vehicle
   Vehicle *   currentVehicle;
   int         vehicleCount;           // # that have gone through the intersection
   Segment_List *  toPorts;            // connections vehicles can be routed to
   Segment_List *  fromPorts;          // connections vehicles can be received from
   Intersection();
   void        initialize();
   int         Check_Data();
   void        Display_Instance();      // Method to display contents of instance
   void        intersectSelected();     // Method to display within control panel
   void        Start_Routing();
   void        Finish_Routing();
   int         sendToSegment(Segment *);  // Routes 'currentVehicle' to Segment
};  /* Intersection Class */
```

Figure 11: Intersection Class.

Terminal or a sink Terminal. Vehicles enter the simulation via source Terminals and leave via sink Terminals. Source Terminals are responsible for producing Vehicles at defined intervals and introducing these same Vehicles into the simulation. Sink Terminals have the corresponding task of removing vehicles from the simulation that have been routed to them. A Terminal should keep track of its location, the Segment to which it is attached, and the number of Vehicles that have passed through it. It should also know if it is currently occupied (blocked) by a Vehicle and the identity of said Vehicle.

The functionality for a Terminal instance is encapsulated in its method definitions. Terminals have a single external interface variable for structural composition which is a pointer to the Segment to which the Terminal is attached.

Intersection Class. An Intersection, whose class definition is shown in Fig. 11, exists where two or more Segments cross. Vehicles are routed at an Intersection according to the turning movements of the associated Segments. An Intersection should know its location, the components (segments) to which it is attached, and the number of `Vehicles` that have passed though it. Similar to `Terminals` in the routing of `Vehicles`, it should also know if it is currently occupied (blocked) by a `Vehicle` and the identity of this `Vehicle`.

As with the other simulator components, the functionality of an `Intersection` instance is encapsulated in its method definition. Different from the other components are an `Intersection`'s external interface variables. Since structural composition can be used to construct arbitrary network configurations, any single `Intersection` instance may have an arbitrary number of `Segments` which connect to it. Thus, the interface variables for an `Intersection` take the form of a linked list of these `Segment` instances.

4.2 Creating Complete Networks Through Structural Composition

As mentioned previously, to simply declare the set of simulator components in the SARNS main program is not enough to form a functional simulation. Structural composition must be utilized to connect the appropriate components through their interface variables.

```
Segment       aTravelSeg;
Intersection  intersect1;
Intersection  intersect2;

aTravelSeg->source = &intersect1;
aTravelSeg->dest = &intersect2;
intersect1->toPorts) = add(&aTravelSeg);     // add to list of toPorts
intersect2->fromPorts) = add(&aTravelSeg);   // add to list of fromPorts
```

Figure 12: Sample Segment to Intersection Connection

Sample Connection. In Fig. 12, `aTravelSeg` is a travel segment with `intersect1` as its source and `intersect2` as its destination. This code fragment is an example of structural composition, setting the interface variable values for each component. The travel segment's source pointer is set to `intersect1`, while the destination pointer is set to `intersect2`. The travel segment is also added to `intersect1`'s list of segments that can be routed to, and `intersect2`'s list of segments that can be routed from.

Structural composition via interface variables for Source Terminal to Intersection connections (entrySegs) and Intersection to Sink Terminal connections (exitSegs) is done similarly; thus allowing each component instance in the simulation to know which other component instances it is connected to. It is through these interface variables that a component can access the states of other components, making inter-object communication throughout the simulation a reality.

4.3 Trigger Setup For Localized Control

Control in a conventional object-oriented program is passed by messages or procedure calls. In both of these cases, methods or procedures can be activated only by explicit external stimulations, which eventually must come from a main program. In a SAOS such as SARNS, control is localized within an individual simulation object. Thus the simulation program itself does not control the sequence of events (such as through a main program). Instead each component implements its own portion of control to initiate its operations. These operations are written as *callback methods* which are executed whenever certain activation events occur.

```
Int         blocked;              // TRUE if occupied by a vehicle
```

Figure 13: Condition Variable for Intersection Class.

```
PROC pf1 = PROC (&(Intersection::Start_Routing));
blocked.tl.addTE((AObject *) this, pf1, ''Start_Routing()'');
```

Figure 14: Trigger Setup for Start Routing() Method.

In SAOS, events which trigger callback methods are controlled through *condition variables.* The type `Int` designates a condition variable for an integer. Fig. 13 shows a fragment of the `Intersection` class definition in which the `blocked` instance variable is designated as a condition variable of type `Int`. A condition variable maintains a list of pointers to functions, called a *trigger list.* Whenever the value of a condition variable is updated, the functions pointed to by the elements of the trigger list are executed. In the case of an `Intersection`, the method `Start Routing()` is activated whenever the `blocked` condition variable is updated. Fig. 14. shows the code fragment required to set up this trigger.

```
(dest -> currentVehicle) = currentVehicle;
(dest -> blocked) = TRUE;
```

Figure 15: Code Fragment to Trigger Start Routing Method.

Interface variables are an essential part of encapsulating control within a component. Rather than explicitly calling a method for another component, components can trigger the callback methods of the components that they are connected to through their interface variables. Thus, when a `Vehicle` arrives at the end of a road `Segment` having an `Intersection` as it's destination, the vehicle can be passed to that `Intersection` using a code fragment similar to Fig. 15. By modifying the `blocked` condition variable, a message is effectively sent to the `Intersection`, asking it to route the current vehicle.

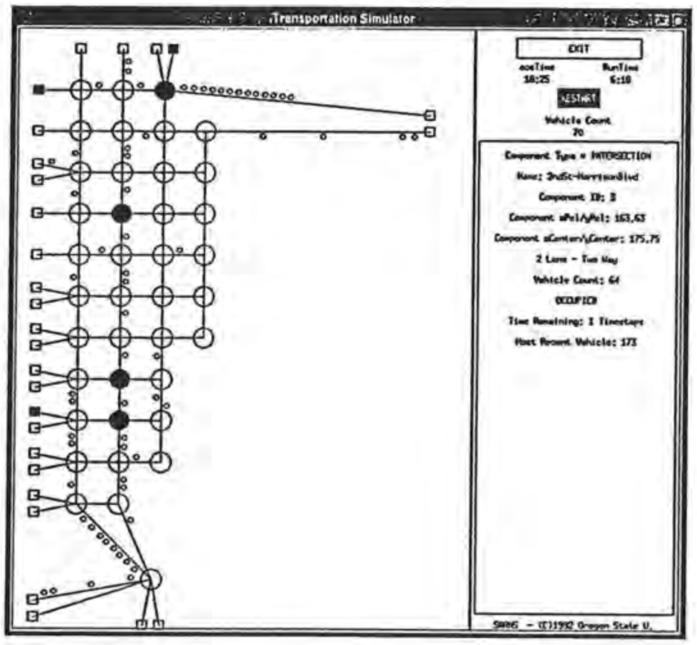
This section has briefly introduced the basic components and functionality of the SAOS Road Network Simulator. Note that much larger road network configurations than this simple example can be created easily using structural composition through interface variables, while encapsulation of control allows all functionality for SARNS to be handled either directly or indirectly through triggered callback methods. These concepts are put together in the next section to show how SARNS can be used to rapidly prototype multiple road network scenarios.

5 Discussion

In this section, examples are shown which detail how SARNS can be utilized to rapidly prototype a set of road network alternatives. In addition, the current features of the simulation system are discussed, together with a description of future work planned for SARNS.

5.1 Rapid Prototyping of Network Alternatives

For any given transportation application, there are only a few component types. Once defined, these components can be repeated hundreds or even thousands of times to produce all the component instances in an entire network. In SARNS, the specifications for the entities which comprise a given road network configuration are read in from external files, giving the application source code independence from the individual configuration of the network. These features, combined with the structural and hierarchical object composition available through the SAOS approach, allow for rapid prototyping of road network scenarios.



acelTime 18:25 RunTime 6:10

Vehicle Count 70

Component Type = INTERSECTION

Name: 2ndSt-HarrisonBlvd

Component ID: 3

Component xPel/yPel: 163,63

Component xCenter/yCenter: 175,75

2 Lane – Two Way

Vehicle Count: 64

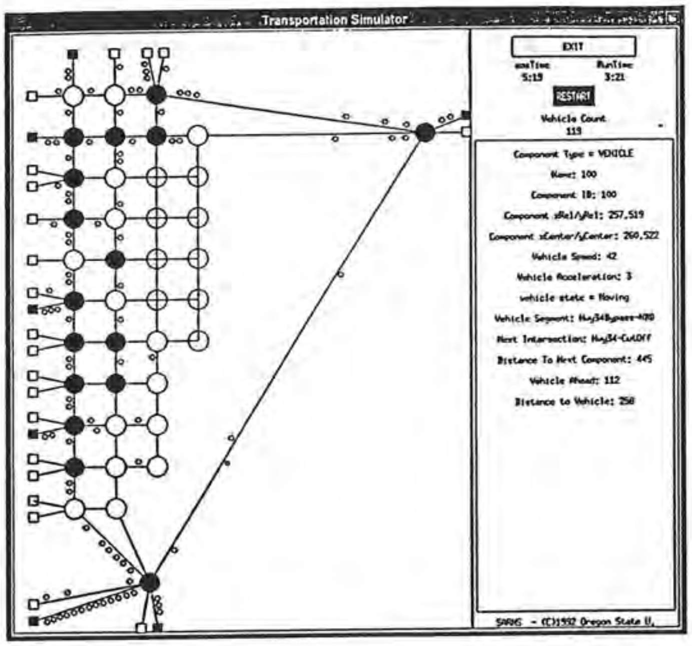
OCCUPIED

Time Remaining: 1 Timesteps

Most Recent Vehicle: 173

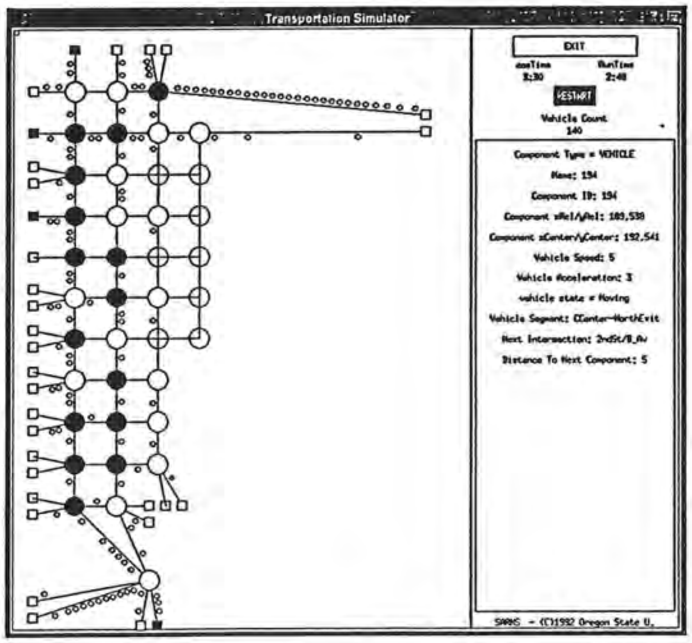
Figure 16: City Center Network

City Center Network . A complex road network configuration consisting of approximately 200 components is represented in Fig. 16. This network is a modest representation of the city center street grid for Corvallis Oregon. As is shown by the backup of vehicles, the network at times is saturated, resulting in unreasonable delay times for commuters. City planners considered the alternatives of building an arterial to bypass the downtown grid and relieve congestion, while at the same time pursuing the development of a downtown convention center which could significantly increase traffic loads in the area.



```
simTime      RunTime
5:19         3:21
Vehicle Count
119
Component Type = VEHICLE
Name: 100
Component ID: 100
Component xRel/yRel: 257,519
Component xCenter/yCenter: 260,522
Vehicle Speed: 42
Vehicle Acceleration: 3
vehicle state = Moving
Vehicle Segment: Hwy34Bypass-400
Next Intersection: Hwy34-CutOff
Distance To Next Component: 445
Vehicle Ahead: 112
Distance to Vehicle: 250
SARS - (C)1992 Oregon State U.
```

Figure 17: City Center Network With Bypass



```
simTime      RunTime
3:30         2:48
Vehicle Count
140
Component Type = VEHICLE
Name: 194
Component ID: 194
Component xRel/yRel: 189,538
Component xCenter/yCenter: 192,541
Vehicle Speed: 5
Vehicle Acceleration: 3
vehicle state = Moving
Vehicle Segment: CCenter-NorthExit
Next Intersection: 2ndSt/B_Av
Distance To Next Component: 5
SARS - (C)1992 Oregon State U.
```

Figure 18: City Center Network With Conference Center

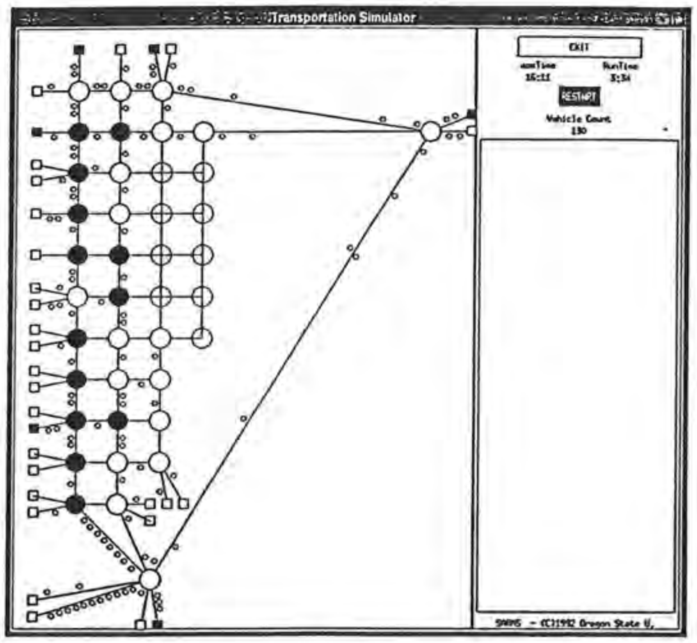


Figure 19: City Center Network With Bypass And Conference Center

Figs. 17 - 19 detail how the options considered by the city's planners can be modeled using SARNS. Adaptation of the network to include the bypass as shown in Fig. 17 requires minor modification/addition of only 12 lines of the original simulation's configuration files. Fig. 18 shows the network when the proposed conference center is included (lower, right corner of the main grid, a vehicle exiting the center is highlighted). This adaptation of the network again requires minor modification/addition of only 12 lines of the original simulation's configuration files. Adaptation of the network to include both proposed additions as shown in Fig. 19 requires both modifications above, that is, the changing of only 24 lines in the simulation's configuration files.

It is important to note that SARNS was not available when these modifications where being considered, thus the application of SARNS to the previous set of configurations is only an exercise. The simplicity of the modifications required nonetheless shows that significant road network alternatives can be rapidly prototyped using SARNS.

5.2 Current Features of SARNS

SARNS has the ability to rapidly prototype road network configurations; however, as a usable transportation planning tool, the current version of SARNS lacks some important features. For example, signalized intersection control is not supported, and a statistical package to evaluate *methods of effectiveness* (MOEs) automatically for the network is also missing.

Accurate data and figures from other research were used whenever possible in determining the values of variables used in the simulation. Values used for the maximum acceleration of a vehicle in one timestep (taken from [DAY90]) are an example of this. However, there is no way to validate the model without performing comparative analysis in the field. It should be noted that the current version of SARNS is a prototype. As such, validation of the SARNS model goes beyond the current scope of the project within which it was designed.

The graphics in SARNS are rudimentary at best; however, the accuracy of the visual representation for transportation networks is not of paramount importance. A visually accurate drawing is impossible to produce for some networks and impractical for others. The graphics need only be sufficiently informative so that the user can readily identify what each component is supposed to be [HORO87].

SARNS `Vehicle` *generation volumes* are specified in terms of the number of vehicles per hour (VPH) to be generated by a source `Terminal`. This volume is used to seed a random number generator according to which the interval between `Vehicles` is determined. This approach differs greatly from most other simulators which usually assume a method based on a probability distribution. It should be noted, however, that this assumption is often violated in real life [COHE77]. Given this tradeoff, the VPH method was judged easier to implement.

Turning movement percentages are attached to each road `Segment` in the simulation. A `Vehicle` is routed to a `Segment` according to these turning percentages. Most other simulation models also use turning percentages to route vehicles within the simulation. A typical alternative is to utilize an *origin-destination* (OD) *matrix* which defines the origin and destination nodes for every `Vehicle` in the simulation. However, (OD) matrices are rarely available [SHEF87], while turning percentages are a standard part of traffic counts and are easily accessible by traffic engineers [ROSS77].

The largest drawback of the SARNS prototype is the lack of intersection control. This feature can be implemented with little difficulty; however, due to time considerations, was not included in the current version of the system. Regardless, SARNS has the potential to overcome this deficiency in over-capacity situations, where traffic signals effectively become useless and transportation engineers become more concerned with bottlenecks and maximizing traffic flow than signal optimization. Such situations include emergencies (such as natural disasters), special event overflow, and rush-hour simulation.

5.3 Future Work

Several additions to SARNS would add to its robustness, making it potentially suitable for actual application in the field. These future additions include specific traffic simulation-oriented features such as:

1. Signalized intersections, including non-actuated, semi-actuated, and actuated options.
2. A range of green time options.
3. Turning pockets and lanes.
4. A statistical package allowing for the automatic collection of MOE data.

Past adding specific traffic simulation functionality, the inclusion of a SAOS editor is the most significant feature to be added to the SAOS Road Network Simulator. This is because the most time consuming part of creating a SARNS network configuration is the task of specifying the coordinates necessary to layout the components. Currently this must be initially done by hand. A SAOS editor can complement the existing GUI and be used to construct the component-declarations part of a SAO class on the display screen, including the locations and interconnections of components.

The creation of a SAOS editor for SARNS has the following advantages that will further enhance the ability of SARNS to rapidly prototype various road network configurations:

1. A top-level SAOS program can be automatically generated from the network created by the editor.

2. This network is used as part of the user interface for the running simulation.

3. This network can be edited while the simulation is running.

6 Conclusions

In a society where more and more traffic is overloading existing road systems, the use of computers to facilitate efficient design and modification of transportation systems has effectively replaced traditional pen and paper methods. The SAOS Road Network Simulator (SARNS) is a graphical simulation program utilizing event-driven control mechanisms in conjunction with structural and hierarchical software construction methodology to model road network alternatives.

SARNS utilizes the SAOS features of *structural and hierarchical object composition* and encapsulated control to create a system in which complete road network configurations are created from a small set of structural active objects (SAOs). These SAOs are connected via interface variables to form a larger interconnected simulation system. SARNS encapsulates the functionality and control of the system within the individual active components using condition variables and triggered callback methods. Whenever the value of a component's condition variable is assigned, the corresponding callback method is executed.

The SARNS graphical user interface (GUI) allows the user to test a scenario, quickly establish the validity of a proposed road network configuration, and locate further enhancements to the network. A SARNS graphical editor can be constructed for use in conjunction with the GUI, potentially eliminating the tedious data input existing in current traffic simulation systems.

The features of SARNS give it the ability to rapidly prototype road network configurations. When robustly developed, SARNS has the potential to be a valuable tool for transportation engineers to use in investigating network alternatives without wasting valuable resources in the field. The functionality of SARNS makes it very useful in a variety of situations, including:

1. Adding road `Segments`, `Intersections`, and `Vehicle` sources to model new development and growth.

2. Modifying signal timings and intersection controls to experiment with traffic flow control alternatives.

3. Increasing traffic volumes in the simulation to model increased load on the network due to rush hour conditions or new growth.

References

[BOOC91] Booch. *Object Oriented Software Design.* Benjamin/Cummings, 1991.

[CHOI91] Choi, S., and Minoura, T. Active object system for discrete system simulation. Proc. Western Simulation Multiconference, 1991, pp. 209-215.

[CHOI92] Choi, S. and Minoura, T. *User interface system based on active objects.* Proc. 2nd Symp. on Environments and Tools for Ada, Jan. 1992.

[COHE77] Cohen, S. L. Analysis of traffic-generated carbon monoxide pollution. In *Simulation Councils Proceedings Series, 7*, 2, Bernard, J. E. (Ed), Simulation Councils, Inc, 1977, pp. 207-211.

[COX87] Cox, B. J. *Object Oriented Programming : An Evolutionary Approach*, Addison Wesley, 1987.

[DAY90] May, A. D. *Traffic Flow Fundamentals.* Prentice-Hall, 1990.

[GOLD80] Goldberg, A., Robson, D. *Smalltalk-80: The language and its implementation.* Addision-Wesley, 1983.

[HORO87] Horowitz, A., Pithavadian, A. Generalized Computer Aided Design of Transportation Networks, *Transportation Quarterly, 41*, 3 (July 1987), 397-409.

[LEWI86] Lewis, S., McNeil, S. Developments in Microcomptuter Network Analysis Tools for Transportation Planning, *ITE Journal, 56*, 10 (Oct. 1986), 31-35.

[LEWI90] Lewis, S., Cook, P., Minc, M. Comprehensive Transportation Models: Past, Present and Future, *Transportation Quarterly, 44*, 2 (Apr. 1990), 249-265.

[MEYE88] Meyer, B. *Object-Oriented Software Construction.* Prentice-Hall, 1988.

[MINO93] Minoura, T., Choi, S., and Robinson, R. Structural active-object systems for manufacturing control. Special issue of *Heuristics* on object-oriented intelligent systems for manufacturing, 1993, to appear.

[ROSS77] Ross, P., Gibson, D. R. P. Survey of models for simulating traffic. In *Simulation Councils Proceedings Series, 7* 1, Bernard, J. E. (Ed), Simulation Councils, Inc, 1977, pp. 39-48.

[RUMB91] Rumbaugh, J., et al. *Object-Oriented Modelling and Design.* Prentice-Hall, 1991.

[SHEF87] Sheffi, Y., Barhhart, C. XNET: Extended Traffic Assignment Model, *Journal of Transportation Engineering, 113*, 7 (July 1987), 450-462.

[STRO86] Stroustrup, B. *The C++ Programming Language.* Addison-Wesly, 1986.

[ZISM78] Zisman, M. D. Use of production systems for modelling asynchronous, concurrent processes. In *Pattern-Directed Inference Systems*, Waterman, D.A. and Hayes-Roth, F. (Eds), Academic Press, 1978, pp. 53-69.