TRANSIMS Network Subsystem for IOC-1

K. P. Berkbigler
Computer Research and Applications Group
Los Alamos National Laboratory
and
B. W. Bush
Energy and Environmental Analysis Group
Los Alamos National Laboratory

20 June 1997

Abstract

The TRANSIMS network representation provides access to detailed information about streets, intersections, and signals in a road network. It forms a layer separating the other subsystems from the actual network data tables so that the other subsystems do not need to access the data tables directly or deal with the format and organization of the tables. This subsystem allows the user to construct multiple subnetworks from the network database tables. It includes road network objects such as nodes (intersections), links (road/street segments), lanes, and traffic controls (signs and signals).

I. Introduction 6
II. Design 7
A. Concepts 7
1. Node 7
2. Link 8
3. Lane 8
4. Pocket Lane 8
5. Parking 8
6. Traffic Control 8
7. Lane Connectivity 8
8. Unsignalized Node 9
9. Signalized Node 9
10. Phasing Plan 9
11. Timing Plan 9
12. Study Areas and Buffer Areas 9
B. Classes 9
1. TNetFactory 12
2. TNetNetwork 13
3. TNetSubnetwork 13

4. TNetReader 14
5. TNetNode 14
6. TNetNodeReader 15
7. TNetLink 16
8. TNetLinkReader 18
9. TNetLocation 20
10. TNetLane 20
11. TNetLaneLocation 21
12. TNetLaneConnectivityReader 21
13. TNetAccessory 22
14. TNetAccessoryReader 23
15. TNetPocket 23
16. TNetPocketReader 24
17. TNetParking 24
18. TNetParkingReader 25
19. TNetTrafficControl 25
20. TNetNullControl 27
21. TNetIsolatedControl 27
22. TNetUnsignalizedControl 28
23. TNetUnsignalizedControlReader 28
24. TNetSignalizedControl 29
25. TNetSignalizedControlReader 31
26. TNetTimedControl 32
27. TNetPhase 32
28. TNetPhaseDescription 33
29. TNetPhasingPlan 34
30. TNetPhasingPlanReader 35
31. TNetTimingPlanReader 36
32. TNetSignalCoordinator 37
33. TNetSimulationArea 37
34. TNetSimulationAreaReader 38
35. TNetSimulationAreaLinkReader 38
36. TGeoPoint 39
37. TGeoRectangle 39
38. TGeoFilterFunction 40
39. TGeoFilterNone 40
40. TGeoFilterRectangle 40
41. TNetException 40
III. Implementation 41
A. C++ Libraries 41
B. Sources for Traffic Engineering Information 41
IV. Usage 42
A. Accessing Network Data via C++ 42
B. Network Data Tables 43
1. File Formats 43

2. Example 49
C. Notes 57
1. Row Order of Phasing and Timing Plans 57
2. Boundary Accessories 57
V. Future Work 57
VI. References 57
VII. APPENDIX: Booch Notation Diagrams 58
VIII. APPENDIX: Source Code 60
A. TNetAccessory Class 60
1. Accessory.h 60
2. Accessory.C 61
B. TNetAccessoryReader Class 62
1. AccessoryReader.h 62
2. AccessoryReader.C 63
C. TNetException Class 64
1. Exception.h 64
2. Exception.C 65
D. TNetFactory Class 66
1. Factory.h 66
2. Factory.C 67
E. TGeoFilterFunction Class 68
1. FilterFunction.h 68
F. TGeoFilterNone Class 69
1. FilterNone.h 69
2. FilterNone.C 69
G. TGeoFilterRectangle Class 70
1. FilterRectangle.h 70
2. FilterRectangle.C 70
H. TNetIsolatedControl Class 71
1. IsolatedControl.h 71
2. IsolatedControl.C 72
I. TNetLane Class 73
1. Lane.h 73
2. Lane.C 74
J. TNetLaneConnectivityReader Class 76
1. LaneConnectivityReader.h 76
2. LaneConnectivityReader.C 77
K. TNetLaneLocation Class 78
1. LaneLocation.h 78
2. LaneLocation.C 79
L. TNetLink Class 79
1. Link.h 79
2. Link.C 82
M. TNetLinkReader Class 85
1. LinkReader.h 85

2. LinkReader.C 88
N. TNetLocation Class 92
1. Location.h 92
2. Location.C 93
O. TNetNetwork Class 94
1. Network.h 94
2. Network.C 96
P. TNetNode Class 97
1. Node.h 97
2. Node.C 99
Q. TNetNodeReader Class 101
1. NodeReader.h 101
2. NodeReader.C 102
R. TNetNullControl Class 103
1. NullControl.h 103
2. NullControl.C 104
S. TNetParking Class 105
1. Parking.h 105
2. Parking.C 106
T. TNetParkingReader Class 107
1. ParkingReader.h 107
2. ParkingReader.C 108
U. TNetPhase Class 109
1. Phase.h 109
2. Phase.C 110
V. TNetPhaseDescription Class 112
1. PhaseDescription.h 112
2. PhaseDescription.C 114
W. TNetPhasingPlan Class 116
1. PhasingPlan.h 116
2. PhasingPlan.C 118
X. TNetPhasingPlanReader Class 119
1. PhasingPlanReader.h 119
2. PhasingPlanReader.C 120
Y. TNetPocket Class 122
1. Pocket.h 122
2. Pocket.C 123
Z. TNetPocketReader Class 123
1. PocketReader.h 123
2. PocketReader.C 124
AA. TGeoPoint Class 125
1. Point.h 125
2. Point.C 126
BB. TNetReader Class 127
1. Reader.h 127

2. Reader.C 128
CC. TGeoRectangle Class 129
1. Rectangle.h 129
2. Rectangle.C 130
DD. TNetSignalCoordinator Class 131
1. SignalCoordinator.h 131
2. SignalCoordinator.C 132
EE. TNetSignalizedControl Class 135
1. SignalizedControl.h 135
2. SignalizedControl.C 137
FF. TNetSignalizedControlReader Class 145
1. SignalizedControlReader.h 145
2. SignalizedControlReader.C 146
GG. TNetSimulationArea Class 147
1. SimulationArea.h 147
2. SimulationArea.C 148
HH. TNetSimulationAreaLinkReader Class 149
1. SimulationAreaLinkReader.h 149
2. SimulationAreaLinkReader.C 150
II. TNetSimulationAreaReader Class 152
1. SimulationAreaReader.h 152
2. SimulationAreaReader.C 152
JJ. TNetSubnetwork Class 153
1. Subnetwork.h 153
2. Subnetwork.C 154
KK. TNetTimedControl Class 161
1. TimedControl.h 161
2. TimedControl.C 162
LL. TNetTimingPlanReader Class 164
1. TimingPlanReader.h 164
2. TimingPlanReader.C 165
MM. TNetTrafficControl Class 167
1. TrafficControl.h 167
2. TrafficControl.C 169
NN. TNetUnsignalizedControl Class 173
1. UnsignalizedControl.h 173
2. UnsignalizedControl.C 175
OO. TNetUnsignalizedControlReader Class 176
1. UnsignalizedControlReader.h 176
2. UnsignalizedControlReader.C 177
PP. Constants 178
1. Id.h 178
IX. APPENDIX: Test Program 179
A. Test.C 179
B. TestFilterFunction.C 181

C. TestGeography.C 181
D. TestIsolatedControl.C 182
E. TestLane.C 183
F. TestLink.C 185
G. TestNetwork.C 187
H. TestNode.C 188
I. TestNullControl.C 189
J. TestParking.C 189
K. TestPhase.C 191
L. TestPhaseDescription.C 192
M. TestPhasingPlan.C 193
N. TestPocket.C 194
O. TestReader.C 196
P. TestSignalCoordinator.C 197
Q. TestSignalizedControl.C 198
R. TestSubnetwork.C 200
S. TestTimedControl.C 201
T. TestTrafficControl.C 202
U. TestUnsignalizedControl.C 212
X. APPENDIX: Data Source Creation Utility 213
A. CreateSources.C 213
XI. APPENDIX: Network Dump Utility 214
A. DumpNetwork.C 214
XII. APPENDIX: Network Data Tables 218
A. Empty Network 219
1. EmptyNetwork.Dat 219
B. Sample Networks 221
1. SampleNetwork2.dat 221
2. SampleNetwork2U.dat 225

I. Introduction

The TRANSIMS network representation provides access to detailed information about streets, intersections, and signals in a road network. It forms a layer separating the other subsystems from the actual network data tables so that the other subsystems do not need to access the data tables directly or deal with the format and organization of the tables. Figure 1 shows the position of the network subsystem within the TRANSIMS software architecture. This subsystem depends only upon the database subsystem, and only upon that during network object construction from data tables.

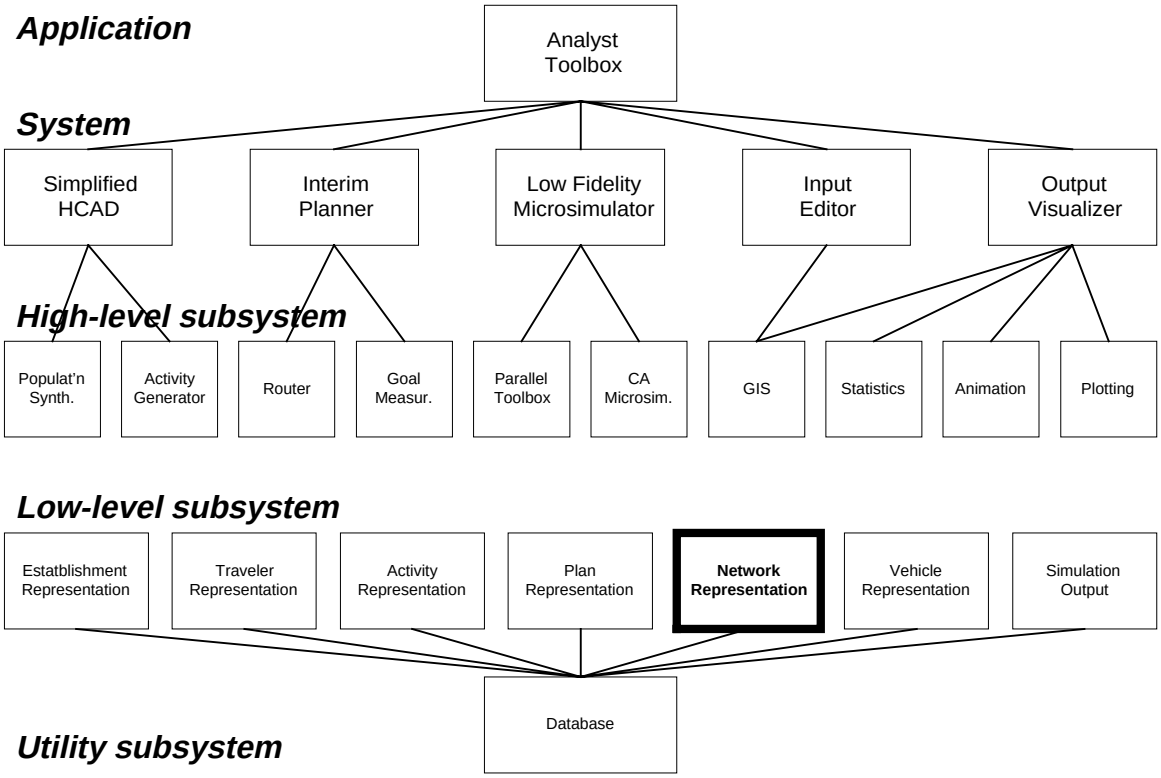

Figure 1. Location of the network subsystem in the TRANSIMS software architecture.

This subsystem allows the user to construct multiple subnetworks from the network database tables. It includes road network objects such as nodes (intersections), links (road/street segments), lanes, and traffic controls (signs and signals). Link attributes for the road network include such characteristics as link type, length, directionality, speed limit, number of lanes, grade, and intersection setbacks. Traffic controls may be either signs (stop and yield) or pre-timed signals.

The body of this document outlines the design, implementation, and usage of the subsystem. The appendices contain the complete C++ source code and sample data.

II. Design

A. Concepts

1. Node

A node is the part of the network corresponding to a *vertex* in graph theory. Nodes typically occur at intersections in the road network. A node must be present where the network branches and where the permanent number of lanes changes. A lane is considered *permanent* if it is not a temporary, pocket lane (see the definition of pocket lane below). A node may be present where neither of the aforementioned occurs, however. Nodes are not required where turn pockets start, as these are not considered

permanent lanes. Each node has a traffic control associated with it (null, unsignalized, pre-timed, actuated, coordinated, etc.).

2. Link

A link is the part of the network corresponding to an *edge* in graph theory. Links represent street and road segments. Each link has a constant number of permanent lanes, but may have a variable number pocket lanes. A link may have lanes in both directions, or the lanes in opposite directions may be on separate links (in which case no passing into oncoming lanes will be possible).

3. Lane

A lane is where traffic flows on a link. The lanes on each side/direction of the link are numbered separately, starting with lane number one as the leftmost lane (relative to the direction of travel). Each successive lane to the right of it is numbered one greater than its predecessor. Pocket lanes (i.e., turn pockets, merges, and pull-outs) are numbered in sequence, even if they do not exist for the full length of the link. A two-way left-turn lane, if present, is considered to be lane number zero.

4. Pocket Lane

A pocket lane is either (a) a right or left turn pocket (a lane that starts after the *from* intersection and ends at the *to* intersection), (b) a right or left pull-out (a lane that starts after the *from* intersection and ends before the *to* intersection), or (c) a right or left merge pocket (a lane that starts at the *from* intersection and ends before the *to* intersection. If a lane starts at the *from* intersection and ends at the *to* intersection, it is considered a permanent lane, not a pocket lane.

5. Parking

Parking areas are located along links and are used as origins and destinations for vehicle trips. Parking may be placed where it actual is physically located in the network, or it may be placed in aggregate *generic* parking areas representing several of the driveways, lots, parking places, etc., on a link. Places where vehicles leave the network are called *boundary* parking areas.

6. Traffic Control

Each node has a traffic control associated with it. The traffic control specifies how lanes are connected across the node and the type of sign or signalized control that determines who has the right-of-way.

7. Lane Connectivity

Lane connectivity specifies how lanes are connected across a node. Lanes are numbered from the median and include turn pockets. Incoming and outgoing links and lanes are defined relative to the node. For each incoming lane on an incoming link, at least one outgoing lane must be specified for each outgoing link that a vehicle on the incoming link can transition to. Multiple outgoing lanes may be defined for an outgoing link, if desired.

8. Unsignalized Node

An unsignalized node represents the type of sign control, if any, that is present at an unsignalized node. Examples are stop and yield signs. Nodes where only the number of permanent lanes is changing are generally considered unsignalized.

9. Signalized Node

A signalized node represents a traffic light. Each signal has a timing plan.

10. Phasing Plan

A phasing plan specifies the turn protection in effect for transitioning from an incoming link to an outgoing link during a particular phase of a specific timing plan.

11. Timing Plan

A timing plan specifies the lengths of the intervals during the specific phases for a traffic light. Many nodes may have the same timing plan. It is possible for each phase to transition to more than one phase if required.

12. Study Areas and Buffer Areas

The microsimulation distinguishes two types of links in its calculations: Study area links are the links of interest for the traffic analyst. The output subsystem, for instance, records events such as when a vehicle leaves or enters the study area. The nature of the microsimulation makes it necessary also to simulate traffic on additional buffer area links. Typically, these link form a *fringe* about two links thick around the study area. A simulation includes buffer links in order to avoid edge effects such as when vehicles enter the study area on its boundary; the *buffer* gives these vehicles time to interact with other traffic and achieve realistic behavior before entering the study area.

B. Classes

The network subsystem has classes for constructing a road network from database tables and for supplying information about the road network to clients. Figure 2 through Figure 6 show the relationships between the classes.

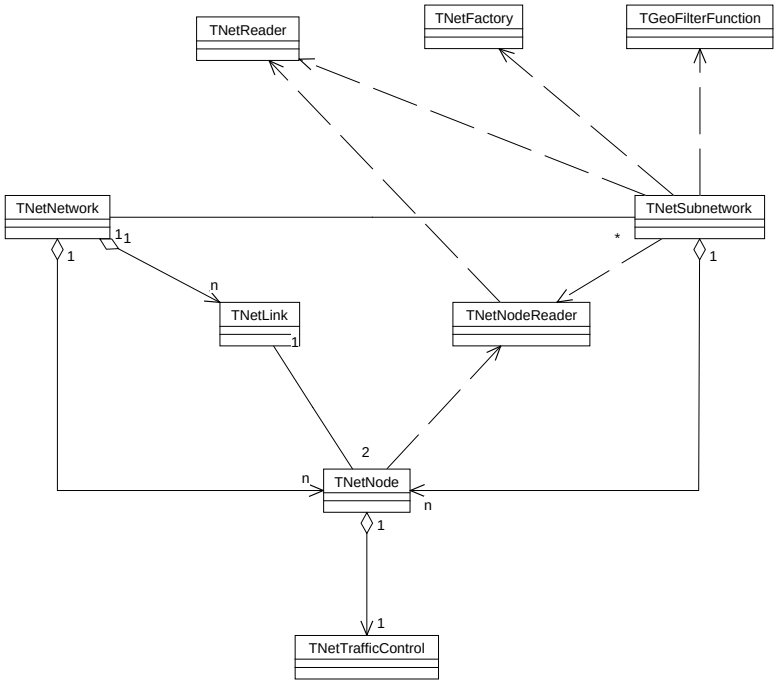

Figure 2. Class diagram for the node-related classes in the TRANSIMS network representation subsystem (unified notation).

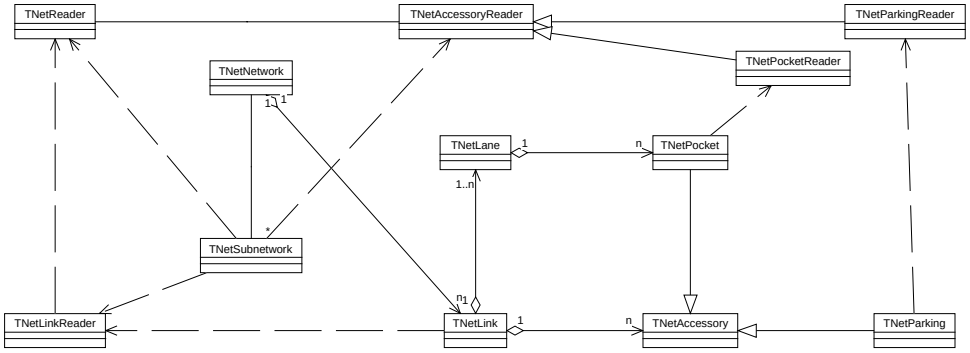

Figure 3. Class diagram for the link-related classes in the TRANSIMS network representation subsystem (unified notation).

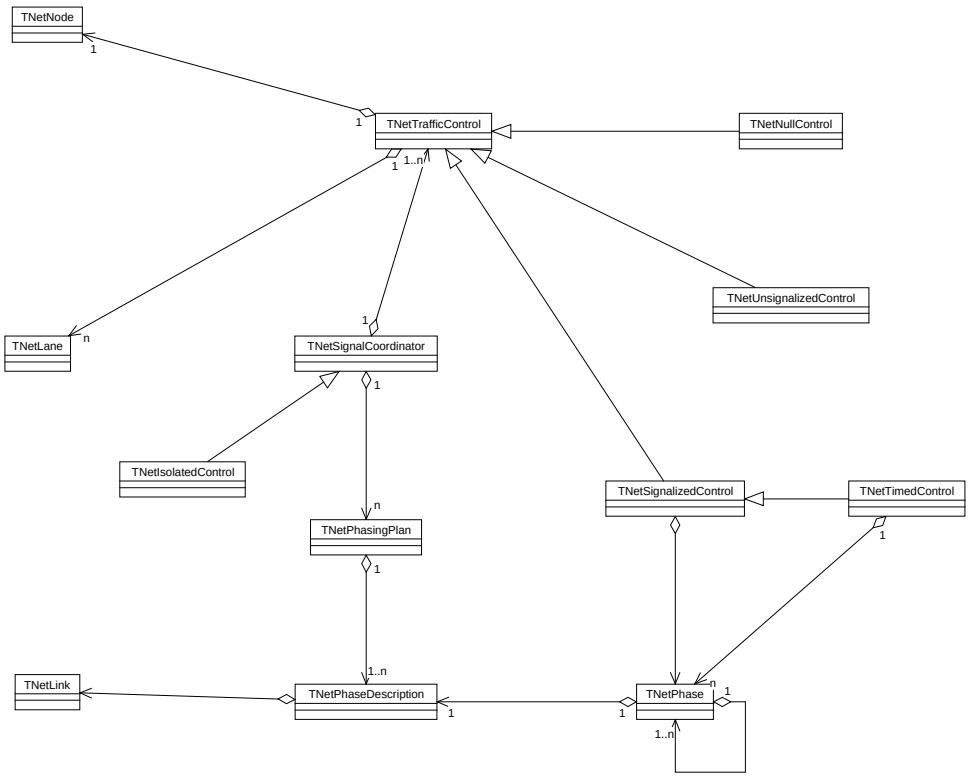

Figure 4. Class diagram for the control-related classes in the TRANSIMS network representation subsystem (unified notation).

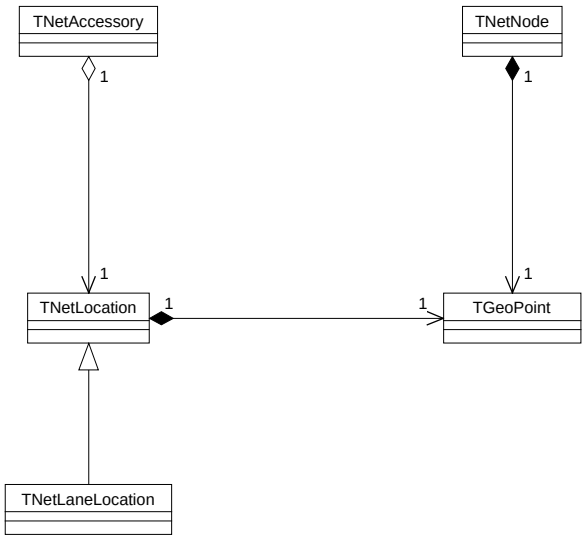

Figure 5. Class diagram for the location-related classes in the TRANSIMS network representation subsystem (unified notation).

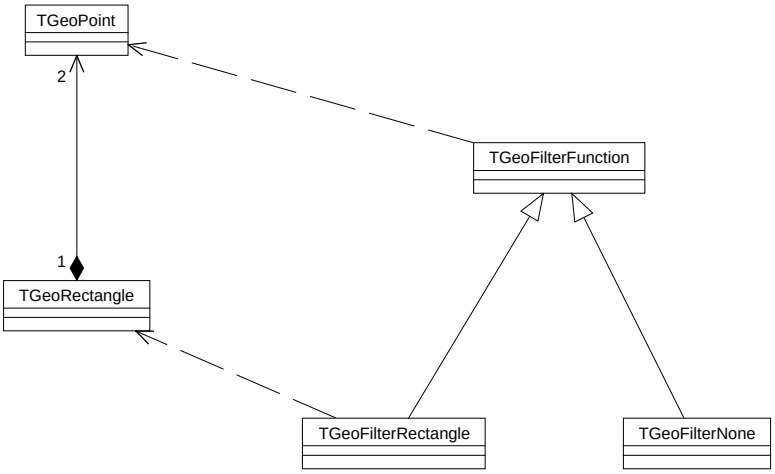

Figure 6. Class diagram for the geography-related classes in the TRANSIMS network representation subsystem (unified notation).

1. TNetFactory

A network factory allocates and constructs new network objects.

`virtual TNetNode* NewNode(TNetNodeReader& reader)`
Return a new node from the specified reader.

`virtual TNetLink* NewLink(TNetLinkReader& reader)`
Return a new link from the specified reader.

`virtual TNetPocket* NewPocket(TNetPocketReader& reader)`
Return a new pocket from the specified reader.

`virtual TNetParking* NewParking(TNetParkingReader& reader)`
Return a new parking place from the specified reader.

```
virtual TNetUnsignalizedControl*
            NewUnsignalizedControl(TNetUnsignalizedControlReader&
            reader, TNetNetwork& network)
```
Return a new unsignalized control from the specified reader and for the specified network.

```
virtual TNetTimedControl*
            NewTimedControl(TNetSignalizedControlReader& reader,
            TNetNetwork& network)
```
Return a new timed control from the specified reader and for the specified network.

`virtual TNetIsolatedControl* NewIsolatedControl(NetNodeId id)`
Return a new isolated control for the specified node.

```
virtual TNetNullControl* NewNullControl(TNetNode& node)
```

Return a new null control for the specified node.

2. TNetNetwork

A network represents all of the network database that has been instantiated. Each network manages/owns the nodes, links and signal coordinators it contains and has a map of parking places.

```
TNetNetwork()
```

Construct a network.

```
NodeMap& GetNodes()
const NodeMap& GetNodes() const
```

Return the set of nodes in the network.

```
LinkMap& GetLinks()
const LinkMap& GetLinks() const
```

Return the set of links in the network.

```
ParkingMap& GetParkings()
const ParkingMap& GetParkings() const
```

Return the set of parking places in the network.

```
SignalCoordinatorMap& GetSignalCoordinators()
const SignalCoordinatorMap& GetSignalCoordinators() const
```

Return the set of signal coordinators in the network.

3. TNetSubnetwork

A subnetwork represents a subset of the instantiated network. Each subnetwork is part of a network and has references to the nodes and links it contains.

```
TNetSubnetwork(TNetReader& reader, TNetNetwork& network,
           TGeoFilterFunction& filter = TGeoFilterNone(),
           TNetFactory& factory = TNetFactory())
```

Construct a subnetwork with geographic filtering using the reader.

```
TNetSubnetwork(TNetReader& reader, TNetNetwork& network,
           NodeIdSet& nodesRequested, LinkIdSet& linksRequested,
           NodeSet& nodesProvided, LinkSet& linksProvided,
           TNetFactory& factory = TNetFactory())
```

Construct a subnetwork of specified nodes and links using the reader.

```
NodeSet& GetNodes()
const NodeSet& GetNodes() const
```

Return the set of nodes in the subnetwork.

```
LinkSet& GetLinks()
const LinkSet& GetLinks() const
```

Return the set of links in the subnetwork.

```
TNetNetwork& GetNetwork()
const TNetNetwork& GetNetwork() const
```

Return the network.

4. TNetReader

A network reader reads a network from the database. Each reader has a node, link, pocket, parking, lane connectivity, unsignalized control, signalized control, phasing plan, and timing plan table.

```
TNetReader(TDbTable nodeTable, TDbTable linkTable, TDbTable
           pocketTable, TDbTable parkingTable, TDbTable
           connTable, TDbTable unsigTable, TDbTable sigTable,
           TDbTable phaseTable, TDbTable timeTable)
```

Construct a reader for the specified tables.

```
TDbTable& GetNodeTable()
```

Return the node table.

```
TDbTable& GetLinkTable()
```

Return the link table.

```
TDbTable& GetPocketTable()
```

Return the pocket table.

```
TDbTable& GetParkingTable()
```

Return the parking table.

```
TDbTable& GetLaneConnectivityTable()
```

Return the lane connectivity table.

```
TDbTable& GetUnsignalizedControlTable()
```

Return the unsignalized control table.

```
TDbTable& GetSignalizedControlTable()
```

Return the signalized control table.

```
TDbTable& GetPhasingPlanTable()
```

Return the phasing plan table.

```
TDbTable& GetTimingPlanTable()
```

Return the timing plan table.

5. TNetNode

A node is the part of the network corresponding to a *vertex* in graph theory. A node must be present where the network branches and where the permanent number of lanes changes. (A node may be present where neither of the aforementioned occurs, however.)

Each node has a unique id, a geographic position, and an associated traffic control, and is connected to links.

`TNetNode(TNetNodeReader& reader)`
Construct a node using the reader.

`TNetNode(NetNodeId id)`
Construct a dummy node with the specified id.

`NetNodeId GetId() const`
Return the id of the node.

`TGeoPoint& GetGeographicPosition()`
`const TGeoPoint& GetGeographicPosition() const`
Return the geographic position of the node.

`LinkRing& GetLinks()`
`const LinkRing& GetLinks() const`
Return the ring of links in order.

`void AddLink(TNetLink* link)`
Add the specified link to the ring of links.

`TNetTrafficControl& GetTrafficControl()`
`const TNetTrafficControl& GetTrafficControl() const`
Return the traffic control for the node.

`void SetTrafficControl (TNetTrafficControl*)`
Define the traffic control for the node.

`bool operator==(const TNetNode& node) const`
Return whether the node has the same id as the given node.

`bool operator!=(const TNetNode& node) const`
Return whether the node has a different id from the given node.

6. TNetNodeReader

A node reader reads node values from the database. Each node reader has a database table accessor and database fields.

`TNetNodeReader(TNetReader& reader)`
Construct a node reader for a given network.

`TNetNodeReader(const TNetNodeReader& reader)`
Construct a copy of the given node reader.

`TNetNodeReader& operator=(const TNetNodeReader& reader)`
Make the reader a copy of the given node reader.

`void Reset()`
Reset the iteration over the table.

`void GetNextNode()`
Get the next node in the table.

`bool MoreNodes() const`
Return whether there are any more nodes in the table.

`NetNodeId GetId() const`
Return the id for the current node.

`TGeoPoint GetGeographicPosition() const`
Return the geographic position for the current node.

7. TNetLink

A link is the part of the network corresponding to an *edge* in graph theory. Each link has a constant number of permanent lanes, but may have turn pocket lanes also. A link may have lanes in both directions, or the lanes in opposite directions may be on separate links (in which case no passing into oncoming lanes will be possible). Each link has a unique id, nodes at its two ends, zero or more accessories, zero or more lanes on each side, zero or more permanent lanes on each side, a length, setback distances from the intersection, through links at either end, speed limits in either direction, and an angle (in radians) from its endpoints.

`enum EFunctionalClass {kOther}`
There are several functions for a link.

`TNetLink(TNetLinkReader& reader)`
Construct a link using the reader.

`TNetLink(NetLinkId id)`
Construct a dummy link with the specified id.

`NetLinkId GetId() const`
Return the id of the link.

```
void GetNodes(NetNodeId& nodeA, NetNodeId& nodeB) const
void GetNodes(TNetNode*& nodeA, TNetNode*& nodeB) const
void GetNodes(const TNetNode*& nodeA, const TNetNode*& nodeB)
          const
```
Return the nodes at the ends of the link.

`void SetNodes(TNetNode* nodeA, TNetNode* nodeB)`
Set the nodes at the ends of the link.

`AccessoryCollection& GetAccessories()`
`const AccessoryCollection& GetAccessories() const`

Return the accessories on the link.

`LaneCollection& GetLanesFrom(const TNetNode& node, bool pockets = FALSE)`
`const LaneCollection& GetLanesFrom(const TNetNode& node, bool pockets = FALSE) const`

Return the lanes going away from the specified node. The lanes are ordered (but not necessarily numbered) from the divider outward. Pockets are included optionally. A `TNetNotFound` exception is thrown if the node is not at one of the link's ends.

`LaneCollection& GetLanesTowards(const TNetNode& node, bool pockets = FALSE)`
`const LaneCollection& GetLanesTowards(const TNetNode& node, bool pockets = FALSE) const`

Return the lanes going toward the specified node. The lanes are ordered (but not necessarily numbered) from the divider outward. Pockets are included optionally. A `TNetNotFound` exception is thrown if the node is not at one of the link's ends.

`REAL GetLength(bool setback = FALSE) const`

Return the length of the link (using the default units). The length is measured from one node to the other, unless setback distance subtraction is requested.

`REAL GetSetback(const TNetNode& node) const`

Return the setback distance (using the default units) at the specified node. A `TNetNotFound` exception is thrown if the node is not at one of the link's ends.

`NetLinkId GetThroughLink(const TNetNode& node) const`

Return the through link at the given node on the current link. A `TNetNotFound` exception is thrown if the node is not at one of the link's ends.

`REAL GetSpeedLimitTowards(const TNetNode& node) const`

Return the speed limit of the lanes heading toward the given node on the current link. A `TNetNotFound` exception is thrown if the node is not at one of the link's ends.

`REAL GetAngle(const TNetNode& node) const`

Return the angle (in radians) of the link from the specified node. A `TNetNotFound` exception is thrown if the node is not at one of the link's ends.

`TNetNode& GetNodeBetween(const TNetLink& link) const`

Return the node between the current link and the given link. A `TNetNotFound` exception is thrown if the links are not adjacent.

`bool operator==(const TNetLink& link) const`

Return whether the link has the same id as the given link.

`bool operator!=(const TNetLink& link) const`

Return whether the link has a different id from the given link

8. TNetLinkReader

A link reader reads link values from the database. Each link reader has a database table accessor and database fields.

`TNetLinkReader(TNetReader& reader)`

Construct a link reader for a given network.

`TNetLinkReader(const TNetLinkReader& reader)`

Construct a copy of the given link reader.

`TNetLinkReader& operator=(const TNetLinkReader& reader)`

Make the reader a copy of the given link reader.

`void Reset()`

Reset the iteration over the table.

`void GetNextLink()`

Get the next link in the table.

`bool MoreLinks() const`

Return whether there are any more links in the table.

`NetLinkId GetId() const`

Return the id for the current link.

`void GetNodeIds(NetNodeId& nodeA, NetNodeId& nodeB) const`

Return the node ids at the ends of the current link.

`BYTE GetPermanentLaneCountTowards(NetNodeId id) const`

Return the number of permanent lanes heading toward the given node on the current link. A `TNetNotFound` exception is thrown if the node is not at one of the link's ends.

`BYTE GetLeftPocketLaneCountTowards(NetNodeId id) const`

Return the number of left pocket lanes heading toward the given node on the current link. A `TNetNotFound` exception is thrown if the node is not at one of the link's ends.

`BYTE GetRightPocketLaneCountTowards(NetNodeId id) const`

Return the number of right pocket lanes heading toward the given node on the current link. A `TNetNotFound` exception is thrown if the node is not at one of the link's ends.

`bool HasTwoWayLeftTurnLane() const`

Return whether there is a two-way left turn lane on the current link.

`REAL GetLength() const`

Return the length of the current link.

`REAL GetGradeTowards(NetNodeId id) const`

Return the percent grade of the lanes heading toward the given node on the current link. A `TNetNotFound` exception is thrown if the node is not at one of the link's ends.

`REAL GetSetbackDistance(NetNodeId id) const`

Return the setback distances at the given node on the current link. A `TNetNotFound` exception is thrown if the node is not at one of the link's ends.

`NetLinkId GetThroughLink(NetNodeId id) const`

Return the through link at the given node on the current link. A `TNetNotFound` exception is thrown if the node is not at one of the link's ends.

`REAL GetCapacityTowards(NetNodeId id) const`

Return the capacity in vehicles-per-hour of the lanes heading toward the given node on the current link. A `TNetNotFound` exception is thrown if the node is not at one of the link's ends.

`REAL GetSpeedLimitTowards(NetNodeId id) const`

Return the speed limit of the lanes heading toward the given node on the current link. A `TNetNotFound` exception is thrown if the node is not at one of the link's ends.

`REAL GetFreeFlowSpeedTowards(NetNodeId id) const`

Return the free-flow speed of the lanes heading toward the given node on the current link. A `TNetNotFound` exception is thrown if the node is not at one of the link's ends.

`REAL GetCrawlSpeedTowards(NetNodeId id) const`

Return the crawl speed of the lanes heading toward the given node on the current link. A `TNetNotFound` exception is thrown if the node is not at one of the link's ends.

`TNetLink::EFunctionalClass GetFunctionalClass() const`

Return the functional class for the link.

`UINT GetCostTowards(NetNodeId id) const`

Return the cost (in arbitrary units) for traveling on the link toward the given node. A `TNetNotFound` exception is thrown if the node is not at one of the link's ends.

9. TNetLocation

A location is a point along a link. Each location is on a link, is measured from a node, and is specified as an offset relative to the start of its link.

`TNetLocation(TNetLink& link, TNetNode& node, REAL offset)`

Construct a location along the given node and link, with the given offset from the node. The exception `TNetNotFound` is thrown if the node is not at one of the link's ends.

`TNetLocation(const TNetLocation& location)`

Construct a copy of the given location.

`TNetLocation& operator=(const TNetLocation& location)`

Make the location a copy of the given location.

`TNetLink& GetLink()`
`const TNetLink& GetLink() const`

Return the link on which the location lies.

`REAL GetOffsetFrom(const TNetNode& node) const`

Return the distance from the given endpoint of the link. The exception `TNetNotFound` is thrown if the node is not at one of the link's ends.

`TGeoPoint GetGeographicPosition() const`

Return the geographic position of the location.

`virtual bool IsOnSpecificLane() const`

Return whether the location is lane-specific.

10. TNetLane

A lane is where traffic flows on a link. Each lane belongs on a link and has a starting node, a number, and zero or more pockets.

`TNetLane(TNetLink& link, NetNodeId node, NetLaneNumber number)`

Construct a lane on the specified link, starting at the specified node, and identified by the specified number.

`TNetLink& GetLink()`
`const TNetLink& GetLink() const`

Return the link on which the lane lies.

`TNetNode& GetStartNode()`
`const TNetNode& GetStartNode() const`

Return the starting node for the lane.

```
TNetNode& GetEndNode()
const TNetNode& GetEndNode() const
```
Return the end node for the lane.

```
PocketCollection& GetPockets()
const PocketCollection& GetPockets() const
```
Return the pockets on the lane.

```
const TNetLane* GetLeftAdjacentLane() const
```
Return the lane to the left, if any.

```
const TNetLane* GetRightAdjacentLane() const
```
Return the lane to the right, if any.

```
NetLaneNumber GetNumber() const
```
Return the number of the lane.

11. TNetLaneLocation

A location is a lane-specific point along a link. Each lane location is on a lane.

```
TNetLaneLocation(TNetLane& lane, REAL offset)
```
Construct a location along the lane, with the given offset from its start.

```
TNetLaneLocation(const TNetLaneLocation& location)
```
Construct a copy of the given location.

```
TNetLaneLocation& operator=(const TNetLaneLocation& location)
```
Make the location a copy of the given location.

```
TNetLane& GetLane()
const TNetLane& GetLane() const
```
Return the lane for the location.

```
bool IsOnSpecificLane() const
```
Return whether the location is lane-specific.

12. TNetLaneConnectivityReader

This reader reads lane connectivity values from the database. Each lane connectivity has a database table accessor and database fields.

```
TNetLaneConnectivityReader(TNetReader& reader)
```
Construct a lane connectivity reader for a given network.

```
TNetLaneConnectivityReader(const TNetLaneConnectivityReader&
            reader)
```
Construct a copy of the given lane connectivity reader.

```
TNetLaneConnectivityReader& operator=(const
            TNetLaneConnectivityReader& reader)
```
Make the reader a copy of the given lane connectivity reader.

```
void Reset()
```
Reset the iteration over the table.

```
void GetNextNode()
```
Get the next node in the table.

```
bool MoreNodes()
```
Return whether there are any more nodes in the table.

```
NetNodeId GetNode() const
```
Return the id for the current node.

```
NetLinkId GetInlink() const
```
Return the id for incoming link.

```
NetLaneNumber GetInlane() const
```
Return the id for incoming lane.

```
NetLinkId GetOutlink() const
```
Return the id for outgoing link.

```
NetLaneNumber GetOutlane() const
```
Return the id for outgoing lane.

13. TNetAccessory

An accessory is where something happens along a link. Each accessory has a unique id, a type, and a location.

```
enum EType {kPocket, kParking}
```
There are several types of accessories.

```
TNetAccessory(NetAccessoryId id, EType type)
```
Construct an accessory with the specified id and type.

```
NetAccessoryId GetId() const
```
Return the id of the accessory.

```
EType GetType() const
```
Return the type of the accessory.

```
TNetLocation& GetLocation()
const TNetLocation& GetLocation() const
```
Return the location of the accessory.

`void SetLocation(const TNetLocation& location)`
Set the location of the accessory.

14. TNetAccessoryReader

An accessory reader reads accessory values from the database. Each accessory has a database table accessor and database fields.

`TNetAccessoryReader(TDbTable& table)`
Construct an accessory reader for a given network.

`TNetAccessoryReader(const TNetParkingReader& reader)`
Construct a copy of the given accessory reader.

`TNetAccessoryReader& operator=(const TNetParkingReader& reader)`
Make the accessory reader a copy of the given accessory reader.

`void Reset()`
Reset the iteration over the table.

`void GetNextAccessory()`
Get the next accessory in the table.

`bool MoreAccessories() const`
Return whether there are any more accessories in the table.

`NetAccessoryId GetId() const`
Return the id for the current accessory.

```
void GetLocation(NetLinkId& link, NetNodeId& node, REAL& offset)
      const
```
Return the location of the current accessory.

15. TNetPocket

A pocket is a length of lane intended for special uses such as buses and pulling out, vehicles waiting for turns, vehicles accelerating in order to merge, etc. Each pocket has a style and a length.

`enum EStyle {kTurn, kPullout, kMerge}`
There are several styles of pockets.

`TNetPocket(TNetPocketReader& reader)`
Construct the pocket accessory from the specified reader.

`EStyle GetStyle() const`
Return the style of the pocket.

`REAL GetLength() const`
Return the length of the pocket.

`TNetLane& GetLane()`
`const TNetLane& GetLane() const`
Return the lane for the pocket.

16. TNetPocketReader

A pocket reader reads pocket values from the database. Each pocket has database table accessor and database fields.

`TNetPocketReader(TNetReader& reader)`
Construct a pocket reader for a given network.

`TNetPocketReader(const TNetPocketReader& reader)`
Construct a copy of the given pocket reader.

`TNetPocketReader& operator=(const TNetPocketReader& reader)`
Make the pocket reader a copy of the given pocket reader.

`NetLaneNumber GetLaneNumber() const`
Return the lane number of the current pocket.

`TNetPocket::EStyle GetStyle() const`
Return the style of the current pocket.

`REAL GetLength() const`
Return the length of the current pocket.

17. TNetParking

A parking place is a source or sink of vehicles along a link. A parking place has a style and a capacity and may be generic.

```
enum EStyle {kParallelOnStreet, kHeadInOnStreet, kDriveway, kLot,
            kBoundary}
```
There are several styles of parking.

`TNetParking(TNetParkingReader& reader)`
Construct the parking from the specified reader.

```
TNetParking (NetAccessoryId, TNetLink&, TNetNode&, REAL offset,
            EStyle style, UINT capacity, bool generic)
```
Construct a parking place with specified values.

`EStyle GetStyle() const`
Return the style of the parking.

`UINT GetCapacity() const`
Return the capacity of the parking.

`bool IsGeneric() const`
Return whether the parking is generic.

`static NetAccessoryId GenerateId(const TNetNetwork&, NetLinkId)`
Generate a unique parking place id.

18. TNetParkingReader

A parking reader reads parking values from the database. Each parking reader has a database table accessor and database fields.

`TNetParkingReader(TNetReader& reader)`
Construct a parking reader for a given network.

`TNetParkingReader(const TNetParkingReader& reader)`
Construct a copy of the given parking reader.

`TNetParkingReader& operator=(const TNetParkingReader& reader)`
Make the parking reader a copy of the given parking reader.

`TNetParking::EStyle GetStyle() const`
Return the style of the current parking.

`UINT GetCapacity() const`
Return the capacity of the current parking.

`bool IsGeneric() const`
Return whether the current parking is generic.

19. TNetTrafficControl

A traffic control is associated with each node. The traffic control specifies how lanes are connected across the node and the type of sign or signalized control that determines who has the right-of-way. A traffic control knows the node with which it is associated, and it contains a table that describes how lanes are connected across the node.

```
enum ETrafficControl {kNone, kStop, kYield, kWait, kCaution,
            kPermitted, kProtected}
```
Vehicle signs and signals indicate right of way for movements.

```
TNetTrafficControl()
TNetTrafficControl(TNetNode& node)
TNetTrafficControl(const TNetTrafficControl& control)
```
Construct a traffic control .

```
virtual void AllowedMovements (LaneCollection& lanes, const
            TNetLink& fromlink, const TNetLane& fromlane, const
            TNetLink& tolink)
```

Return the lanes on next link to which transition from specified lane on this link can be made. Return an empty collection if transition is not possible.

```
virtual void AllowedMovements (LaneCollection& lanes, const
            TNetLink& fromlink, const TNetLink& tolink, const
            TNetLane& tolane)
```

Return the lanes on this link from which transition to specified lane on next link can be made. Return an empty collection if transition is not possible.

```
virtual void AllowedMovements (LaneCollection& lanes, const
            TNetLink& fromlink, const TNetLink& tolink, bool phase
            = FALSE)
```

Return the lanes ordered from median that may be used to transition from current link to next link.

```
virtual void InterferingLanes (LaneCollection& lanes, const
            TNetLane& fromlane, const TNetLane& tolane, bool phase
            = FALSE)
```

Return the lanes that must be examined for interference when transitioning from current lane to next lane.

```
virtual ETrafficControl GetVehicleControl(const TNetLane& lane)
            const
```

Return the vehicle control signal for the specified lane.

```
TNetNode& GetNode()
const TNetNode& GetNode() const
```

Return the associated node.

```
void SetConnectivity(TNetLane& inlane, TNetLane& outlane)
```

Define the lane connectivity for the specified lane.

```
void SetConnectivity (TNetLaneConnectivityReader& reader,
            TNetNetwork& network)
```

Define the lane connectivity using the reader.

```
TNetTrafficControl::ConnectedCollection& GetConnectivity(const
            TNetLane& lane)
const TNetTrafficControl::ConnectedCollection&
            GetConnectivity(const TNetLane& lane) const
```

Return the lane connectivity for the specified lane.

```
TNetTrafficControl::ConnectivityMap& GetConnectivity()
const TNetTrafficControl::ConnectivityMap& GetConnectivity()
            const
```

Return the lane connectivity map.

20. TNetNullControl

Null controls are used by nodes just outside the network boundary that have a `TNetBoundary` accessory on an attached link.

```
TNetNullControl()
TNetNullControl(TNetNode& node)
TNetNullControl(const TNetNullControl& control)
```

Construct a null traffic control.

```
TNetNullControl& operator=(const TNetNullControl& control)
```

Assign a null traffic control.

```
bool operator==(const TNetNullControl& control) const
bool operator!=(const TNetNullControl& control) const
```

Return whether two null controls are the same.

```
void AllowedMovements(LaneCollection& lanes, const TNetLink&
           fromlink, const TNetLane& fromlane, const TNetLink&
           tolink)
void AllowedMovements(LaneCollection& lanes, const TNetLink&
           fromlink, const TNetLink& tolink, const TNetLane&
           tolane)
void AllowedMovements(LaneCollection& lanes, const TNetLink&
           fromlink, const TNetLink& tolink, bool phase = FALSE)
void InterferingLanes(LaneCollection& lanes, const TNetLane&
           fromlane, const TNetLane& tolane, bool phase = FALSE)
TNetTrafficControl::ETrafficControl GetVehicleControl(const
           TNetLane& lane) const
```

Provide definitions for pure virtual functions to throw exceptions.

21. TNetIsolatedControl

An isolated signal requires no coordination. It defers to the signalized control for operations.

```
TNetIsolatedControl()
TNetIsolatedControl(NetCoordinatorId id )
TNetIsolatedControl(const TNetIsolatedControl& control)
```

Construct an isolated control.

```
TNetIsolatedControl& operator=(const TNetIsolatedControl&
           control)
```

Assign an isolated traffic control.

```
bool operator==(const TNetIsolatedControl& control) const
bool operator!=(const TNetIsolatedControl& control) const
```

Return whether two isolated controls are the same.

```
void UpdateSignalizedControl(REAL sim_time)
```

Isolated control requires no coordination.

22. TNetUnsignalizedControl

An unsignalized control specifies the sign control at a node. Some type of sign control is associated with each link attached to an unsignalized intersection. Example values are stop, yield, and no control on this link.

```
TNetUnsignalizedControl()
TNetUnsignalizedControl(TNetNode& node)
TNetUnsignalizedControl(const TNetUnsignalizedControl& control)
```

Construct an unsignalized traffic control.

```
TNetUnsignalizedControl(TNetUnsignalizedControlReader& reader,
        TNetNetwork& network)
```

Construct an unsignalized traffic control using the reader.

```
TNetUnsignalizedControl& operator=(const TNetUnsignalizedControl&
        control)
```

Assign an unsignalized traffic control.

```
bool operator==(const TNetUnsignalizedControl& control) const
bool operator!=(const TNetUnsignalizedControl& control) const
```

Return whether two unsignalized controls are the same.

```
TNetTrafficControl::ETrafficControl GetVehicleControl(const
        TNetLane& lane) const
```

Return the vehicle control signal for the specified lane.

```
void SetVehicleControl(TNetLink& link,
        TNetTrafficControl::ETrafficControl)
```

Define the vehicle control sign.

```
void SetVehicleControl(TNetUnsignalizedControlReader& reader,
        TNetNetwork& network)
```

Define the vehicle control sign using the reader.

23. TNetUnsignalizedControlReader

This reader reads unsignalized control values from the database. Each unsignalized control reader has a database table accessor and database fields.

```
TNetUnsignalizedControlReader(TNetReader& reader)
```

Construct an unsignalized control reader for a given network.

```
TNetUnsignalizedControlReader(const
        TNetUnsignalizedControlReader& reader)
```

Construct a copy of the given unsignalized control reader.

```
TNetUnsignalizedControlReader& operator=(const
        TNetUnsignalizedControlReader& reader)
```

Make the reader a copy of the given unsignalized control reader.

```
void Reset()
```
Reset the iteration over the table.

```
void GetNextNode()
```
Get the next node in the table.

```
bool MoreNodes()
```
Return whether there are any more nodes in the table.

```
NetNodeId GetNode() const
```
Return the id for the current node.

```
NetLinkId GetInlink() const
```
Return the id for incoming link.

```
string GetSign() const
```
Return the sign control indication on incoming link.

24. TNetSignalizedControl

A signalized control specifies the signal phases and phasing plan at a node. A signalized control has a sequence of phases, has a phasing plan, a phasing plan offset, a coordinator, and a current phase.

```
TNetSignalizedControl()
TNetSignalizedControl(TNetNode& node)
TNetSignalizedControl(const TNetSignalizedControl& control)
```
Construct a signalized traffic control.

```
TNetSignalizedControl(TNetSignalizedControlReader& reader,
          TNetNetwork& network)
```
Construct a signalized traffic control using the reader.

```
virtual void AllowedMovements (LaneCollection& lanes, const
          TNetLink& fromlink, const TNetLane& fromlane, const
          TNetLink& tolink)
```
Return the lanes on next link to which transition from specified lane on this link can be made. Return an empty collection if transition is not possible.

```
virtual void AllowedMovements (LaneCollection& lanes, const
          TNetLink& fromlink, const TNetLink& tolink, const
          TNetLane& tolane)
```
Return the lanes on this link from which transition to specified lane on next link can be made. Return an empty collection if transition is not possible.

```
virtual void AllowedMovements (LaneCollection& lanes, const
            TNetLink& fromlink, const TNetLink& tolink, bool phase
            = FALSE)
```

Return the lanes ordered from median that may be used to transition from current link to next link.

```
virtual void InterferingLanes (LaneCollection& lanes, const
            TNetLane& fromlane, const TNetLane& tolane, bool phase
            = FALSE)
```

Return the lanes that must be examined for interference when transitioning from current lane to next lane.

```
virtual TNetTrafficControl::ETrafficControl
            GetVehicleControl(const TNetLane&) const
```

Return the vehicle control signal for the specified lane.

```
virtual void UpdateSignalizedControl (REAL sim_time)
```

Update the traffic control state based on current simulation time.

```
virtual TNetPhasingPlan& GetPhasingPlan()
virtual const TNetPhasingPlan& GetPhasingPlan() const
```

Return the current phasing plan.

```
virtual REAL GetPhasingPlanOffset() const
```

Return the current phasing plan offset.

```
virtual void SetPhasingPlan(TNetPhasingPlan& plan)
```

Define the phasing plan.

```
virtual void SetPhasingPlanOffset(REAL offset)
```

Define the phasing plan offset.

```
virtual TNetSignalCoordinator& GetCoordinator()
virtual const TNetSignalCoordinator& GetCoordinator() const
```

Return the signalized control coordinator for this signal.

```
virtual void SetCoordinator(TNetSignalCoordinator& coordinator)
```

Define the signalized control coordinator for this signal.

```
virtual TNetPhase& CreatePhase(TNetPhaseDescription& description)
```

Create a phase.

```
virtual PhaseCollection& GetPhases()
virtual const PhaseCollection& GetPhases() const
```

Return the phases for this signal.

```
virtual TNetPhase& GetPhase()
virtual const TNetPhase& GetPhase() const
```

Return the current phase.

`virtual void SetPhase(TNetPhase& phase)`
Update the current phase.

`virtual void InitPhase(TNetPhase& phase)`
Initialize the signal to specified phase.

25. TNetSignalizedControlReader

This reader reads signalized control values from the database. Each signalized control reader has a database table accessor and database fields.

```
TNetSignalizedControlReader(TNetReader& reader)
```
Construct a signalized control reader for a given network.

```
TNetSignalizedControlReader(const TNetSignalizedControlReader&
          reader)
```
Construct a copy of the given signalized control reader.

```
TNetSignalizedControlReader& operator=(const
          TNetSignalizedControlReader& reader)
```
Make the reader a copy of the given signalized control reader.

`void Reset()`
Reset the iteration over the table.

`void GetNextNode()`
Get the next node in the table.

`bool MoreNodes()`
Return whether there are any more nodes in the table.

`NetNodeId GetNode() const`
Return the id for the current node.

`string GetType() const`
Return the type of the signal.

`NetPlanId GetPlan() const`
Return the timing plan id.

`REAL GetOffset() const`
Return the offset for coordinated signals.

`string GetStarttime() const`
Return the starting time for the plan.

26. TNetTimedControl

A timed control specifies the performance of a pre-timed signal. Each timed control has a cycle length and times at which each phase ends.

```
TNetTimedControl()
TNetTimedControl(TNetNode& node)
TNetTimedControl(const TNetTimedControl& control)
```

Construct a timed signalized traffic control.

```
TNetTimedControl(TNetSignalizedControlReader& reader,
           TNetNetwork& network)
```

Construct a timed signalized traffic control using the reader.

```
TNetTimedControl& operator=(const TNetTimedControl& control)
```

Assign a timed control.

```
bool operator==(const TNetTimedControl& control) const
bool operator!=(const TNetTimedControl& control) const
```

Return whether two timed controls are the same.

```
virtual void UpdateSignalizedControl(REAL sim_time)
```

Update the traffic control state according to algorithm for fixed time controllers.

```
virtual TNetPhase& CreatePhase(TNetPhaseDescription& description)
```

Create a phase.

27. TNetPhase

A phase is a portion of a traffic signal cycle when the allowed movements are unchanged. A phase is composed of intervals where the traffic displays are constant. Each phase has a sequence of intervals, an associated phase description, and one or more next phases to which it can transition.

```
enum EInterval {kGreen, kYellow, kRed}
```

Each phase has three intervals.

```
TNetPhase()
TNetPhase(TNetPhaseDescription& description)
TNetPhase(const TNetPhase& phase)
```

Construct a phase.

```
TNetPhase& operator=(const TNetPhase& phase)
```

Assign a phase.

```
bool operator==(const TNetPhase&) const
bool operator!=(const TNetPhase&) const
```

Return whether two phases are the same.

```
TNetPhaseDescription& GetPhaseDescription()
const TNetPhaseDescription& GetPhaseDescription() const
```

Return phase description associated with this phase.

```
TNetPhase::EInterval GetInterval() const
```

Return current interval.

```
void SetInterval(EInterval interval)
```

Update signal interval.

```
PhaseCollection& GetNextPhases()
const PhaseCollection& GetNextPhases() const
```

Return phases to which this phase can transition.

```
void SetNextPhase(TNetPhase& phase)
```

Define the next phase.

28. TNetPhaseDescription

A phase description specifies the interval lengths and allowed movements and associated turn protections during a phase. Each phase description has a phase number, a minimum green interval length (or green interval length for fixed time), a maximum green interval length (undefined for fixed time), a green interval extension increment (which equals zero for fixed time), a yellow interval length, a red clearance interval length, and movements allowed during the phase.

```
enum EProtection {kPermitted, kProtected}
```

Turn protections are a subset of ETrafficControl

```
TNetPhaseDescription(NetPhaseNumber phase)
TNetPhaseDescription(const TNetPhaseDescription& description)
```

Construct a phasing plan description.

```
TNetPhaseDescription& operator=(const TNetPhaseDescription&
            description)
```

Assign a phasing plan description.

```
bool operator==(const TNetPhaseDescription& description) const
bool operator!=(const TNetPhaseDescription& description) const
```

Return whether two phase descriptions are the same.

```
NetPhaseNumber GetPhaseNumber() const
```

Return the phase number.

```
REAL GetGreenLength() const
```

Return the green interval length.

```
REAL GetMinGreenLength() const
```

Return the minimum green interval length.

```
REAL GetMaxGreenLength() const
```
Return the maximum green interval length.

```
REAL GetExtGreenLength() const
```
Return the green extension interval length.

```
REAL GetYellowLength() const
```
Return the yellow interval length.

```
REAL GetRedLength() const
```
Return the red interval length.

```
LinkMovementMap& GetLinkMovements()
const LinkMovementMap& GetLinkMovements() const
```
Return all link movements.

```
LinkProtectionMap& GetLinkMovements(const TNetLink& link)
const LinkProtectionMap& GetLinkMovements(const TNetLink& link)
          const
```
Set the interval lengths.

```
void SetLengths(REAL min, REAL max, REAL ext, REAL yellow, REAL
          red)
```
Return the link movements for the specified link.

```
void SetLinkMovements(TNetLink& inlink, TNetLink& outlink,
          TNetPhaseDescription::EProtection protection)
```
Set link movements

29. TNetPhasingPlan

A phasing plan is composed of a series of phase descriptions. Each phasing plan has an id and a sequence of interval length descriptions, one for each phase.

```
TNetPhasingPlan()
TNetPhasingPlan(NetPlanId id)
TNetPhasingPlan(const TNetPhasingPlan& plan)
```
Construct a phasing plan.

```
TNetPhasingPlan& operator=(const TNetPhasingPlan& plan)
```
Assign a phasing plan.

```
bool operator==(const TNetPhasingPlan& plan) const
bool operator!=(const TNetPhasingPlan& plan) const
```
Return whether two phasing plans are the same.

```
PhaseDescriptionCollection& GetPhaseDescriptions()
const PhaseDescriptionCollection& GetPhaseDescriptions() const
```
Create a phase description.

```
TNetPhaseDescription& CreatePhaseDescription(NetPhaseNumber
            phase)
```
Return this phasing plan.

```
NetPlanId GetId() const
```
Return the plan id.

30. TNetPhasingPlanReader

This reader reads phasing plan values from the database. Each phasing plan has a database table accessor and database fields.

```
TNetPhasingPlanReader(TNetReader& reader)
```
Construct a phasing plan reader for a given network.

```
TNetPhasingPlanReader(const TNetPhasingPlanReader& reader)
```
Construct a copy of the given phasing plan reader.

```
TNetPhasingPlanReader& operator=(const TNetPhasingPlanReader&
            reader)
```
Make the reader a copy of the given phasing plan reader.

```
void Reset()
```
Reset the iteration over the table.

```
void GetNextNode()
```
Get the next node in the table.

```
bool MoreNodes()
```
Return whether there are any more nodes in the table.

```
NetNodeId GetNode() const
```
Return the id for the current node.

```
NetPlanId GetPlan() const
```
Return the timing plan id.

```
NetPhaseNumber GetPhase() const
```
Return the phase number.

```
NetLinkId GetInlink() const
```
Return the incoming link id.

```
NetLinkId GetOutlink() const
```
Return the outgoing link id.

```
string GetProtection() const
```
Return the turn protection indicator.

31. TNetTimingPlanReader

This reader reads timing plan values from the database. Each timing plan reader has a database table accessor and database fields.

`TNetTimingPlanReader(TNetReader& reader)`
Construct a timing plan reader for a given network.

`TNetTimingPlanReader(const TNetTimingPlanReader& reader)`
Construct a copy of the given timing plan reader.

```
TNetTimingPlanReader& operator=(const TNetTimingPlanReader&
            reader)
```
Make the reader a copy of the given timing plan reader.

`void Reset()`
Reset the iteration over the table.

`void GetNextPlan()`
Get the next plan in the table.

`bool MorePlans()`
Return whether there are any more plans in the table.

`NetPlanId GetPlan() const`
Return the timing plan id.

`NetPhaseNumber GetPhase() const`
Return the phase number.

`string GetNextPhases() const`
Return the phase numbers of the next phases.

`REAL GetGreenMin() const`
Return the minimum length of green interval.

`REAL GetGreenMax() const`
Return the maximum length of green interval.

`REAL GetGreenExt() const`
Return the length of green interval extension.

`REAL GetYellow() const`
Return the length of yellow interval.

`REAL GetRedClear() const`
Return the length of red clearance interval.

32. TNetSignalCoordinator

A signal coordinator coordinates the operation of several traffic signals. Each coordinator has a unique id, one or more signalized controllers, and a group of phasing plans that it can tell its controllers to use.

```
TNetSignalCoordinator()
TNetSignalCoordinator(NetCoordinatorId id)
TNetSignalCoordinator(const TNetSignalCoordinator& coordinator)
```

Construct a signalized control coordinator.

```
virtual void UpdateSignalizedControl(REAL sim_time)
```

Coordinate signal controls when necessary and then call the `UpdateSignalizedControl` method for each of the controllers.

```
PhasingPlanSet& GetPhasingPlans()
const PhasingPlanSet& GetPhasingPlans() const
```

Return the coordinator's phasing plans.

```
TNetPhasingPlan& CreatePhasingPlan(NetNodeId node, NetPlanId
           phasing, TNetTimingPlanReader& timing,
           TNetPhasingPlanReader& reader, TNetNetwork& network,
           PhaseNumberMap& numbers)
```

Define the phasing plan.

```
TNetPhasingPlan& GetPhasingPlan(NetPlanId id)
const TNetPhasingPlan& GetPhasingPlan(NetPlanId id) const
```

Return the phasing plan with specified id.

```
ControllerCollection& GetControllers()
const ControllerCollection& GetControllers() const
```

Define a controller.

```
void SetController(TNetSignalizedControl& controller)
```

Return the signalized controls coordinated by this coordinator.

```
NetCoordinatorId GetId() const
```

Return the coordinator's id.

33. TNetSimulationArea

The simulation area describes the simulation region of interest. A simulation area is described by links and whether the link is in the buffer portion of the simulation area.

```
enum EType {kStudy, kBuffer}
```

Link type indicates whether the link is in the buffer or study area.

```
TNetSimulationArea(TNetSimulationAreaLinkReader& reader)
TNetSimulationArea(const TNetSimulationArea& area)
```

Construct a simulation area.

```
TNetSimulationArea& operator=(const TNetSimulationArea& area)
```
Assign a simulation area.

```
bool operator==(const TNetSimulationArea& area) const
bool operator!=(const TNetSimulationArea& area) const
```
Return whether two simulation areas are the same.

```
bool IsInSimulationArea(NetLinkId id)
```
Return whether a link is in the simulation area (study area + buffer).

```
bool IsInStudyArea(NetLinkId id)
```
Return whether a link is in the study area.

```
bool IsInBufferArea(NetLinkId id)
```
Return whether a link is in the simulation area buffer region.

```
LinkIdMap& GetLinks()
const LinkIdMap& GetLinks() const
```
Return all links in simulation area.

34. TNetSimulationAreaReader

A simulation area reader reads a simulation area from the database. Each reader has a simulation area database table accessor and database fields.

```
TNetSimulationAreaReader(TDbTable areaTable)
```
Construct a reader for the specified tables.

```
TDbTable& GetSimulationAreaTable()
```
Return the simulation area table.

35. TNetSimulationAreaLinkReader

A simulation area link reader reads link values from the database. Each simulation area link reader has a database table accessor and database fields.

```
TNetSimulationAreaLinkReader(TNetSimulationAreaReader& reader)
```
Construct a simulation area link reader.

```
TNetSimulationAreaLinkReader(const TNetSimulationAreaLinkReader&
            reader)
```
Construct a copy of the given simulation area link reader.

```
TNetSimulationAreaLinkReader& operator=(const
            TNetSimulationAreaLinkReader& reader)
```
Make the reader a copy of the given simulation area link reader.

```
void Reset()
```
Reset the iteration over the table.

`void GetNextLink()`
Get the next link in the table.

`bool MoreLinks() const`
Return whether there are any more links in the table.

`LinkIdMap GetLinks()`
Return the links in the simulation area.

36. TGeoPoint

A geographic point contains the coordinates of a position on a map. Each point has an *x* coordinate and a *y* coordinate.

`TGeoPoint(REAL x = 0, REAL y = 0)`
Construct a point with the given *x* and *y* coordinates.

`TGeoPoint(const TGeoPoint& point)`
Construct a copy of the given point.

`TGeoPoint& operator=(const TGeoPoint& point)`
Make the point a copy of the given point.

`REAL GetX() const`
Return the *x* coordinate.

`REAL GetY() const`
Return the *y* coordinate.

`REAL GetAngleTo(const TGeoPoint& point) const`
Return the angle to the specified point.

37. TGeoRectangle

A geographic rectangle is a rectangle in a map coordinate system. Each rectangle has a minimum corner and a maximum corner.

`TGeoRectangle(const TGeoPoint& corner1, const TGeoPoint& corner2)`
Construct a rectangle with the given corners.

`TGeoRectangle(const TGeoRectangle& rectangle)`
Construct a copy of the given rectangle.

`TGeoRectangle& operator=(const TGeoRectangle& rectangle)`
Make the rectangle a copy of the given rectangle.

`void GetCorners(TGeoPoint& corner1, TGeoPoint& corner2) const`
Return the corners.

`bool Contains(const TGeoPoint& point) const`
Return whether the rectangle contains the given point.

38. TGeoFilterFunction

A geographic filter function selects geographic points. This abstract class must be subclassed to be used.

`virtual bool operator()(const TGeoPoint& point) const`
Return whether a point is acceptable.

39. TGeoFilterNone

This geographic filter accepts all points.

`TGeoFilterNone(const TGeoFilterNone& filter)`
Construct a copy of the given filter function.

`TGeoFilterNone& operator=(const TGeoFilterNone& filter)`
Make the filter a copy of the given filter function.

`bool operator()(const TGeoPoint& point) const`
Return whether a point is acceptable.

40. TGeoFilterRectangle

This geographic filter accepts all points within a rectangle. Each rectangular filter has a rectangle.

`TGeoFilterRectangle(const TGeoRectangle& rectangle)`
Construct a rectangular filter for the given rectangle.

`TGeoFilterRectangle(const TGeoFilterRectangle& filter)`
Construct a copy of the given filter function.

`TGeoFilterRectangle& operator=(const TGeoFilterRectangle& filter)`
Make the filter a copy of the given filter function.

`bool operator()(const TGeoPoint& point) const`
Return whether a point is acceptable.

41. TNetException

A network exception signals the failure of a network subsystem function. Each exception has a message. Figure 7 shows the hierarchy of exception classes.

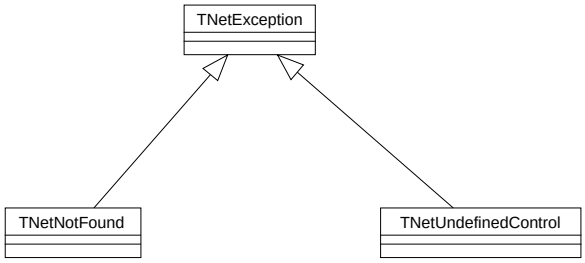

Figure 7. Exception hierarchy for the TRANSIMS network subsystem (unified notation).

`TNetException(const string& message = "Network error.")`
Construct an exception with the specified message text.

`TNetException(const TNetException& exception)`
Construct a copy of the given exception.

`TNetException& operator=(const TNetException& exception)`
Make the exception a copy of the given exception.

`const string& GetMessage() const`
Return the message text for the exception.

`class TNetNotFound`
This exception is thrown when an attempt is made to access something that cannot be found.

`class TNetUndefinedControl`
This exception is thrown when an attempt is made to access the member functions of a null control.

III. Implementation

A. C++ Libraries

The Booch Components [RW 94] the provide C++ container classes that the network representation subsystem uses extensively. The subsystem also uses the standard C++ library [Pl 95], the standard C library [Pl 92], and the POSIX library [Ga 95b]. All of these libraries compile on a wide variety of platforms (UNIX and otherwise).

B. Sources for Traffic Engineering Information

References [MM 84] and [PP 93] provide an overview of traffic engineering practice incorporated into the TRANSIMS network represenation.

IV. Usage

A. Accessing Network Data via C++

The following example shows how to create a network from data tables and to retrieve the node and link objects in the network:

```
//  Open the data directory.
TDbDirectory directory(TDbDirectoryDescription("IOC-1"));

//  Open the network data sources.
TDbSource nodeSource(directory, directory.GetSource("Node"));
TDbSource linkSource(directory, directory.GetSource("Link"));
TDbSource pocketSource(directory,
        directory.GetSource("Pocket Lane"));
TDbSource parkingSource(directory,
        directory.GetSource("Parking"));
TDbSource laneSource(directory,
        directory.GetSource("Lane Connectivity"));
TDbSource ucontrolSource(directory,
        directory.GetSource("Unsignalized Node"));
TDbSource scontrolSource(directory,
        directory.GetSource("Signalized Node"));
TDbSource phasingSource(directory,
        directory.GetSource("Phasing Plan"));
TDbSource timingSource(directory,
        directory.GetSource("Timing Plan"));

//  Read the table names from standard input and open them.
char line[80];
cin.getline(line, 80);
TDbTable nodeTable(nodeSource, nodeSource.GetTable(line));
cin.getline(line, 80);
TDbTable linkTable(linkSource, linkSource.GetTable(line));
cin.getline(line, 80);
TDbTable pocketTable(pocketSource, pocketSource.GetTable(line));
cin.getline(line, 80);
TDbTable parkingTable(parkingSource,
        parkingSource.GetTable(line));
cin.getline(line, 80);
TDbTable laneTable(laneSource, laneSource.GetTable(line));
cin.getline(line, 80);
TDbTable ucontrolTable(ucontrolSource,
        ucontrolSource.GetTable(line));
cin.getline(line, 80);
TDbTable scontrolTable(scontrolSource,
        scontrolSource.GetTable(line));
cin.getline(line, 80);
TDbTable phasingTable(phasingSource,
        phasingSource.GetTable(line));
cin.getline(line, 80);
TDbTable timingTable(timingSource, timingSource.GetTable(line));

//  Create a network reader for the tables.
TNetReader reader(nodeTable, linkTable, pocketTable, parkingTable,
        laneTable, ucontrolTable, scontrolTable, phasingTable,
        timingTable);
```

```
//  Create the network.
TNetNetwork network;

//  Read all of the network data.
TNetSubnetwork subnetwork(reader, network);

//  Get the nodes and links.
TNetSubnetwork::NodeSet& nodes = subnetwork.GetNodes();
TNetSubnetwork::LinkSet& links = subnetwork.GetLinks();
```

B. Network Data Tables

Nine data tables are required to describe a TRANSIMS format road network. The tables, their fields, and an example are shown below.

1. File Formats

The preferred format for data files is ASCII, with columns delimited by tab characters. Records are terminated by an end-of-line character. Formats such as dBASE III+ are also acceptable. Table 1 through Table 11 below defined the fields for the network data tables.

Table 1. Node table format.

Column Name	*Description*	*Allowed Values*
ID	id # of the node	integer: 1 through 4,294,967,295
ABSCISSA	x-coordinate of the node	floating-point number
ORDINATE	y-coordinate of the node	floating-point number

Table 2. Link table format.

Column Name	*Description*	*Allowed Values*
ID	id # of the link	integer: 1 through 4,294,967,295
NODEA	id # of the node at A	integer: 1 through 4,294,967,295
NODEB	id # of the node at B	integer: 1 through 4,294,967,295
PERMLANESA	number of lanes on the link heading toward node A, not including pocket lanes	integer: 1 through 255
PERMLANESB	number of lanes on the link heading toward node B, not including pocket lanes	integer: 1 through 255
LEFTPCKTSA	number of pocket lanes to the left of the permanent lanes heading toward node A	integer: 1 through 255
LEFTPCKTSB	number of pocket lanes to the left of the permanent lanes heading toward node B	integer: 1 through 255
RGHTPCKTSA	number of pocket lanes to the right of the permanent lanes heading toward node A	integer: 1 through 255
RGHTPCKTSB	number of pocket lanes to the right of the permanent lanes heading toward node B	integer: 1 through 255
TWOWAYTURN	whether there is a two-way left-turn lane in the center of the link	one character: ‘F’ = false/no, ‘T’ = true/yes
LENGTH	length of the link	floating-point number
GRADE	percent grade from node A to node B, uphill being a positive number	floating-point number

Table 3. Link table format (continued).

Column Name	*Description*	*Allowed Values*
SETBACKA	set-back distance from the center of the intersection at node A	floating-point number
SETBACKB	set-back distance from the center of the intersection at node B	floating-point number
CAPACITYA	total capacity (in vehicles-per-hour) for the lanes traveling to node A	floating-point number
CAPACITYB	total capacity (in vehicles-per-hour) for the lanes traveling to node B	floating-point number
SPEEDLMTA	speed limit for vehicles traveling toward node A	floating-point number
SPEEDLMTB	speed limit for vehicles traveling toward node B	floating-point number
FREESPDA	free-flow speed for vehicles traveling toward node A	floating-point number
FREESPDB	free-flow speed for vehicles traveling toward node B	floating-point number
CRAWLSPDA	crawl speed for vehicles traveling toward node	floating-point number
CRAWLSPDB	crawl speed for vehicles traveling toward node B	floating-point number
FUNCTCLASS	functional class of the link	ten characters
COSTA	travel cost for vehicles traveling toward node A	integer: 0 through 4,294,967,295
COSTB	travel cost for vehicles traveling toward node B	integer: 0 through 4,294,967,295
THRUA	default through link connected at node A	integer: 1 through 4,294,967,295
THRUB	default through link connected at node B	integer: 1 through 4,294,967,295

Table 4. Pocket lane table format.*

Column Name	*Description*	*Allowed Values*
ID	id # of the pocket lane accessory	integer: 1 through 4,294,967,295
NODE	id # of the node into which the pocket lane leads	integer: 1 through 4,294,967,295
LINK	id # of the link on which the pocket lane lies	integer: 1 through 4,294,967,295
OFFSET	starting position of the pocket lane, measured from the node away from which it is traveling (pull-out pockets only)	floating-point number
LANE	lane number of the pocket lane	integer: 1 through 255
STYLE	type of the pocket lane	one character: ‘T’ = turn pocket, ‘P’ = pull-out pocket, ‘M’ = merge pocket
LENGTH	length of the pocket lane (turn pockets and merge pockets always start or end at the appropriate limit line)	floating-point number

* Note that pocket lanes are accessories and so have an accessory id; this id must be unique over all types of accessories, not just pocket lane accessories.

Table 5. Parking table format.*

Column Name	*Description*	*Allowed Values*
ID	id # of the parking place	integer: 1 through 4,294,967,295
NODE	id # of the node into which the parking leads	integer: 1 through 4,294,967,295
LINK	id # of the link on which the parking lies	integer: 1 through 4,294,967,295
OFFSET	starting position of the parking, measured from the node away from which it is traveling	floating-point number
STYLE	type of the parking	five characters: 'PRSTR' = parallel on street, 'HISTR' = head in on street, 'DRVWY' = driveway, 'LOT' = parking lot, 'BNDRY' = network boundary
CAPACITY	number of vehicles the parking can accommodate, zero for unlimited capacity	integer: 0 through 65,535
GENERIC	whether the accessory represents generic parking (not an actual driveway, lot, etc., but a group/aggregate of them used to simplify modeling)	one character: 'T' = true/yes, 'F' = false/no

Table 6. Lane connectivity table format.

Column Name	*Description*	*Allowed Values*
NODE	id # of the node	integer: 1 through 4,294,967,295
INLINK	id # of an incoming link	integer: 1 through 4,294,967,295
INLANE	lane number of an incoming lane	integer: 1 through 255
OUTLINK	id # of an outgoing link	integer: 1 through 4,294,967,295
OUTLANE	lane number of an outgoing lane	integer: 1 through 255

Table 7. Unsignalized node table format.

Column Name	*Description*	*Allowed Values*
NODE	id # of the node	integer: 1 through 4,294,967,295
INLINK	id # of an incoming link	integer: 1 through 4,294,967,295
SIGN	type of sign control on link	one character: 'S' = stop, 'Y' = yield, 'N' = none

* Note that parking areas are accessories and so have an accessory id; this id must be unique over all of types of accessories, not just parking area accessories.

Table 8. Signalized node table format.

Column Name	*Description*	*Allowed Values*
NODE	id # of the node	integer: 1 through 4,294,967,295
TYPE	the type of the signal	one character: ‘T’ = timed, ‘A’ = actuated
PLAN	id # of a timing plan	integer: 1 through 255
OFFSET	offset in seconds for coordinated signals	floating-point number
STARTTIME	starting time for this plan	a character string with the day of week (‘SUN’ = Sunday, ‘MON’ = Monday, ‘TUE’ = Tuesday, ‘WED’ = Wednesday, ‘THU’ = Thursday, ‘FRI’ = Friday, ‘SAT’ = Saturday, ‘WKE’ = any weekend day, ‘WKD’ = any weekday, ‘ALL’ = any day) followed by the time of day (on a 24-hour clock), for example ‘WKD13:20’ is any weekday at 1:20 in the afternoon

Table 9. Phasing plan table format.

Column Name	*Description*	*Allowed Values*
NODE	id # of the node	integer: 1 through 4,294,967,295
PLAN	id # of a timing plan	integer: 1 through 255
PHASE	phase number	integer: 1 through 255
INLINK	id # of an incoming link	integer: 1 through 4,294,967,295
OUTLINK	id # of an outgoing link	integer: 1 through 4,294,967,295
PROTECTION	turn protection indicator	one character: ‘P’ = protected, ‘U’ = unprotected

Table 10. Timing plan table format.*

Column Name	*Description*	*Allowed Values*
PLAN	id # of a timing plan	integer: 1 through 255
PHASE	phase number	integer: 1 through 255
NEXTPHASES	phase number(s) of the next phase(s) in sequence	string of phase numbers, separated by slashes
GREENMIN	minimum length in seconds of green interval, or fixed green length for timed signal	floating-point number
GREENMAX	maximum length in seconds of green interval	floating-point number
GREENEXT	length in seconds of green extension interval	floating-point number
YELLOW	length in seconds of yellow interval	floating-point number
REDCLEAR	length in seconds of red clearance interval	floating-point number

Table 11. Study area link table format.

Column Name	*Description*	*Allowed Values*
ID	id # of the link	integer: 1 through 4,294,967,295
BUFFER	whether the link is in the buffer area or the study area	one character: 'Y' = in buffer area, 'N' = in study area

2. Example

An example network is depicted in Figure 8, followed by Table 12 through Table 25 which illustrate the data tables that correspond to this network. This example shows how the following are represented in this specification:

- signalized intersection
- unsignalized intersection
- a node without an intersection
- turn pocket
- merge pocket
- pull-out pocket
- lane use
- lane connectivity
- phasing plan
- timing plan

* GREENMAX and GREENEXT are undefined for pre-timed signals and should be indicated in the table as 0.0. REDCLEAR should be specified as 0.0 when there is no red clearance interval.

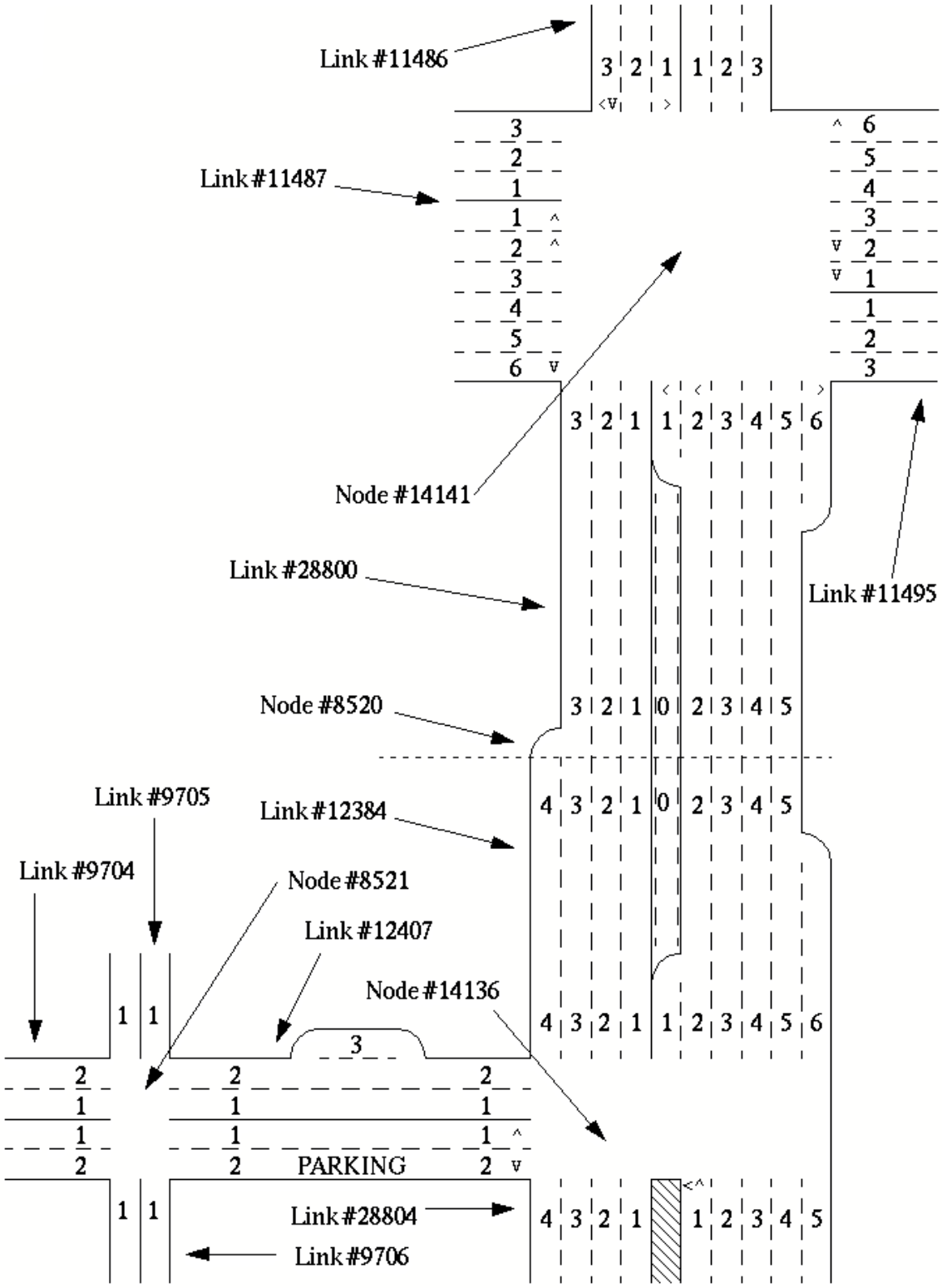

Figure 8 . Example network.

Three of the nodes (#8521, #14136, and #14141) in Table 12 are associated with intersections, while the other (#8520) is associated with a change in the permanent number of lanes.

Table 12. Example node table.

ID	*ABSCISSA*	*ORDINATE*
8520	1.0	0.5
8521	0.0	0.0
14136	1.0	0.0
14141	1.0	1.0

The three links in Table 13 illustrate how the permanent number of lanes, left pocket lanes, and right pocket lanes are counted and numbered.

Table 13. Example link table (part I).

ID	*NODEA*	*NODEB*	*PERMLANESA*	*PERMLANESB*	*LEFTPCKTSA*	*LEFTPCKTSB*
12384	14136	8520	4	4	0	1
12407	8521	14136	2	2	0	0
28800	8520	14141	3	4	0	1

Table 14. Example link table (part II).

RGHTPCKTSA	*RGHTPCKTSB*	*TWOWAYTURN*	*LENGTH*	*GRADE*	*SETBACKA*	*SETBACKB*
0	1	T	1000.0	1.0	15.0	0.0
1	0	F	1200.0	0.0	10.0	15.0
0	1	T	1500.00	1.0	0.0	15.0

Table 15. Example link table (part III).

CAPACITYA	*CAPACITYB*	*SPEEDLMTA*	*SPEEDLMTB*	*FREESPDA*	*FREESPDB*
800	1000	45.0	45.0	50.0	50.0
500	500	35.0	35.0	40.0	40.0
800	1000	45.0	45.0	50.0	50.0

Table 16. Example link table (part IV).

CRAWLSPDA	*CRAWLSPDB*	*THRUA*	*THRUB*	*COSTA*	*COSTB*	*FUNCTCLASS*
15.0	15.0	28804	28800	1	1	OTHER
10.0	10.0	9704	0	1	1	OTHER
15.0	15.0	12384	11486	1	1	OTHER

All three types of pocket lanes (turn pocket, merge pocket, and pull-out pocket) are represented in Table 17.

Table 17. Example pocket lane table.

ID	*NODE*	*LINK*	*OFFSET*	*LANE*	*STYLE*	*LENGTH*
85201	8520	12384	0	1	M	100.0
852062	8520	12384	0	6	M	200.0
85213	8521	12407	600.0	3	P	30.0
141411	14141	28800	0	1	T	200.0
141416	14141	28800	0	6	T	300.0

Referring to Figure 8, the first six rows in Table 18 may be understood as follows. Lanes 1 and 2 on link 11487 (attached to node 14141) are exclusive left turn lanes connecting only to lanes 1 and 2 on link 11486. Lanes 3, 4 and 5 on link 11487 are through lanes to lanes 1, 2 and 3 on link 11495. Lane 6 on link 11487 is a right-turn-only lane connecting to lane 3 on link 28800. Lane 3 on link 11486 connects to both lane 2 on link 28800 and lane 3 on link 11487, as shown in rows 9 and 10 of the table.

Table 18. Example lane connectivity table.

NODE	*INLINK*	*INLANE*	*OUTLINK*	*OUTLANE*
14141	11487	1	11486	1
14141	11487	2	11486	2
14141	11487	3	11495	1
14141	11487	4	11495	2
14141	11487	5	11495	3
14141	11487	6	28800	3
14141	11486	1	11495	1
14141	11486	2	28800	1
14141	11486	3	28800	2
14141	11486	3	11487	3
14141	11495	1	28800	1
14141	11495	2	28800	2
14141	11495	3	11487	1
14141	11495	4	11487	2
14141	11495	5	11487	3
14141	11495	6	11486	3
14141	28800	1	11487	1
14141	28800	2	11487	2
14141	28800	3	11486	1
14141	28800	4	11486	2
14141	28800	5	11486	3
14141	28800	6	11495	3
8520	12384	2	28800	2
8520	12384	3	28800	3
8520	12384	4	28800	4
8520	12384	5	28800	5
8520	28800	1	12384	1
8520	28800	2	12384	2
8520	28800	3	12384	3
8520	28800	3	12384	4

Table 19. Example lane connectivity table (continued).

NODE	*INLINK*	*INLANE*	*OUTLINK*	*OUTLANE*
14136	12407	1	12384	1
14136	12407	2	28804	4
14136	12384	1	28804	1
14136	12384	2	28804	2
14136	12384	3	28804	3
14136	12384	4	28804	4
14136	12384	4	12407	2
14136	28804	1	12407	1
14136	28804	1	12384	2
14136	28804	2	12384	3
14136	28804	3	12384	4
14136	28804	4	12384	5
14136	28804	5	12384	6
8521	12407	1	9704	1
8521	12407	1	9706	1
8521	12407	2	9704	2
8521	12407	2	9705	1
8521	9704	1	12407	1
8521	9704	1	9705	1
8521	9704	2	12407	2
8521	9704	2	9706	1
8521	9705	1	9706	1
8521	9705	1	9704	2
8521	9705	1	12407	1
8521	9706	1	9705	1
8521	9706	1	12407	2
8521	9706	1	9704	1

Several types of parking (lot, street, driveway, and generic vs. actual) are represented in the Table 20.

Table 20. Example parking table.

ID	*NODE*	*LINK*	*OFFSET*	*STYLE*	*CAPACITY*	*GENERIC*
1001	28800	8520	400	LOT	50	T
1002	12384	14136	300	PRSTR	10	T
1003	12407	14136	200	HISTR	10	T
1004	12407	8521	100	DRVWY	1	F

The number of permanent lanes changes from three lanes on link 28800 to four lanes on link 12384 at node 8520. No right-of-way sign control is required at this node. A stop sign is indicated on link 12407 at node 14136, with no sign control on the other two links at this node. Table 21 illustrates these.

Table 21. Example unsignalized node table.

NODE	*INLINK*	*SIGN*
8520	12384	N
8520	28800	N
14136	12407	S
14136	12384	N
14136	28804	N

Node 14141 has a pre-timed signal control with an offset of 19.0 seconds. A single timing and phasing plan is always in effect. Node 8521 is defined as having an actuated signal and two timing and phasing plans. Table 22 illustrates these.

Table 22. Example signalized node table.

NODE	*TYPE*	*PLAN*	*OFFSET*	*STARTTIME*
14141	T	1	19.0	‘ALL0:00’
8521	A	2	0.0	‘ALL18:00’
8521	A	3	0.0	‘WKD7:00’

The movements permitted during phase 1 at node 14141 are through movements between links 11487 and 11495, as well as right turn movements from these links. The right turns are protected, and the protection for the through movements is blank because we don’t know whether to consider them as protected or unprotected. (This issue was raised earlier.) Additionally, unprotected right turns are permitted from links 11486 and 28800 during phase 1. The first six rows of Table 23 specify this information.

Table 23. Example phasing plan table.

NODE	*PLAN*	*PHASE*	*INLINK*	*OUTLINK*	*PROTECTION*
14141	1	1	11487	11495	U
14141	1	1	11487	28800	P
14141	1	1	11495	11487	U
14141	1	1	11495	11486	P
14141	1	1	11486	11487	U
14141	1	1	28800	11495	U
14141	1	2	11487	28800	P
14141	1	2	11495	11486	P
14141	1	2	11486	11495	P
14141	1	2	28800	11487	P
14141	1	2	28800	11495	U
14141	1	2	11486	11487	U
14141	1	3	11487	28800	P
14141	1	3	28800	11487	P
14141	1	3	28800	11486	U
14141	1	3	28800	11495	P
14141	1	3	11495	11486	U
14141	1	3	11486	11487	U
14141	1	4	11487	28800	P
14141	1	4	11486	11495	U
14141	1	4	11486	28800	U
14141	1	4	11486	11487	P
14141	1	4	28800	11486	U
14141	1	4	28800	11495	P
14141	1	4	11495	11486	U
14141	1	5	11487	11486	P
14141	1	5	11487	28800	P
14141	1	5	11495	28800	P
14141	1	5	11495	11486	U
14141	1	5	11486	11487	P
14141	1	5	28800	11495	P
14141	1	6	11487	28800	P
14141	1	6	11495	28800	P
14141	1	6	11495	11487	U
14141	1	6	11495	11486	P
14141	1	6	11486	11487	U
14141	1	6	28800	11495	P

Table 24. Example phasing plan table (continued).

NODE	*PLAN*	*PHASE*	*INLINK*	*OUTLINK*	*PROTECTION*
8521	2	1	9705	9704	U
8521	2	1	9705	9706	U
8521	2	1	9705	12407	U
8521	2	1	9706	9705	U
8521	2	1	9706	12407	U
8521	2	1	9706	9704	U
8521	2	2	12407	9704	U
8521	2	2	12407	9705	U
8521	2	2	12407	9706	U
8521	2	2	9704	12407	U
8521	2	2	9704	9705	U
8521	2	2	9704	9706	U
8521	3	1	9705	9704	U
8521	3	1	9705	9706	U
8521	3	1	9705	12407	U
8521	3	1	9706	9705	U
8521	3	1	9706	12407	U
8521	3	1	9706	9704	U
8521	3	2	12407	9706	P
8521	3	2	9704	9705	P
8521	3	3	12407	9704	U
8521	3	3	12407	9705	U
8521	3	3	12407	9706	U
8521	3	3	9704	12407	U
8521	3	3	9704	9705	U
8521	3	3	9704	9706	U

Plan 1 in the Table 25 was specified in the Table 22 as applicable to node 14141. This is a timed signal with green, yellow, and red clearance intervals as indicated in row 1 of the table. Plans 2 and 3 for node 8521 were invented as illustrations and may not make sense as real timing plans.

Table 25. Example timing plan table.

PLAN	*PHASE*	*NEXTPHASES*	*GREENMIN*	*GREENMAX*	*GREENEXT*	*YELLOW*	*REDCLEAR*
1	1	2	35.0	0.0	0.0	4.0	0.0
1	2	3	5.0	0.0	0.0	3.0	0.0
1	3	4	8.0	0.0	0.0	3.0	0.0
1	4	5	32.0	0.0	0.0	4.0	0.0
1	5	6	9.0	0.0	0.0	3.0	0.0
1	6	1	1.0	0.0	0.0	3.0	0.0
2	1	2	12.0	30.0	4.0	3.0	0.0
2	2	1	10.0	40.0	4.0	3.0	0.0
3	1	2	12.0	30.0	4.0	3.0	1.0
3	2	3	4.0	8.0	2.0	3.0	0.0
3	3	1	10.0	20.0	4.0	3.0	1.0

C. Notes

1. Row Order of Phasing and Timing Plans

Because of a limitation in how the network construction is organized in the C++ code, it is necessary to use network data tables that have phasing plans and timing plans in sequential order.

2. Boundary Accessories

Boundary accessories have not been implemented for the IOC-1 versions of the network representation subsystem.

V. Future Work

Future work planned for the TRANSIMS network representation subsystem will focus on the inclusion of additional modes of travel (bus, rail, etc.), time-of-day variation in network properties, and more complicated traffic controls (actuated signals and signals coordinated over a wide area). We will also provide a highly portable alternative implementation that is not based on commercial products such as the Booch Components. Performance enhancements are also anticipated.

VI. References

[Ga 95b] B. O. Gallmeister, *POSIX.4: Programming for the Real World*, (Sebastopol, California: O'Reilly & Associates, 1995).

[MM 84] M. D. Meyer and E. J. Miller, *Urban Transportation Planning*, (New York: McGraw-Hill, 1984).

[Pl 92] P. J. Plauger, *The Standard C Library*, (Englewood Cliffs, New Jersey: Prentice Hall, 1992).

[Pl 95] P. J. Plauger, *The Draft Standard C++ Library*, (Englewood Cliffs, New Jersey: Prentice Hall, 1995).

[PP 93] C. S. Papacostas and P. D. Prevedouros, *Transportation Engineering and Planning*, (Englewood Cliffs, New Jersey: Prentice Hall, 1993).

[RW 94] Rogue Wave Software, *The C++ Booch Components*, Version 2.3, (Corvallis, Oregon: Rogue Wave Software, 1994).

VII. APPENDIX: Booch Notation Diagrams

Figure 9 through Figure 13 repeat the class diagrams of Figure 2 through Figure 6 using Booch notation and Figure 14 repeats the exception hierarchy of Figure 7 using Booch notation.

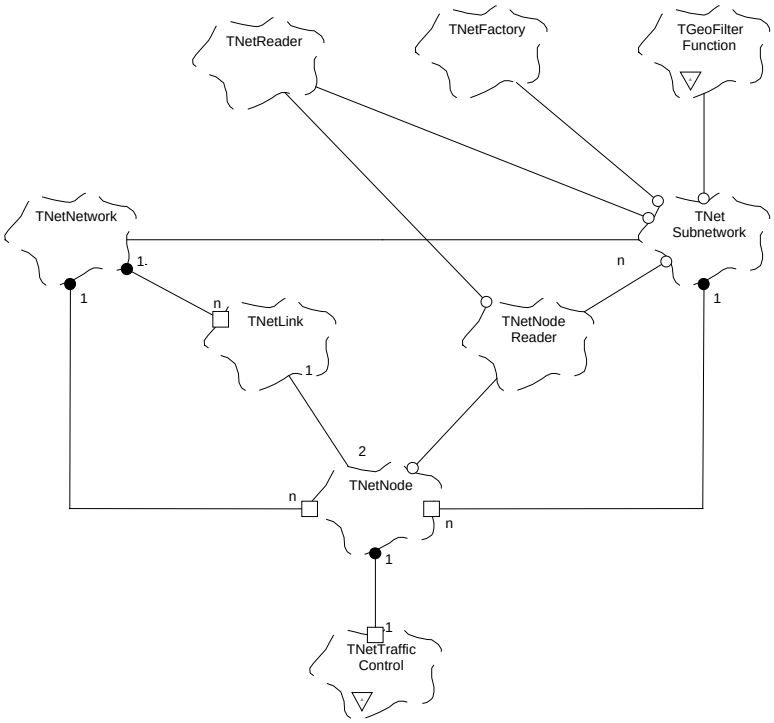

Figure 9. Class diagram for the node-related classes in the TRANSIMS network representation subsystem (Booch notation).

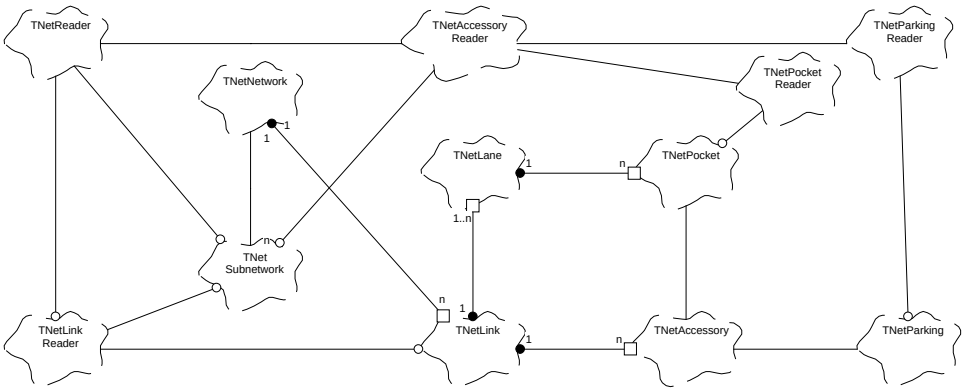

Figure 10. Class diagram for the node-related classes in the TRANSIMS network representation subsystem (Booch notation).

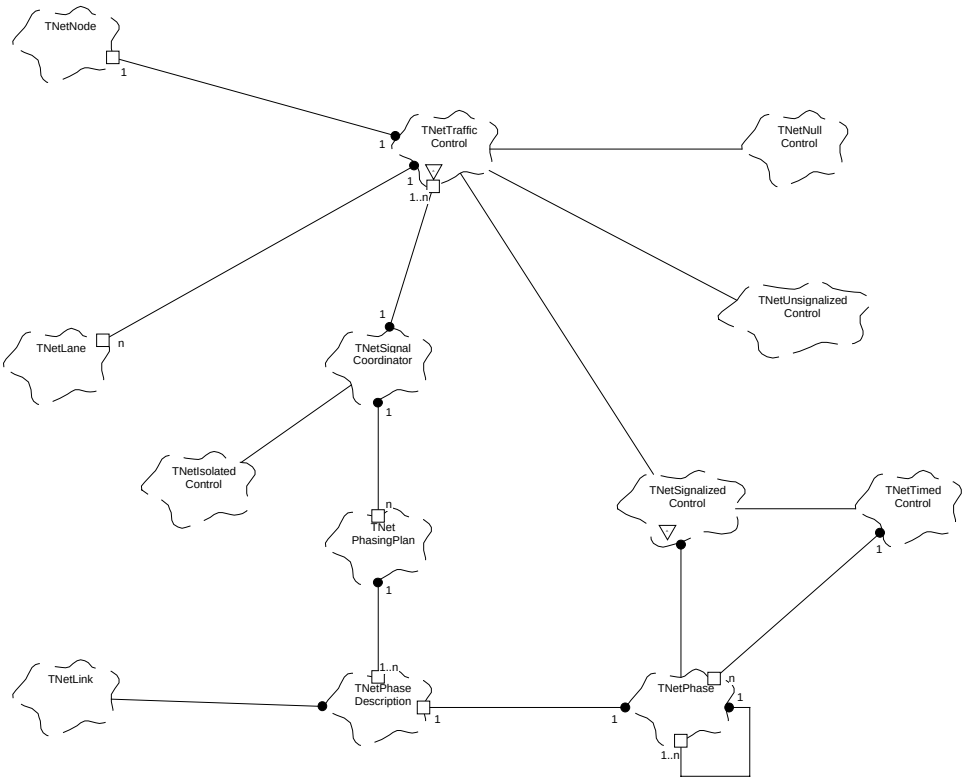

Figure 11. Class diagram for the control-related classes in the TRANSIMS network representation subsystem (Booch notation).

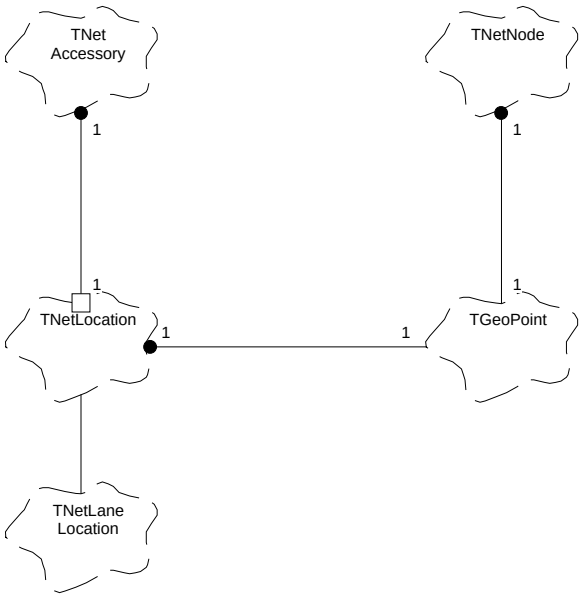

Figure 12. Class diagram for the location-related classes in the TRANSIMS network representation subsystem (Booch notation).

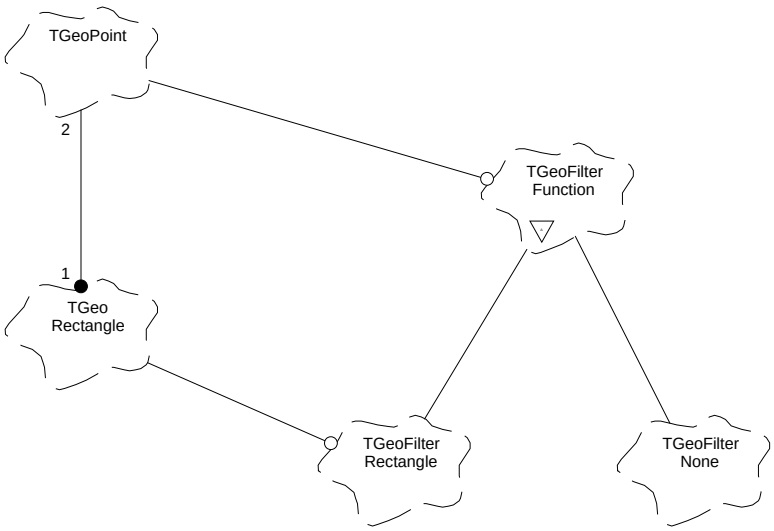

Figure 13. Class diagram for the geography-related classes in the TRANSIMS network representation subsystem (Booch notation).

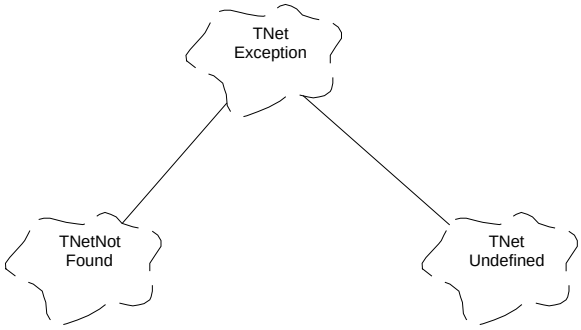

Figure 14. Exception hierarchy for the TRANSIMS network representation subsystem (Booch notation).

VIII. APPENDIX: Source Code

This appendix contains the complete C++ source code for the network subsystem classes.

A. TNetAccessory Class

1. Accessory.h

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: Accessory.h,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 19:29:51 $
//  $State: Rel $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


#ifndef TRANSIMS_NET_ACCESSORY
#define TRANSIMS_NET_ACCESSORY


//  Include TRANSIMS header files.
```

```
#include <NET/Id.h>


//  Forward declarations.
class TNetLocation;


//  An accessory is where something happens along a link.
class TNetAccessory
{
  public:

    //  There are several types of accessories.
    enum EType {kPocket, kParking};

    //  Construct an accessory with the specified id and type.
    TNetAccessory(NetAccessoryId id, EType type);

    //  Destroy the accessory.
    virtual ~TNetAccessory();

    //  Return the id of the accessory.
    NetAccessoryId GetId() const;

    //  Return the type of the accessory.
    EType GetType() const;

    //  Return the location of the accessory.
    TNetLocation& GetLocation();
    const TNetLocation& GetLocation() const;

    //  Set the location of the accessory.
    void SetLocation(const TNetLocation& location);

  private:

    //  Do not allow accessories to be copied.
    TNetAccessory(const TNetAccessory&) {}

    //  Do not allow accessories to be assigned.
    TNetAccessory& operator=(const TNetAccessory&) {return *this;}

    //  Each accessory has a unique id.
    NetAccessoryId fId;

    //  Each accessory has a type.
    EType fType;

    //  Each accessory has a location.
    TNetLocation* fLocation;
};


#endif // TRANSIMS_NET_ACCESSORY
```

2. Accessory.C

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: Accessory.C,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 19:29:51 $
//  $State: Rel $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


//  Include TRANSIMS header files.
#include <NET/Accessory.h>
#include <NET/Location.h>
#include <NET/LaneLocation.h>


//  Construct an accessory with the specified id and type.
```

```
TNetAccessory::TNetAccessory(NetAccessoryId id, EType type)
  : fId(id),
    fType(type),
    fLocation(NULL)
{
}


//  Destroy the accessory.
TNetAccessory::~TNetAccessory()
{
    if (fLocation != NULL) {
        if (fLocation->IsOnSpecificLane())
            delete (TNetLaneLocation*) fLocation;
        else
           delete fLocation;
    }
}


//  Return the id of the accessory
NetAccessoryId TNetAccessory::GetId() const
{
    return fId;
}


//  Return the type of the accessory.
TNetAccessory::EType TNetAccessory::GetType() const
{
    return fType;
}


//  Return the location of the accessory.
TNetLocation& TNetAccessory::GetLocation()
{
    return *fLocation;
}

const TNetLocation& TNetAccessory::GetLocation() const
{
    return *fLocation;
}


//  Set the location of the accessory.
void TNetAccessory::SetLocation(const TNetLocation& location)
{
    delete fLocation;
    fLocation = NULL;
    fLocation = location.IsOnSpecificLane() ? new TNetLaneLocation((const
            TNetLaneLocation&) location) : new TNetLocation(location);
}
```

B. TNetAccessoryReader Class

1. AccessoryReader.h

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: AccessoryReader.h,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 19:29:51 $
//  $State: Rel $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


#ifndef TRANSIMS_NET_ACCESSORYREADER
#define TRANSIMS_NET_ACCESSORYREADER
```

```
//  Include TRANSIMS header files.
#include <GBL/Globals.h>
#include <DBS/Accessor.h>
#include <NET/Id.h>
#include <NET/Accessory.h>


//  Forward declarations.
class TNetReader;


//  An accessory reader reads accessory values from the database.
class TNetAccessoryReader
{
  public:

    //  Construct an accessory reader for a given network.
    TNetAccessoryReader(TDbTable& table);

    //  Construct a copy of the given accessory reader.
//  TNetAccessoryReader(const TNetParkingReader& reader);

    //  Make the accessory reader a copy of the given accessory reader.
//  TNetAccessoryReader& operator=(const TNetParkingReader& reader);

    //  Reset the iteration over the table.
    void Reset();

    //  Get the next accessory in the table.
    void GetNextAccessory();

    //  Return whether there are any more accessories in the table.
    bool MoreAccessories() const;

    //  Return the id for the current accessory.
    NetAccessoryId GetId() const;

    //  Return the location of the current accessory.
    void GetLocation(NetLinkId& link, NetNodeId& node, REAL& offset) const;

  protected:

    //  Each accessory has a database table accessor.
    TDbAccessor fAccessor;

    //  Each accessory has an ID field.
    const TDbField fIdField;

    //  Each accessory has a LINK field.
    const TDbField fLinkField;

    //  Each accessory has a NODE field.
    const TDbField fNodeField;

    //  Each accessory has an OFFSET field.
    const TDbField fOffsetField;
};


#endif // TRANSIMS_NET_ACCESSORYREADER
```

2. AccessoryReader.C

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: AccessoryReader.C,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 19:29:51 $
//  $State: Rel $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


//  Include TRANSIMS header files.
```

```
#include <NET/AccessoryReader.h>
#include <NET/Reader.h>


//  Construct an accessory reader for a given network.
TNetAccessoryReader::TNetAccessoryReader(TDbTable& table)
  : fAccessor(table),
    fIdField(table.GetField("ID")),
    fLinkField(table.GetField("LINK")),
    fNodeField(table.GetField("NODE")),
    fOffsetField(table.GetField("OFFSET"))
{
}


//  Reset the iteration over the table.
void TNetAccessoryReader::Reset()
{
    fAccessor.GotoFirst();
}


//  Get the next accessory in the table.
void TNetAccessoryReader::GetNextAccessory()
{
    fAccessor.GotoNext();
}


//  Return whether there are any more accessories in the table.
bool TNetAccessoryReader::MoreAccessories() const
{
    return fAccessor.IsAtRecord();
}


//  Return the id for the current accessory.
NetAccessoryId TNetAccessoryReader::GetId() const
{
    NetAccessoryId id;
    fAccessor.GetField(fIdField, id);
    return id;
}


//  Return the location of the current accessory.
void TNetAccessoryReader::GetLocation(NetLinkId& link, NetNodeId& node, REAL&
        offset) const
{
    fAccessor.GetField(fLinkField, link);
    fAccessor.GetField(fNodeField, node);
    fAccessor.GetField(fOffsetField, offset);
}
```

C. TNetException Class

1. Exception.h

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: Exception.h,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 19:29:51 $
//  $State: Rel $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


#ifndef TRANSIMS_NET_EXCEPTION
#define TRANSIMS_NET_EXCEPTION


//  Include TRANSIMS header files.
```

```
#include <GBL/Globals.h>


//  A network exception signals the failure of a member function.
class TNetException
{
  public:

    //  Construct an exception with the specified message text.
    TNetException(const string& message = "Network error.");

    //  Construct a copy of the given exception.
//  TNetException(const TNetException& exception);

    //  Make the exception a copy of the given exception.
//  TNetException& operator=(const TNetException& exception);

    //  Return the message text for the exception.
    const string& GetMessage() const;

  private:

    //  Each exception has a message.
    string fMessage;
};


//  This exception is thrown when an attempt is made to access something that
//  cannot be found.
class TNetNotFound
  : public TNetException
{
  public:

    //  Construct an exception with the specified message text.
    TNetNotFound(const string& message = "Not found.");

    //  Construct a copy of the given exception.
//  TNetException(const TNetException& exception);

    //  Make the exception a copy of the given exception.
//  TNetException& operator=(const TNetException& exception);
};


//  This exception is thrown when an attempt is made to access the member
//  functions of a NullControl.
class TNetUndefinedControl
  : public TNetException
{
  public:

    //  Construct an exception with the specified message text.
    TNetUndefinedControl(const string& message = "Null control.");

    //  Construct a copy of the given exception.
//  TNetException(const TNetException& exception);

    //  Make the exception a copy of the given exception.
//  TNetException& operator=(const TNetException& exception);
};


#endif // TRANSIMS_NET_EXCEPTION
```

2. Exception.C

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: Exception.C,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 19:29:51 $
//  $State: Rel $
//  $Author: bwb $
//  U.S. Government Copyright 1995
```

```
//  All rights reserved


//  Include TRANSIMS header files.
#include <NET/Exception.h>


//  Construct an exception with the specified message text.
TNetException::TNetException(const string& message)
  : fMessage(message)
{
}


//  Return the message text for the exception.
const string& TNetException::GetMessage() const
{
    return fMessage;
}


//  Construct a "not found" exception with the specified message text.
TNetNotFound::TNetNotFound(const string& message)
  : TNetException(message)
{
}


//  Construct an "undefined control" exception with the specified message text.
TNetUndefinedControl::TNetUndefinedControl(const string& message)
  : TNetException(message)
{
}
```

D. TNetFactory Class

1. Factory.h

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: Factory.h,v $
//  $Revision: 2.2 $
//  $Date: 1995/08/11 16:12:39 $
//  $State: Stab $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


#ifndef TRANSIMS_NET_FACTORY
#define TRANSIMS_NET_FACTORY


//  Include TRANSIMS header files.
#include <GBL/Globals.h>
#include <NET/Id.h>


//  Forward declarations.
class TNetNetwork;
class TNetNode;
class TNetLink;
class TNetPocket;
class TNetParking;
class TNetUnsignalizedControl;
class TNetTimedControl;
class TNetIsolatedControl;
class TNetNullControl;
class TNetNodeReader;
class TNetLinkReader;
class TNetPocketReader;
class TNetParkingReader;
class TNetUnsignalizedControlReader;
class TNetSignalizedControlReader;
```

```
//  A network factory allocates and constructs new network objects.
class TNetFactory
{
  public:

    //  Return a new node from the specified reader.
    virtual TNetNode* NewNode(TNetNodeReader& reader);

    //  Return a new link from the specified reader.
    virtual TNetLink* NewLink(TNetLinkReader& reader);

    //  Return a new pocket from the specified reader.
    virtual TNetPocket* NewPocket(TNetPocketReader& reader);

    //  Return a new parking place from the specified reader.
    virtual TNetParking* NewParking(TNetParkingReader& reader);

    //  Return a new unsignalized control from the specified reader and for
    //  the specified network.
    virtual TNetUnsignalizedControl*
            NewUnsignalizedControl(TNetUnsignalizedControlReader& reader,
            TNetNetwork& network);

    //  Return a new timed control from the specified reader and for the
    //  specified network.
    virtual TNetTimedControl* NewTimedControl(TNetSignalizedControlReader&
            reader, TNetNetwork& network);

    //  Return a new isolated control for the specified node.
    virtual TNetIsolatedControl* NewIsolatedControl(NetNodeId id);

    //  Return a new null control for the specified node.
    virtual TNetNullControl* NewNullControl(TNetNode& node);
};


#endif // TRANSIMS_NET_FACTORY
```

2. Factory.C

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: Factory.C,v $
//  $Revision: 2.1 $
//  $Date: 1995/08/09 14:32:04 $
//  $State: Exp $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


//  Include TRANSIMS header files.
#include <NET/Factory.h>
#include <NET/Network.h>
#include <NET/Node.h>
#include <NET/Link.h>
#include <NET/Pocket.h>
#include <NET/Parking.h>
#include <NET/UnsignalizedControl.h>
#include <NET/SignalizedControl.h>
#include <NET/PhaseDescription.h>
#include <NET/TimedControl.h>
#include <NET/IsolatedControl.h>
#include <NET/NullControl.h>
#include <NET/NodeReader.h>
#include <NET/LinkReader.h>
#include <NET/PocketReader.h>
#include <NET/ParkingReader.h>
#include <NET/UnsignalizedControlReader.h>
#include <NET/SignalizedControlReader.h>


//  Return a new node from the specified reader.
```

```
TNetNode* TNetFactory::NewNode(TNetNodeReader& reader)
{
    return new TNetNode(reader);
}


//  Return a new link from the specified reader.
TNetLink* TNetFactory::NewLink(TNetLinkReader& reader)
{
    return new TNetLink(reader);
}


//  Return a new pocket from the specified reader.
TNetPocket* TNetFactory::NewPocket(TNetPocketReader& reader)
{
    return new TNetPocket(reader);
}


//  Return a new parking place from the specified reader.
TNetParking* TNetFactory::NewParking(TNetParkingReader& reader)
{
    return new TNetParking(reader);
}


//  Return a new unsignalized control from the specified reader and for the
//  specified network.
TNetUnsignalizedControl*
        TNetFactory::NewUnsignalizedControl(TNetUnsignalizedControlReader&
        reader, TNetNetwork& network)
{
    return new TNetUnsignalizedControl(reader, network);
}


//  Return a new timed control from the specified reader and for the specified
//  network.
TNetTimedControl* TNetFactory::NewTimedControl(TNetSignalizedControlReader&
        reader, TNetNetwork& network)
{
    return new TNetTimedControl(reader, network);
}


//  Return a new isolated control for the specified node.
TNetIsolatedControl* TNetFactory::NewIsolatedControl(NetNodeId id)
{
    return new TNetIsolatedControl(id);
}


//  Return a new null control for the specified node.
TNetNullControl* TNetFactory::NewNullControl(TNetNode& node)
{
    return new TNetNullControl(node);
}
```

E. TGeoFilterFunction Class

1. FilterFunction.h

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: FilterFunction.h,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 19:29:51 $
//  $State: Rel $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved
```

```
#ifndef TRANSIMS_NET_FILTERFUNCTION
#define TRANSIMS_NET_FILTERFUNCTION


//  Include TRANSIMS header files.
#include <GBL/Globals.h>
#include <NET/Point.h>


//  A geographic filter function selects geographic points.  This abstract class
//  must be subclassed to be used.
class TGeoFilterFunction
{
  public:

    //  Return whether a point is acceptable.
    virtual bool operator()(const TGeoPoint& point) const = 0;
};


#endif // TRANSIMS_NET_FILTERFUNCTION
```

F. TGeoFilterNone Class

1. FilterNone.h

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: FilterNone.h,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 19:29:51 $
//  $State: Rel $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


#ifndef TRANSIMS_NET_FILTERNONE
#define TRANSIMS_NET_FILTERNONE


//  Include TRANSIMS header files.
#include <GBL/Globals.h>
#include <NET/FilterFunction.h>


//  This geographic filter accepts all points.
class TGeoFilterNone
  : public TGeoFilterFunction
{
  public:

    //  Construct a copy of the given filter function.
//  TGeoFilterNone(const TGeoFilterNone& filter);

    //  Make the filter a copy of the given filter function.
//  TGeoFilterNone& operator=(const TGeoFilterNone& filter);

    //  Return whether a point is acceptable.
    bool operator()(const TGeoPoint& point) const;
};


#endif // TRANSIMS_NET_FILTERNONE
```

2. FilterNone.C

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: FilterNone.C,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 19:29:51 $
//  $State: Rel $
```

```
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


//  Include TRANSIMS header files.
#include <NET/FilterNone.h>


//  Return whether a point is acceptable.
bool TGeoFilterNone::operator()(const TGeoPoint&) const
{
    return TRUE;
}
```

G. TGeoFilterRectangle Class

1. FilterRectangle.h

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: FilterRectangle.h,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 19:29:51 $
//  $State: Rel $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


#ifndef TRANSIMS_NET_FILTERRECTANGLE
#define TRANSIMS_NET_FILTERRECTANGLE


//  Include TRANSIMS header files.
#include <GBL/Globals.h>
#include <NET/FilterFunction.h>
#include <NET/Rectangle.h>


//  This geographic filter accepts all points within a rectangle.
class TGeoFilterRectangle
  : public TGeoFilterFunction
{
  public:

    //  Construct a rectangular filter for the given rectangle.
    TGeoFilterRectangle(const TGeoRectangle& rectangle);

    //  Construct a copy of the given filter function.
//  TGeoFilterRectangle(const TGeoFilterRectangle& filter);

    //  Make the filter a copy of the given filter function.
//  TGeoFilterRectangle& operator=(const TGeoFilterRectangle& filter);

    //  Return whether a point is acceptable.
    bool operator()(const TGeoPoint& point) const;

  private:

    //  Each rectangular filter has a rectangle.
    TGeoRectangle fRectangle;
};


#endif // TRANSIMS_NET_FILTERRECTANGLE
```

2. FilterRectangle.C

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: FilterRectangle.C,v $
//  $Revision: 2.0 $
```

```
//  $Date: 1995/08/04 19:29:51 $
//  $State: Rel $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


//  Include TRANSIMS header files.
#include <NET/FilterRectangle.h>


//  Construct a rectangular filter for the given rectangle.
TGeoFilterRectangle::TGeoFilterRectangle(const TGeoRectangle& rectangle)
  : fRectangle(rectangle)
{
}


//  Return whether a point is acceptable.
bool TGeoFilterRectangle::operator()(const TGeoPoint& point) const
{
    return fRectangle.Contains(point);
}
```

H. TNetIsolatedControl Class

1. IsolatedControl.h

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: IsolatedControl.h,v $
//  $Revision: 2.1 $
//  $Date: 1996/02/14 21:41:46 $
//  $State: Exp $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


#ifndef TRANSIMS_NET_ISOLATEDCONTROL
#define TRANSIMS_NET_ISOLATEDCONTROL


//  Include TRANSIMS header files.
#include "GBL/Globals.h"
#include "NET/SignalCoordinator.h"


//  Forward declarations.
class TNetPhasingPlan;


//  An isolated signal requires no coordination.  It defers to the signalized
//  control for operations.
class TNetIsolatedControl
  : public TNetSignalCoordinator
{
  public:

    //  Construct an isolated control.
    TNetIsolatedControl();
    TNetIsolatedControl(NetCoordinatorId id );
    TNetIsolatedControl(const TNetIsolatedControl& control);

    //  Destroy an isolated control.
    ~TNetIsolatedControl();

    //  Assign an isolated traffic control.
    TNetIsolatedControl& operator=(const TNetIsolatedControl& control);

    //  Return whether two isolated controls are the same.
    bool operator==(const TNetIsolatedControl& control) const;
    bool operator!=(const TNetIsolatedControl& control) const;
```

```
    //  Isolated control requires no coordination.
    void UpdateSignalizedControl(REAL sim_time);
};


#endif // TRANSIMS_NET_ISOLATEDCONTROL
```

2. IsolatedControl.C

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: IsolatedControl.C,v $
//  $Revision: 2.1 $
//  $Date: 1996/02/14 21:41:46 $
//  $State: Exp $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


//  Include TRANSIMS header files.
#include "NET/IsolatedControl.h"
#include "NET/SignalizedControl.h"
#include "NET/PhasingPlan.h"
#include "NET/Exception.h"


//  Construct an isolated control.
TNetIsolatedControl::TNetIsolatedControl()
  : TNetSignalCoordinator()
{
}

TNetIsolatedControl::TNetIsolatedControl(NetCoordinatorId id)
  : TNetSignalCoordinator(id)
{
}

TNetIsolatedControl::TNetIsolatedControl(const TNetIsolatedControl& c)
  : TNetSignalCoordinator(c)
{
}


//  Destroy an isolated control.
TNetIsolatedControl::~TNetIsolatedControl()
{
}


//  Assign an isolated traffic control.
TNetIsolatedControl& TNetIsolatedControl::operator=(const TNetIsolatedControl&
        c)
{
    if (this == &c)
        return *this;
    TNetSignalCoordinator::operator=(c);
    return *this;
}


//  Return whether two isolated controls are the same.
bool TNetIsolatedControl::operator==(const TNetIsolatedControl& c) const
{
    return (this == &c);
}

bool TNetIsolatedControl::operator!=(const TNetIsolatedControl& c) const
{
    return !(this == &c);
}


//  Isolated control requires no coordination.
void TNetIsolatedControl::UpdateSignalizedControl(REAL sim_time)
```

```
{
    if (GetControllers().Length() == 1)
        GetControllers().First()->UpdateSignalizedControl(sim_time);
    else
        throw TNetException("Isolated controllers must have exactly one “
                "signal.");
}
```

I. TNetLane Class

1. Lane.h

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: Lane.h,v $
//  $Revision: 2.3 $
//  $Date: 1996/11/08 18:12:44 $
//  $State: Stab $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


#ifndef TRANSIMS_NET_LANE
#define TRANSIMS_NET_LANE


//  Include Booch Components header files.
#include <BCStoreM.h>
#include <BCCollB.h>
#include <BCCollU.h>


//  Include TRANSIMS header files.
#include <NET/Id.h>


//  Forward declarations.
class TNetNode;
class TNetLink;
class TNetPocket;


//  A lane is where traffic flows on a link.
class TNetLane
{
  public:

    //  Type definitions.
    typedef BC_TUnboundedCollection<TNetPocket*, BC_CManaged> PocketCollection;
    typedef BC_TCollectionActiveIterator<TNetPocket*> PocketCollectionIterator;

    //  Construct a lane on the specified link, starting at the specified node,
    //  and identified by the specified number.
    TNetLane(TNetLink& link, NetNodeId node, NetLaneNumber number);

    //  Return the link on which the lane lies.
    TNetLink& GetLink();
    const TNetLink& GetLink() const;

    //  Return the starting node for the lane.
    TNetNode& GetStartNode();
    const TNetNode& GetStartNode() const;

    //  Return the end node for the lane.
    TNetNode& GetEndNode();
    const TNetNode& GetEndNode() const;

    //  Return the pockets on the lane.
    PocketCollection& GetPockets();
    const PocketCollection& GetPockets() const;

    //  Return the lane to the left, if any.
    const TNetLane* GetLeftAdjacentLane() const;
```

```
    //  Return the lane to the right, if any.
    const TNetLane* GetRightAdjacentLane() const;

    //  Return the number of the lane.
    NetLaneNumber GetNumber() const;

  private:

    //  Do not allow lanes to be copied.
    TNetLane(const TNetLane*) {}

    //  Do not allow lanes to be assigned.
    TNetLane& operator=(const TNetLane&) {return *this;}

    //  Each lane belongs on a link.
    TNetLink* fLink;

    //  Each lane has a starting node.
    NetNodeId fNode;

    //  Each lane has a number.
    NetLaneNumber fNumber;

    //  Each lane has zero or more pockets.
    PocketCollection fPockets;
};


#endif // TRANSIMS_NET_LANE
```

2. Lane.C

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: Lane.C,v $
//  $Revision: 2.1 $
//  $Date: 1996/11/08 18:13:14 $
//  $State: Stab $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


//  Include TRANSIMS header files.
#include <NET/Lane.h>
#include <NET/Node.h>
#include <NET/Link.h>
#include <NET/Pocket.h>


//  Construct a lane on the specified link, starting at the specified node, and
//  identified by the specified number.
TNetLane::TNetLane(TNetLink& link, NetNodeId node, NetLaneNumber number)
  : fLink(&link),
    fNode(node),
    fNumber(number)
{
    //ISSUE(bwb): We really should check here that the node and number are
    //valid.
}


//  Return the link on which the lane lies.
TNetLink& TNetLane::GetLink()
{
    return *fLink;
}

const TNetLink& TNetLane::GetLink() const
{
    return *fLink;
}
```

```
//  Return the starting node for the lane.
TNetNode& TNetLane::GetStartNode()
{
    TNetNode *nodeA, *nodeB;
    fLink->GetNodes(nodeA, nodeB);
    return fNode == nodeA->GetId() ? *nodeA : *nodeB;
}

const TNetNode& TNetLane::GetStartNode() const
{
    TNetNode *nodeA, *nodeB;
    fLink->GetNodes(nodeA, nodeB);
    return fNode == nodeA->GetId() ? *nodeA : *nodeB;
}


//  Return the end node for the lane.
TNetNode& TNetLane::GetEndNode()
{
    TNetNode *nodeA, *nodeB;
    fLink->GetNodes(nodeA, nodeB);
    return fNode == nodeA->GetId() ? *nodeB : *nodeA;
}

const TNetNode& TNetLane::GetEndNode() const
{
    TNetNode *nodeA, *nodeB;
    fLink->GetNodes(nodeA, nodeB);
    return fNode == nodeA->GetId() ? *nodeB : *nodeA;
}


//  Return the pockets on the lane.
TNetLane::PocketCollection& TNetLane::GetPockets()
{
    return fPockets;
}

const TNetLane::PocketCollection& TNetLane::GetPockets() const
{
    return fPockets;
}


//  Return the lane to the left, if any.
const TNetLane* TNetLane::GetLeftAdjacentLane() const
{
    for (TNetLink::LaneCollectionIterator i(fLink->GetLanesFrom(GetStartNode(),
            TRUE)); !i.IsDone(); i.Next())
        if ((*i.CurrentItem())->fNumber + 1 == fNumber)
            return *i.CurrentItem();
    return NULL;
}


//  Return the lane to the right, if any.
const TNetLane* TNetLane::GetRightAdjacentLane() const
{
    for (TNetLink::LaneCollectionIterator i(fLink->GetLanesFrom(GetStartNode(),
            TRUE)); !i.IsDone(); i.Next())
        if ((*i.CurrentItem())->fNumber - 1 == fNumber)
            return *i.CurrentItem();
    return NULL;
}


//  Return the number of the lane.
NetLaneNumber TNetLane::GetNumber() const
{
    return fNumber;
}
```

J. TNetLaneConnectivityReader Class

1. LaneConnectivityReader.h

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: LaneConnectivityReader.h,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 19:29:51 $
//  $State: Rel $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


#ifndef TRANSIMS_NET_LCONNREADER
#define TRANSIMS_NET_LCONNREADER


//  Include TRANSIMS header files.
#include <GBL/Globals.h>
#include <DBS/Accessor.h>
#include <NET/Id.h>


//  Forward declarations.
class TNetReader;


//  This reader reads lane connectivity values from the database.
class TNetLaneConnectivityReader
{
  public:

    //  Construct a lane connectivity reader for a given network.
    TNetLaneConnectivityReader(TNetReader& reader);

    //  Construct a copy of the given lane connectivity reader.
//  TNetLaneConnectivityReader(const TNetLaneConnectivityReader& reader);

    //  Make the reader a copy of the given lane connectivity reader.
//  TNetLaneConnectivityReader& operator=(const TNetLaneConnectivityReader&
//          reader);

    //  Reset the iteration over the table.
    void Reset();

    //  Get the next node in the table.
    void GetNextNode();

    //  Return whether there are any more nodes in the table.
    bool MoreNodes();

    //  Return the id for the current node.
    NetNodeId GetNode() const;

    //  Return the id for incoming link.
    NetLinkId GetInlink() const;

    //  Return the id for incoming lane.
    NetLaneNumber GetInlane() const;

    //  Return the id for outgoing link.
    NetLinkId GetOutlink() const;

    //  Return the id for outgoing lane.
    NetLaneNumber GetOutlane() const;

  private:

    //  Each lane connectivity reader has a database table accessor.
    TDbAccessor fAccessor;

    //  Each record has a node field.
```

```
    const TDbField fNodeField;

    //  Each record has an incoming link field.
    const TDbField fInlinkField;

    //  Each record has an incoming lane field.
    const TDbField fInlaneField;

    //  Each record has an outgoing link field.
    const TDbField fOutlinkField;

    //  Each record has an outgoing lane field.
    const TDbField fOutlaneField;
};


#endif // TRANSIMS_NET_LCONNREADER
```

2. LaneConnectivityReader.C

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: LaneConnectivityReader.C,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 19:29:51 $
//  $State: Rel $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


//  Include TRANSIMS header files.
#include <NET/LaneConnectivityReader.h>
#include <NET/Reader.h>


//  Construct a lane connectivity reader for a given network.
TNetLaneConnectivityReader::TNetLaneConnectivityReader(TNetReader& reader)
  : fAccessor(reader.GetLaneConnectivityTable()),
    fNodeField(reader.GetLaneConnectivityTable().GetField("NODE")),
    fInlinkField(reader.GetLaneConnectivityTable().GetField("INLINK")),
    fInlaneField(reader.GetLaneConnectivityTable().GetField("INLANE")),
    fOutlinkField(reader.GetLaneConnectivityTable().GetField("OUTLINK")),
    fOutlaneField(reader.GetLaneConnectivityTable().GetField("OUTLANE"))
{
}


//  Reset the iteration over the table.
void TNetLaneConnectivityReader::Reset()
{
    fAccessor.GotoFirst();
}


//  Get the next node in the table.
void TNetLaneConnectivityReader::GetNextNode()
{
    fAccessor.GotoNext();
}


//  Return whether there are any more nodes in the table.
bool TNetLaneConnectivityReader::MoreNodes()
{
    return fAccessor.IsAtRecord();
}


//  Return the id for the current node.
NetNodeId TNetLaneConnectivityReader::GetNode() const
{
    NetNodeId node;
    fAccessor.GetField(fNodeField, node);
    return node;
```

```
}


//  Return the id for incoming link.
NetLinkId TNetLaneConnectivityReader::GetInlink() const
{
    NetLinkId inlink;
    fAccessor.GetField(fInlinkField, inlink);
    return inlink;
}


//  Return the id for incoming lane.
NetLaneNumber TNetLaneConnectivityReader::GetInlane() const
{
    NetLaneNumber inlane;
    fAccessor.GetField(fInlaneField, inlane);
    return inlane;
}


//  Return the id for outgoing link.
NetLinkId TNetLaneConnectivityReader::GetOutlink() const
{
    NetLinkId outlink;
    fAccessor.GetField(fOutlinkField, outlink);
    return outlink;
}


//  Return the id for outgoing lane.
NetLaneNumber TNetLaneConnectivityReader::GetOutlane() const
{
    NetLaneNumber outlane;
    fAccessor.GetField(fOutlaneField, outlane);
    return outlane;
}
```

K. TNetLaneLocation Class

1. LaneLocation.h

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: LaneLocation.h,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 19:29:51 $
//  $State: Rel $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


#ifndef TRANSIMS_NET_LANELOCATION
#define TRANSIMS_NET_LANELOCATION


//  Include TRANSIMS header files.
#include <NET/Location.h>
#include <NET/Lane.h>


//  A location is a lane-specific point along a link.
class TNetLaneLocation
  : public TNetLocation
{
  public:

    //  Construct a location along the lane, with the given offset from its
    //  start.
    TNetLaneLocation(TNetLane& lane, REAL offset);

    //  Construct a copy of the given location.
//  TNetLaneLocation(const TNetLaneLocation& location);
```

```
    //  Make the location a copy of the given location.
//  TNetLaneLocation& operator=(const TNetLaneLocation& location);

    //  Return the lane for the location.
    TNetLane& GetLane();
    const TNetLane& GetLane() const;

    //  Return whether the location is lane-specific.
    bool IsOnSpecificLane() const;

  private:

    //  Each lane location is on a lane.
    TNetLane* fLane;
};


#endif // TRANSIMS_NET_LANELOCATION
```

2. LaneLocation.C

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: LaneLocation.C,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 19:29:51 $
//  $State: Rel $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


//  Include TRANSIMS header files.
#include <NET/LaneLocation.h>


//  Construct a location along the lane, with the given offset from its start.
TNetLaneLocation::TNetLaneLocation(TNetLane& lane, REAL offset)
  : TNetLocation(lane.GetLink(), lane.GetStartNode(), offset),
    fLane(&lane)
{
}


//  Return the lane for the location.
TNetLane& TNetLaneLocation::GetLane()
{
    return *fLane;
}

const TNetLane& TNetLaneLocation::GetLane() const
{
    return *fLane;
}


//  Return whether the location is lane-specific.
bool TNetLaneLocation::IsOnSpecificLane() const
{
    return TRUE;
}
```

L. TNetLink Class

1. Link.h

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: Link.h,v $
//  $Revision: 2.4 $
//  $Date: 1996/06/03 22:35:51 $
//  $State: Stab $
```

```
//  $Author: bwb $
//  U.S. Government Copyright, 1995
//  All rights reserved


#ifndef TRANSIMS_NET_LINK
#define TRANSIMS_NET_LINK


//  Include TRANSIMS header files.
#include <GBL/Globals.h>
#include <NET/Id.h>


//  Include Booch Components header files.
#include <BCStoreM.h>
#include <BCCollB.h>
#include <BCCollU.h>


//  Forward Declarations.
class TNetNode;
class TNetLane;
class TNetAccessory;
class TNetLinkReader;


//  A link is the part of the network corresponding to an "edge" in graph
//  theory.  Each link has a constant number of permanent lanes, but may have
//  turn pocket lanes also.  A link may have lanes in both directions, or the
//  lanes in opposite directions may be on separate links (in which case no
//  passing into oncoming lanes will be possible).
class TNetLink
{
  public:

    // There are several functions for a link.
    enum EFunctionalClass {kOther};

    // Type definitions.
    typedef BC_TUnboundedCollection<TNetAccessory*, BC_CManaged>
            AccessoryCollection;
    typedef BC_TCollectionActiveIterator<TNetAccessory*>
            AccessoryCollectionIterator;
    typedef BC_TUnboundedCollection<TNetLane*, BC_CManaged> LaneCollection;
    typedef BC_TCollectionActiveIterator<TNetLane*> LaneCollectionIterator;

    //  Construct a link using the reader.
    TNetLink(TNetLinkReader& reader);

    //  Construct a dummy link with the specified id.
    TNetLink(NetLinkId id);

    //  Destroy a link.
    virtual ~TNetLink();

    //  Return the id of the link.
    NetLinkId GetId() const;

    //  Return the nodes at the ends of the link.
    void GetNodes(NetNodeId& nodeA, NetNodeId& nodeB) const;
    void GetNodes(TNetNode*& nodeA, TNetNode*& nodeB) const;
    void GetNodes(const TNetNode*& nodeA, const TNetNode*& nodeB) const;

    //  Set the nodes at the ends of the link.
    void SetNodes(TNetNode* nodeA, TNetNode* nodeB);

    //  Return the accessories on the link.
    AccessoryCollection& GetAccessories();
    const AccessoryCollection& GetAccessories() const;

    //  Return the lanes going away from the specified node.  The lanes are
    //  ordered (but not necessarily numbered) from the divider outward.
    //  Pockets are included optionally.  A TNetNotFound exception is thrown if
    //  the node is not at one of the link's ends.
    LaneCollection& GetLanesFrom(const TNetNode& node, bool pockets = FALSE);
```

```
  const LaneCollection& GetLanesFrom(const TNetNode& node, bool pockets =
          FALSE) const;

  //  Return the lanes going toward the specified node.  The lanes are
  //  ordered (but not necessarily numbered) from the divider outward.
  //  Pockets are included optionally.  A TNetNotFound exception is thrown if
  //  the node is not at one of the link's ends.
  LaneCollection& GetLanesTowards(const TNetNode& node, bool pockets = FALSE);
  const LaneCollection& GetLanesTowards(const TNetNode& node, bool pockets =
          FALSE) const;

  //  Return the length of the link (using the default units).  The length is
  //  measured from one node to the other, unless setback distance subtraction
  //  is requested.
  REAL GetLength(bool setback = FALSE) const;

  //  Return the setback distance (using the default units) at the specified
  //  node.  A TNetNotFound exception is thrown if the node is not at one of
  //  the link's ends.
  REAL GetSetback(const TNetNode& node) const;

  //  Return the through link at the given node on the current link.  A
  //  TNetNotFound exception is thrown if the node is not at one of the link's
  //  ends.
  NetLinkId GetThroughLink(const TNetNode& node) const;

  //  Return the speed limit of the lanes heading toward the given node on
  //  the current link.  A TNetNotFound exception is thrown if the node is not
  //  at one of the link's ends.
  REAL GetSpeedLimitTowards(const TNetNode& node) const;

  //  Return the angle (in radians) of the link from the specified node.  A
  //  TNetNotFound exception is thrown if the node is not at one of the link's
  //  ends.
  REAL GetAngle(const TNetNode& node) const;

  //  Return the node between the current link and the given link.  A
  //  TNetNotFound exception is thrown if the links are not adjacent.
  TNetNode& GetNodeBetween(const TNetLink& link) const;

  //  Return whether the link has the same id as the given link.
  bool operator==(const TNetLink& link) const;

  //  Return whether the link has a different id from the given link
  bool operator!=(const TNetLink& link) const;

private:

  //  Do not allow links to be copied.
  TNetLink(const TNetLink&) {}

  //  Do not allow links to be assigned.
  TNetLink& operator=(const TNetLink&) {return *this;}

  //  Return the index of the given node.
  size_t GetNodeIndex(const TNetNode& node) const;

  //  Each link has a unique id.
  NetLinkId fId;

  //  Each link has nodes at its two ends.
  NetNodeId fNodeIds[2];
  TNetNode* fNodes[2];

  //  Each link has zero or more accessories.
  AccessoryCollection fAccessories;

  //  Each link has zero or more lanes on each side.
  LaneCollection fLanes[2];

  //  Each link has zero or more permanent lanes on each side.
  LaneCollection fPermanentLanes[2];

  //  Each link has a length.
  REAL fLength;
```

```
    //  Each link has setback distances from the intersection.
    REAL fSetbacks[2];

    //  Each link has through links at either end.
    NetLinkId fThroughLinks[2];

    //  Each link has speed limits in either direction.
    REAL fSpeedLimits[2];

    //  Each link has an angle (in radians) from its endpoints.
    REAL fAngles[2];
};


#endif // TRANSIMS_NET_LINK
```

2. Link.C

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: Link.C,v $
//  $Revision: 2.2 $
//  $Date: 1996/06/03 22:36:10 $
//  $State: Stab $
//  $Author: bwb $
//  U.S. Government Copyright, 1995
//  All rights reserved


//  Include TRANSIMS header files.
#include <NET/Node.h>
#include <NET/Link.h>
#include <NET/Lane.h>
#include <NET/Accessory.h>
#include <NET/Pocket.h>
#include <NET/LinkReader.h>
#include <NET/Exception.h>


//  Construct a link using the reader.
TNetLink::TNetLink(TNetLinkReader& reader)
  : fId(reader.GetId()),
    fLength(reader.GetLength())
{
    reader.GetNodeIds(fNodeIds[0], fNodeIds[1]);
    for (size_t side = 0; side < 2; ++side) {
        const NetNodeId id = fNodeIds[side];
        fNodes[side] = NULL;
        const nLeft = reader.GetLeftPocketLaneCountTowards(id);
        const nPerm = reader.GetPermanentLaneCountTowards(id);
        const nRight = reader.GetRightPocketLaneCountTowards(id);
        for (size_t iPerm = 0; iPerm < nPerm; ++iPerm)
            fLanes[side].Append(new TNetLane(*this, fNodeIds[1 - side], nLeft +
                    iPerm + 1));
        fPermanentLanes[side] = fLanes[side];
        for (size_t iLeft = 0; iLeft < nLeft; ++iLeft)
            fLanes[side].Insert(new TNetLane(*this, fNodeIds[1 - side], iLeft +
                    1), iLeft);
        for (size_t iRight = 0; iRight < nRight; ++iRight)
            fLanes[side].Append(new TNetLane(*this, fNodeIds[1 - side], nLeft +
                    nPerm + iRight + 1));
        fSetbacks[side] = reader.GetSetbackDistance(id);
        fThroughLinks[side] = reader.GetThroughLink(id);
        fSpeedLimits[side] = reader.GetSpeedLimitTowards(id);
    }
}


//  Construct a dummy link with the specified id.
TNetLink::TNetLink(NetLinkId id)
  : fId(id)
{
}
```

```
//  Destroy a link.
TNetLink::~TNetLink()
{
    for (AccessoryCollectionIterator i(fAccessories); !i.IsDone(); i.Next())
        delete *i.CurrentItem();
    for (size_t side = 0; side < 2; ++side)
        for (LaneCollectionIterator j(fLanes[side]); !j.IsDone(); j.Next()) {
            TNetLane* const lane = *j.CurrentItem();
            for (TNetLane::PocketCollectionIterator k(lane->GetPockets());
                    !k.IsDone(); k.Next())
                delete *k.CurrentItem();
            delete lane;
        }
}


//  Return the id of the link.
NetLinkId TNetLink::GetId() const
{
    return fId;
}


//  Return the nodes at the ends of the link.
void TNetLink::GetNodes(NetNodeId& nodeA, NetNodeId& nodeB) const
{
    nodeA = fNodeIds[0];
    nodeB = fNodeIds[1];
}

void TNetLink::GetNodes(TNetNode*& nodeA, TNetNode*& nodeB) const
{
    nodeA = fNodes[0];
    nodeB = fNodes[1];
}

void TNetLink::GetNodes(const TNetNode*& nodeA, const TNetNode*& nodeB) const
{
    nodeA = fNodes[0];
    nodeB = fNodes[1];
}


//  Set the nodes at the ends of the link.
void TNetLink::SetNodes(TNetNode* nodeA, TNetNode* nodeB)
{
    fNodes[0] = nodeA;
    fNodes[1] = nodeB;
    for (size_t side = 0; side < 2; ++side) {
        const TGeoPoint& head = fNodes[side]->GetGeographicPosition();
        const TGeoPoint& tail = fNodes[1 - side]->GetGeographicPosition();
        fAngles[side] = head.GetAngleTo(tail);
    }
}


//  Return the accessories on the link.
TNetLink::AccessoryCollection& TNetLink::GetAccessories()
{
    return fAccessories;
}

const TNetLink::AccessoryCollection& TNetLink::GetAccessories() const
{
    return fAccessories;
}


//  Return the lanes going away from the specified node.  The lanes are ordered
//  (but not necessarily numbered) from the divider outward.  Pockets are
//  included optionally.  A TNetNotFound exception is thrown if the node is
//  not at one of the link's ends.
TNetLink::LaneCollection& TNetLink::GetLanesFrom(const TNetNode& node, bool
        pockets)
{
    const size_t i = 1 - GetNodeIndex(node);
```

```
    return pockets ? fLanes[i] : fPermanentLanes[i];
}

const TNetLink::LaneCollection& TNetLink::GetLanesFrom(const TNetNode& node,
        bool pockets) const
{
    const size_t i = 1 - GetNodeIndex(node);
    return pockets ? fLanes[i] : fPermanentLanes[i];
}


//  Return the lanes going toward the specified node.  The lanes are ordered
//  (but not necessarily numbered) from the divider outward.  Pockets are
//  included optionally.  A TNetNotFound exception is thrown if the node is not
//  at one of the link's ends.
TNetLink::LaneCollection& TNetLink::GetLanesTowards(const TNetNode& node, bool
        pockets)
{
    const size_t i = GetNodeIndex(node);
    return pockets ? fLanes[i] : fPermanentLanes[i];
}

const TNetLink::LaneCollection& TNetLink::GetLanesTowards(const TNetNode& node,
        bool pockets) const
{
    const size_t i = GetNodeIndex(node);
    return pockets ? fLanes[i] : fPermanentLanes[i];
}


//  Return the length of the link (using the default units).  The length is
//  measured from one node to the other, unless setback distance subtraction is
//  requested.
REAL TNetLink::GetLength(bool setback) const
{
    if (setback)
        return fLength - fSetbacks[0] - fSetbacks[1];
    else
        return fLength;
}


//  Return the setback distance (usings the default units) at the specified
//  node.  A TNetNotFound exception is thrown if the node is not at one of
//  the link's ends.
REAL TNetLink::GetSetback(const TNetNode& node) const
{
    const size_t i = GetNodeIndex(node);
    return fSetbacks[i];
}


//  Return the through link at the given node on the current link.  A
//  TNetNotFound exception is thrown if the node is not at one of the link's
//  ends.
NetLinkId TNetLink::GetThroughLink(const TNetNode& node) const
{
    const size_t i = GetNodeIndex(node);
    return fThroughLinks[i];
}

//  Return the speed limit of the lanes heading toward the given node on
//  the current link.  A TNetNotFound exception is thrown if the node is not
//  at one of the link's ends.
REAL TNetLink::GetSpeedLimitTowards(const TNetNode& node) const
{
    const size_t i = GetNodeIndex(node);
    return fSpeedLimits[i];
}

//  Return the angle (in radians) of the link from the specified node.  A
//  TNetNotFound exception is thrown if the node is not at one of the link's
//  ends.
REAL TNetLink::GetAngle(const TNetNode& node) const
{
    const size_t i = GetNodeIndex(node);
```

```
    return fAngles[i];
}


//  Return the node between the current link and the given link.  A
//  TNetNotFound exception is thrown if the links are not adjacent.
TNetNode& TNetLink::GetNodeBetween(const TNetLink& link) const
{
    TNetNode* nodeA;
    TNetNode* nodeB;
    link.GetNodes(nodeA, nodeB);
    if (fNodes[0] == nodeA || fNodes[0] == nodeB)
        return *fNodes[0];
    else if (fNodes[1] == nodeA || fNodes[1] == nodeB)
        return *fNodes[1];
    else
    {
        throw TNetNotFound ("The links are not adjacent.");
        return *fNodes[0]; //ISSUE(kpb): This statement never reached.
    }
}


//  Return whether the link has the same id as the given link.
bool TNetLink::operator==(const TNetLink& link) const
{
    return GetId() == link.GetId();
}


//  Return whether the link has a different id from the given link.
bool TNetLink::operator!=(const TNetLink& link) const
{
    return GetId() != link.GetId();
}


//  Return the index of the given node. A TNetNotFound exception is thrown if
//  the node is not at one of the link's ends.
size_t TNetLink::GetNodeIndex(const TNetNode& node) const
{
    if (&node == fNodes[0])
        return 0;
    else if (&node == fNodes[1])
        return 1;
    else
        throw TNetNotFound("The node is not on the link.");
}
```

M. TNetLinkReader Class

1. LinkReader.h

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: LinkReader.h,v $
//  $Revision: 2.2 $
//  $Date: 1996/05/02 19:50:38 $
//  $State: Rel $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


#ifndef TRANSIMS_NET_LINKREADER
#define TRANSIMS_NET_LINKREADER


//  Include TRANSIMS header files.
#include <GBL/Globals.h>
#include <DBS/Accessor.h>
#include <NET/Id.h>
#include <NET/Link.h>
#include <NET/Point.h>
```

```
//  Forward declarations.
class TNetReader;


//  A link reader reads link values from the database.
class TNetLinkReader
{
  public:

    //  Construct a link reader for a given network.
    TNetLinkReader(TNetReader& reader);

    //  Construct a copy of the given link reader.
//  TNetLinkReader(const TNetLinkReader& reader);

    //  Make the reader a copy of the given link reader.
//  TNetLinkReader& operator=(const TNetLinkReader& reader);

    //  Reset the iteration over the table.
    void Reset();

    //  Get the next link in the table.
    void GetNextLink();

    //  Return whether there are any more links in the table.
    bool MoreLinks() const;

    //  Return the id for the current link.
    NetLinkId GetId() const;

    //  Return the node ids at the ends of the current link.
    void GetNodeIds(NetNodeId& nodeA, NetNodeId& nodeB) const;

    //  Return the number of permanent lanes heading toward the given node on
    //  the current link.  A TNetNotFound exception is thrown if the node is not
    //  at one of the link's ends.
    BYTE GetPermanentLaneCountTowards(NetNodeId id) const;

    //  Return the number of left pocket lanes heading toward the given node on
    //  the current link.  A TNetNotFound exception is thrown if the node is not
    //  at one of the link's ends.
    BYTE GetLeftPocketLaneCountTowards(NetNodeId id) const;

    //  Return the number of right pocket lanes heading toward the given node
    //  on the current link.  A TNetNotFound exception is thrown if the node is
    //  not at one of the link's ends.
    BYTE GetRightPocketLaneCountTowards(NetNodeId id) const;

    //  Return whether there is a two-way left turn lane on the current link.
    bool HasTwoWayLeftTurnLane() const;

    //  Return the length of the current link.
    REAL GetLength() const;

    //  Return the percent grade of the lanes heading toward the given node on
    //  the current link.  A TNetNotFound exception is thrown if the node is
    //  not at one of the link's ends.
    REAL GetGradeTowards(NetNodeId id) const;

    //  Return the setback distances at the given node on the current link.  A
    //  TNetNotFound exception is thrown if the node is not at one of the
    //  link's ends.
    REAL GetSetbackDistance(NetNodeId id) const;

    //  Return the through link at the given node on the current link.  A
    //  TNetNotFound exception is thrown if the node is not at one of the link's
    //  ends.
    NetLinkId GetThroughLink(NetNodeId id) const;

    //  Return the capacity in vehicles-per-hour of the lanes heading toward
    //  the given node on the current link.  A TNetNotFound exception is
    //  thrown if the node is not at one of the link's ends.
    REAL GetCapacityTowards(NetNodeId id) const;
```

```
  //  Return the speed limit of the lanes heading toward the given node on
  //  the current link.  A TNetNotFound exception is thrown if the node is not
  //  at one of the link's ends.
  REAL GetSpeedLimitTowards(NetNodeId id) const;

  //  Return the free-flow speed of the lanes heading toward the given node
  //  on the current link.  A TNetNotFound exception is thrown if the node is
  //  not at one of the link's ends.
  REAL GetFreeFlowSpeedTowards(NetNodeId id) const;

  //  Return the crawl speed of the lanes heading toward the given node on
  //  the current link.  A TNetNotFound exception is thrown if the node is not
  //  at one of the link's ends.
  REAL GetCrawlSpeedTowards(NetNodeId id) const;

  //  Return the functional class for the link.
  TNetLink::EFunctionalClass GetFunctionalClass() const;

  //  Return the cost (in arbitrary units) for traveling on the link toward
  //  the given node.  A TNetNotFound exception is thrown if the node is not
  //  at one of the link's ends.
  UINT GetCostTowards(NetNodeId id) const;

private:

  //  Update the current nodes for the link.  A TNetNotFound exception is
  //  thrown if the node is not at one of the link's ends.
  void UpdateCurrentNodes();

  //  Each link reader has a database table accessor.
  TDbAccessor fAccessor;

  //  Each link has a current A node.
  NetNodeId fNodeA;

  //  Each link has a current B node.
  NetNodeId fNodeB;

  //  Each link has an ID field.
  const TDbField fIdField;

  //  Each link has a NODEA field.
  const TDbField fNodeAField;

  //  Each link has a NODEB field.
  const TDbField fNodeBField;

  //  Each link has a PERMLANESA field.
  const TDbField fPermanentLanesAField;

  //  Each link has a PERMLANESB field.
  const TDbField fPermanentLanesBField;

  //  Each link has a LEFTPCKTSA field.
  const TDbField fLeftPocketsAField;

  //  Each link has a LEFTPCKTSB field.
  const TDbField fLeftPocketsBField;

  //  Each link has a RGHTPCKTSA field.
  const TDbField fRightPocketsAField;

  //  Each link has a RGHTPCKTSB field.
  const TDbField fRightPocketsBField;

  //  Each link has a TWOWAYTURN field.
  const TDbField fTwoWayTurnField;

  //  Each link has a LENGTH field.
  const TDbField fLengthField;

  //  Each link has a GRADE field.
  const TDbField fGradeField;

  //  Each link has a SETBACKA field.
  const TDbField fSetbackAField;
```

```
    //  Each link has a SETBACKB field.
    const TDbField fSetbackBField;

    //  Each link has a THROUGHA field.
    const TDbField fThroughAField;

    //  Each link has a THROUGHB field.
    const TDbField fThroughBField;

    //  Each link has a CAPACITYA field.
    const TDbField fCapacityAField;

    //  Each link has a CAPACITYB field.
    const TDbField fCapacityBField;

    //  Each link has a SPEEDLMTA field.
    const TDbField fSpeedLimitAField;

    //  Each link has a SPEEDLMTB field.
    const TDbField fSpeedLimitBField;

    //  Each link has a FREESPDA field.
    const TDbField fFreeSpeedAField;

    //  Each link has a FREESPDB field.
    const TDbField fFreeSpeedBField;

    //  Each link has a CRAWLSPDA field.
    const TDbField fCrawlSpeedAField;

    //  Each link has a CRAWLSPDB field.
    const TDbField fCrawlSpeedBField;

    //  Each link has a FUNCTCLASS field.
    const TDbField fFunctionalClassField;

    //  Each link has a COSTA field.
    const TDbField fCostAField;

    //  Each link has a COSTB field.
    const TDbField fCostBField;
};


#endif // TRANSIMS_NET_LINKREADER
```

2. LinkReader.C

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: LinkReader.C,v $
//  $Revision: 2.2 $
//  $Date: 1996/05/02 19:50:38 $
//  $State: Rel $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


//  Include TRANSIMS header files.
#include <NET/LinkReader.h>
#include <NET/Reader.h>
#include <NET/Exception.h>


//  Construct a link reader for a given network.
TNetLinkReader::TNetLinkReader(TNetReader& reader)
  : fAccessor(reader.GetLinkTable()),
    fNodeA(0),
    fNodeB(0),
    fIdField(reader.GetLinkTable().GetField("ID")),
    fNodeAField(reader.GetLinkTable().GetField("NODEA")),
    fNodeBField(reader.GetLinkTable().GetField("NODEB")),
    fPermanentLanesAField(reader.GetLinkTable().GetField("PERMLANESA")),
```

```
    fPermanentLanesBField(reader.GetLinkTable().GetField("PERMLANESB")),
    fLeftPocketsAField(reader.GetLinkTable().GetField("LEFTPCKTSA")),
    fLeftPocketsBField(reader.GetLinkTable().GetField("LEFTPCKTSB")),
    fRightPocketsAField(reader.GetLinkTable().GetField("RGHTPCKTSA")),
    fRightPocketsBField(reader.GetLinkTable().GetField("RGHTPCKTSB")),
    fTwoWayTurnField(reader.GetLinkTable().GetField("TWOWAYTURN")),
    fLengthField(reader.GetLinkTable().GetField("LENGTH")),
    fGradeField(reader.GetLinkTable().GetField("GRADE")),
    fSetbackAField(reader.GetLinkTable().GetField("SETBACKA")),
    fSetbackBField(reader.GetLinkTable().GetField("SETBACKB")),
    fThroughAField(reader.GetLinkTable().GetField("THRUA")),
    fThroughBField(reader.GetLinkTable().GetField("THRUB")),
    fCapacityAField(reader.GetLinkTable().GetField("CAPACITYA")),
    fCapacityBField(reader.GetLinkTable().GetField("CAPACITYB")),
    fSpeedLimitAField(reader.GetLinkTable().GetField("SPEEDLMTA")),
    fSpeedLimitBField(reader.GetLinkTable().GetField("SPEEDLMTB")),
    fFreeSpeedAField(reader.GetLinkTable().GetField("FREESPDA")),
    fFreeSpeedBField(reader.GetLinkTable().GetField("FREESPDB")),
    fCrawlSpeedAField(reader.GetLinkTable().GetField("CRAWLSPDA")),
    fCrawlSpeedBField(reader.GetLinkTable().GetField("CRAWLSPDB")),
    fFunctionalClassField(reader.GetLinkTable().GetField("FUNCTCLASS")),
    fCostAField(reader.GetLinkTable().GetField("COSTA")),
    fCostBField(reader.GetLinkTable().GetField("COSTB"))
{
}


//  Reset the iteration over the table.
void TNetLinkReader::Reset()
{
    fAccessor.GotoFirst();
    UpdateCurrentNodes();
}


//  Get the next link in the table.
void TNetLinkReader::GetNextLink()
{
    fAccessor.GotoNext();
    UpdateCurrentNodes();
}


//  Return whether there are any more links in the table.
bool TNetLinkReader::MoreLinks() const
{
    return fAccessor.IsAtRecord();
}


//  Return the id for the current link.
NetLinkId TNetLinkReader::GetId() const
{
    NetLinkId id;
    fAccessor.GetField(fIdField, id);
    return id;
}


//  Return the node ids at the ends of the current link.
void TNetLinkReader::GetNodeIds(NetNodeId& nodeA, NetNodeId& nodeB) const
{
    nodeA = fNodeA;
    nodeB = fNodeB;
}


//  Return the number of permanent lanes heading toward the given node on the
//  current link.  A TNetNotFound exception is thrown if the node is not at one
//  of the link's ends.
BYTE TNetLinkReader::GetPermanentLaneCountTowards(NetNodeId id) const
{
    BYTE count;
    if (id == fNodeA)
        fAccessor.GetField(fPermanentLanesAField, count);
    else if (id == fNodeB)
```

```
        fAccessor.GetField(fPermanentLanesBField, count);
    else
        throw TNetNotFound("Invalid node id.");
    return count;
}


//  Return the number of left pocket lanes heading toward the given node on the
//  current link.  A TNetNotFound exception is thrown if the node is not at one
//  of the link's ends.
BYTE TNetLinkReader::GetLeftPocketLaneCountTowards(NetNodeId id) const
{
    BYTE count;
    if (id == fNodeA)
        fAccessor.GetField(fLeftPocketsAField, count);
    else if (id == fNodeB)
        fAccessor.GetField(fLeftPocketsBField, count);
    else
        throw TNetNotFound("Invalid node id.");
    return count;
}


//  Return the number of right pocket lanes heading toward the given node on
//  the current link.  A TNetNotFound exception is thrown if the node is not at
//  one of the link's ends.
BYTE TNetLinkReader::GetRightPocketLaneCountTowards(NetNodeId id) const
{
    BYTE count;
    if (id == fNodeA)
        fAccessor.GetField(fRightPocketsAField, count);
    else if (id == fNodeB)
        fAccessor.GetField(fRightPocketsBField, count);
    else
        throw TNetNotFound("Invalid node id.");
    return count;
}


//  Return whether there is a two-way left turn lane on the current link.
bool TNetLinkReader::HasTwoWayLeftTurnLane() const
{
    string flag;
    fAccessor.GetField(fTwoWayTurnField, flag);
    return flag == "Y";
}


//  Return the length of the current link.
REAL TNetLinkReader::GetLength() const
{
    REAL length;
    fAccessor.GetField(fLengthField, length);
    return length;
}


//  Return the percent grade of the lanes heading toward the given node on the
//  current link.  A TNetNotFound exception is thrown if the node is not at one
//  of the link's ends.
REAL TNetLinkReader::GetGradeTowards(NetNodeId id) const
{
    REAL grade;
    fAccessor.GetField(fGradeField, grade);
    if (id == fNodeA)
        return - grade;
    else if (id == fNodeB)
        return grade;
    else
        throw TNetNotFound("Invalid node id.");
}


//  Return the setback distances at the given node on the current link.  A
//  TNetNotFound exception is thrown if the node is not at one of the link's
//  ends.
```

```
REAL TNetLinkReader::GetSetbackDistance(NetNodeId id) const
{
    REAL distance;
    if (id == fNodeA)
        fAccessor.GetField(fSetbackAField, distance);
    else if (id == fNodeB)
        fAccessor.GetField(fSetbackBField, distance);
    else
        throw TNetNotFound("Invalid node id.");
    return distance;
}


//  Return the through link at the given node on the current link.  A
//  TNetNotFound exception is thrown if the node is not at one of the link's
//  ends.
NetLinkId TNetLinkReader::GetThroughLink(NetNodeId id) const
{
    NetLinkId link;
    if (id == fNodeA)
        fAccessor.GetField(fThroughAField, link);
    else if (id == fNodeB)
        fAccessor.GetField(fThroughBField, link);
    else
        throw TNetNotFound("Invalid node id.");
    return link;
}


//  Return the capacity in vehicles-per-hour of the lanes heading toward the
//  given node on the current link.  A TNetNotFound exception is thrown if the
//  node is not at one of the link's ends.
REAL TNetLinkReader::GetCapacityTowards(NetNodeId id) const
{
    REAL capacity;
    if (id == fNodeA)
        fAccessor.GetField(fCapacityAField, capacity);
    else if (id == fNodeB)
        fAccessor.GetField(fCapacityBField, capacity);
    else
        throw TNetNotFound("Invalid node id.");
    return capacity;
}


//  Return the speed limit of the lanes heading toward the given node on the
//  current link.  A TNetNotFound exception is thrown if the node is not at one
//  of the link's ends.
REAL TNetLinkReader::GetSpeedLimitTowards(NetNodeId id) const
{
    REAL limit;
    if (id == fNodeA)
        fAccessor.GetField(fSpeedLimitAField, limit);
    else if (id == fNodeB)
        fAccessor.GetField(fSpeedLimitBField, limit);
    else
        throw TNetNotFound("Invalid node id.");
    return limit;
}


//  Return the free-flow speed of the lanes heading toward the given node on
//  the current link.  A TNetNotFound exception is thrown if the node is not at
//  one of the link's ends.
REAL TNetLinkReader::GetFreeFlowSpeedTowards(NetNodeId id) const
{
    REAL speed;
    if (id == fNodeA)
        fAccessor.GetField(fFreeSpeedAField, speed);
    else if (id == fNodeB)
        fAccessor.GetField(fFreeSpeedBField, speed);
    else
        throw TNetNotFound("Invalid node id.");
    return speed;
}
```

```
//  Return the crawl speed of the lanes heading toward the given node on the
//  current link.  A TNetNotFound exception is thrown if the node is not at one
//  of the link's ends.
REAL TNetLinkReader::GetCrawlSpeedTowards(NetNodeId id) const
{
    REAL speed;
    if (id == fNodeA)
        fAccessor.GetField(fCrawlSpeedAField, speed);
    else if (id == fNodeB)
        fAccessor.GetField(fCrawlSpeedBField, speed);
    else
        throw TNetNotFound("Invalid node id.");
    return speed;
}


//  Update the current nodes for the link.
void TNetLinkReader::UpdateCurrentNodes()
{
    if (fAccessor.IsAtRecord()) {
        fAccessor.GetField(fNodeAField, fNodeA);
        fAccessor.GetField(fNodeBField, fNodeB);
    }
}


//  Return the functional class for the link.
TNetLink::EFunctionalClass TNetLinkReader::GetFunctionalClass() const
{
    string functionalClass;
    fAccessor.GetField(fFunctionalClassField, functionalClass);
    return TNetLink::kOther;
}


//  Return the cost (in arbitrary units) for traveling on the link toward
//  the given node.  A TNetNotFound exception is thrown if the node is not
//  at one of the link's ends.
UINT TNetLinkReader::GetCostTowards(NetNodeId id) const
{
    UINT cost;
    if (id == fNodeA)
        fAccessor.GetField(fCostAField, cost);
    else if (id == fNodeB)
        fAccessor.GetField(fCostBField, cost);
    else
        throw TNetNotFound("Invalid node id.");
    return cost;
}
```

N. TNetLocation Class

1. Location.h

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: Location.h,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 19:29:51 $
//  $State: Rel $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


#ifndef TRANSIMS_NET_LOCATION
#define TRANSIMS_NET_LOCATION


//  Include TRANSIMS header files.
#include <GBL/Globals.h>
#include <NET/Node.h>
#include <NET/Link.h>
```

```
#include <NET/Point.h>


//  A location is a point along a link.
class TNetLocation
{
  public:

    //  Construct a location along the given node and link, with the given
    //  offset from the node.  The exception TNetNotFound is thrown if the node
    //  is not at one of the link's ends.
    TNetLocation(TNetLink& link, TNetNode& node, REAL offset);

    //  Construct a copy of the given location.
//  TNetLocation(const TNetLocation& location);

    //  Make the location a copy of the given location.
//  TNetLocation& operator=(const TNetLocation& location);

    //  Return the link on which the location lies.
    TNetLink& GetLink();
    const TNetLink& GetLink() const;

    //  Return the distance from the given endpoint of the link.  The exception
    //  TNetNotFound is thrown if the node is not at one of the link's ends.
    REAL GetOffsetFrom(const TNetNode& node) const;

    //  Return the geographic position of the location.
    TGeoPoint GetGeographicPosition() const;

    //  Return whether the location is lane-specific.
    virtual bool IsOnSpecificLane() const;

  private:

    //  Each location is on a link.
    TNetLink* fLink;

    //  Each location is measured from a node.
    TNetNode* fNode;

    //  Each location is specified as an offset relative to the start of its
    //  link.
    REAL fOffset;
};


#endif // TRANSIMS_NET_LOCATION
```

2. Location.C

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: Location.C,v $
//  $Revision: 2.1 $
//  $Date: 1995/09/08 12:21:54 $
//  $State: Stab $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


//  Include TRANSIMS header files.
#include <NET/Exception.h>
#include <NET/Location.h>


//  Construct a location along the given node and link, with the given offset
//  from the node.  The exception TNetNotFound is thrown if the node is not at
//  one of the link's ends.
TNetLocation::TNetLocation(TNetLink& link, TNetNode& node, REAL offset)
  : fLink(&link)
{
    TNetNode *nodeA, *nodeB;
    fLink->GetNodes(nodeA, nodeB);
```

```
    fNode = nodeA;
    if (node == *nodeA)
        fOffset = offset;
    else if (node == *nodeB)
        fOffset = fLink->GetLength() - offset;
    else
        throw TNetNotFound("Incorrect node.");
}


//  Return the link on which the location lies.
TNetLink& TNetLocation::GetLink()
{
    return *fLink;
}

const TNetLink& TNetLocation::GetLink() const
{
    return *fLink;
}


//  Return the distance from the given endpoint of the link.  The exception
//  TNetNotFound is thrown if the node is not at one of the link's ends.
REAL TNetLocation::GetOffsetFrom(const TNetNode& node) const
{
    TNetNode *nodeA, *nodeB;
    fLink->GetNodes(nodeA, nodeB);
    if (node == *nodeA)
        return fOffset;
    else if (node == *nodeB)
        return fLink->GetLength() - fOffset;
    else
        throw TNetNotFound("Incorrect node.");
}


//  Return the geographic position of the location.
TGeoPoint TNetLocation::GetGeographicPosition() const
{
    TNetNode *nodeA, *nodeB;
    fLink->GetNodes(nodeA, nodeB);
    const TGeoPoint& pointA = nodeA->GetGeographicPosition();
    const TGeoPoint& pointB = nodeB->GetGeographicPosition();
    if (fLink->GetLength() == 0)
        return pointA;
    const REAL fractionA = 1 - fOffset / fLink->GetLength();
    const REAL fractionB = 1 - fractionA;
    return TGeoPoint(pointA.GetX() * fractionA + pointB.GetX() * fractionB,
        pointA.GetY() * fractionA + pointB.GetY() * fractionB);
}


//  Return whether the location is lane-specific.
bool TNetLocation::IsOnSpecificLane() const
{
    return FALSE;
}
```

O. TNetNetwork Class

1. Network.h

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: Network.h,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 19:29:51 $
//  $State: Rel $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved
```

```
#ifndef TRANSIMS_NET_NETWORK
#define TRANSIMS_NET_NETWORK


//  Include TRANSIMS header files.
#include <GBL/Globals.h>
#include <NET/Id.h>


//  Include Booch Components header files.
#include <BCStoreM.h>
#include <BCMapU.h>


//  Forward declarations.
class TNetNode;
class TNetLink;
class TNetParking;
class TNetSignalCoordinator;


//  A network represents all of the network database that has been instantiated.
class TNetNetwork
{
  public:

    //  Type definitions.
    typedef BC_TUnboundedMap<NetNodeId, TNetNode*, 10000U, BC_CManaged> NodeMap;
    typedef BC_TMapActiveIterator<NetNodeId, TNetNode*> NodeMapIterator;
    typedef BC_TUnboundedMap<NetLinkId, TNetLink*, 10000U, BC_CManaged> LinkMap;
    typedef BC_TMapActiveIterator<NetLinkId, TNetLink*> LinkMapIterator;
    typedef BC_TUnboundedMap<NetAccessoryId, TNetParking*, 10000U, BC_CManaged>
            ParkingMap;
    typedef BC_TMapActiveIterator<NetAccessoryId, TNetParking*>
            ParkingMapIterator;
    typedef BC_TUnboundedMap<NetCoordinatorId, TNetSignalCoordinator*, 2000U,
            BC_CManaged> SignalCoordinatorMap;
    typedef BC_TMapActiveIterator<NetCoordinatorId, TNetSignalCoordinator*>
            SignalCoordinatorMapIterator;

    //  Construct a network.
    TNetNetwork();

    //  Destroy a network.
    ~TNetNetwork();

    //  Return the set of nodes in the network.
    NodeMap& GetNodes();
    const NodeMap& GetNodes() const;

    //  Return the set of links in the network.
    LinkMap& GetLinks();
    const LinkMap& GetLinks() const;

    //  Return the set of parking places in the network.
    ParkingMap& GetParkings();
    const ParkingMap& GetParkings() const;

    //  Return the set of signal coordinators in the network.
    SignalCoordinatorMap& GetSignalCoordinators();
    const SignalCoordinatorMap& GetSignalCoordinators() const;

  private:

    //  Do not allow networks to be copied.
    TNetNetwork(const TNetNetwork&) {}

    //  Do not allow networks to be assigned.
    TNetNetwork& operator=(const TNetNetwork&) {return *this;}

    //  Each network manages/owns the nodes it contains.
    NodeMap fNodes;

    //  Each network manages/owns the links it contains.
    LinkMap fLinks;
```

```
    //  Each network has a map for parking places.
    ParkingMap fParkings;

    //  Each network manages/owns the signal coordinators it contains.
    SignalCoordinatorMap fCoordinators;
};


#endif // TRANSIMS_NET_NETWORK
```

2. Network.C

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: Network.C,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 19:29:51 $
//  $State: Rel $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


//  Include TRANSIMS header files.
#include <NET/Network.h>
#include <NET/Node.h>
#include <NET/Link.h>
#include <NET/Parking.h>
#include <NET/SignalCoordinator.h>


//  Define the hash function for nodes.
static BC_Index NodeHashValue(const NetNodeId& id)
{
    return BC_Index(id);
}


//  Define the hash function for links.
static BC_Index LinkHashValue(const NetLinkId& id)
{
    return BC_Index(id);
}


//  Define the hash function for parking places.
static BC_Index ParkingHashValue(const NetAccessoryId& id)
{
    return BC_Index(id);
}


//  Define the hash function for signal coordinators.
static BC_Index CoordinatorHashValue(const NetCoordinatorId& id)
{
    return BC_Index(id);
}


//  Construct a network using the reader.
TNetNetwork::TNetNetwork()
  : fNodes(NodeHashValue),
    fLinks(LinkHashValue),
    fParkings(ParkingHashValue),
    fCoordinators(CoordinatorHashValue)
{
}


//  Destroy a network.
TNetNetwork::~TNetNetwork()
{
    for (LinkMapIterator i(fLinks); !i.IsDone(); i.Next())
        delete *i.CurrentValue();
    for (NodeMapIterator j(fNodes); !j.IsDone(); j.Next())
```

```
        delete *j.CurrentValue();
    for (SignalCoordinatorMapIterator k(fCoordinators); !k.IsDone(); k.Next())
        delete *k.CurrentValue();
}


//  Return the set of nodes in the network.
TNetNetwork::NodeMap& TNetNetwork::GetNodes()
{
    return fNodes;
}

const TNetNetwork::NodeMap& TNetNetwork::GetNodes() const
{
    return fNodes;
}


//  Return the set of links in the network.
TNetNetwork::LinkMap& TNetNetwork::GetLinks()
{
    return fLinks;
}

const TNetNetwork::LinkMap& TNetNetwork::GetLinks() const
{
    return fLinks;
}


//  Return the set of parkings in the network.
TNetNetwork::ParkingMap& TNetNetwork::GetParkings()
{
    return fParkings;
}

const TNetNetwork::ParkingMap& TNetNetwork::GetParkings() const
{
    return fParkings;
}


//  Return the set of signal coordinators in the network.
TNetNetwork::SignalCoordinatorMap& TNetNetwork::GetSignalCoordinators()
{
    return fCoordinators;
}

const TNetNetwork::SignalCoordinatorMap& TNetNetwork::GetSignalCoordinators()
        const
{
    return fCoordinators;
}
```

P. TNetNode Class

1. Node.h

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: Node.h,v $
//  $Revision: 2.1 $
//  $Date: 1995/08/09 14:32:04 $
//  $State: Exp $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


#ifndef TRANSIMS_NET_NODE
#define TRANSIMS_NET_NODE


//  Include TRANSIMS header files.
```

```
#include <GBL/Globals.h>
#include <NET/Id.h>
#include <NET/Point.h>


//  Include Booch Components header files.
#include <BCStoreM.h>
#include <BCRingU.h>


//  Forward declarations.
class TNetNodeReader;
class TNetLink;
class TNetTrafficControl;


//  A node is the part of the network corresponding to a "vertex" in graph
//  theory.  A node must be present where the network branches and where the
//  permanent number of lanes changes.  (A node may be present where neither of
//  the aforementioned occurs, however.)
class TNetNode
{
  public:

    //  Type definitions.
    typedef BC_TUnboundedRing<TNetLink*, BC_CManaged> LinkRing;
    typedef BC_TRingActiveIterator<TNetLink*> LinkRingIterator;

    //  Construct a node using the reader.
    TNetNode(TNetNodeReader& reader);

    //  Construct a dummy node with the specified id.
    TNetNode(NetNodeId id);

    //  Destroy a node.
    virtual ~TNetNode();

    //  Return the id of the node.
    NetNodeId GetId() const;

    //  Return the geographic position of the node.
    TGeoPoint& GetGeographicPosition();
    const TGeoPoint& GetGeographicPosition() const;

    //  Return the ring of links in order.
    LinkRing& GetLinks();
    const LinkRing& GetLinks() const;

    //  Add the specified link to the ring of links.
    void AddLink(TNetLink* link);

    //  Return the traffic control for the node.
    TNetTrafficControl& GetTrafficControl();
    const TNetTrafficControl& GetTrafficControl() const;

    //  Define the traffic control for the node.
    void SetTrafficControl (TNetTrafficControl*);

    //  Return whether the node has the same id as the given node.
    bool operator==(const TNetNode& node) const;

    //  Return whether the node has a different id from the given node.
    bool operator!=(const TNetNode& node) const;

  private:

    //  Do not allow nodes to be copied.
    TNetNode(const TNetNode&) {}

    //  Do not allows nodes to be assigned.
    TNetNode& operator=(const TNetNode&) {return *this;}

    //  Each node has a unique id.
    NetNodeId fId;

    //  Each node has a geographic position.
```

```
    TGeoPoint fPosition;

    //  Each node is connected to links.
    LinkRing fLinks;

    //  Each node has an associated traffic control
    TNetTrafficControl* fTrafficControl;
};


#endif // TRANSIMS_NET_NODE
```

2. Node.C

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: Node.C,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 19:29:51 $
//  $State: Rel $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


//  Include Booch Components header files.
#include <BCOQueU.h>


//  Include TRANSIMS header files.
#include <NET/Node.h>
#include <NET/NodeReader.h>
#include <NET/Link.h>
#include <NET/TrafficControl.h>


//  Construct a node using the reader.
TNetNode::TNetNode(TNetNodeReader& reader)
  : fId(reader.GetId()),
    fPosition(reader.GetGeographicPosition()),
    fLinks(),
    fTrafficControl(NULL)
{
}


//  Construct a dummy node with the specified id.
TNetNode::TNetNode(NetNodeId id)
  : fId(id),
    fPosition(),
    fLinks(),
    fTrafficControl(NULL)
{
}


//  Destroy a node.
TNetNode::~TNetNode()
{
    delete fTrafficControl;
}


//  Return the id of the node.
NetNodeId TNetNode::GetId() const
{
    return fId;
}


//  Return the geographic position of the node.
TGeoPoint& TNetNode::GetGeographicPosition()
{
    return fPosition;
}
```

```
const TGeoPoint& TNetNode::GetGeographicPosition() const
{
    return fPosition;
}


//  Return the ring of links in order.
TNetNode::LinkRing& TNetNode::GetLinks()
{
    return fLinks;
}

const TNetNode::LinkRing& TNetNode::GetLinks() const
{
    return fLinks;
}


//  Return whether a link belongs before another link in the link ring.
//ISSUE(bwb): This function is not thread-safe.
static const TNetNode* THIS = NULL;
static BC_Boolean LinkIsBefore(TNetLink *const& a, TNetLink *const& b)
{
    return a->GetAngle(*THIS) < b->GetAngle(*THIS);
}


//  Add the specified link to the ring of links.
void TNetNode::AddLink(TNetLink* link)
{
    //ISSUE(bwb): The BC_TRing::Rotate() and BC_TRing::Insert(...) functions do
    //not work as advertised, so the following work-around is necessary.

    THIS = this;
    BC_TUnboundedOrderedQueue<TNetLink*, BC_CManaged> queue;
    queue.SetIsLessThanFunction(LinkIsBefore);

    queue.Append(link);
    while (!fLinks.IsEmpty()) {
        queue.Append(fLinks.Top());
        fLinks.Pop();
    }

    while (!queue.IsEmpty()) {
        fLinks.Insert(queue.Front());
        queue.Pop();
    }

    THIS = NULL;
}


//  Return the traffic control for the node.
TNetTrafficControl&
TNetNode::GetTrafficControl()
{
    return *fTrafficControl;
}

const TNetTrafficControl&
TNetNode::GetTrafficControl() const
{
    return *fTrafficControl;
}


//  Define the traffic control for the node.
void  TNetNode::SetTrafficControl(TNetTrafficControl* c)
{
    fTrafficControl = c;
}


//  Return whether the node has the same id as the given node.
bool TNetNode::operator==(const TNetNode& node) const
```

```
{
    return GetId() == node.GetId();
}


//  Return whether the node has a different id from the given node.
bool TNetNode::operator!=(const TNetNode& node) const
{
    return GetId() != node.GetId();
}
```

Q. TNetNodeReader Class

1. NodeReader.h

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: NodeReader.h,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 19:29:51 $
//  $State: Rel $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


#ifndef TRANSIMS_NET_NODEREADER
#define TRANSIMS_NET_NODEREADER


//  Include TRANSIMS header files.
#include <GBL/Globals.h>
#include <DBS/Accessor.h>
#include <NET/Id.h>
#include <NET/Point.h>


//  Forward declarations.
class TNetReader;


//  A node reader reads node values from the database.
class TNetNodeReader
{
  public:

    //  Construct a node reader for a given network.
    TNetNodeReader(TNetReader& reader);

    //  Construct a copy of the given node reader.
//  TNetNodeReader(const TNetNodeReader& reader);

    //  Make the reader a copy of the given node reader.
//  TNetNodeReader& operator=(const TNetNodeReader& reader);

    //  Reset the iteration over the table.
    void Reset();

    //  Get the next node in the table.
    void GetNextNode();

    //  Return whether there are any more nodes in the table.
    bool MoreNodes() const;

    //  Return the id for the current node.
    NetNodeId GetId() const;

    //  Return the geographic position for the current node.
    TGeoPoint GetGeographicPosition() const;

  private:

    //  Each node reader has a database table accessor.
    TDbAccessor fAccessor;
```

```
    //  Each node has an ID field.
    const TDbField fIdField;

    //  Each node has an ABSCISSA field.
    const TDbField fAbscissaField;

    //  Each node has an ORDINATE field.
    const TDbField fOrdinateField;
};


#endif // TRANSIMS_NET_NODEREADER
```

2. NodeReader.C

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: NodeReader.C,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 19:29:51 $
//  $State: Rel $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


//  Include TRANSIMS header files.
#include <NET/NodeReader.h>
#include <NET/Reader.h>


//  Construct a node reader for a given network.
TNetNodeReader::TNetNodeReader(TNetReader& reader)
  : fAccessor(reader.GetNodeTable()),
    fIdField(reader.GetNodeTable().GetField("ID")),
    fAbscissaField(reader.GetNodeTable().GetField("ABSCISSA")),
    fOrdinateField(reader.GetNodeTable().GetField("ORDINATE"))
{
}


//  Reset the iteration over the table.
void TNetNodeReader::Reset()
{
    fAccessor.GotoFirst();
}


//  Get the next node in the table.
void TNetNodeReader::GetNextNode()
{
    fAccessor.GotoNext();
}


//  Return whether there are any more nodes in the table.
bool TNetNodeReader::MoreNodes() const
{
    return fAccessor.IsAtRecord();
}


//  Return the id for the current node.
NetNodeId TNetNodeReader::GetId() const
{
    NetNodeId id;
    fAccessor.GetField(fIdField, id);
    return id;
}


//  Return the geographic position for the current node.
TGeoPoint TNetNodeReader::GetGeographicPosition() const
{
```

```
    REAL x, y;
    fAccessor.GetField(fAbscissaField, x);
    fAccessor.GetField(fOrdinateField, y);
    return TGeoPoint(x, y);
}
```

R. TNetNullControl Class

1. NullControl.h

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: NullControl.h,v $
//  $Revision: 2.2 $
//  $Date: 1996/12/13 15:23:38 $
//  $State: Stab $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


#ifndef TRANSIMS_NET_NULLCONTROL
#define TRANSIMS_NET_NULLCONTROL


//  Include TRANSIMS header files.
#include "GBL/Globals.h"
#include "NET/TrafficControl.h"


//  Forward declarations.
class TNetNode;
class TNetLink;
class TNetLane;


//  Null controls are used by nodes just outside the network boundary that have
//  a TNetBoundary accessory on an attached link.
class TNetNullControl
  : public TNetTrafficControl
{
  public:

    //  Construct a null traffic control.
    TNetNullControl();
    TNetNullControl(TNetNode& node);
    TNetNullControl(const TNetNullControl& control);

    //  Destroy a null traffic control.
    ~TNetNullControl();

    //  Assign a null traffic control.
    TNetNullControl& operator=(const TNetNullControl& control);

    //  Return whether two null controls are the same.
    bool operator==(const TNetNullControl& control) const;
    bool operator!=(const TNetNullControl& control) const;

    //  Provide definitions for pure virtual functions to throw exceptions.
    void AllowedMovements(LaneCollection& lanes, const TNetLink& fromlink, const
            TNetLane& fromlane, const TNetLink& tolink);
    void AllowedMovements(LaneCollection& lanes, const TNetLink& fromlink, const
            TNetLink& tolink, const TNetLane& tolane);
    void AllowedMovements(LaneCollection& lanes, const TNetLink& fromlink, const
           TNetLink& tolink, bool phase = FALSE);
    void InterferingLanes(LaneCollection& lanes, const TNetLane& fromlane, const
            TNetLane& tolane, bool phase = FALSE);
    TNetTrafficControl::ETrafficControl GetVehicleControl(const TNetLane& lane)
            const;
};


#endif // TRANSIMS_NET_NULLCONTROL
```

2. NullControl.C

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: NullControl.C,v $
//  $Revision: 2.3 $
//  $Date: 1996/12/13 15:24:17 $
//  $State: Stab $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


//  Include TRANSIMS header files.
#include "NET/NullControl.h"
#include "NET/Node.h"
#include "NET/Link.h"
#include "NET/Lane.h"
#include "NET/Exception.h"


//  Construct a null traffic control.
TNetNullControl::TNetNullControl()
  : TNetTrafficControl()
{
}

TNetNullControl::TNetNullControl(TNetNode& node)
  : TNetTrafficControl(node)
{
}

TNetNullControl::TNetNullControl(const TNetNullControl& n)
  : TNetTrafficControl(n)
{
}


//  Destroy a null traffic control.
TNetNullControl::~TNetNullControl()
{
}


//  Assign a null traffic control.
TNetNullControl& TNetNullControl::operator=(const TNetNullControl& n)
{
    if (this == &n)
        return *this;
    TNetTrafficControl::operator=(n);
    return *this;
}


//  Return whether two null controls are the same.
bool TNetNullControl::operator==(const TNetNullControl& n) const
{
    return (this == &n);
}

bool TNetNullControl::operator!=(const TNetNullControl& n) const
{
    return !(this == &n);
}


//  Provide definitions for pure virtual functions.
void TNetNullControl::AllowedMovements(LaneCollection& /*c*/, const TNetLink&
        /*fromlink*/, const TNetLane& /*fromlane*/, const TNetLink& /*tolink*/)
{
    throw TNetUndefinedControl("Nodes with null traffic control must not be "
            "reached.");
}

void TNetNullControl::AllowedMovements(LaneCollection& /*c*/, const TNetLink&
        /*fromlink*/, const TNetLink& /*tolink*/, const TNetLane& /*tolane*/)
```

```
{
    throw TNetUndefinedControl ("Nodes with null traffic control must not be "
            "reached.");
}

void TNetNullControl::AllowedMovements(LaneCollection& /*c*/, const TNetLink&
        /*fromlink*/, const TNetLink& /*tolink*/, bool /*phase*/)
{
    throw TNetUndefinedControl("Nodes with null traffic control must not be "
            "reached.");
}

void TNetNullControl::InterferingLanes(LaneCollection& /*c*/, const TNetLane&
        /*fromlane*/, const TNetLane& /*tolane*/, bool /*phase*/)
{
    throw TNetUndefinedControl("Nodes with null traffic control must not be "
            "reached.");
}

TNetTrafficControl::ETrafficControl TNetNullControl::GetVehicleControl(const
        TNetLane& /*lane*/) const
{
    throw TNetUndefinedControl("Nodes with null traffic control must not be "
            "reached.");
    return TNetTrafficControl::kNone; // this statement never reached
}
```

S. TNetParking Class

1. Parking.h

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: Parking.h,v $
//  $Revision: 2.1 $
//  $Date: 1996/06/03 22:36:32 $
//  $State: Stab $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


#ifndef TRANSIMS_NET_PARKING
#define TRANSIMS_NET_PARKING


//  Include TRANSIMS header files.
#include <NET/Accessory.h>


//  Forward declarations.
class TNetParkingReader;
class TNetNetwork;
class TNetLink;
class TNetNode;


//  A parking place is a source or sink of vehicles along a link.
class TNetParking
  : public TNetAccessory
{
  public:

    //  There are several styles of parking.
    enum EStyle {kParallelOnStreet, kHeadInOnStreet, kDriveway, kLot,
            kBoundary};

    //  Construct the parking from the specified reader.
    TNetParking(TNetParkingReader& reader);

    //  Construct a parking place with specified values.
    TNetParking (NetAccessoryId, TNetLink&, TNetNode&, REAL offset,
                 EStyle style, UINT capacity, bool generic);
```

```
    //  Return the style of the parking.
    EStyle GetStyle() const;

    //  Return the capacity of the parking.
    UINT GetCapacity() const;

    //  Return whether the parking is generic.
    bool IsGeneric() const;

    //  Generate a unique parking place id.
    static NetAccessoryId GenerateId(const TNetNetwork&, NetLinkId);

  private:

    //  A parking place has a style.
    EStyle fStyle;

    //  A parking place has a capacity.
    UINT fCapacity;

    //  A parking place may be generic.
    bool fGeneric;
};


#endif // TRANSIMS_NET_PARKING
```

2. Parking.C

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: Parking.C,v $
//  $Revision: 2.1 $
//  $Date: 1996/06/03 22:36:58 $
//  $State: Stab $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


//  Include TRANSIMS header files.
#include <NET/Parking.h>
#include <NET/ParkingReader.h>
#include <NET/Network.h>
#include <NET/Location.h>
#include <NET/Exception.h>


//  Construct the parking from the specified reader.
TNetParking::TNetParking(TNetParkingReader& reader)
  : TNetAccessory(reader.GetId(), kParking),
    fStyle(reader.GetStyle()),
    fCapacity(reader.GetCapacity()),
    fGeneric(reader.IsGeneric())
{
}


//  Construct a parking place with specified values.
TNetParking::TNetParking (NetAccessoryId id, TNetLink& link, TNetNode& node,
        REAL offset, EStyle style, UINT capacity, bool generic)
  : TNetAccessory(id, kParking),
    fStyle(style),
    fCapacity(capacity),
    fGeneric(generic)
{
    TNetLocation location(link, node, offset);
    SetLocation(location);
}


//  Return the style of the parking.
TNetParking::EStyle TNetParking::GetStyle() const
{
    return fStyle;
```

```
}


//  Return the capacity of the parking.
UINT TNetParking::GetCapacity() const
{
    return fCapacity;
}


//  Return whether the parking is generic.
bool TNetParking::IsGeneric() const
{
    return fGeneric;
}


//  Generate a unique parking place id.
NetAccessoryId TNetParking::GenerateId(const TNetNetwork& network, NetLinkId
        linkid)
{
    UINT bit_mask_32 = 1 << 31;
    NetAccessoryId id = linkid | bit_mask_32;
    if (!network.GetParkings().IsBound(id))
        return id;
    else {
        UINT bit_mask_31 = 1 << 30;
        id = linkid | bit_mask_31;
        if (!network.GetParkings().IsBound(id))
            return id;
        else {
            UINT bit_mask_3231 = 3 << 30;
            id = linkid | bit_mask_3231;
            if (!network.GetParkings().IsBound(id))
                return id;
            else {
                throw TNetException("Cannot generate accessory id.");
                return 0; //ISSUE(kpb): This statement never reached.
            }
        }
    }
}
```

T. TNetParkingReader Class

1. ParkingReader.h

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: ParkingReader.h,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 19:29:51 $
//  $State: Rel $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


#ifndef TRANSIMS_NET_PARKINGREADER
#define TRANSIMS_NET_PARKINGREADER


//  Include TRANSIMS header files.
#include <GBL/Globals.h>
#include <DBS/Accessor.h>
#include <NET/Id.h>
#include <NET/Parking.h>
#include <NET/AccessoryReader.h>


//  Forward declarations.
class TNetReader;
```

```
//  A parking reader reads parking values from the database.
class TNetParkingReader
  : public TNetAccessoryReader
{
  public:

    //  Construct a parking reader for a given network.
    TNetParkingReader(TNetReader& reader);

    //  Construct a copy of the given parking reader.
//  TNetParkingReader(const TNetParkingReader& reader);

    //  Make the parking reader a copy of the given parking reader.
//  TNetParkingReader& operator=(const TNetParkingReader& reader);

    //  Return the style of the current parking.
    TNetParking::EStyle GetStyle() const;

    //  Return the capacity of the current parking.
    UINT GetCapacity() const;

    //  Return whether the current parking is generic.
    bool IsGeneric() const;

  private:

    //  Each parking has a STYLE field.
    const TDbField fStyleField;

    //  Each parking has a CAPACITY field.
    const TDbField fCapacityField;

    //  Each parking has a GENERIC field.
    const TDbField fGenericField;
};


#endif // TRANSIMS_NET_PARKINGREADER
```

2. ParkingReader.C

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: ParkingReader.C,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 19:29:51 $
//  $State: Rel $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


//  Include TRANSIMS header files.
#include <NET/Exception.h>
#include <NET/ParkingReader.h>
#include <NET/Reader.h>


//  Construct a parking reader for a given network.
TNetParkingReader::TNetParkingReader(TNetReader& reader)
  : TNetAccessoryReader(reader.GetParkingTable()),
    fStyleField(reader.GetParkingTable().GetField("STYLE")),
    fCapacityField(reader.GetParkingTable().GetField("CAPACITY")),
    fGenericField(reader.GetParkingTable().GetField("GENERIC"))
{
}


//  Return the style of the current parking.
TNetParking::EStyle TNetParkingReader::GetStyle() const
{
    string style;
    fAccessor.GetField(fStyleField, style);
    if (style == "PRSTR")
        return TNetParking::kParallelOnStreet;
```

```
    else if (style == "HISTR")
        return TNetParking::kHeadInOnStreet;
    else if (style == "DRVWY")
        return TNetParking::kDriveway;
    else if (style == "LOT  ")
        return TNetParking::kLot;
    else if (style == "BNDRY")
        return TNetParking::kBoundary;
    else
        throw TNetNotFound("Invalid parking style.");
}


//  Return the capacity of the parking.
UINT TNetParkingReader::GetCapacity() const
{
    UINT capacity;
    fAccessor.GetField(fCapacityField, capacity);
    return capacity;
}


//  Return whether the current parking is generic.
bool TNetParkingReader::IsGeneric() const
{
    string generic;
    fAccessor.GetField(fGenericField, generic);
    return generic == "T";
}
```

U. TNetPhase Class

1. Phase.h

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: Phase.h,v $
//  $Revision: 2.2 $
//  $Date: 1996/11/08 18:27:21 $
//  $State: Stab $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


#ifndef TRANSIMS_NET_PHASE
#define TRANSIMS_NET_PHASE


//  Include TRANSIMS header files.
#include "GBL/Globals.h"


//  Include Booch Components header files.
#include "BCStoreM.h"
#include "BCRingB.h"
#include "BCCollU.h"


//  Forward declarations.
class TNetLink;
class TNetLane;
class TNetPhaseDescription;


//  A phase is a portion of a traffic signal cycle when the allowed movements
//  are unchanged.  A phase is composed of intervals where the traffic displays
//  are constant.
class TNetPhase
{
  public:

    //  Each phase has three intervals.
    enum EInterval {kGreen, kYellow, kRed};
```

```
    //  Type definitions.
    typedef BC_TBoundedRing<EInterval, 3U> IntervalRing;
    typedef BC_TRingActiveIterator<EInterval> IntervalRingIterator;
    typedef BC_TUnboundedCollection<TNetPhase*, BC_CManaged> PhaseCollection;
    typedef BC_TCollectionActiveIterator<TNetPhase*> PhaseCollectionIterator;

    //  Construct a phase.
    TNetPhase();
    TNetPhase(TNetPhaseDescription& description);
    TNetPhase(const TNetPhase& phase);

    //  Destroy a phase.
    ~TNetPhase();

    //  Assign a phase.
    TNetPhase& operator=(const TNetPhase& phase);

    //  Return whether two phases are the same.
    bool operator==(const TNetPhase&) const;
    bool operator!=(const TNetPhase&) const;

    //  Return phase description associated with this phase.
    TNetPhaseDescription& GetPhaseDescription();
    const TNetPhaseDescription& GetPhaseDescription() const;

    //  Return current interval.
    TNetPhase::EInterval GetInterval() const;

    //  Update signal interval.
    void SetInterval(EInterval interval);

    //  Return phases to which this phase can transition.
    PhaseCollection& GetNextPhases();
    const PhaseCollection& GetNextPhases() const;

    //  Define the next phase.
    void SetNextPhase(TNetPhase& phase);

  private:

    //  Each phase has a sequence of intervals.
    IntervalRing  fIntervals;

    //  Each phase has an associated phase description.
    TNetPhaseDescription* fPhaseDescription;

    //  A phase has one or more next phases that it can transition to.
    PhaseCollection fNextPhases;
};


#endif // TRANSIMS_NET_PHASE
```

2. Phase.C

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: Phase.C,v $
//  $Revision: 2.2 $
//  $Date: 1996/11/08 18:27:21 $
//  $State: Stab $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


//  Include TRANSIMS header files.
#include "NET/TrafficControl.h"
#include "NET/Phase.h"
#include "NET/PhasingPlan.h"


//  Construct a phase.
TNetPhase::TNetPhase()
```

```
  : fPhaseDescription(NULL),
    fNextPhases()
{
    fIntervals.Insert(TNetPhase::kRed);
    fIntervals.Insert(TNetPhase::kYellow);
    fIntervals.Insert(TNetPhase::kGreen);
}

TNetPhase::TNetPhase(TNetPhaseDescription& d)
  : fPhaseDescription(&d),
    fNextPhases()
{
    fIntervals.Insert(TNetPhase::kRed);
    fIntervals.Insert(TNetPhase::kYellow);
    fIntervals.Insert(TNetPhase::kGreen);
}

TNetPhase::TNetPhase(const TNetPhase& p)
  : fNextPhases()
{
    fIntervals = p.fIntervals;
    fPhaseDescription = p.fPhaseDescription;
    fNextPhases = p.fNextPhases;
}


//  Destroy a phase.
TNetPhase::~TNetPhase()
{
}


//  Assign a phase.
TNetPhase& TNetPhase::operator=(const TNetPhase& p)
{
    if (this == &p)
        return *this;
    fIntervals = p.fIntervals;
    fPhaseDescription = p.fPhaseDescription;
    fNextPhases = p.fNextPhases;
    return *this;
}


//  Return whether two phases are the same.
bool TNetPhase::operator==(const TNetPhase& p) const
{
    return (this == &p);
}

bool TNetPhase::operator!=(const TNetPhase& p) const
{
    return !(this == &p);
}


//  Return phase description associated with this phase.
TNetPhaseDescription& TNetPhase::GetPhaseDescription()
{
    return *fPhaseDescription;
}

const TNetPhaseDescription& TNetPhase::GetPhaseDescription() const
{
    return *fPhaseDescription;
}


//  Return current interval.
TNetPhase::EInterval TNetPhase::GetInterval() const
{
    return fIntervals.Top();
}


//  Update signal interval.
```

```
void TNetPhase::SetInterval(EInterval intv)
{
    for (int i=0; I < 2; ++i)
        if (fIntervals.Top() != intv)
            fIntervals.Rotate();
        else
            break;
}


//  Return phases to which this phase can transition.
TNetPhase::PhaseCollection& TNetPhase::GetNextPhases()
{
    return fNextPhases;
}

const TNetPhase::PhaseCollection& TNetPhase::GetNextPhases() const
{
    return fNextPhases;
}


//  Define the next phases.
void TNetPhase::SetNextPhase(TNetPhase& p)
{
    fNextPhases.Append(&p);
}
```

V. TNetPhaseDescription Class

1. PhaseDescription.h

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: PhaseDescription.h,v $
//  $Revision: 2.2 $
//  $Date: 1996/11/08 18:28:08 $
//  $State: Stab $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


#ifndef TRANSIMS_NET_PHASEDESCRIPTION
#define TRANSIMS_NET_PHASEDESCRIPTION


//  Include TRANSIMS header files.
#include "GBL/Globals.h"
#include "NET/Id.h"


//  Include Booch Components header files.
#include "BCStoreM.h"
#include "BCMapU.h"
#include "BCCollU.h"


//  Forward declarations
class TNetLink;


//  A phase description specifies the interval lengths and allowed movements
//  and associated turn protections during a phase.
class TNetPhaseDescription
{
  public:

    //  Turn protections are a subset of ETrafficControl
    enum EProtection {kPermitted, kProtected};

    //  Type definitions.
    typedef BC_TUnboundedMap<TNetLink*, EProtection, 4U, BC_CManaged>
            LinkProtectionMap;
```

```
  typedef BC_TMapActiveIterator<TNetLink*, EProtection>
          LinkProtectionMapIterator;
  typedef BC_TUnboundedMap<TNetLink*, LinkProtectionMap*, 4U, BC_CManaged>
          LinkMovementMap;
  typedef BC_TMapActiveIterator<TNetLink*, LinkProtectionMap*>
          LinkMovementMapIterator;

  //  Construct a phasing plan description.
  TNetPhaseDescription(NetPhaseNumber phase);
  TNetPhaseDescription(const TNetPhaseDescription& description);

  //  Destroy a phasing plan description.
  ~TNetPhaseDescription();

  //  Assign a phasing plan description.
  TNetPhaseDescription& operator=(const TNetPhaseDescription& description);

  //  Return whether two phase descriptions are the same.
  bool operator==(const TNetPhaseDescription& description) const;
  bool operator!=(const TNetPhaseDescription& description) const;

  //  Return the phase number.
  NetPhaseNumber GetPhaseNumber() const;

  //  Return the green interval length.
  REAL GetGreenLength() const;

  //  Return the minimum green interval length.
  REAL GetMinGreenLength() const;

  //  Return the maximum green interval length.
  REAL GetMaxGreenLength() const;

  //  Return the green extension interval length.
  REAL GetExtGreenLength() const;

  //  Return the yellow interval length.
  REAL GetYellowLength() const;

  //  Return the red interval length.
  REAL GetRedLength() const;

  //  Return all link movements.
  LinkMovementMap& GetLinkMovements();
  const LinkMovementMap& GetLinkMovements() const;

  //  Return the link movements for the specified link.
  LinkProtectionMap& GetLinkMovements(const TNetLink& link);
  const LinkProtectionMap& GetLinkMovements(const TNetLink& link) const;

  //  Set the interval lengths.
  void SetLengths(REAL min, REAL max, REAL ext, REAL yellow, REAL red);

  //  Set link movements
  void SetLinkMovements(TNetLink& inlink, TNetLink& outlink,
          TNetPhaseDescription::EProtection protection);

private:

  //  Phase number.
  NetPhaseNumber fPhaseNumber;

  //  Minimum green interval length, or green interval length for fixed time.
  union {
      REAL fMinGreenLength;
      REAL fGreenLength;
  };

  //  Maximum green interval length, undefined for fixed time.
  REAL fMaxGreenLength;

  //  Green interval extension increment, equals zero for fixed time.
  REAL fExtGreenLength;

  //  Yellow interval length.
  REAL fYellowLength;
```

```
    //  Red clearance interval length.
    REAL fRedLength;

    //  Movements allowed during this phase.
    LinkMovementMap fLinkMovements;

    //  Null map is returned when no link movement binding exists.
    static LinkProtectionMap fNull;

    //  Walk interval length.
    //  REAL fWalkLength;

    //  Flashing don't walk interval length.
    //  REAL fFDWLength;
};


#endif // TRANSIMS_NET_PHASEDESCRIPTION
```

2. PhaseDescription.C

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: PhaseDescription.C,v $
//  $Revision: 2.3 $
//  $Date: 1996/11/08 18:28:08 $
//  $State: Stab $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


//  Include TRANSIMS header files
#include "NET/PhaseDescription.h"


//  Define the hash function for links.
static BC_Index LinkHashValue(TNetLink* const & l)
{
    return ((int)l & 0xf0) >> 4;
}


//  Null map is returned when no link movement binding exists.
TNetPhaseDescription::LinkProtectionMap TNetPhaseDescription::fNull;


//  Construct a phasing plan description.
TNetPhaseDescription::TNetPhaseDescription(NetPhaseNumber n)
  : fPhaseNumber(n),
    fMinGreenLength(0.0),
    fMaxGreenLength(0.0),
    fExtGreenLength(0.0),
    fYellowLength(0.0),
    fRedLength(0.0),
    fLinkMovements(LinkHashValue)
{
}

TNetPhaseDescription::TNetPhaseDescription(const TNetPhaseDescription& p)
  : fLinkMovements(LinkHashValue)
{
    fPhaseNumber = p.fPhaseNumber;
    fMinGreenLength = p.fMinGreenLength;
    fMaxGreenLength = p.fMaxGreenLength;
    fExtGreenLength = p.fExtGreenLength;
    fYellowLength = p.fYellowLength;
    fRedLength = p.fRedLength;
    fLinkMovements = p.fLinkMovements;
}


//  Destroy a phasing plan description.
TNetPhaseDescription::~TNetPhaseDescription()
```

```
{
}


//  Assign a phasing plan description.
TNetPhaseDescription& TNetPhaseDescription::operator=(const
        TNetPhaseDescription& p)
{
    if (this == &p)
        return *this;
    fPhaseNumber = p.fPhaseNumber;
    fMinGreenLength = p.fMinGreenLength;
    fMaxGreenLength = p.fMaxGreenLength;
    fExtGreenLength = p.fExtGreenLength;
    fYellowLength = p.fYellowLength;
    fRedLength = p.fRedLength;
    fLinkMovements = p.fLinkMovements;
    return *this;
}


//  Return whether two phase descriptions are the same.
bool TNetPhaseDescription::operator==(const TNetPhaseDescription& p) const
{
    return (this == &p);
}

bool TNetPhaseDescription::operator!=(const TNetPhaseDescription& p) const
{
    return !(this == &p);
}


//  Return the phase number.
NetPhaseNumber TNetPhaseDescription::GetPhaseNumber() const
{
    return fPhaseNumber;
}


//  Return the green interval length.
REAL TNetPhaseDescription::GetGreenLength () const
{
    return fGreenLength;
}


//  Return the minimum green interval length.
REAL TNetPhaseDescription::GetMinGreenLength() const
{
    return fMinGreenLength;
}


//  Return the maximum green interval length.
REAL TNetPhaseDescription::GetMaxGreenLength() const
{
    return fMaxGreenLength;
}


//  Return the green extension interval length.
REAL TNetPhaseDescription::GetExtGreenLength() const
{
    return fExtGreenLength;
}


//  Return the yellow interval length.
REAL TNetPhaseDescription::GetYellowLength() const
{
    return fYellowLength;
}


//  Return the red interval length.
```

```
REAL TNetPhaseDescription::GetRedLength() const
{
    return fRedLength;
}


//  Return all link movements.
TNetPhaseDescription::LinkMovementMap& TNetPhaseDescription::GetLinkMovements()
{
    return fLinkMovements;
}

const TNetPhaseDescription::LinkMovementMap&
        TNetPhaseDescription::GetLinkMovements () const
{
    return fLinkMovements;
}


//  Return the link movements for the specified link.
TNetPhaseDescription::LinkProtectionMap&
        TNetPhaseDescription::GetLinkMovements(const TNetLink& l1)
{
    if (fLinkMovements.IsBound((TNetLink*)&l1))
        return **fLinkMovements.ValueOf((TNetLink*)&l1);
    else
        return fNull;
}

const TNetPhaseDescription::LinkProtectionMap&
        TNetPhaseDescription::GetLinkMovements(const TNetLink& l1) const
{
    if (fLinkMovements.IsBound((TNetLink*)&l1))
        return **fLinkMovements.ValueOf((TNetLink*)&l1);
    else
        return fNull;
}


//  Set the interval lengths.
void TNetPhaseDescription::SetLengths(REAL min, REAL max, REAL ext, REAL yellow,
        REAL red)
{
    fMinGreenLength = min;
    fMaxGreenLength = max;
    fExtGreenLength = ext;
    fYellowLength = yellow;
    fRedLength = red;
}


//  Set the link movements.
void TNetPhaseDescription::SetLinkMovements(TNetLink& l1, TNetLink& l2,
        TNetPhaseDescription::EProtection p)
{
    if (!fLinkMovements.IsBound(&l1)) {
        LinkProtectionMap* prot = new LinkProtectionMap(LinkHashValue);
        prot->Bind(&l2, p);
        fLinkMovements.Bind(&l1, prot);
    } else
        (*fLinkMovements.ValueOf(&l1))->Bind(&l2, p);
}
```

W. TNetPhasingPlan Class

1. PhasingPlan.h

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: PhasingPlan.h,v $
//  $Revision: 2.3 $
//  $Date: 1996/11/08 18:28:08 $
//  $State: Stab $
//  $Author: bwb $
```

```
//  U.S. Government Copyright 1995
//  All rights reserved


#ifndef TRANSIMS_NET_PHASINGPLAN
#define TRANSIMS_NET_PHASINGPLAN


//  Include TRANSIMS header files.
#include "GBL/Globals.h"
#include "NET/Id.h"
#include "NET/PhaseDescription.h"


//  Include Booch Components header files.
#include "BCStoreM.h"
#include "BCCollU.h"


//  Forward declarations.
class TNetLink;
class TNetPhaseDescription;


//  A phasing plan is composed of a series of phase descriptions.
class TNetPhasingPlan
{
  public:

    //  Type definitions.
    typedef BC_TUnboundedCollection<TNetPhaseDescription*, BC_CManaged>
            PhaseDescriptionCollection;
    typedef BC_TCollectionActiveIterator<TNetPhaseDescription*>
            PhaseDescriptionCollectionIterator;

    //  Construct a phasing plan.
    TNetPhasingPlan();
    TNetPhasingPlan(NetPlanId id);
    TNetPhasingPlan(const TNetPhasingPlan& plan);

    //  Destroy a phasing plan.
    ~TNetPhasingPlan();

    //  Assign a phasing plan.
    TNetPhasingPlan& operator=(const TNetPhasingPlan& plan);

    //  Return whether two phasing plans are the same.
    bool operator==(const TNetPhasingPlan& plan) const;
    bool operator!=(const TNetPhasingPlan& plan) const;

    //  Return this phasing plan.
    PhaseDescriptionCollection& GetPhaseDescriptions();
    const PhaseDescriptionCollection& GetPhaseDescriptions() const;

    //  Create a phase description.
    TNetPhaseDescription& CreatePhaseDescription(NetPhaseNumber phase);

    //  Set values in phase description
    //void SetPhaseDescription(NetPhaseNumber, REAL, REAL, REAL, REAL, REAL);
    //void SetPhaseDescription(NetPhaseNumber, const TNetLink&, const TNetLink&,
    //        TNetPhaseDescription::EProtection);

    //  Return the plan id.
    NetPlanId GetId() const;

  private:

    //  Phasing plan has an id.
    NetPlanId fId;

    //  A phasing plan is a sequence of interval length descriptions, one
    //  for each phase.
   PhaseDescriptionCollection  fPhasingPlan;
};
```

```
#endif // TRANSIMS_NET_PHASINGPLAN
```

2. PhasingPlan.C

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: PhasingPlan.C,v $
//  $Revision: 2.3 $
//  $Date: 1996/11/08 18:28:08 $
//  $State: Stab $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


//  Include TRANSIMS header files
#include "NET/PhasingPlan.h"


//  Define the hash function for links.
static BC_Index LinkHashValue (TNetLink* const & l)
{
    return ((int)l & 0xf0) >> 4;
}


//  Construct a phasing plan.
TNetPhasingPlan::TNetPhasingPlan ()
  : fId(0),
    fPhasingPlan()
{
}

TNetPhasingPlan::TNetPhasingPlan(NetPlanId id)
  : fId(id),
    fPhasingPlan()
{
}

TNetPhasingPlan::TNetPhasingPlan(const TNetPhasingPlan& p)
  : fPhasingPlan()
{
    fId = p.fId;
    fPhasingPlan = p.fPhasingPlan;
}


//  Destroy a phasing plan.
TNetPhasingPlan::~TNetPhasingPlan()
{
    for (PhaseDescriptionCollectionIterator it(fPhasingPlan); !it.IsDone();
            it.Next())
        delete *it.CurrentItem();
}


//  Assign a phasing plan.
TNetPhasingPlan& TNetPhasingPlan::operator=(const TNetPhasingPlan& p)
{
    if (this == &p)
        return *this;
    fId = p.fId;
    fPhasingPlan = p.fPhasingPlan;
    return *this;
}


//  Return whether two phasing plans are the same.
bool TNetPhasingPlan::operator==(const TNetPhasingPlan& p) const
{
    return (this == &p);
}

bool TNetPhasingPlan::operator!=(const TNetPhasingPlan& p) const
{
```

```
    return !(this == &p);
}


//  Return this phasing plan.
TNetPhasingPlan::PhaseDescriptionCollection&
        TNetPhasingPlan::GetPhaseDescriptions()
{
    return fPhasingPlan;
}

const TNetPhasingPlan::PhaseDescriptionCollection&
        TNetPhasingPlan::GetPhaseDescriptions() const
{
    return fPhasingPlan;
}


//  Create a phase description.
TNetPhaseDescription& TNetPhasingPlan::CreatePhaseDescription(NetPhaseNumber n)
{
    TNetPhaseDescription* d = new TNetPhaseDescription(n);
    fPhasingPlan.Append(d);
    return *d;
}


//  Set values in phase description
//void TNetPhasingPlan::SetPhaseDescription(NetPhaseNumber phase, REAL min,
//        REAL max, REAL ext, REAL yellow, REAL red)
//{
//    fPhasingPlan[phase-1]->SetLengths (min, max, ext, yellow, red);
//}
//
//void TNetPhasingPlan::SetPhaseDescription(NetPhaseNumber phase, const
//        TNetLink& l1, const TNetLink& l2, TNetPhaseDescription::EProtection p)
//{
//    fPhasingPlan[phase-1]->SetLinkMovements ((TNetLink&)l1, (TNetLink&)l2, p);
//}


//  Return the plan id.
NetPlanId TNetPhasingPlan::GetId() const
{
    return fId;
}
```

X. TNetPhasingPlanReader Class

1. PhasingPlanReader.h

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: PhasingPlanReader.h,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 19:29:51 $
//  $State: Rel $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


#ifndef TRANSIMS_NET_PHASEREADER
#define TRANSIMS_NET_PHASEREADER


//  Include TRANSIMS header files.
#include <GBL/Globals.h>
#include <DBS/Accessor.h>
#include <NET/Id.h>


//  Forward declarations.
class TNetReader;
```

```
//  This reader reads phasing plan values from the database.
class TNetPhasingPlanReader
{
  public:

    //  Construct a phasing plan reader for a given network.
    TNetPhasingPlanReader(TNetReader& reader);

    //  Construct a copy of the given phasing plan reader.
//  TNetPhasingPlanReader(const TNetPhasingPlanReader& reader);

    //  Make the reader a copy of the given phasing plan reader.
//  TNetPhasingPlanReader& operator=(const TNetPhasingPlanReader& reader);

    //  Reset the iteration over the table.
    void Reset();

    //  Get the next node in the table.
    void GetNextNode();

    //  Return whether there are any more nodes in the table.
    bool MoreNodes();

    //  Return the id for the current node.
    NetNodeId GetNode() const;

    //  Return the timing plan id.
    NetPlanId GetPlan() const;

    //  Return the phase number.
    NetPhaseNumber GetPhase() const;

    //  Return the incoming link id.
    NetLinkId GetInlink() const;

    //  Return the outgoing link id.
    NetLinkId GetOutlink() const;

    //  Return the turn protection indicator.
    string GetProtection() const;

  private:

    //  Each phasing plan reader has a database table accessor.
    TDbAccessor fAccessor;

    //  Each record has a node field.
    const TDbField fNodeField;

    //  Each record has a timing plan field.
    const TDbField fPlanField;

    //  Each timing plan has phase numbers.
    const TDbField fPhaseField;

    //  Each record has an incoming link.
    const TDbField fInlinkField;

    //  Each record has an outgoing link.
    const TDbField fOutlinkField;

    //  Each node’s phasing plan has a turn protection indicator.
    const TDbField fProtectionField;
};


#endif // TRANSIMS_NET_PHASEREADER
```

2. PhasingPlanReader.C

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: PhasingPlanReader.C,v $
```

```
//  $Revision: 2.0 $
//  $Date: 1995/08/04 19:29:51 $
//  $State: Rel $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


//  Include TRANSIMS header files.
#include <NET/PhasingPlanReader.h>
#include <NET/Reader.h>


//  Construct a phasing plan reader for a given network.
TNetPhasingPlanReader::TNetPhasingPlanReader(TNetReader& reader)
  : fAccessor(reader.GetPhasingPlanTable()),
    fNodeField(reader.GetPhasingPlanTable().GetField("NODE")),
    fPlanField(reader.GetPhasingPlanTable().GetField("PLAN")),
    fPhaseField(reader.GetPhasingPlanTable().GetField("PHASE")),
    fInlinkField(reader.GetPhasingPlanTable().GetField("INLINK")),
    fOutlinkField(reader.GetPhasingPlanTable().GetField("OUTLINK")),
    fProtectionField(reader.GetPhasingPlanTable().GetField("PROTECTION"))
{
}


//  Reset the iteration over the table.
void TNetPhasingPlanReader::Reset()
{
    fAccessor.GotoFirst();
}


//  Get the next node in the table.
void TNetPhasingPlanReader::GetNextNode()
{
    fAccessor.GotoNext();
}


//  Return whether there are any more nodes in the table.
bool TNetPhasingPlanReader::MoreNodes()
{
    return fAccessor.IsAtRecord();
}


//  Return the id for the current node.
NetNodeId TNetPhasingPlanReader::GetNode() const
{
    NetNodeId node;
    fAccessor.GetField(fNodeField, node);
    return node;
}


//  Return the timing plan id.
NetPlanId TNetPhasingPlanReader::GetPlan() const
{
    NetPlanId plan;
    fAccessor.GetField(fPlanField, plan);
    return plan;
}


//  Return the phase number.
NetPhaseNumber TNetPhasingPlanReader::GetPhase() const
{
    NetPhaseNumber phase;
    fAccessor.GetField(fPhaseField, phase);
    return phase;
}


//  Return the incoming link id.
NetLinkId TNetPhasingPlanReader::GetInlink() const
```

```
{
    NetLinkId inlink;
    fAccessor.GetField(fInlinkField, inlink);
    return inlink;
}


//  Return the outgoing link id.
NetLinkId TNetPhasingPlanReader::GetOutlink() const
{
    NetLinkId outlink;
    fAccessor.GetField(fOutlinkField, outlink);
    return outlink;
}


//  Return the turn protection indicator.
string TNetPhasingPlanReader::GetProtection() const
{
    string prot;
    fAccessor.GetField(fProtectionField, prot);
    return prot;
}
```

Y. TNetPocket Class

1. Pocket.h

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: Pocket.h,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 19:29:51 $
//  $State: Rel $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


#ifndef TRANSIMS_NET_POCKET
#define TRANSIMS_NET_POCKET


//  Include TRANSIMS header files.
#include <NET/Accessory.h>
#include <NET/Lane.h>


//  Forward declarations.
class TNetPocketReader;


//  A pocket is a length of lane intended for special uses such as buses and
//  pulling out, vehicles waiting for turns, vehicles accelerating in order to
//  merge, etc.
class TNetPocket
  : public TNetAccessory
{
  public:

    //  There are several styles of pockets.
    enum EStyle {kTurn, kPullout, kMerge};

    //  Construct the pocket accessory from the specified reader.
    TNetPocket(TNetPocketReader& reader);

    //  Return the style of the pocket.
    EStyle GetStyle() const;

    //  Return the length of the pocket.
    REAL GetLength() const;

    //  Return the lane for the pocket.
    TNetLane& GetLane();
```

```
    const TNetLane& GetLane() const;

  private:

    //  Each pocket has a style.
    EStyle fStyle;

    //  Each pocket has a length.
    REAL fLength;
};


#endif // TRANSIMS_NET_POCKET
```

2. Pocket.C

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: Pocket.C,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 19:29:51 $
//  $State: Rel $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


//  Include TRANSIMS header files.
#include <NET/Pocket.h>
#include <NET/LaneLocation.h>
#include <NET/PocketReader.h>


//  Construct the pocket accessory from the specified reader.
TNetPocket::TNetPocket(TNetPocketReader& reader)
  : TNetAccessory(reader.GetId(), kPocket),
    fStyle(reader.GetStyle()),
    fLength(reader.GetLength())
{
}


//  Return the style of the pocket.
TNetPocket::EStyle TNetPocket::GetStyle() const
{
    return fStyle;
}


//  Return the length of the pocket.
REAL TNetPocket::GetLength() const
{
    return fLength;
}


//  Return the lane for the pocket.
TNetLane& TNetPocket::GetLane()
{
    return ((TNetLaneLocation&) GetLocation()).GetLane();
}

const TNetLane& TNetPocket::GetLane() const
{
    return ((TNetLaneLocation&) GetLocation()).GetLane();
}
```

Z. TNetPocketReader Class

1. PocketReader.h

```
//  Project: TRANSIMS
//  Subsystem: Network
```

```
//  $RCSfile: PocketReader.h,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 19:29:51 $
//  $State: Rel $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


#ifndef TRANSIMS_NET_POCKETREADER
#define TRANSIMS_NET_POCKETREADER


//  Include TRANSIMS header files.
#include <GBL/Globals.h>
#include <DBS/Accessor.h>
#include <NET/Id.h>
#include <NET/Pocket.h>
#include <NET/AccessoryReader.h>


//  Forward declarations.
class TNetReader;


//  A pocket reader reads pocket values from the database.
class TNetPocketReader
  : public TNetAccessoryReader
{
  public:

    //  Construct a pocket reader for a given network.
    TNetPocketReader(TNetReader& reader);

    //  Construct a copy of the given pocket reader.
//  TNetPocketReader(const TNetPocketReader& reader);

    //  Make the pocket reader a copy of the given pocket reader.
//  TNetPocketReader& operator=(const TNetPocketReader& reader);

    //  Return the lane number of the current pocket.
    NetLaneNumber GetLaneNumber() const;

    //  Return the style of the current pocket.
    TNetPocket::EStyle GetStyle() const;

    //  Return the length of the current pocket.
    REAL GetLength() const;

  private:

    //  Each pocket has a LANE field.
    const TDbField fLaneField;

    //  Each pocket has a STYLE field.
    const TDbField fStyleField;

    //  Each pocket has a LENGTH field.
    const TDbField fLengthField;

};


#endif // TRANSIMS_NET_POCKETREADER
```

2. PocketReader.C

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: PocketReader.C,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 19:29:51 $
//  $State: Rel $
//  $Author: bwb $
//  U.S. Government Copyright 1995
```

```
//  All rights reserved


//  Include TRANSIMS header files.
#include <NET/Exception.h>
#include <NET/PocketReader.h>
#include <NET/Reader.h>


//  Construct a pocket reader for a given network.
TNetPocketReader::TNetPocketReader(TNetReader& reader)
  : TNetAccessoryReader(reader.GetPocketTable()),
    fLaneField(reader.GetPocketTable().GetField("LANE")),
    fStyleField(reader.GetPocketTable().GetField("STYLE")),
    fLengthField(reader.GetPocketTable().GetField("LENGTH"))
{
}


//  Return the lane number of the current pocket.
NetLaneNumber TNetPocketReader::GetLaneNumber() const
{
    NetLaneNumber lane;
    fAccessor.GetField(fLaneField, lane);
    return lane;
}


//  Return the style of the current pocket.
TNetPocket::EStyle TNetPocketReader::GetStyle() const
{
    string style;
    fAccessor.GetField(fStyleField, style);
    if (style == "T")
        return TNetPocket::kTurn;
    else if (style == "P")
        return TNetPocket::kPullout;
    else if (style == "M")
        return TNetPocket::kMerge;
    else
        throw TNetNotFound("Invalid pocket style.");
}


//  Return the length of the pocket.
REAL TNetPocketReader::GetLength() const
{
    REAL length;
    fAccessor.GetField(fLengthField, length);
    return length;
}
```

AA. TGeoPoint Class

1. Point.h

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: Point.h,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 19:29:51 $
//  $State: Rel $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


#ifndef TRANSIMS_NET_POINT
#define TRANSIMS_NET_POINT


//  Include TRANSIMS header files.
#include <GBL/Globals.h>
```

```
//  A geographic point contains the coordinates of a position on a map.
class TGeoPoint
{
  public:

    //  Construct a point with the given x- and y-coordinates.
    TGeoPoint(REAL x = 0, REAL y = 0);

    //  Construct a copy of the given point.
//  TGeoPoint(const TGeoPoint& point);

    //  Make the point a copy of the given point.
//  TGeoPoint& operator=(const TGeoPoint& point);

    //  Return the x-coordinate.
    REAL GetX() const;

    //  Return the y-coordinate.
    REAL GetY() const;

    //  Return the angle to the specified point.
    REAL GetAngleTo(const TGeoPoint& point) const;

  private:

    //  Each point has an x-coordinate.
    REAL fX;

    //  Each point has an y-coordinate.
    REAL fY;
};


#endif // TRANSIMS_NET_POINT
```

2. Point.C

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: Point.C,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 19:29:51 $
//  $State: Rel $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


//  Include Standard C header files.
#include <math.h>


//  Include TRANSIMS header files.
#include <NET/Point.h>


//  Construct a point with the given x- and y-coordinates.
TGeoPoint::TGeoPoint(REAL x, REAL y)
  : fX(x),
    fY(y)
{
}


//  Return the x-coordinate.
REAL TGeoPoint::GetX() const
{
    return fX;
}


//  Return the y-coordinate.
REAL TGeoPoint::GetY() const
{
```

```
    return fY;
}


//  Return the angle to the specified point.
REAL TGeoPoint::GetAngleTo(const TGeoPoint& point) const
{
    const REAL dx = point.fX - fX;
    const REAL dy = point.fY - fY;
    return atan2(dy, dx);
}
```

BB. TNetReader Class

1. Reader.h

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: Reader.h,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 19:29:51 $
//  $State: Rel $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


#ifndef TRANSIMS_NET_READER
#define TRANSIMS_NET_READER


//  Include TRANSIMS header files.
#include <GBL/Globals.h>
#include <DBS/Table.h>


//  Forward declarations.
class TDbTable;


//  A network reader reads a network from the database.
class TNetReader
{
  public:

    //  Construct a reader for the specified tables.
    TNetReader(TDbTable nodeTable, TDbTable linkTable, TDbTable pocketTable,
            TDbTable parkingTable, TDbTable connTable, TDbTable unsigTable,
            TDbTable sigTable, TDbTable phaseTable, TDbTable timeTable);

    //  Return the node table.
    TDbTable& GetNodeTable();

    //  Return the link table.
    TDbTable& GetLinkTable();

    //  Return the pocket table.
    TDbTable& GetPocketTable();

    //  Return the parking table.
    TDbTable& GetParkingTable();

    //  Return the lane connectivity table.
    TDbTable& GetLaneConnectivityTable();

    //  Return the unsignalized control table.
    TDbTable& GetUnsignalizedControlTable();

    //  Return the signalized control table.
    TDbTable& GetSignalizedControlTable();

    //  Return the phasing plan table.
    TDbTable& GetPhasingPlanTable();
```

```
    //  Return the timing plan table.
    TDbTable& GetTimingPlanTable();

  private:

    //  Each reader has a node table.
    TDbTable fNodeTable;

    //  Each reader has a link table.
    TDbTable fLinkTable;

    //  Each reader has a pocket table.
    TDbTable fPocketTable;

    //  Each reader has a parking table.
    TDbTable fParkingTable;

    //  Each reader has a lane connectivity table.
    TDbTable fLaneConnectivityTable;

    //  Each reader has an unsignalized control table.
    TDbTable fUnsignalizedControlTable;

    //  Each reader has a signalized control table.
    TDbTable fSignalizedControlTable;

    //  Each reader has a phasing plan table.
    TDbTable fPhasingPlanTable;

    //  Each reader has a timing plan table.
    TDbTable fTimingPlanTable;
};


#endif // TRANSIMS_NET_READER
```

2. Reader.C

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: Reader.C,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 19:29:51 $
//  $State: Rel $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


//  Include TRANSIMS header files.
#include <NET/Reader.h>


//  Construct a reader for the specified tables.
TNetReader::TNetReader(TDbTable nodeTable, TDbTable linkTable, TDbTable
        pocketTable, TDbTable parkingTable, TDbTable connTable, TDbTable
        unsigTable, TDbTable sigTable, TDbTable phaseTable, TDbTable timeTable)
  : fNodeTable(nodeTable),
    fLinkTable(linkTable),
    fPocketTable(pocketTable),
    fParkingTable(parkingTable),
    fLaneConnectivityTable(connTable),
    fUnsignalizedControlTable(unsigTable),
    fSignalizedControlTable(sigTable),
    fPhasingPlanTable(phaseTable),
    fTimingPlanTable(timeTable)
{
}

//  Return the node table.
TDbTable& TNetReader::GetNodeTable()
{
    return fNodeTable;
}
```

```
//  Return the link table.
TDbTable& TNetReader::GetLinkTable()
{
    return fLinkTable;
}


//  Return the pocket table.
TDbTable& TNetReader::GetPocketTable()
{
    return fPocketTable;
}


//  Return the parking table.
TDbTable& TNetReader::GetParkingTable()
{
    return fParkingTable;
}


//  Return the lane connectivity table.
TDbTable& TNetReader::GetLaneConnectivityTable()
{
    return fLaneConnectivityTable;
}


//  Return the unsignalized control table.
TDbTable& TNetReader::GetUnsignalizedControlTable()
{
    return fUnsignalizedControlTable;
}


//  Return the signalized control table.
TDbTable& TNetReader::GetSignalizedControlTable()
{
    return fSignalizedControlTable;
}


//  Return the phasing plan table.
TDbTable& TNetReader::GetPhasingPlanTable()
{
    return fPhasingPlanTable;
}


//  Return the timing plan table.
TDbTable& TNetReader::GetTimingPlanTable()
{
    return fTimingPlanTable;
}
```

CC. TGeoRectangle Class

1. Rectangle.h

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: Rectangle.h,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 19:29:51 $
//  $State: Rel $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


#ifndef TRANSIMS_NET_RECTANGLE
#define TRANSIMS_NET_RECTANGLE
```

```
//  Include TRANSIMS header files.
#include <GBL/Globals.h>
#include <NET/Point.h>


//  A geographic rectangle is a rectangle in a map coordinate system.
class TGeoRectangle
{
  public:

    //  Construct a rectangle with the given corners.
    TGeoRectangle(const TGeoPoint& corner1, const TGeoPoint& corner2);

    //  Construct a copy of the given rectangle.
//  TGeoRectangle(const TGeoRectangle& rectangle);

    //  Make the rectangle a copy of the given rectangle.
//  TGeoRectangle& operator=(const TGeoRectangle& rectangle);

    //  Return the corners.
    void GetCorners(TGeoPoint& corner1, TGeoPoint& corner2) const;

    //  Return whether the rectangle contains the given point.
    bool Contains(const TGeoPoint& point) const;

  private:

    //  Each rectangle has a minimum corner.
    TGeoPoint fMinimum;

    //  Each rectangle has a maximum corner.
    TGeoPoint fMaximum;
};


#endif // TRANSIMS_NET_RECTANGLE
```

2. Rectangle.C

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: Rectangle.C,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 19:29:51 $
//  $State: Rel $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


//  Include TRANSIMS header files.
#include <NET/Rectangle.h>


//  Define minimum and maximum functions.
inline static REAL Minimum(REAL a, REAL b) {return a < b ? a : b;}
inline static REAL Maximum(REAL a, REAL b) {return a > b ? a : b;}


//  Construct a rectangle with the given corners.
TGeoRectangle::TGeoRectangle(const TGeoPoint& corner1, const TGeoPoint&
        corner2)
  : fMinimum(Minimum(corner1.GetX(), corner2.GetX()), Minimum(corner1.GetY(),
            corner2.GetY())),
    fMaximum(Maximum(corner1.GetX(), corner2.GetX()), Maximum(corner1.GetY(),
            corner2.GetY()))
{
}


//  Return the corners.
void TGeoRectangle::GetCorners(TGeoPoint& corner1, TGeoPoint& corner2) const
{
    corner1 = fMinimum;
```

```
    corner2 = fMaximum;
}


//  Return whether the rectangle contains the given point.
bool TGeoRectangle::Contains(const TGeoPoint& point) const
{
    return point.GetX() >= fMinimum.GetX() && point.GetX() <= fMaximum.GetX() &&
            point.GetY() >= fMinimum.GetY() && point.GetY() <= fMaximum.GetY();
}
```

DD. TNetSignalCoordinator Class

1. SignalCoordinator.h

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: SignalCoordinator.h,v $
//  $Revision: 2.3 $
//  $Date: 1996/11/08 18:28:56 $
//  $State: Stab $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


#ifndef TRANSIMS_NET_SIGNALCOORDINATOR
#define TRANSIMS_NET_SIGNALCOORDINATOR


//  Include TRANSIMS header files.
#include "GBL/Globals.h"
#include "NET/Id.h"


//  Include Booch Components header files.
#include "BCStoreM.h"
#include "BCCollU.h"
#include "BCSetU.h"
#include "BCMapU.h"


//  Forward declarations.
class TNetSignalizedControl;
class TNetPhasingPlan;
class TNetSensor;
class TNetPhasingPlanSchedule;
class TNetTimingPlanReader;
class TNetPhasingPlanReader;
class TNetNetwork;


//  A signal coordinator coordinates the operation of several traffic signals.
class TNetSignalCoordinator
{
  public:

    //  Type definitions.
    typedef BC_TUnboundedCollection<TNetSignalizedControl*, BC_CManaged>
            ControllerCollection;
    typedef BC_TCollectionActiveIterator<TNetSignalizedControl*>
            ControllerCollectionIterator;
    typedef BC_TUnboundedSet<TNetPhasingPlan*, 4U, BC_CManaged> PhasingPlanSet;
    typedef BC_TSetActiveIterator<TNetPhasingPlan*> PhasingPlanSetIterator;
    typedef BC_TUnboundedMap<NetPhaseNumber,
            BC_TUnboundedCollection<NetPhaseNumber, BC_CManaged>, 1U,
            BC_CManaged> PhaseNumberMap;

    //  Construct a signalized control coordinator.
    TNetSignalCoordinator();
    TNetSignalCoordinator(NetCoordinatorId id);
    TNetSignalCoordinator(const TNetSignalCoordinator& coordinator);

    //  Destroy a coordinator.
```

```
    virtual ~TNetSignalCoordinator();

    //  Coordinate signal controls when necessary and then call the
    //  UpdateSignalizedControl method for each of the controllers.
    virtual void UpdateSignalizedControl(REAL sim_time) = 0;

    //  Return the coordinator's phasing plans.
    PhasingPlanSet& GetPhasingPlans();
    const PhasingPlanSet& GetPhasingPlans() const;

    //  Define the phasing plan.
    TNetPhasingPlan& CreatePhasingPlan(NetNodeId node, NetPlanId phasing,
            TNetTimingPlanReader& timing, TNetPhasingPlanReader& reader,
            TNetNetwork& network, PhaseNumberMap& numbers);

    //  Return the phasing plan with specified id.
    TNetPhasingPlan& GetPhasingPlan(NetPlanId id);
    const TNetPhasingPlan& GetPhasingPlan(NetPlanId id) const;

    //  Return the signalized controls coordinated by this coordinator.
    ControllerCollection& GetControllers();
    const ControllerCollection& GetControllers() const;

    //  Define a controller.
    void SetController(TNetSignalizedControl& controller);

    //  Return the coordinator's id.
    NetCoordinatorId GetId() const;

  protected:

    //  Assign a signal coordinator.
    TNetSignalCoordinator& operator=(const TNetSignalCoordinator&);

  private:

    //  Each coordinator has a unique id.
    NetCoordinatorId fId;

    //  A signalized control coordinator has a group of associated sensors.
    //BC_TUnboundedSet<const TNetSensor*, 16U, BC_CManaged> fSensors;

    //  A signalized control coordinator controls one or more signalized
    //  controllers.
    ControllerCollection fControllers;

    //  A signalized control coordinator has a group of phasing plans that it
    //  can tell its controllers to use.
    PhasingPlanSet fPhasingPlans;
};


#endif // TRANSIMS_NET_SIGNALCOORDINATOR
```

2. SignalCoordinator.C

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: SignalCoordinator.C,v $
//  $Revision: 2.3 $
//  $Date: 1996/11/08 18:28:56 $
//  $State: Stab $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


//  Include TRANSIMS header files.
#include "NET/SignalCoordinator.h"
#include "NET/Network.h"
#include "NET/Node.h"
#include "NET/SignalizedControl.h"
#include "NET/PhasingPlan.h"
#include "NET/PhaseDescription.h"
#include "NET/Exception.h"
```

```
#include "NET/TimingPlanReader.h"
#include "NET/PhasingPlanReader.h"


//  Include standard C++ header files.
#include <strstream.h>


//  Define the phasing plan hash value.
static BC_Index PlanHashValue (TNetPhasingPlan* const & p)
{
    return ((int)p & 0xf0) >> 4;
}


//  Define the timing plan hash value.
static BC_Index PhaseHashValue (const NetPhaseNumber& n)
{
    return BC_Index(n);
}


//  Construct a signalized control coordinator.
TNetSignalCoordinator::TNetSignalCoordinator()
  : fId(0),
    fControllers(),
    fPhasingPlans(PlanHashValue)
{
}

TNetSignalCoordinator::TNetSignalCoordinator(NetCoordinatorId id)
  : fId(id),
    fControllers(),
    fPhasingPlans(PlanHashValue)
{
}

TNetSignalCoordinator::TNetSignalCoordinator(const TNetSignalCoordinator& c)
  : fControllers(),
    fPhasingPlans(PlanHashValue)
{
    fId = c.fId;
    fControllers = c.fControllers;
    fPhasingPlans = c.fPhasingPlans;
}


//  Destroy a signal coordinator.
TNetSignalCoordinator::~TNetSignalCoordinator()
{
    for (PhasingPlanSetIterator it(fPhasingPlans); !it.IsDone(); it.Next())
        delete *it.CurrentItem();
}


//  Assign a signal coordinator.
TNetSignalCoordinator& TNetSignalCoordinator::operator=(const
        TNetSignalCoordinator& c)
{
    if (this == &c) return *this;
    fId = c.fId;
    fControllers = c.fControllers;
    fPhasingPlans = c.fPhasingPlans;
    return *this;
}


//  Return the coordinator’s phasing plans.
TNetSignalCoordinator::PhasingPlanSet& TNetSignalCoordinator::GetPhasingPlans()
{
    return fPhasingPlans;
}

const TNetSignalCoordinator::PhasingPlanSet&
        TNetSignalCoordinator::GetPhasingPlans() const
{
```

```
    return fPhasingPlans;
}


//  Define a phasing plan.
TNetPhasingPlan& TNetSignalCoordinator::CreatePhasingPlan(NetNodeId nodeid,
        NetPlanId planid, TNetTimingPlanReader& timingReader,
        TNetPhasingPlanReader& phasingReader, TNetNetwork& network,
        PhaseNumberMap& nextPhases)
{
    bool found = FALSE;
    BC_TUnboundedMap<NetPhaseNumber, TNetPhaseDescription*, 2U, BC_CManaged>
            pdmap(PhaseHashValue);

    TNetPhasingPlan *plan = new TNetPhasingPlan(planid);
    fPhasingPlans.Add(plan);
    for (timingReader.Reset(); timingReader.MorePlans();
            timingReader.GetNextPlan()) {
        if (timingReader.GetPlan() == planid) {
            found = TRUE;
            NetPhaseNumber phase = timingReader.GetPhase();
            string next = timingReader.GetNextPhases();
            REAL grmin = timingReader.GetGreenMin();
            REAL grmax = timingReader.GetGreenMax();
            REAL grext = timingReader.GetGreenExt();
            REAL yellow = timingReader.GetYellow();
            REAL red = timingReader.GetRedClear();
            TNetPhaseDescription& desc = plan->CreatePhaseDescription(phase);
            desc.SetLengths (grmin, grmax, grext, yellow, red);
            pdmap.Bind (phase, &desc);
            // parse next string and store values in a collection
            BC_TUnboundedCollection<NetPhaseNumber, BC_CManaged> phases;
            NetPhaseNumber n;
            while (next.contains("/")){
                string str = next;
                str = str.remove(str.index("/")); // truncate after first slash
                istrstream (str, sizeof(str)) >> n;
                phases.Append(n);
                next = next.remove(0,next.index("/")+1); // delete up to first slash
            }
            istrstream (next, sizeof(next)) >> n;
            phases.Append(n);
            nextPhases.Bind (phase, phases);
        }
//ISSUE(kpb): Can't rely on records being in order.
//        else
//        {
//            if (found) break;
//        }
    }

    found = FALSE;
    for (phasingReader.Reset(); phasingReader.MoreNodes();
            phasingReader.GetNextNode()) {
        if (phasingReader.GetNode() == nodeid && phasingReader.GetPlan() ==
                planid) {
            found = TRUE;
            NetPhaseNumber phase = phasingReader.GetPhase();
            TNetLink* inlink =
                    *(network.GetLinks().ValueOf(phasingReader.GetInlink()));
            TNetLink* outlink =
                    *(network.GetLinks().ValueOf(phasingReader.GetOutlink()));
            string prot = phasingReader.GetProtection();
            TNetPhaseDescription* desc;
            if (pdmap.IsBound(phase))
                desc = *pdmap.ValueOf(phase);
            else
                throw TNetNotFound ("Invalid phase number");
            if (prot == "P")
                desc->SetLinkMovements (*inlink, *outlink,
                        TNetPhaseDescription::kProtected);
            else if (prot == "U")
                desc->SetLinkMovements (*inlink, *outlink,
                        TNetPhaseDescription::kPermitted);
            else
                throw TNetNotFound ("Invalid value for protection code.");
```

```
        }
//ISSUE(kpb): Can’t rely on records being in order.
//        else
//        {
//            if (found) break;
//        }
    }
    return found ? *plan : *((TNetPhasingPlan*) NULL);
}


//  Return phasing plan with specified id.
TNetPhasingPlan& TNetSignalCoordinator::GetPhasingPlan(NetPlanId id)
{
    for (PhasingPlanSetIterator it(fPhasingPlans); !it.IsDone(); it.Next())
        if ((*it.CurrentItem())->GetId() == id)
            return **it.CurrentItem();
}

const TNetPhasingPlan& TNetSignalCoordinator::GetPhasingPlan(NetPlanId id) const
{
    for (PhasingPlanSetIterator it(fPhasingPlans); !it.IsDone(); it.Next())
        if ((*it.CurrentItem())->GetId() == id)
            return **it.CurrentItem();
}


//  Return the signalized controls coordinated by this coordinator.
TNetSignalCoordinator::ControllerCollection&
        TNetSignalCoordinator::GetControllers()
{
    return fControllers;
}

const TNetSignalCoordinator::ControllerCollection&
        TNetSignalCoordinator::GetControllers() const
{
    return fControllers;
}


//  Define a controller.
void TNetSignalCoordinator::SetController(TNetSignalizedControl& controller)
{
    if (fControllers.Location(&controller) == -1)
        fControllers.Append(&controller);
    if (&controller.GetCoordinator() != this)
        controller.SetCoordinator(*this);
}


//  Return the coordinator’s id.
NetCoordinatorId TNetSignalCoordinator::GetId() const
{
    return fId;
}
```

EE. TNetSignalizedControl Class

1. SignalizedControl.h

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: SignalizedControl.h,v $
//  $Revision: 2.5 $
//  $Date: 1996/12/13 15:27:11 $
//  $State: Stab $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


#ifndef TRANSIMS_NET_SIGNALIZEDCONTROL
#define TRANSIMS_NET_SIGNALIZEDCONTROL
```

```
//  Include TRANSIMS header files.
#include "GBL/Globals.h"
#include "NET/TrafficControl.h"
#include "NET/PhaseDescription.h"


//  Include Booch Components header files.
#include "BCStoreM.h"
#include "BCCollU.h"
#include "BCSetU.h"


//  Forward declarations.
class TNetNode;
class TNetLink;
class TNetLane;
class TNetPhase;
class TNetPhasingPlan;
class TNetSignalCoordinator;
class TNetSignalizedControlReader;


//  A signalized control specifies the signal phases and phasing plan at a node.
class TNetSignalizedControl
  : public TNetTrafficControl
{
  public:

    //  Type definitions.
    typedef BC_TUnboundedCollection<TNetPhase*, BC_CManaged> PhaseCollection;
    typedef BC_TCollectionActiveIterator<TNetPhase*> PhaseCollectionIterator;

    //  Construct a signalized traffic control.
    TNetSignalizedControl();
    TNetSignalizedControl(TNetNode& node);
    TNetSignalizedControl(const TNetSignalizedControl& control);

    //  Construct a signalized traffic control using the reader.
    TNetSignalizedControl(TNetSignalizedControlReader& reader, TNetNetwork&
            network);

    //  Destroy a signalized traffic control.
    virtual ~TNetSignalizedControl();

    //  Return the lanes on next link to which transition from specified lane on
    //  this link can be made.  Return an empty collection if transition is not
    //  possible.
    virtual void AllowedMovements(LaneCollection& lanes, const TNetLink&
        fromlink, const TNetLane& fromlane, const TNetLink& tolink);

    //  Return the lanes on this link from which transition to specified lane on
    //  next link can be made.  Return an empty collection if transition is not
    //  possible.
    virtual void AllowedMovements(LaneCollection& lanes, const TNetLink&
            fromlink, const TNetLink& tolink, const TNetLane& tolane);

    //  Return the lanes ordered from median that may be used to transition
    //  from current link to next link.
    virtual void AllowedMovements(LaneCollection& lanes, const TNetLink&
            fromlink, const TNetLink& tolink, bool phase = FALSE);

    //  Return the lanes that must be examined for interference when
    //  transitioning from current lane to next lane.
    virtual void InterferingLanes(LaneCollection& lanes, const TNetLane&
            fromlane, const TNetLane& tolane, bool phase = FALSE);

    //  Return the vehicle control signal for the specified lane.
    virtual TNetTrafficControl::ETrafficControl GetVehicleControl(const
            TNetLane&) const;

    //  Update the traffic control state based on current simulation time.
    virtual void UpdateSignalizedControl (REAL sim_time) = 0;

    //  Return the current phasing plan.
```

```
    virtual TNetPhasingPlan& GetPhasingPlan();
    virtual const TNetPhasingPlan& GetPhasingPlan() const;

    //  Return the current phasing plan offset.
    virtual REAL GetPhasingPlanOffset() const;

    //  Define the phasing plan.
    virtual void SetPhasingPlan(TNetPhasingPlan& plan);

    //  Define the phasing plan offset.
    virtual void SetPhasingPlanOffset(REAL offset);

    //  Return the signalized control coordinator for this signal.
    virtual TNetSignalCoordinator& GetCoordinator();
    virtual const TNetSignalCoordinator& GetCoordinator() const;

    //  Define the signalized control coordinator for this signal.
    virtual void SetCoordinator(TNetSignalCoordinator& coordinator);

    //  Create a phase.
    virtual TNetPhase& CreatePhase(TNetPhaseDescription& description);

    //  Return the phases for this signal.
    virtual PhaseCollection& GetPhases();
    virtual const PhaseCollection& GetPhases() const;

    //  Return the current phase.
    virtual TNetPhase& GetPhase();
    virtual const TNetPhase& GetPhase() const;

    //  Update the current phase.
    virtual void SetPhase(TNetPhase& phase);

    //  Initialize the signal to specified phase.
    virtual void InitPhase(TNetPhase& phase);

  protected:

    //  Type definitions.
    typedef BC_TUnboundedSet<TNetLink*, 4U, BC_CManaged> LinkSet;
    typedef BC_TSetActiveIterator<TNetLink*> LinkSetIterator;
    typedef BC_TUnboundedSet<TNetPhaseDescription::EProtection, 1U, BC_CManaged>
            ProtectionSet;
    typedef BC_TSetActiveIterator<TNetPhaseDescription::EProtection>
            ProtectionSetIterator;

    //  Assign a signalized traffic control.
    TNetSignalizedControl& operator=(const TNetSignalizedControl&);

  private:

    //  A signalized control has a sequence of phases.
    PhaseCollection fPhases;

    //  A signalized controller has a phasing plan.
    TNetPhasingPlan* fPhasingPlan;

    //  A signalized controller may have a non-zero phasing plan offset.
    REAL fPhasingPlanOffset;

    //  A signalized controller knows its coordinator.
    TNetSignalCoordinator* fCoordinator;

    //  Current phase.
    TNetPhase* fCurrentPhase;
};


#endif // TRANSIMS_NET_SIGNALIZEDCONTROL
```

2. SignalizedControl.C

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: SignalizedControl.C,v $
```

```
//  $Revision: 2.7 $
//  $Date: 1996/12/13 15:27:11 $
//  $State: Stab $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


//  Include standard C header files.
#include <math.h>


//  Include TRANSIMS header files.
#include "NET/SignalizedControl.h"
#include "NET/Network.h"
#include "NET/Node.h"
#include "NET/Phase.h"
#include "NET/PhasingPlan.h"
#include "NET/SignalCoordinator.h"
#include "NET/Link.h"
#include "NET/Lane.h"
#include "NET/Exception.h"
#include "NET/SignalizedControlReader.h"


//  Construct a signalized traffic control.
TNetSignalizedControl::TNetSignalizedControl()
  : TNetTrafficControl(),
    fPhases(),
    fPhasingPlan((TNetPhasingPlan*)NULL),
    fPhasingPlanOffset(0.0),
    fCoordinator((TNetSignalCoordinator*)NULL),
    fCurrentPhase((TNetPhase*)NULL)
{
}

TNetSignalizedControl::TNetSignalizedControl(TNetNode& node)
  : TNetTrafficControl(node),
    fPhases(),
    fPhasingPlan((TNetPhasingPlan*)NULL),
    fPhasingPlanOffset(0.0),
    fCoordinator((TNetSignalCoordinator*)NULL),
    fCurrentPhase((TNetPhase*)NULL)
{
}

TNetSignalizedControl::TNetSignalizedControl(const TNetSignalizedControl& c)
  : TNetTrafficControl(c),
    fPhases()
{
    fPhases = c.fPhases;
    fPhasingPlan = c.fPhasingPlan;
    fPhasingPlanOffset = c.fPhasingPlanOffset;
    fCoordinator = c.fCoordinator;
    fCurrentPhase = c.fCurrentPhase;
}


//  Construct a signalized traffic control using the reader.
TNetSignalizedControl::TNetSignalizedControl(TNetSignalizedControlReader&
        reader, TNetNetwork& network)
  : TNetTrafficControl(**network.GetNodes().ValueOf(reader.GetNode())),
    fPhases(),
    fPhasingPlan((TNetPhasingPlan*)NULL),
    fPhasingPlanOffset(0.0),
    fCoordinator((TNetSignalCoordinator*)NULL),
    fCurrentPhase((TNetPhase*)NULL)
{
}


//  Destroy a signalized traffic control.
TNetSignalizedControl::~TNetSignalizedControl()
{
    for (PhaseCollectionIterator it(fPhases); !it.IsDone(); it.Next())
        delete *it.CurrentItem();
```

```
}


//  Assign a signalized traffic control.
TNetSignalizedControl& TNetSignalizedControl::operator=(const
        TNetSignalizedControl& c)
{
    if (this == &c)
            return *this;
    TNetTrafficControl::operator =(c);
    fPhases = c.fPhases;
    fPhasingPlan = c.fPhasingPlan;
    fPhasingPlanOffset = c.fPhasingPlanOffset;
    fCoordinator = c.fCoordinator;
    fCurrentPhase = c.fCurrentPhase;
    return *this;
}


//  Define the hash function for links.
static BC_Index LinkHashValue(TNetLink* const & l)
{
    return ((int)l & 0xf0) >> 4;
}


//  Define the hash function for protections.
static BC_Index ProtHashValue (TNetPhaseDescription::EProtection const& /*p*/)
{
    return 1;
}


//  Return the lanes on next link to which transition from specified lane on
//  this link can be made.  Return an empty collection if transition is not
//  possible.
void TNetSignalizedControl::AllowedMovements(LaneCollection& tolanes,  const
        TNetLink& fromlink, const TNetLane& fromlane, const TNetLink& tolink)
{
    TNetTrafficControl::AllowedMovements (tolanes, fromlink, fromlane, tolink);
}


//  Return the lanes on this link from which transition to specified lane on
//  next link can be made.  Return an empty collection if transition is not
//  possible.
void TNetSignalizedControl::AllowedMovements (LaneCollection& fromlanes, const
        TNetLink& fromlink, const TNetLink& tolink, const TNetLane& tolane)
{
    TNetTrafficControl::AllowedMovements (fromlanes, fromlink, tolink, tolane);
}


//  Return the lanes ordered from median that may be used to transition
//  from current link to next link.
void TNetSignalizedControl::AllowedMovements (LaneCollection& fromlanes, const
        TNetLink& fromlink, const TNetLink& tolink, bool phase)
{
    LaneCollectionIterator itto(tolink.GetLanesFrom(GetNode(),TRUE));

    fromlanes.Clear();
    if (phase) {
        TNetPhaseDescription::LinkProtectionMap& prot = GetPhase().
            GetPhaseDescription().GetLinkMovements(fromlink);
        if (prot.IsEmpty() || !prot.IsBound((TNetLink*) &tolink))
            return; // no movement from fromlink to tolink during this phase
    }

    for (LaneCollectionIterator itfrom(fromlink.GetLanesTowards(GetNode(),TRUE));
            !itfrom.IsDone(); itfrom.Next()) {
        TNetLane* current = *itfrom.CurrentItem();
        if (GetConnectivity().IsBound (current)) {
            ConnectedCollection& lanes = *GetConnectivity().ValueOf (current);
            for (itto.Reset(); !itto.IsDone(); itto.Next())
                if (lanes.Location (*itto.CurrentItem()) != -1  &&
                      fromlanes.Location(current) == -1)
```

```
                    fromlanes.Append (current);
        }
    }
}


//  Return the lanes that must be examined for interference when transitioning
//  from current lane to next lane.
void TNetSignalizedControl::InterferingLanes(LaneCollection& fromlanes, const
        TNetLane& fromlane, const TNetLane& tolane, bool phase)
{
    fromlanes.Clear();
    bool found = FALSE;
    if (GetConnectivity().IsBound ((TNetLane*)&fromlane))
        for (LaneCollectionIterator
                itl(*GetConnectivity().ValueOf((TNetLane*)&fromlane));
                !itl.IsDone(); itl.Next())
            if (*itl.CurrentItem() == &tolane) {
                found = TRUE;
                break;
            }
    if (!found)
        return; // no path from fromlane to tolane

    const TNetLink *fromlink = &fromlane.GetLink();
    const TNetLink *tolink = &tolane.GetLink();
    if (phase) {
        TNetPhaseDescription::LinkProtectionMap& prot = GetPhase().
            GetPhaseDescription().GetLinkMovements(*fromlink);
        if (prot.IsEmpty() || !prot.IsBound((TNetLink*)tolink))
            return; // movement not allowed
        else if (prot.IsBound((TNetLink*)tolink) &&
                *prot.ValueOf((TNetLink*)tolink) ==
                TNetPhaseDescription::kProtected)
            return; // movement protected
        else
            ; // movement unprotected
    }

    TNetNode::LinkRing& links = GetNode().GetLinks();
    while (links.Top() != fromlink)
        links.Rotate();

    //  Can rotate in either direction first, but more efficient to rotate in
    //  shortest direction first.  dir is shortest direction between fromlink
    //  and tolink; opsdir is longest (opposite) direction.
    BC_Direction dir, opsdir;
    REAL angle1 = fromlink->GetAngle(GetNode());
    if (angle1 < 0.0)
        angle1 += 2.0 * M_PI;
    REAL angle2 = tolink->GetAngle(GetNode());
    if (angle2 < 0.0)
        angle2 += 2.0 * M_PI;
    if (angle2 < angle1)
        angle2 += 2.0 * M_PI;
    REAL angle = angle2 - angle1;
    if (angle <= 3.66) { // == 210 deg.  (Use 180 deg. minimim; a slightly
                         // larger angle may be more efficient for intersections
                         // that are not quite square.)
        dir = BC_kReverse;
        opsdir = BC_kForward;
    } else {
        dir = BC_kForward;
        opsdir = BC_kReverse;
    }

    ConnectedCollection tolanes;
    ConnectedCollection frlanes;
    if (dir == BC_kForward) {
        //  tolanes includes lanes leaving on fromlink
        for (LaneCollectionIterator itto(links.Top()->GetLanesFrom(GetNode(),
                TRUE)); !itto.IsDone(); itto.Next())
            tolanes.Append(*itto.CurrentItem());
    }
    links.Rotate(dir);
    while (links.Top() != tolink) {
```

```
        //  tolanes includes lanes leaving on links between fromlink and tolink
        for (LaneCollectionIterator itto(links.Top()->GetLanesFrom(GetNode(),
                TRUE)); !itto.IsDone(); itto.Next())
            tolanes.Append(*itto.CurrentItem());
        //  frlanes includes lanes arriving on links between fromlink and tolink
        //  if movements are allowed from the link during this phase or if phase
        //  is ignored
        if (!phase || !GetPhase().GetPhaseDescription()
                .GetLinkMovements(*links.Top()).IsEmpty()) {
            for (LaneCollectionIterator
                    itfr(links.Top()->GetLanesTowards(GetNode(), TRUE));
                    !itfr.IsDone(); itfr.Next())
                frlanes.Append(*itfr.CurrentItem());
        }
        links.Rotate(dir);
    }
    if (dir == BC_kForward) {
        //  frlanes includes lanes arriving on tolink if movements are allowed
        //  from this link during this phase or if phase is ignored
        if (!phase || !GetPhase().GetPhaseDescription()
                .GetLinkMovements(*links.Top()).IsEmpty()) {
            for (LaneCollectionIterator
                    itfr(links.Top()->GetLanesTowards(GetNode(), TRUE));
                    !itfr.IsDone(); itfr.Next())
                frlanes.Append(*itfr.CurrentItem());
        }
    }

    //  include lanes leaving tolink up to tolane in tolanes
    int index = tolink->GetLanesFrom(GetNode(), TRUE).Location((TNetLane*)
            &tolane);
    int i=0;
    for (LaneCollectionIterator itto(tolink->GetLanesFrom(GetNode(),TRUE));
        !itto.IsDone(); itto.Next()) {
        if (dir == BC_kForward && i < index)
            tolanes.Append(*itto.CurrentItem());
        else if (dir == BC_kReverse && i > index)
            tolanes.Append(*itto.CurrentItem());
        else
            ; //  do nothing
        ++i;
    }

    for (LaneCollectionIterator itf1(frlanes); !itf1.IsDone(); itf1.Next()) {
        TNetLane* current = *itf1.CurrentItem();
        TNetPhaseDescription::LinkProtectionMap& prot = GetPhase().
                GetPhaseDescription().GetLinkMovements((current)->GetLink());
        if (GetConnectivity().IsBound (current))
            //  fromlanes includes frlanes having connectivity to any lane not
            //  in tolanes if movements are allowed to the link during this
            //  phase or if phase is ignored
            for (LaneCollectionIterator
                    itl(*GetConnectivity().ValueOf(current)); !itl.IsDone();
                    itl.Next())
                if (tolanes.Location (*itl.CurrentItem()) == -1)
                    if (!phase ||
                            prot.IsBound(&(*itl.CurrentItem())->GetLink())) {
                        fromlanes.Append (current);
                        break;
                    }
    }

    //  include tolane in tolanes
    tolanes.Append((TNetLane*)&tolane);
    while (links.Top() != fromlink)
        links.Rotate(opsdir);
    links.Rotate(opsdir);

    frlanes.Clear();
    while (links.Top() != tolink) {
        //  new frlanes includes lanes arriving on links between fromlink and
        //  tolink in opposite direction if movements are allowed from the link
        //  during this phase or if phase is ignored
        if (!phase || !GetPhase().GetPhaseDescription()
                .GetLinkMovements(*links.Top()).IsEmpty()) {
            for (LaneCollectionIterator
```

```
                    itfr(links.Top()->GetLanesTowards(GetNode(),TRUE));
                    !itfr.IsDone(); itfr.Next())
                frlanes.Append(*itfr.CurrentItem());
        }
        links.Rotate(opsdir);
    }
    if (opsdir == BC_kForward) {
        //  frlanes includes lanes arriving on tolink if movements are allowed
        //  from this link during this phase or if phase is ignored
        if (!phase || !GetPhase().GetPhaseDescription()
                .GetLinkMovements(*links.Top()).IsEmpty())
            for (LaneCollectionIterator
                    itfr(links.Top()->GetLanesTowards(GetNode(),TRUE));
                    !itfr.IsDone(); itfr.Next())
                frlanes.Append(*itfr.CurrentItem());
    }

    for (LaneCollectionIterator itf2(frlanes); !itf2.IsDone(); itf2.Next()) {
        TNetLane* current = *itf2.CurrentItem();
        TNetPhaseDescription::LinkProtectionMap& prot = GetPhase().
                GetPhaseDescription().GetLinkMovements((current)->GetLink());
        if (GetConnectivity().IsBound (current))
            //  fromlanes includes frlanes having connectivity to any lane in
            //  tolanes if movements are allowed to the link during this phase
            //  or if phase is ignored
            for (LaneCollectionIterator
                    itl(*GetConnectivity().ValueOf(current)); !itl.IsDone();
                    itl.Next())
                if (tolanes.Location (*itl.CurrentItem()) != -1 &&
                        fromlanes.Location(current) == -1)
                    if (!phase ||
                            prot.IsBound(&(*itl.CurrentItem())->GetLink())) {
                        fromlanes.Append (current);
                        break;
                    }
    }
}


//  Return the vehicle control signal for the specified lane.
TNetTrafficControl::ETrafficControl
        TNetSignalizedControl::GetVehicleControl(const TNetLane& lane) const
{
    LinkSet tolinks(LinkHashValue);
    ProtectionSet prot(ProtHashValue);
    if (!GetConnectivity().IsBound((TNetLane*)&lane))
        return TNetTrafficControl::kWait;  // no place to go
    for (TNetTrafficControl::LaneCollectionIterator itc(GetConnectivity(lane));
            !itc.IsDone(); itc.Next())
        tolinks.Add (&(*itc.CurrentItem())->GetLink());

    const TNetPhaseDescription::LinkProtectionMap& protMap =
            GetPhase().GetPhaseDescription().GetLinkMovements(lane.GetLink());
    for (TNetPhaseDescription::LinkProtectionMapIterator itl(protMap);
            !itl.IsDone(); itl.Next())
        if (tolinks.IsMember(*itl.CurrentItem())) {
            //  don't include protection for right turn on red
            REAL angle1 = lane.GetLink().GetAngle(GetNode());
            if (angle1 < 0.0)
                angle1 += 2.0*M_PI;
            REAL angle2 = (*itl.CurrentItem())->GetAngle(GetNode());
            if (angle2 < 0.0)
                angle2 += 2.0*M_PI;
            if (angle2 < angle1)
                angle2 += 2.0*M_PI;
            REAL angle = angle2 - angle1;
            if (angle < 0.01 || angle > 2.35 || *itl.CurrentValue() ==
                    TNetPhaseDescription::kProtected)
                prot.Add (*itl.CurrentValue());
        }

    if (prot.Extent() == 0)
        return TNetTrafficControl::kWait;  // no movement from this lane during
                                           // this phase
    else {
        TNetPhase::EInterval intv = GetPhase().GetInterval();
```

```
        if (intv == TNetPhase::kGreen)
            //  when protected and permitted movements allowed, return permitted
            if (prot.IsMember(TNetPhaseDescription::kPermitted))
                return TNetTrafficControl::kPermitted;
            else if (prot.IsMember(TNetPhaseDescription::kProtected))
                   return TNetTrafficControl::kProtected;
            else
                throw TNetException ("Invalid protection code found.");

        //  when interval is yellow or red and same movement is allowed in next phase
        //  return the appropriate green signal
        bool possible = FALSE;
        bool mismatch = FALSE;
        for (TNetPhase::PhaseCollectionIterator itn(GetPhase().GetNextPhases());
                !itn.IsDone(); itn.Next())
            for (TNetPhaseDescription::LinkProtectionMapIterator
                    itm((*itn.CurrentItem())->GetPhaseDescription()
                    .GetLinkMovements(lane.GetLink())); !itm.IsDone();
                    itm.Next()) {
                TNetLink* current = *itm.CurrentItem();
                if (tolinks.IsMember(current)) {
                    possible = TRUE;
                    if (protMap.IsBound(current) && *protMap.ValueOf(current)
                            != *itm.CurrentValue())
                        mismatch = TRUE;
                }
            }
          if (possible && !mismatch)
            if (prot.IsMember(TNetPhaseDescription::kPermitted))
                return TNetTrafficControl::kPermitted;
            else if (prot.IsMember(TNetPhaseDescription::kProtected))
                   return TNetTrafficControl::kProtected;
            else
                       throw TNetException ("Invalid protection code found.");

        if (intv == TNetPhase::kYellow)
                return TNetTrafficControl::kCaution;
        if (intv == TNetPhase::kRed)
                return TNetTrafficControl::kWait;
    }
    return TNetTrafficControl::kNone;  //  never reached
}


//  Return the current phasing plan.
TNetPhasingPlan& TNetSignalizedControl::GetPhasingPlan()
{
    return *fPhasingPlan;
}

const TNetPhasingPlan& TNetSignalizedControl::GetPhasingPlan() const
{
    return *fPhasingPlan;
}


//  Return the current phasing plan offset.
REAL TNetSignalizedControl::GetPhasingPlanOffset() const
{
    return fPhasingPlanOffset;
}


//  Define the phasing plan.
void TNetSignalizedControl::SetPhasingPlan(TNetPhasingPlan& plan)
{
    fPhasingPlan = &plan;
}


//  Define the phasing plan offset.
void TNetSignalizedControl::SetPhasingPlanOffset(REAL offset)
{
    fPhasingPlanOffset = offset;
}
```

```
//  Return the signalized control coordinator for this signal.
TNetSignalCoordinator& TNetSignalizedControl::GetCoordinator()
{
    return *fCoordinator;
}

const TNetSignalCoordinator& TNetSignalizedControl::GetCoordinator() const
{
    return *fCoordinator;
}


//  Define the signalized control coordinator for this signal.
void TNetSignalizedControl::SetCoordinator(TNetSignalCoordinator& coord)
{
    fCoordinator = &coord;
    if (coord.GetControllers().Location(this) == -1)
        coord.SetController(*this);
}


//  Create a phase.
TNetPhase& TNetSignalizedControl::CreatePhase(TNetPhaseDescription& d)
{
    TNetPhase* p = new TNetPhase(d);
    fPhases.Append(p);
    return *p;
}


//  Return the phases for this signal.
TNetSignalizedControl::PhaseCollection& TNetSignalizedControl::GetPhases()
{
    return fPhases;
}

const TNetSignalizedControl::PhaseCollection&
        TNetSignalizedControl::GetPhases() const
{
    return fPhases;
}


//  Return the current phase.
TNetPhase& TNetSignalizedControl::GetPhase()
{
    return *fCurrentPhase;
}

const TNetPhase& TNetSignalizedControl::GetPhase() const
{
    return *fCurrentPhase;
}


//  Update the current phase.
void TNetSignalizedControl::SetPhase(TNetPhase& p)
{
    if (fCurrentPhase->GetNextPhases().Location(&p) != -1)
          fCurrentPhase = &p;
    else
        throw TNetException ("Error in SetPhase.");
}


//  Initialize the signal to specified phase.
void TNetSignalizedControl::InitPhase(TNetPhase& p)
{
    fCurrentPhase = &p;
}
```

FF. TNetSignalizedControlReader Class

1. SignalizedControlReader.h

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: SignalizedControlReader.h,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 19:29:51 $
//  $State: Rel $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


#ifndef TRANSIMS_NET_SIGREADER
#define TRANSIMS_NET_SIGREADER


//  Include TRANSIMS header files.
#include <GBL/Globals.h>
#include <DBS/Accessor.h>
#include <NET/Id.h>


//  Forward declarations.
class TNetReader;


//  This reader reads signalized control values from the database.
class TNetSignalizedControlReader
{
  public:

    //  Construct a signalized control reader for a given network.
    TNetSignalizedControlReader(TNetReader& reader);

    //  Construct a copy of the given signalized control reader.
//  TNetSignalizedControlReader(const TNetSignalizedControlReader& reader);

    //  Make the reader a copy of the given signalized control reader.
//  TNetSignalizedControlReader& operator=(const TNetSignalizedControlReader&
//          reader);

    //  Reset the iteration over the table.
    void Reset();

    //  Get the next node in the table.
    void GetNextNode();

    //  Return whether there are any more nodes in the table.
    bool MoreNodes();

    //  Return the id for the current node.
    NetNodeId GetNode() const;

    //  Return the type of the signal.
    string GetType() const;

    //  Return the timing plan id.
    NetPlanId GetPlan() const;

    //  Return the offset for coordinated signals.
    REAL GetOffset() const;

    //  Return the starting time for the plan.
    string GetStarttime() const;

  private:

    //  Each signalized control reader has a database table accessor.
    TDbAccessor fAccessor;

    //  Each record has a node field.
```

```
    const TDbField fNodeField;

    //  Each record has signal type.
    const TDbField fTypeField;

    //  Each record has a timing plan.
    const TDbField fPlanField;

    //  Each timing plan has an offset.
    const TDbField fOffsetField;

    //  Each timing plan has a start time.
    const TDbField fStarttimeField;
};


#endif // TRANSIMS_NET_SIGREADER
```

2. SignalizedControlReader.C

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: SignalizedControlReader.C,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 19:29:51 $
//  $State: Rel $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


//  Include TRANSIMS header files.
#include <NET/SignalizedControlReader.h>
#include <NET/Reader.h>


//  Construct a signalized control reader for a given network.
TNetSignalizedControlReader::TNetSignalizedControlReader(TNetReader& reader)
  : fAccessor(reader.GetSignalizedControlTable()),
    fNodeField(reader.GetSignalizedControlTable().GetField("NODE")),
    fTypeField(reader.GetSignalizedControlTable().GetField("TYPE")),
    fPlanField(reader.GetSignalizedControlTable().GetField("PLAN")),
    fOffsetField(reader.GetSignalizedControlTable().GetField("OFFSET")),
    fStarttimeField(reader.GetSignalizedControlTable().GetField("STARTTIME"))
{
}


//  Reset the iteration over the table.
void TNetSignalizedControlReader::Reset()
{
    fAccessor.GotoFirst();
}


//  Get the next node in the table.
void TNetSignalizedControlReader::GetNextNode()
{
    fAccessor.GotoNext();
}


//  Return whether there are any more nodes in the table.
bool TNetSignalizedControlReader::MoreNodes()
{
    return fAccessor.IsAtRecord();
}


//  Return the id for the current node.
NetNodeId TNetSignalizedControlReader::GetNode() const
{
    NetNodeId node;
    fAccessor.GetField(fNodeField, node);
    return node;
```

```
}


//  Return the type of the signal.
string TNetSignalizedControlReader::GetType() const
{
    string type;
    fAccessor.GetField(fTypeField, type);
    return type;
}


//  Return the timing plan id.
NetPlanId TNetSignalizedControlReader::GetPlan() const
{
    NetPlanId plan;
    fAccessor.GetField(fPlanField, plan);
    return plan;
}


//  Return the offset for coordinated signals.
REAL TNetSignalizedControlReader::GetOffset() const
{
    REAL offset;
    fAccessor.GetField(fOffsetField, offset);
    return offset;
}


//  Return the starting time for the plan.
string TNetSignalizedControlReader::GetStarttime() const
{
    string start;
    fAccessor.GetField(fStarttimeField, start);
    return start;
}
```

GG. TNetSimulationArea Class

1. SimulationArea.h

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: SimulationArea.h,v $
//  $Revision: 2.0 $
//  $Date: 1996/06/03 22:37:43 $
//  $State: Stab $
//  $Author: bwb $
//  U.S. Government Copyright 1996
//  All rights reserved


#ifndef TRANSIMS_NET_SIMAREA
#define TRANSIMS_NET_SIMAREA


//  Include TRANSIMS header files.
#include "GBL/Globals.h"
#include "NET/Id.h"


//  Include Booch Components header files.
#include <BCStoreM.h>
#include <BCMapU.h>


//  Forward declarations.
class TNetSimulationAreaLinkReader;


//  The simulation area describes the simulation region of interest.
class TNetSimulationArea
{
```

```
  public:

    //  Link type indicates whether the link is in the buffer or study area.
    enum EType {kStudy, kBuffer};

    //  Type definitions.
    typedef BC_TUnboundedMap<NetLinkId, EType, 500U, BC_CManaged> LinkIdMap;
    typedef BC_TMapActiveIterator<NetLinkId, EType> LinkIdMapIterator;

    //  Construct a simulation area.
    TNetSimulationArea(TNetSimulationAreaLinkReader& reader);
    TNetSimulationArea(const TNetSimulationArea& area);

    //  Destroy a simulation area.
    ~TNetSimulationArea();

    //  Assign a simulation area.
    TNetSimulationArea& operator=(const TNetSimulationArea& area);

    //  Return whether two simulation areas are the same.
    bool operator==(const TNetSimulationArea& area) const;
    bool operator!=(const TNetSimulationArea& area) const;

    //  Return whether a link is in the simulation area (study area + buffer).
    bool IsInSimulationArea(NetLinkId id);

    //  Return whether a link is in the study area.
    bool IsInStudyArea(NetLinkId id);

    //  Return whether a link is in the simulation area buffer region.
    bool IsInBufferArea(NetLinkId id);

    //  Return all links in simulation area.
    LinkIdMap& GetLinks();
    const LinkIdMap& GetLinks() const;

  private:

    //  A simulation area is described by links and whether the link is in the
    //  buffer portion of the simulation area.
    LinkIdMap fLinks;
};


#endif // TRANSIMS_NET_SIMAREA
```

2. SimulationArea.C

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: SimulationArea.C,v $
//  $Revision: 2.0 $
//  $Date: 1996/06/03 22:38:35 $
//  $State: Stab $
//  $Author: bwb $
//  U.S. Government Copyright 1996
//  All rights reserved


//  Include TRANSIMS header files
#include "NET/SimulationArea.h"
#include "NET/SimulationAreaLinkReader.h"


//  Construct a simulation area
TNetSimulationArea::TNetSimulationArea(TNetSimulationAreaLinkReader& reader)
  : fLinks(reader.GetLinks())
{
}


TNetSimulationArea::TNetSimulationArea(const TNetSimulationArea& a)
{
    fLinks = a.fLinks;
}
```

```
//  Destroy a simulation area.
TNetSimulationArea::~TNetSimulationArea()
{
}


//  Assign a simulation area.
TNetSimulationArea& TNetSimulationArea::operator=(const TNetSimulationArea& a)
{
    if (this == &a)
        return *this;
    fLinks = a.fLinks;
    return *this;
}


//  Return whether two simulation areas are the same.
bool TNetSimulationArea::operator==(const TNetSimulationArea& a) const
{
    return (this == &a);
}

bool TNetSimulationArea::operator!=(const TNetSimulationArea& a) const
{
    return (this == &a);
}


//  Return whether a link is in the simulation area (study area + buffer).
bool TNetSimulationArea::IsInSimulationArea(NetLinkId link)
{
    return fLinks.IsBound(link);
}


//  Return whether a link is in the study area.
bool TNetSimulationArea::IsInStudyArea(NetLinkId link)
{
    return !(*fLinks.ValueOf(link));
}


//  Return whether a link is in the simulation area buffer region.
bool TNetSimulationArea::IsInBufferArea(NetLinkId link)
{
    return *fLinks.ValueOf(link);
}


//  Return all links in simulation area.
TNetSimulationArea::LinkIdMap& TNetSimulationArea::GetLinks()
{
    return fLinks;
}

const TNetSimulationArea::LinkIdMap& TNetSimulationArea::GetLinks() const
{
    return fLinks;
}
```

HH. TNetSimulationAreaLinkReader Class

1. SimulationAreaLinkReader.h

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: SimulationAreaLinkReader.h,v $
//  $Revision: 2.0 $
//  $Date: 1996/06/03 22:39:54 $
//  $State: Stab $
//  $Author: bwb $
//  U.S. Government Copyright 1995
```

```
//  All rights reserved


#ifndef TRANSIMS_NET_AREALINKREADER
#define TRANSIMS_NET_AREALINKREADER


//  Include TRANSIMS header files.
#include <GBL/Globals.h>
#include <DBS/Accessor.h>
#include <NET/Id.h>
#include <NET/SimulationArea.h>


//  Include Booch Component header files.
#include <BCStoreM.h>
#include <BCMapU.h>


//  Forward declarations.
class TNetSimulationAreaReader;


//  A simulation area link reader reads link values from the database.
class TNetSimulationAreaLinkReader
{
  public:

    //  Type definitions.
    typedef BC_TUnboundedMap<NetLinkId, TNetSimulationArea::EType, 500U, BC_CManaged>
            LinkIdMap;
    typedef BC_TMapActiveIterator<NetLinkId, TNetSimulationArea::EType>
            LinkIdMapIterator;

    //  Construct a simulation area link reader.
    TNetSimulationAreaLinkReader(TNetSimulationAreaReader& reader);

    //  Construct a copy of the given simulation area link reader.
//  TNetSimulationAreaLinkReader(const TNetSimulationAreaLinkReader& reader);

    //  Make the reader a copy of the given simulation area link reader.
//  TNetSimulationAreaLinkReader& operator=(const TNetSimulationAreaLinkReader&
//          reader);

    //  Reset the iteration over the table.
    void Reset();

    //  Get the next link in the table.
    void GetNextLink();

    //  Return whether there are any more links in the table.
    bool MoreLinks() const;

    //  Return the links in the simulation area.
    LinkIdMap GetLinks();

  private:

    //  Each study area link reader has a database table accessor.
    TDbAccessor fAccessor;

    //  Each link has an ID field.
    const TDbField fIdField;

    //  Each link has a field that indicates if it is in the buffer area.
    const TDbField fBufferField;
};


#endif // TRANSIMS_NET_AREALINKREADER
```

2. SimulationAreaLinkReader.C

```
//  Project: TRANSIMS
//  Subsystem: Network
```

```
//  $RCSfile: SimulationAreaLinkReader.C,v $
//  $Revision: 2.0 $
//  $Date: 1996/06/03 22:40:18 $
//  $State: Stab $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


//  Include TRANSIMS header files.
#include <NET/SimulationAreaLinkReader.h>
#include <NET/SimulationAreaReader.h>
#include <NET/SimulationArea.h>
#include <NET/Exception.h>


//  Define the hash function for link ids.
static BC_Index LinkIdHashValue(const NetLinkId& id)
{
    return BC_Index(id);
}


//  Construct a simulation area link reader .
TNetSimulationAreaLinkReader::TNetSimulationAreaLinkReader(
        TNetSimulationAreaReader& reader)
  : fAccessor(reader.GetSimulationAreaTable()),
    fIdField(reader.GetSimulationAreaTable().GetField("ID")),
    fBufferField(reader.GetSimulationAreaTable().GetField("BUFFER"))
{
}


//  Reset the iteration over the table.
void TNetSimulationAreaLinkReader::Reset()
{
    fAccessor.GotoFirst();
}


//  Get the next link in the table.
void TNetSimulationAreaLinkReader::GetNextLink()
{
    fAccessor.GotoNext();
}


//  Return whether there are any more links in the table.
bool TNetSimulationAreaLinkReader::MoreLinks() const
{
    return fAccessor.IsAtRecord();
}


//  Return the links in the simulation area.
TNetSimulationAreaLinkReader::LinkIdMap TNetSimulationAreaLinkReader::GetLinks()
{
    LinkIdMap links(LinkIdHashValue);

    for (Reset(); MoreLinks(); GetNextLink()) {
        NetLinkId id;
        fAccessor.GetField(fIdField, id);
        string buf;
        fAccessor.GetField(fBufferField, buf);
        if (buf == 'Y')
            links.Bind(id, TNetSimulationArea::kBuffer);
        else if (buf == 'N')
            links.Bind(id, TNetSimulationArea::kStudy);
        else
            throw TNetException("Invalid BUFFER field.");
    }

    return links;
}
```

II. TNetSimulationAreaReader Class

1. SimulationAreaReader.h

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: SimulationAreaReader.h,v $
//  $Revision: 2.0 $
//  $Date: 1996/06/03 22:39:01 $
//  $State: Stab $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


#ifndef TRANSIMS_NET_SIMAREAREADER
#define TRANSIMS_NET_SIMAREAREADER


//  Include TRANSIMS header files.
#include <GBL/Globals.h>
#include <DBS/Table.h>


//  Forward declarations.
class TDbTable;


//  A simulation area reader reads a simulation area from the database.
class TNetSimulationAreaReader
{
  public:

    //  Construct a reader for the specified tables.
    TNetSimulationAreaReader(TDbTable areaTable);

    //  Return the simulation area table.
    TDbTable& GetSimulationAreaTable();

  private:

    //  Each reader has a simulation area table.
    TDbTable fSimulationAreaTable;
};


#endif // TRANSIMS_NET_SIMAREAREADER
```

2. SimulationAreaReader.C

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: SimulationAreaReader.C,v $
//  $Revision: 2.0 $
//  $Date: 1996/06/03 22:39:26 $
//  $State: Stab $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


//  Include TRANSIMS header files.
#include <NET/SimulationAreaReader.h>


//  Construct a reader for the specified tables.
TNetSimulationAreaReader::TNetSimulationAreaReader(TDbTable areaTable)
  : fSimulationAreaTable(areaTable)
{
}


//  Return the study area table.
TDbTable& TNetSimulationAreaReader::GetSimulationAreaTable()
```

```
{
    return fSimulationAreaTable;
}
```

JJ. TNetSubnetwork Class

1. Subnetwork.h

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: Subnetwork.h,v $
//  $Revision: 2.3 $
//  $Date: 1996/02/14 21:43:45 $
//  $State: Exp $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


#ifndef TRANSIMS_NET_SUBNETWORK
#define TRANSIMS_NET_SUBNETWORK


//  Include TRANSIMS header files.
#include <GBL/Globals.h>
#include <NET/Id.h>
#include <NET/Factory.h>
#include <NET/FilterFunction.h>
#include <NET/FilterNone.h>


//  Include Booch Components header files.
#include <BCStoreM.h>
#include <BCSetU.h>


//  Forward declarations.
class TNetReader;
class TNetNetwork;
class TNetNode;
class TNetLink;
class TNetFactory;


//  A subnetwork represents a subset of the instantiated network.
class TNetSubnetwork
{
  public:

    //  Type definitions.
    typedef BC_TUnboundedSet<TNetNode*, 10000U, BC_CManaged> NodeSet;
    typedef BC_TSetActiveIterator<TNetNode*> NodeSetIterator;
    typedef BC_TUnboundedSet<TNetLink*, 10000U, BC_CManaged> LinkSet;
    typedef BC_TSetActiveIterator<TNetLink*> LinkSetIterator;
    typedef BC_TUnboundedSet<NetNodeId, 10000U, BC_CManaged> NodeIdSet;
    typedef BC_TSetActiveIterator<NetNodeId> NodeIdSetIterator;
    typedef BC_TUnboundedSet<NetLinkId, 10000U, BC_CManaged> LinkIdSet;
    typedef BC_TSetActiveIterator<NetLinkId> LinkIdSetIterator;

    //  Construct a subnetwork with geographic filtering using the reader.
    TNetSubnetwork(TNetReader& reader, TNetNetwork& network, TGeoFilterFunction&
            filter = TGeoFilterNone(), TNetFactory& factory = TNetFactory());

    //  Construct a subnetwork of specified nodes and links using the reader.
    TNetSubnetwork(TNetReader& reader, TNetNetwork& network,
            NodeIdSet& nodesRequested, LinkIdSet& linksRequested,
            NodeSet& nodesProvided, LinkSet& linksProvided,
            TNetFactory& factory = TNetFactory());

    //  Return the set of nodes in the subnetwork.
    NodeSet& GetNodes();
    const NodeSet& GetNodes() const;

    //  Return the set of links in the subnetwork.
```

```
    LinkSet& GetLinks();
    const LinkSet& GetLinks() const;

    //  Return the network.
    TNetNetwork& GetNetwork();
    const TNetNetwork& GetNetwork() const;

  protected:

    //  Create the network nodes.
    void CreateNodes (TNetReader&, TNetFactory&, NodeIdSet&);

    //  Create the network links.
    void CreateLinks (TNetReader&, TNetFactory&, LinkIdSet&, LinkIdSet&);

    //  Create the network accessories.
    void CreateAccessories (TNetReader&, TNetFactory&, LinkIdSet&);

    //  Create the network traffic controls.
    void CreateTrafficControls (TNetReader&, TNetFactory&, NodeIdSet&);

  private:

    //  Do not allow subnetworks to be copied.
    TNetSubnetwork(const TNetSubnetwork&) : fNetwork((TNetNetwork*) NULL){}

    //  Do not allow subnetworks to be assigned.
    TNetSubnetwork& operator=(const TNetSubnetwork&) {return *this;}

    //  Each subnetwork is part of a network.
    TNetNetwork* const fNetwork;

    //  Each subnetwork has references to the nodes it contains.
    NodeSet fNodes;

    //  Each subnetwork has references to the links it contains.
    LinkSet fLinks;
};


#endif // TRANSIMS_NET_SUBNETWORK
```

2. Subnetwork.C

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: Subnetwork.C,v $
//  $Revision: 2.6 $
//  $Date: 1996/08/29 15:13:22 $
//  $State: Stab $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


//  Include TRANSIMS header files.
#include <NET/Subnetwork.h>
#include <NET/Network.h>
#include <NET/Node.h>
#include <NET/Link.h>
#include <NET/Lane.h>
#include <NET/Pocket.h>
#include <NET/Parking.h>
#include <NET/Location.h>
#include <NET/LaneLocation.h>
#include <NET/TrafficControl.h>
#include <NET/UnsignalizedControl.h>
#include <NET/NullControl.h>
#include <NET/SignalizedControl.h>
#include <NET/TimedControl.h>
#include <NET/Phase.h>
#include <NET/PhasingPlan.h>
#include <NET/SignalCoordinator.h>
#include <NET/IsolatedControl.h>
#include <NET/Reader.h>
```

```
#include <NET/NodeReader.h>
#include <NET/LinkReader.h>
#include <NET/PocketReader.h>
#include <NET/ParkingReader.h>
#include <NET/LaneConnectivityReader.h>
#include <NET/UnsignalizedControlReader.h>
#include <NET/SignalizedControlReader.h>
#include <NET/TimingPlanReader.h>
#include <NET/PhasingPlanReader.h>
#include <NET/Exception.h>


//  Include C header files.
#include <iostream.h>


//  Global variable for logging errors from Network subsystem.
ostream* gNetLogStream = &cerr;


//  Define the hash function for nodes.
static BC_Index NodeHashValue(TNetNode* const& node)
{
    return BC_Index(node->GetId());
}


//  Define the hash function for links.
static BC_Index LinkHashValue(TNetLink* const& link)
{
    return BC_Index(link->GetId());
}


//  Define the hash function for node ids.
static BC_Index NodeIdHashValue(const NetNodeId& id)
{
    return BC_Index(id);
}


//  Define the hash function for link ids.
static BC_Index LinkIdHashValue(const NetLinkId& id)
{
    return BC_Index(id);
}


//  Define the hash function for phase numbers.
static BC_Index PhaseHashValue(const NetPhaseNumber& n)
{
    return BC_Index(n);
}


//  Construct a subnetwork with geographic filtering using the reader.
TNetSubnetwork::TNetSubnetwork(TNetReader& reader, TNetNetwork& network,
        TGeoFilterFunction& filter, TNetFactory& factory)
  : fNetwork(&network),
    fNodes(NodeHashValue),
    fLinks(LinkHashValue)
{
    //  Build set of nodes in subnetwork.
    NodeIdSet subnetNodes(NodeIdHashValue);
    TNetNodeReader nodeReader(reader);
    for (nodeReader.Reset(); nodeReader.MoreNodes(); nodeReader.GetNextNode())
        if (filter(nodeReader.GetGeographicPosition()))
            subnetNodes.Add(nodeReader.GetId());

    //  Build set of links in subnetwork and adjust set of nodes to include any
    //  nodes associated with links that cross the subnetwork boundary.
    LinkIdSet subnetLinks(LinkIdHashValue);
    NodeIdSet extraNodes(NodeIdHashValue);
    TNetLinkReader linkReader(reader);
    for (linkReader.Reset(); linkReader.MoreLinks(); linkReader.GetNextLink()) {
        const NetLinkId id = linkReader.GetId();
```

```
        NetNodeId idA, idB;
        linkReader.GetNodeIds(idA, idB);
        const bool includesA = subnetNodes.IsMember(idA);
        const bool includesB = subnetNodes.IsMember(idB);
        if (includesA || includesB) {
            subnetLinks.Add(id);
            if (!includesA)
                extraNodes.Add(idA);
            if (!includesB)
                extraNodes.Add(idB);
        }
    }

    //  Add the extra nodes to the subnetwork.
    for (NodeIdSetIterator i(extraNodes); !i.IsDone(); i.Next())
        subnetNodes.Add(*i.CurrentItem());

    //  Construct nodes
    CreateNodes (reader, factory, subnetNodes);

    //  Construct links
    LinkIdSet newLinks(LinkIdHashValue);
    CreateLinks (reader, factory, subnetLinks, newLinks);

    //  Construct accessories
    CreateAccessories (reader, factory, newLinks);

    //  Construct traffic controls
    CreateTrafficControls (reader, factory, extraNodes);
}


//  Construct a subnetwork of specified nodes and links using the reader.
    //  nodesRequested and linksRequested are the desired sets of nodes and
    //  links in the subnetwork.  nodesRequested and/or linksRequested are
    //  filled in by the caller; either may be empty, but not both.
    //  nodesProvided and linksProvided are filled in by this constructor.
    //  When nodesRequested is empty, nodesProvided includes those nodes
    //  implied by linksRequested.  Similarly for linksProvided.
    //  nodesProvided and linksProvided do not include extra nodes and links
    //  that may be needed to complete the traffic control at a node.
TNetSubnetwork::TNetSubnetwork(TNetReader& reader, TNetNetwork& network,
        NodeIdSet& requestedNodes, LinkIdSet& requestedLinks,
        NodeSet& providedNodes, LinkSet& providedLinks, TNetFactory& factory)
  : fNetwork(&network),
    fNodes(NodeHashValue),
    fLinks(LinkHashValue)
{
    //  Build set of nodes in subnetwork.
    NodeIdSet subnetNodes(NodeIdHashValue);
    subnetNodes = requestedNodes;
    const bool emptyNodes = requestedNodes.IsEmpty();

    //  Build set of links in subnetwork.
    LinkIdSet subnetLinks(LinkIdHashValue);
    subnetLinks = requestedLinks;
    const bool emptyLinks = requestedLinks.IsEmpty();
    if (emptyNodes && emptyLinks)
        throw TNetException ("Requested nodes and requested links may not "
                "both be empty.");

    TNetLinkReader linkReader(reader);
    //  Adjust set of nodes and links to include those that are implied when the
    //  requested set is empty.
    if (emptyNodes || emptyLinks)
        for (linkReader.Reset(); linkReader.MoreLinks();
                linkReader.GetNextLink()) {
            const NetLinkId id = linkReader.GetId();
            NetNodeId idA, idB;
            linkReader.GetNodeIds(idA, idB);
            const bool includesA = requestedNodes.IsMember(idA);
            const bool includesB = requestedNodes.IsMember(idB);
            if (requestedLinks.IsMember(id) && emptyNodes) {
                subnetNodes.Add(idA);
                subnetNodes.Add(idB);
            }
```

```
            if (includesA || includesB)
                if (emptyLinks)
                    subnetLinks.Add(id);
        }

    //  Identify additional links and nodes needed to construct complete
    //  traffic controls at nodes in subnet.
    NodeIdSet extraNodes(NodeIdHashValue);
    LinkIdSet extraLinks(LinkIdHashValue);
    for (linkReader.Reset(); linkReader.MoreLinks(); linkReader.GetNextLink()) {
        const NetLinkId id = linkReader.GetId();
        NetNodeId idA, idB;
        linkReader.GetNodeIds(idA, idB);
        const bool includesA = subnetNodes.IsMember(idA);
        const bool includesB = subnetNodes.IsMember(idB);
        const bool includesL = subnetLinks.IsMember(id);
        if (includesA || includesB) {
            if (!includesL)
                extraLinks.Add(id);
            if (!includesA)
                extraNodes.Add(idA);
            if (!includesB)
                extraNodes.Add(idB);
        } else if (includesL) {
            extraNodes.Add(idA);
            extraNodes.Add(idB);
        }
    }

    //  Construct nodes.
    subnetNodes.Union(extraNodes);
    CreateNodes (reader, factory, subnetNodes);
    for (NodeMapIterator i(fNetwork->GetNodes()); !i.IsDone(); i.Next())
        if (!extraNodes.IsMember(*i.CurrentItem()))
                providedNodes.Add(*i.CurrentValue());

    //  Construct links.
    LinkIdSet newLinks(LinkIdHashValue);
    subnetLinks.Union(extraLinks);
    CreateLinks (reader, factory, subnetLinks, newLinks);
    for (LinkMapIterator j(fNetwork->GetLinks()); !j.IsDone(); j.Next())
        if (!extraLinks.IsMember(*j.CurrentItem()))
                providedLinks.Add(*j.CurrentValue());

    //  Construct accessories.
    CreateAccessories (reader, factory, newLinks);

    //  Construct traffic controls.
    CreateTrafficControls (reader, factory, extraNodes);
}


//  Create the network nodes.
void TNetSubnetwork::CreateNodes(TNetReader& reader, TNetFactory& factory,
        NodeIdSet& subnetNodes)
{
    //  Get network data structures.
    TNetNetwork::NodeMap& nodes = fNetwork->GetNodes();
    TNetNetwork::LinkMap& links = fNetwork->GetLinks();

    //  Construct nodes.
    TNetNodeReader nodeReader(reader);
    for (nodeReader.Reset(); nodeReader.MoreNodes(); nodeReader.GetNextNode()) {
        const NetNodeId id = nodeReader.GetId();
        if (subnetNodes.IsMember(id)) {
            if (nodes.IsBound(id))
                fNodes.Add(*nodes.ValueOf(id));
            else {
                TNetNode* const node = factory.NewNode(nodeReader);
                nodes.Bind(id, node);
                fNodes.Add(node);
            }
        }
    }
}
```

```
//  Create the network links.
void TNetSubnetwork::CreateLinks (TNetReader& reader, TNetFactory& factory,
        LinkIdSet& subnetLinks, LinkIdSet& newLinks)
{
    //  Get network data structures.
    TNetNetwork::NodeMap& nodes = fNetwork->GetNodes();
    TNetNetwork::LinkMap& links = fNetwork->GetLinks();

    //  Construct links.
    TNetLinkReader linkReader(reader);
    for (linkReader.Reset(); linkReader.MoreLinks(); linkReader.GetNextLink()) {
        const NetLinkId id = linkReader.GetId();
        if (subnetLinks.IsMember(id)) {
            if (links.IsBound(id))
                fLinks.Add(*links.ValueOf(id));
            else {
                newLinks.Add(id);
                TNetLink* const link = factory.NewLink(linkReader);
                NetNodeId idA, idB;
                linkReader.GetNodeIds(idA, idB);
                TNetNode* const nodeA = *nodes.ValueOf(idA);
                TNetNode* const nodeB = *nodes.ValueOf(idB);
                link->SetNodes(nodeA, nodeB);
                nodeA->AddLink(link);
                nodeB->AddLink(link);
                links.Bind(id, link);
                fLinks.Add(link);
            }
        }
    }
}


//  Create the network accessories.
void TNetSubnetwork::CreateAccessories (TNetReader& reader, TNetFactory&
        factory, LinkIdSet& newLinks)
{
    //  Get network data structures.
    TNetNetwork::NodeMap& nodes = fNetwork->GetNodes();
    TNetNetwork::LinkMap& links = fNetwork->GetLinks();
    TNetNetwork::ParkingMap& parkings = fNetwork->GetParkings();

    //  Construct pocket accessories.
    TNetPocketReader pocketReader(reader);
    for (pocketReader.Reset(); pocketReader.MoreAccessories();
            pocketReader.GetNextAccessory()) {
        NetLinkId linkId;
        NetNodeId nodeId;
        REAL offset;
        pocketReader.GetLocation(linkId, nodeId, offset);
        if (newLinks.IsMember(linkId)) {
            TNetLink* const link = *links.ValueOf(linkId);
            TNetNode* const node = *nodes.ValueOf(nodeId);
            TNetLink::LaneCollection& lanes = link->GetLanesTowards(*node,
                    TRUE);
            TNetLane *const lane = lanes[pocketReader.GetLaneNumber() - 1];
            TNetPocket* const pocket = factory.NewPocket(pocketReader);
            pocket->SetLocation(TNetLaneLocation(*lane, offset));
            lane->GetPockets().Append(pocket);
        }
    }

    //  Construct parking places.
    TNetParkingReader parkingReader(reader);
    for (parkingReader.Reset(); parkingReader.MoreAccessories();
            parkingReader.GetNextAccessory()) {
        NetLinkId linkId;
        NetNodeId nodeId;
        REAL offset;
        parkingReader.GetLocation(linkId, nodeId, offset);
        if (newLinks.IsMember(linkId)) {
            TNetLink* const link = *links.ValueOf(linkId);
            TNetNode* const node = *nodes.ValueOf(nodeId);
            TNetParking* const parking = factory.NewParking(parkingReader);
            parking->SetLocation(TNetLocation(*link, *node, offset));
```

```
            link->GetAccessories().Append(parking);
            parkings.Bind(parking->GetId(), parking);
        }
    }
}


//  Create the network traffic controls.
void TNetSubnetwork::CreateTrafficControls (TNetReader& reader, TNetFactory&
        factory, NodeIdSet& extraNodes)
{
    //  Get network data structures.
    TNetNetwork::NodeMap& nodes = fNetwork->GetNodes();
    TNetNetwork::LinkMap& links = fNetwork->GetLinks();

    //  Read and construct unsignalized traffic controls.
    TNetUnsignalizedControlReader unsigReader(reader);
    for (unsigReader.Reset(); unsigReader.MoreNodes(); unsigReader.GetNextNode())
    {
        const NetNodeId id = unsigReader.GetNode();
        TNetUnsignalizedControl* u;
        if (nodes.IsBound(id) && &(*nodes.ValueOf(id))->GetTrafficControl() ==
                NULL && !extraNodes.IsMember(id)) {
            u = factory.NewUnsignalizedControl(unsigReader, *fNetwork);
            u->SetVehicleControl(unsigReader, *fNetwork);
        } else if (nodes.IsBound(id) && (u = (TNetUnsignalizedControl*)
                &(*nodes.ValueOf(id))->GetTrafficControl()) != NULL &&
                !extraNodes.IsMember(id))
            u->SetVehicleControl(unsigReader, *fNetwork);
    }

    //  Read and construct signalized traffic controls.
    TNetNetwork::SignalCoordinatorMap& coordinators =
            fNetwork->GetSignalCoordinators();
    TNetSignalizedControlReader sigReader(reader);
    TNetTimingPlanReader timingReader(reader);
    TNetPhasingPlanReader phasingReader(reader);
    TNetSignalCoordinator::PhaseNumberMap nextPhases(PhaseHashValue);
    BC_TUnboundedMap<NetPhaseNumber, TNetPhase*, 2U, BC_CManaged>
            phaseMap(PhaseHashValue);
    for (sigReader.Reset(); sigReader.MoreNodes(); sigReader.GetNextNode()) {
        const NetNodeId id = sigReader.GetNode();
        TNetSignalizedControl* s;
        TNetSignalCoordinator* c;
        if (nodes.IsBound(id) && &(*nodes.ValueOf(id))->GetTrafficControl() ==
                NULL && !extraNodes.IsMember(id)) {
            string type = sigReader.GetType();
            if (type == "T")
                s = factory.NewTimedControl(sigReader, *fNetwork);
            else if (type == "A") {
                *gNetLogStream << "Actuated controls are not currently "
                        "supported.  Creating timed control." << endl;
                s = factory.NewTimedControl(sigReader, *fNetwork);
            } else
                throw TNetNotFound ("Invalid signal type.");

            //  only isolated controls are currently supported.
            c = factory.NewIsolatedControl(id);
            coordinators.Bind(c->GetId(), (TNetSignalCoordinator*)c);
            c->SetController(*s);
            s->SetPhasingPlanOffset(sigReader.GetOffset());
            nextPhases.Clear();
            TNetPhasingPlan& plan = c->CreatePhasingPlan(id,
                    sigReader.GetPlan(), timingReader, phasingReader, *fNetwork,
                    nextPhases);
            s->SetPhasingPlan(plan);

            //  create phases according to phasing plan.
            phaseMap.Clear();
            for (TNetPhasingPlan::PhaseDescriptionCollectionIterator
                    i1(plan.GetPhaseDescriptions()); !i1.IsDone(); i1.Next()) {
                TNetPhase& phase = s->CreatePhase(**i1.CurrentItem());
                phaseMap.Bind ((*i1.CurrentItem())->GetPhaseNumber(), &phase);
            }
            //  bind next phases according to map
            for (TNetSignalizedControl::PhaseCollectionIterator
```

```
                    i2(s->GetPhases()); !i2.IsDone(); i2.Next())
                for (BC_TCollectionActiveIterator<NetPhaseNumber>
                       i3(*(nextPhases.ValueOf((*i2.CurrentItem())
                       ->GetPhaseDescription().GetPhaseNumber())));
                       !i3.IsDone(); i3.Next())
                    (*i2.CurrentItem())->SetNextPhase(**(phaseMap
                            .ValueOf(*i3.CurrentItem())));
        } else if (nodes.IsBound(id) &&
                &(*nodes.ValueOf(id))->GetTrafficControl() != NULL &&
                extraNodes.IsMember(id))
            *gNetLogStream << "Warning: A second traffic control for node "
                << id << " was not created." << endl;
    }

    //  Construct null controls for nodes without traffic controls.
    for (TNetNetwork::NodeMapIterator it(nodes); !it.IsDone(); it.Next())
        if (&(*it.CurrentValue())->GetTrafficControl() == NULL)
        {
            if (!extraNodes.IsMember(*it.CurrentItem()))
                *gNetLogStream << "Warning: Node " << *it.CurrentItem() <<
                        " has no traffic control.  Assigning null control." <<
                        endl;
            factory.NewNullControl (**it.CurrentValue());
        }

    //  Fill in lane connectivity in traffic controls.
    TNetLaneConnectivityReader connReader(reader);
    for (connReader.Reset(); connReader.MoreNodes(); connReader.GetNextNode()) {
            const NetNodeId id = connReader.GetNode();
        TNetTrafficControl* c;
        if (nodes.IsBound(id) && (c=&(*nodes.ValueOf(id))->GetTrafficControl())
                != NULL && !extraNodes.IsMember(id))
            c->SetConnectivity(connReader, *fNetwork);
    }
}


//  Return the set of nodes in the subnetwork.
TNetSubnetwork::NodeSet& TNetSubnetwork::GetNodes()
{
    return fNodes;
}

const TNetSubnetwork::NodeSet& TNetSubnetwork::GetNodes() const
{
    return fNodes;
}


//  Return the set of links in the subnetwork.
TNetSubnetwork::LinkSet& TNetSubnetwork::GetLinks()
{
    return fLinks;
}

const TNetSubnetwork::LinkSet& TNetSubnetwork::GetLinks() const
{
    return fLinks;
}


//  Return the network.
TNetNetwork& TNetSubnetwork::GetNetwork()
{
    return *fNetwork;
}

const TNetNetwork& TNetSubnetwork::GetNetwork() const
{
    return *fNetwork;
}
```

KK. TNetTimedControl Class

1. TimedControl.h

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: TimedControl.h,v $
//  $Revision: 2.3 $
//  $Date: 1996/08/29 15:22:01 $
//  $State: Stab $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


#ifndef TRANSIMS_NET_TIMEDCONTROL
#define TRANSIMS_NET_TIMEDCONTROL


//  Include TRANSIMS header files.
#include "GBL/Globals.h"
#include "NET/SignalizedControl.h"


//  Include Booch Components header files.
#include "BCStoreM.h"
#include "BCMapU.h"


//  Forward declarations.
class TNetNode;
class TNetPhase;
class TNetPhaseDescription;
class TNetSignalizedControlReader;


//  A timed control specifies the performance of a pre-timed signal.
class TNetTimedControl
  : public TNetSignalizedControl
{
  public:

    //  Type definitions.
    typedef BC_TUnboundedMap<TNetPhase*, REAL, 4U, BC_CManaged> PhaseEndMap;
    typedef BC_TMapActiveIterator<TNetPhase*, REAL> PhaseEndMapIterator;

    //  Construct a timed signalized traffic control.
    TNetTimedControl();
    TNetTimedControl(TNetNode& node);
    TNetTimedControl(const TNetTimedControl& control);

    //  Construct a timed signalized traffic control using the reader.
    TNetTimedControl(TNetSignalizedControlReader& reader, TNetNetwork& network);

    //  Destroy a timed signalized traffic control.
    ~TNetTimedControl();

    //  Assign a timed control.
    TNetTimedControl& operator=(const TNetTimedControl& control);

    //  Return whether two timed controls are the same.
    bool operator==(const TNetTimedControl& control) const;
    bool operator!=(const TNetTimedControl& control) const;

     //  Update the traffic control state according to algorithm for fixed time
     //  controllers.
    virtual void UpdateSignalizedControl(REAL sim_time);

    //  Create a phase.
    virtual TNetPhase& CreatePhase(TNetPhaseDescription& description);

  private:

    //  Cycle length.
```

```
    REAL fCycleLength;

    //  Times at which each phase ends.
    PhaseEndMap  fPhaseEnd;
};


#endif // TRANSIMS_NET_TIMEDCONTROL
```

2. TimedControl.C

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: TimedControl.C,v $
//  $Revision: 2.3 $
//  $Date: 1996/09/20 22:07:56 $
//  $State: Stab $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


//  Include TRANSIMS header files.
#include "NET/SignalizedControl.h"
#include "NET/Network.h"
#include "NET/TimedControl.h"
#include "NET/Phase.h"
#include "NET/PhasingPlan.h"
#include "NET/Exception.h"
#include "NET/SignalizedControlReader.h"


//  Include C header files.
#include <math.h>
#include <float.h>


//  Define the hash function for phases.
static BC_Index PhaseHashValue(TNetPhase* const & p)
{
    return ((int)p & 0xf0) >> 4;
}


//  Construct a timed signalized traffic control.
TNetTimedControl::TNetTimedControl()
  : TNetSignalizedControl(),
    fCycleLength(0.0),
    fPhaseEnd()
{
    fPhaseEnd.SetHashFunction(PhaseHashValue);
}

TNetTimedControl::TNetTimedControl(TNetNode& node)
  : TNetSignalizedControl(node),
    fCycleLength(0.0),
    fPhaseEnd()
{
    fPhaseEnd.SetHashFunction(PhaseHashValue);
}

TNetTimedControl::TNetTimedControl(const TNetTimedControl& c)
  : TNetSignalizedControl(c),
    fPhaseEnd()
{
    fPhaseEnd.SetHashFunction(PhaseHashValue);
    fCycleLength = c.fCycleLength;
    fPhaseEnd = c.fPhaseEnd;
}


//  Construct a timed signalized traffic control using the reader.
TNetTimedControl::TNetTimedControl(TNetSignalizedControlReader& reader,
        TNetNetwork& network)
  : TNetSignalizedControl(**network.GetNodes().ValueOf(reader.GetNode())),
```

```
    fCycleLength(0.0),
    fPhaseEnd()
{
    fPhaseEnd.SetHashFunction(PhaseHashValue);
}


//  Destroy a timed signalized traffic control.
TNetTimedControl::~TNetTimedControl()
{
}


//  Assign a timed control.
TNetTimedControl& TNetTimedControl::operator=(const TNetTimedControl& c)
{
    if (this == &c)
        return *this;
    TNetSignalizedControl::operator=(c);
    fCycleLength = c.fCycleLength;
    fPhaseEnd = c.fPhaseEnd;
    return *this;
}


//  Return whether timed controls are the same.
bool TNetTimedControl::operator==(const TNetTimedControl& c) const
{
    return (this == &c);
}

bool TNetTimedControl::operator!=(const TNetTimedControl& c) const
{
    return !(this == &c);
}


//  Update the traffic control state according to algorithm for fixed time
//  controllers.
void TNetTimedControl::UpdateSignalizedControl(REAL sim_time)
{
    TNetPhase* current_phase;
    REAL current_value = 1000000.;        //  any number > max. cycle length
    REAL offset = GetPhasingPlanOffset();
    offset = fmod (offset, fCycleLength);
    REAL t = fmod (sim_time + fCycleLength - offset, fCycleLength);
    if (t < FLT_EPSILON) t = fCycleLength;
    for (PhaseEndMapIterator iter(fPhaseEnd);
         !iter.IsDone(); iter.Next())
        if (*iter.CurrentValue() < current_value  &&  t <=
                *iter.CurrentValue()) {
            current_phase = *iter.CurrentItem();
            current_value = *iter.CurrentValue();
        }
    if (*current_phase != GetPhase())
        SetPhase (*current_phase);

    TNetPhase::EInterval current_intv;
    TNetPhaseDescription& desc = current_phase->GetPhaseDescription ();
    REAL phase_length = desc.GetGreenLength() + desc.GetYellowLength() +
            desc.GetRedLength();
    t = t + phase_length - current_value;
    if (t <= desc.GetGreenLength())
        current_intv = TNetPhase::kGreen;
    else if (t <= desc.GetGreenLength() + desc.GetYellowLength())
        current_intv = TNetPhase::kYellow;
    else if (t <= phase_length)
        current_intv = TNetPhase::kRed;
    else
        throw TNetException("Error in interval determination.");
    if (current_intv != current_phase->GetInterval())
        current_phase->SetInterval (current_intv);
}


//  Create a phase.
```

```
TNetPhase& TNetTimedControl::CreatePhase(TNetPhaseDescription& d)
{
    TNetPhase* p = new TNetPhase(d);
    GetPhases().Append(p);
    REAL phase_length = d.GetGreenLength() + d.GetYellowLength() +
            d.GetRedLength();
    fPhaseEnd.Bind (p, fCycleLength+phase_length);
    fCycleLength += phase_length;
    return *p;
}
```

LL. TNetTimingPlanReader Class

1. TimingPlanReader.h

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: TimingPlanReader.h,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 19:29:51 $
//  $State: Rel $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


#ifndef TRANSIMS_NET_TIMEREADER
#define TRANSIMS_NET_TIMEREADER


//  Include TRANSIMS header files.
#include <GBL/Globals.h>
#include <DBS/Accessor.h>
#include <NET/Id.h>


//  Forward declarations.
class TNetReader;


//  This reader reads timing plan values from the database.
class TNetTimingPlanReader
{
  public:

    //  Construct a timing plan reader for a given network.
    TNetTimingPlanReader(TNetReader& reader);

    //  Construct a copy of the given timing plan reader.
//  TNetTimingPlanReader(const TNetTimingPlanReader& reader);

    //  Make the reader a copy of the given timing plan reader.
//  TNetTimingPlanReader& operator=(const TNetTimingPlanReader& reader);

    //  Reset the iteration over the table.
    void Reset();

    //  Get the next plan in the table.
    void GetNextPlan();

    //  Return whether there are any more plans in the table.
    bool MorePlans();

    //  Return the timing plan id.
    NetPlanId GetPlan() const;

    //  Return the phase number.
    NetPhaseNumber GetPhase() const;

    //  Return the phase numbers of the next phases.
    string GetNextPhases() const;

    //  Return the minimum length of green interval.
    REAL GetGreenMin() const;
```

```
    //  Return the maximum length of green interval.
    REAL GetGreenMax() const;

    //  Return the length of green interval extension.
    REAL GetGreenExt() const;

    //  Return the length of yellow interval.
    REAL GetYellow() const;

    //  Return the length of red clearance interval.
    REAL GetRedClear() const;

  private:

    //  Each timing plan has a database table accessor.
    TDbAccessor fAccessor;

    //  Each record has a timing plan ID.
    const TDbField fPlanField;

    //  Each timing plan has phase numbers.
    const TDbField fPhaseField;

    //  Each phase has one or more next phases.
    const TDbField fNextphasesField;

    //  Each phase has a minimum green length.
    const TDbField fGreenminField;

    //  Each phase has a maximum green length.
    const TDbField fGreenmaxField;

    //  Each phase has a green extension length.
    const TDbField fGreenextField;

    //  Each phase has a yellow length.
    const TDbField fYellowField;

    //  Each phase has a red clearance length.
    const TDbField fRedclearField;
};


#endif // TRANSIMS_NET_TIMEREADER
```

2. TimingPlanReader.C

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: TimingPlanReader.C,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 19:29:51 $
//  $State: Rel $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


//  Include TRANSIMS header files.
#include <NET/TimingPlanReader.h>
#include <NET/Reader.h>


//  Construct a timing plan reader for a given network.
TNetTimingPlanReader::TNetTimingPlanReader(TNetReader& reader)
  : fAccessor(reader.GetTimingPlanTable()),
    fPlanField(reader.GetTimingPlanTable().GetField("PLAN")),
    fPhaseField(reader.GetTimingPlanTable().GetField("PHASE")),
    fNextphasesField(reader.GetTimingPlanTable().GetField("NEXTPHASES")),
    fGreenminField(reader.GetTimingPlanTable().GetField("GREENMIN")),
    fGreenmaxField(reader.GetTimingPlanTable().GetField("GREENMAX")),
    fGreenextField(reader.GetTimingPlanTable().GetField("GREENEXT")),
    fYellowField(reader.GetTimingPlanTable().GetField("YELLOW")),
    fRedclearField(reader.GetTimingPlanTable().GetField("REDCLEAR"))
```

```
{
}


//  Reset the iteration over the table.
void TNetTimingPlanReader::Reset()
{
    fAccessor.GotoFirst();
}


//  Get the next plan in the table.
void TNetTimingPlanReader::GetNextPlan()
{
    fAccessor.GotoNext();
}


//  Return whether there are any more plans in the table.
bool TNetTimingPlanReader::MorePlans()
{
    return fAccessor.IsAtRecord();
}


//  Return the timing plan id.
NetPlanId TNetTimingPlanReader::GetPlan() const
{
    NetPlanId plan;
    fAccessor.GetField(fPlanField, plan);
    return plan;
}


//  Return the phase number.
NetPhaseNumber TNetTimingPlanReader::GetPhase() const
{
    NetPhaseNumber phase;
    fAccessor.GetField(fPhaseField, phase);
    return phase;
}


//  Return the phase numbers of the next phases.
string TNetTimingPlanReader::GetNextPhases() const
{
    string next;
    fAccessor.GetField(fNextphasesField, next);
    return next;
}


//  Return the minimum length of green interval.
REAL TNetTimingPlanReader::GetGreenMin() const
{
    REAL min;
    fAccessor.GetField(fGreenminField, min);
    return min;
}


//  Return the maximum length of green interval.
REAL TNetTimingPlanReader::GetGreenMax() const
{
    REAL max;
    fAccessor.GetField(fGreenmaxField, max);
    return max;
}


//  Return the length of green interval extension.
REAL TNetTimingPlanReader::GetGreenExt() const
{
    REAL ext;
    fAccessor.GetField(fGreenextField, ext);
    return ext;
```

```
}


//  Return the length of yellow interval.
REAL TNetTimingPlanReader::GetYellow() const
{
    REAL yellow;
    fAccessor.GetField(fYellowField, yellow);
    return yellow;
}


//  Return the length of red clearance interval.
REAL TNetTimingPlanReader::GetRedClear() const
{
    REAL red;
    fAccessor.GetField(fRedclearField, red);
    return red;
}
```

MM. TNetTrafficControl Class

1. TrafficControl.h

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: TrafficControl.h,v $
//  $Revision: 2.5 $
//  $Date: 1996/12/13 15:27:41 $
//  $State: Stab $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


#ifndef TRANSIMS_NET_TRAFFICCONTROL
#define TRANSIMS_NET_TRAFFICCONTROL


//  Include TRANSIMS header files.
#include "GBL/Globals.h"


//  Include Booch Components header files.
#include "BCStoreM.h"
#include "BCMapU.h"
#include "BCCollU.h"


//  Include C header files.
#include <stddef.h>


//  Forward declarations.
class TNetNetwork;
class TNetNode;
class TNetLink;
class TNetLane;
class TNetLaneConnectivityReader;


//  A traffic control is associated with each node.  The traffic control
//  specifies how lanes are connected across the node and the type of
//  sign or signalized control that determines who has the right-of-way.
class TNetTrafficControl
{
  public:

    //  Type definitions.
    typedef BC_TUnboundedCollection<TNetLane*, BC_CManaged> LaneCollection;
    typedef BC_TUnboundedCollection<TNetLane*, BC_CManaged> ConnectedCollection;
    typedef BC_TCollectionActiveIterator<TNetLane*> LaneCollectionIterator;
    typedef BC_TUnboundedMap<TNetLane*, ConnectedCollection, 16U, BC_CManaged>
            ConnectivityMap;
```

```
  typedef BC_TMapActiveIterator<TNetLane*, ConnectedCollection>
          ConnectivityMapIterator;

  //  Vehicle signs and signals indicate right of way for movements.
  enum ETrafficControl {kNone, kStop, kYield, kWait, kCaution, kPermitted,
          kProtected};

  //  Pedestrian signals display a walk indicator
  //  enum EPedestrianControl { ... };

  //  Construct a traffic control .
  TNetTrafficControl();
  TNetTrafficControl(TNetNode& node);
  TNetTrafficControl(const TNetTrafficControl& control);

  //  Destroy a traffic control.
  virtual ~TNetTrafficControl();

  //  Return the lanes on next link to which transition from specified lane on
  //  this link can be made.  Return an empty collection if transition is not
  //  possible.
  virtual void AllowedMovements (LaneCollection& lanes, const TNetLink&
          fromlink, const TNetLane& fromlane, const TNetLink& tolink);

  //  Return the lanes on this link from which transition to specified lane on
  //  next link can be made.  Return an empty collection if transition is not
  //  possible.
  virtual void AllowedMovements (LaneCollection& lanes, const TNetLink&
          fromlink, const TNetLink& tolink, const TNetLane& tolane);

  //  Return the lanes ordered from median that may be used to transition
  //  from current link to next link.
  virtual void AllowedMovements (LaneCollection& lanes, const TNetLink&
          fromlink, const TNetLink& tolink, bool phase = FALSE);

  //  Return the lanes that must be examined for interference when
  //  transitioning from current lane to next lane.
  virtual void InterferingLanes (LaneCollection& lanes, const TNetLane&
          fromlane, const TNetLane& tolane, bool phase = FALSE);

  //  Return the vehicle control signal for the specified lane.
  virtual ETrafficControl GetVehicleControl(const TNetLane& lane) const = 0;

  //  Return the associated node.
  TNetNode& GetNode();
  const TNetNode& GetNode() const;

  //  Define the lane connectivity for the specified lane.
  void SetConnectivity(TNetLane& inlane, TNetLane& outlane);

  //  Define the lane connectivity using the reader.
  void SetConnectivity (TNetLaneConnectivityReader& reader, TNetNetwork&
         network);

  //  Return the lane connectivity for the specified lane.
  TNetTrafficControl::ConnectedCollection& GetConnectivity(const TNetLane&
          lane);
  const TNetTrafficControl::ConnectedCollection& GetConnectivity(const
          TNetLane& lane) const;

  //  Return the lane connectivity map.
  TNetTrafficControl::ConnectivityMap& GetConnectivity();
  const TNetTrafficControl::ConnectivityMap& GetConnectivity() const;

protected:

  //  Assign a traffic control.
  TNetTrafficControl& operator=(const TNetTrafficControl& control);

private:

  //  A traffic control contains a table that describes how lanes are
  //  connected across the node.
  ConnectivityMap fLaneConnectivity;

  //  Traffic control knows its associated node
```

```
    TNetNode* fNode;
};


#endif  // TRANSIMS_NET_TRAFFICCONTROL
```

2. TrafficControl.C

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: TrafficControl.C,v $
//  $Revision: 2.4 $
//  $Date: 1996/12/13 15:27:41 $
//  $State: Stab $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


//  Include standard C header files
#include <math.h>


//  Include TRANSIMS header files.
#include "NET/TrafficControl.h"
#include "NET/Network.h"
#include "NET/Node.h"
#include "NET/Link.h"
#include "NET/Lane.h"
#include "NET/Exception.h"
#include "NET/LaneConnectivityReader.h"


//  Define the hash function for lanes.
static BC_Index LaneHashValue (TNetLane* const & l)
{
    return ((int)l & 0xf0) >> 4;
}


//  Construct a traffic control.
TNetTrafficControl::TNetTrafficControl()
  : fNode((TNetNode*)NULL),
    fLaneConnectivity(LaneHashValue)
{
}

TNetTrafficControl::TNetTrafficControl(TNetNode& node)
  : fNode(&node),
    fLaneConnectivity(LaneHashValue)
{
    node.SetTrafficControl (this);
}

TNetTrafficControl::TNetTrafficControl(const TNetTrafficControl& tc)
  : fLaneConnectivity(LaneHashValue)
{
    fLaneConnectivity = tc.fLaneConnectivity;
    fNode = tc.fNode;
}


//  Destroy a traffic control.
TNetTrafficControl::~TNetTrafficControl ()
{
}


//  Assign a traffic control.
TNetTrafficControl& TNetTrafficControl::operator=(const TNetTrafficControl& tc)
{
    if (this == &tc)
        return *this;
    fLaneConnectivity = tc.fLaneConnectivity;
    fNode = tc.fNode;
```

```
    return *this;
}


//  Return the lanes on next link to which transition from specified lane on
//  this link can be made.  Return an empty collection if transition is not
//  possible.
void TNetTrafficControl::AllowedMovements(LaneCollection& tolanes, const
        TNetLink& fromlink, const TNetLane& fromlane, const TNetLink& tolink)
{
    tolanes.Clear();
    if (fromlink.GetLanesTowards(*fNode,TRUE).Location((TNetLane*)&fromlane) !=
            -1) {
        if (fLaneConnectivity.IsBound ((TNetLane*)&fromlane)) {
            ConnectedCollection& lanes = *fLaneConnectivity.ValueOf((TNetLane*)
                    &fromlane);
            for (LaneCollectionIterator itto(tolink.GetLanesFrom(*fNode,TRUE));
                    !itto.IsDone(); itto.Next()) {
                TNetLane* current = *itto.CurrentItem();
                if (lanes.Location (current) != -1)
                    tolanes.Append (current);
            }
        }
    }
}


//  Return the lanes on this link from which transition to specified lane on
//  next link can be made.  Return an empty collection if transition is not
//  possible.
void TNetTrafficControl::AllowedMovements(LaneCollection& fromlanes, const
        TNetLink& fromlink, const TNetLink& tolink, const TNetLane& tolane)
{
    fromlanes.Clear();
    if (tolink.GetLanesFrom(*fNode,TRUE).Location((TNetLane*)&tolane) != -1) {
        for (LaneCollectionIterator itfrom(fromlink.GetLanesTowards(*fNode,
                TRUE)); !itfrom.IsDone(); itfrom.Next()) {
            TNetLane* current = *itfrom.CurrentItem();
            if (fLaneConnectivity.IsBound (current)) {
                ConnectedCollection& lanes =
                        *fLaneConnectivity.ValueOf(current);
                if (lanes.Location((TNetLane*)&tolane) != -1  &&
                        fromlanes.Location(current) == -1)
                    fromlanes.Append (current);
            }
        }
    }
}


//  Return the lanes ordered from median that may be used to transition
//  from current link to next link, ignoring phase.
void TNetTrafficControl::AllowedMovements(LaneCollection& fromlanes, const
        TNetLink& fromlink, const TNetLink& tolink, bool /*phase*/)
{
    LaneCollectionIterator itto(tolink.GetLanesFrom(*fNode,TRUE));

    fromlanes.Clear();
    for (LaneCollectionIterator itfrom(fromlink.GetLanesTowards(*fNode,TRUE));
             !itfrom.IsDone(); itfrom.Next()) {
        TNetLane* current = *itfrom.CurrentItem();
        if (fLaneConnectivity.IsBound (current)) {
            ConnectedCollection& lanes = *fLaneConnectivity.ValueOf (current);
            for (itto.Reset(); !itto.IsDone(); itto.Next())
                if (lanes.Location (*itto.CurrentItem()) != -1  &&
                        fromlanes.Location(current) == -1)
                    fromlanes.Append (current);
        }
    }
}


//  Return the lanes that must be examined for interference when transitioning
//  from current lane to next lane, ignoring phase.
void TNetTrafficControl::InterferingLanes(LaneCollection& fromlanes, const
        TNetLane& fromlane, const TNetLane& tolane, bool /*phase*/)
```

```
{
    fromlanes.Clear();
    bool found = FALSE;
    if (fLaneConnectivity.IsBound ((TNetLane*)&fromlane))
        for (LaneCollectionIterator itl(*fLaneConnectivity.ValueOf((TNetLane*)
                &fromlane)); !itl.IsDone(); itl.Next())
            if (*itl.CurrentItem() == &tolane) {
                found = TRUE;
                break;
            }
    if (!found)
        return;        // no path from fromlane to tolane
    if (GetVehicleControl(fromlane) == TNetTrafficControl::kNone)
        return;                        // fromlane always has right-of-way

    const TNetLink *fromlink = &fromlane.GetLink();
    const TNetLink *tolink = &tolane.GetLink();
    TNetNode::LinkRing& links = fNode->GetLinks();
    while (links.Top() != fromlink)
        links.Rotate();

    //  Can rotate in either direction first, but more efficient to rotate in
    //  shortest direction first.  dir is shortest direction between fromlink
    //  and tolink; opsdir is longest (opposite) direction.
    BC_Direction dir, opsdir;
    REAL angle1 = fromlink->GetAngle(*fNode);
    if (angle1 < 0.0)
        angle1 += 2.0*M_PI;
    REAL angle2 = tolink->GetAngle(*fNode);
    if (angle2 < 0.0)
        angle2 += 2.0*M_PI;
    if (angle2 < angle1) angle2 += 2.0*M_PI;
    REAL angle = angle2 - angle1;
    if (angle <= 3.66) { // == 210 deg.  (Use 180 deg. minimim; a slightly
                         // larger angle may be more efficient for intersections
                         // that are not quite square.)
        dir = BC_kReverse;
        opsdir = BC_kForward;
    } else {
        dir = BC_kForward;
        opsdir = BC_kReverse;
    }

    ConnectedCollection tolanes;
    ConnectedCollection frlanes;
    if (dir == BC_kForward) {
        //  tolanes includes lanes leaving on fromlink
        for (LaneCollectionIterator itto(links.Top()->GetLanesFrom(*fNode,
                TRUE)); !itto.IsDone(); itto.Next())
            tolanes.Append(*itto.CurrentItem());
    }
    links.Rotate(dir);
    while (links.Top() != tolink) {
        //  tolanes includes lanes leaving on links between fromlink and tolink
        for (LaneCollectionIterator itto(links.Top()->GetLanesFrom(*fNode,
                TRUE)); !itto.IsDone(); itto.Next())
            tolanes.Append(*itto.CurrentItem());
        //  frlanes includes lanes arriving on links between fromlink and
        //  tolink
        for (LaneCollectionIterator itfr(links.Top()->GetLanesTowards(*fNode,
                TRUE)); !itfr.IsDone(); itfr.Next())
            frlanes.Append(*itfr.CurrentItem());
        links.Rotate(dir);
    }
    if (dir == BC_kForward) {
        //  frlanes includes lanes arriving on tolink
        for (LaneCollectionIterator itfr(links.Top()->GetLanesTowards(*fNode,
                TRUE)); !itfr.IsDone(); itfr.Next())
            frlanes.Append(*itfr.CurrentItem());
    }

    //  include lanes leaving tolink up to tolane in tolanes
    int index = tolink->GetLanesFrom(*fNode, TRUE).Location((TNetLane*)
            &tolane);
    int i=0;
    for (LaneCollectionIterator itto(tolink->GetLanesFrom(*fNode, TRUE));
```

```
            !itto.IsDone(); itto.Next()) {
        if (dir == BC_kForward && i < index)
            tolanes.Append(*itto.CurrentItem());
        else if (dir == BC_kReverse && i > index)
            tolanes.Append(*itto.CurrentItem());
        else
            ;   //  do nothing
        ++i;
    }

    for (LaneCollectionIterator itf1(frlanes); !itf1.IsDone(); itf1.Next()) {
        TNetLane* current = *itf1.CurrentItem();
        if (fLaneConnectivity.IsBound (current))
            //  fromlanes includes frlanes having connectivity to any lane not
            //  in tolanes
            for (LaneCollectionIterator
                    itl(*fLaneConnectivity.ValueOf(current)); !itl.IsDone();
                    itl.Next())
                if (tolanes.Location(*itl.CurrentItem()) == -1) {
                    fromlanes.Append (current);
                    break;
                }
    }

    //  include tolane in tolanes.
    tolanes.Append((TNetLane*)&tolane);
    while (links.Top() != fromlink)
        links.Rotate(opsdir);
    links.Rotate(opsdir);

    frlanes.Clear();
    while (links.Top() != tolink) {
        //  new frlanes includes lanes arriving on links between fromlink and
        //  tolink in opposite direction
        for (LaneCollectionIterator itfr(links.Top()->GetLanesTowards(*fNode,
                TRUE)); !itfr.IsDone(); itfr.Next())
            frlanes.Append(*itfr.CurrentItem());
        links.Rotate(opsdir);
    }
    if (opsdir == BC_kForward) {
        //  frlanes includes lanes arriving on tolink
        for (LaneCollectionIterator itfr(links.Top()->GetLanesTowards(*fNode,
                TRUE)); !itfr.IsDone(); itfr.Next())
            frlanes.Append(*itfr.CurrentItem());
    }

    for (LaneCollectionIterator itf2(frlanes); !itf2.IsDone(); itf2.Next()) {
        TNetLane* current = *itf2.CurrentItem();
        if (fLaneConnectivity.IsBound (current))
            //  fromlanes includes frlanes having connectivity to any lane in
            //  tolanes
            for (LaneCollectionIterator
                    itl(*fLaneConnectivity.ValueOf(current)); !itl.IsDone();
                    itl.Next())
                if (tolanes.Location (*itl.CurrentItem()) != -1 &&
                        fromlanes.Location(current) == -1) {
                    fromlanes.Append (current);
                    break;
                }
    }
}


//  Return the associated node.
TNetNode& TNetTrafficControl::GetNode()
{
    return *fNode;
}

const TNetNode& TNetTrafficControl::GetNode() const
{
    return *fNode;
}


//  Define the lane connectivity for the specified lane.
```

```
void TNetTrafficControl::SetConnectivity(TNetLane& l1, TNetLane& l2)
{
    if (!fLaneConnectivity.IsBound(&l1)) {
        ConnectedCollection col;
        col.Append(&l2);
        fLaneConnectivity.Bind(&l1, col);
    } else
        fLaneConnectivity.ValueOf(&l1)->Append(&l2);
}


//  Define the lane connectivity using the reader.
void TNetTrafficControl::SetConnectivity(TNetLaneConnectivityReader& reader,
        TNetNetwork& network)
{
    TNetNode* node = *(network.GetNodes().ValueOf(reader.GetNode()));
    TNetLink* inlink = *(network.GetLinks().ValueOf(reader.GetInlink()));
    TNetLink* outlink = *(network.GetLinks().ValueOf(reader.GetOutlink()));
    TNetLane* inlane = inlink->GetLanesTowards(*node,TRUE)[reader.GetInlane() –
            1];
    TNetLane* outlane = outlink->GetLanesFrom(*node,TRUE)[reader.GetOutlane() –
            1];

    if (!fLaneConnectivity.IsBound(inlane)) {
        ConnectedCollection col;
        col.Append(outlane);
        fLaneConnectivity.Bind(inlane, col);
    } else
        fLaneConnectivity.ValueOf(inlane)->Append(outlane);
}


//  Return the lane connectivity for the specified lane.
TNetTrafficControl::ConnectedCollection&
        TNetTrafficControl::GetConnectivity(const TNetLane& l1)
{
    return *fLaneConnectivity.ValueOf((TNetLane*)&l1);
}

const TNetTrafficControl::ConnectedCollection&
        TNetTrafficControl::GetConnectivity(const TNetLane& l1) const
{
    return *fLaneConnectivity.ValueOf((TNetLane*)&l1);
}


//  Return the lane Connectivity map.
TNetTrafficControl::ConnectivityMap& TNetTrafficControl::GetConnectivity()
{
    return fLaneConnectivity;
}

const TNetTrafficControl::ConnectivityMap& TNetTrafficControl::GetConnectivity()
        const
{
    return fLaneConnectivity;
}
```

NN. TNetUnsignalizedControl Class

1. UnsignalizedControl.h

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: UnsignalizedControl.h,v $
//  $Revision: 2.2 $
//  $Date: 1996/11/08 18:28:56 $
//  $State: Stab $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


#ifndef TRANSIMS_NET_UNSIGNALIZEDCONTROL
```

```
#define TRANSIMS_NET_UNSIGNALIZEDCONTROL


//  Include TRANSIMS header files.
#include "GBL/Globals.h"
#include "NET/TrafficControl.h"


//  Include Booch Components header files.
#include "BCStoreM.h"
#include "BCMapU.h"


//  Forward declarations.
class TNetNetwork;
class TNetNode;
class TNetLink;
class TNetLane;
class TNetUnsignalizedControlReader;


//  An unsignalized control specifies the sign control at a node.
class TNetUnsignalizedControl
  : public TNetTrafficControl
{
  public:

    //  Type definitions.
    typedef BC_TUnboundedMap<TNetLink*, TNetTrafficControl::ETrafficControl, 4U,
            BC_CManaged> SignMap;
    typedef BC_TMapActiveIterator<TNetLink*,
            TNetTrafficControl::ETrafficControl> SignMapIterator;

    //  Construct an unsignalized traffic control.
    TNetUnsignalizedControl();
    TNetUnsignalizedControl(TNetNode& node);
    TNetUnsignalizedControl(const TNetUnsignalizedControl& control);

    //  Construct an unsignalized traffic control using the reader.
    TNetUnsignalizedControl(TNetUnsignalizedControlReader& reader, TNetNetwork&
            network);

    //  Destroy an unsignalized traffic control.
    ~TNetUnsignalizedControl();

    //  Assign an unsignalized traffic control.
    TNetUnsignalizedControl& operator=(const TNetUnsignalizedControl&
            control);

    //  Return whether two unsignalized controls are the same.
    bool operator==(const TNetUnsignalizedControl& control) const;
    bool operator!=(const TNetUnsignalizedControl& control) const;

    //  Return the vehicle control signal for the specified lane.
    TNetTrafficControl::ETrafficControl GetVehicleControl(const TNetLane& lane)
            const;

    //  Define the vehicle control sign.
    void SetVehicleControl(TNetLink& link, TNetTrafficControl::ETrafficControl);

    //  Define the vehicle control sign using the reader.
    void SetVehicleControl(TNetUnsignalizedControlReader& reader, TNetNetwork&
            network);

  private:

    //  Some type of sign control is associated with each link attached to an
    //  unsignalized intersection.  Example values are stop, yield, and no control
    //  on this link.
    SignMap fSigns;
};


#endif  // TRANSIMS_NET_UNSIGNALIZEDCONTROL
```

2. UnsignalizedControl.C

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: UnsignalizedControl.C,v $
//  $Revision: 2.2 $
//  $Date: 1996/11/08 18:28:56 $
//  $State: Stab $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


//  Include TRANSIMS header files.
#include "NET/UnsignalizedControl.h"
#include "NET/Network.h"
#include "NET/Node.h"
#include "NET/Link.h"
#include "NET/Lane.h"
#include "NET/Id.h"
#include "NET/Exception.h"
#include "NET/UnsignalizedControlReader.h"


//  Define the hash function for links.
static BC_Index LinkHashValue(TNetLink* const & l)
{
    return ((int)l & 0xf0) >> 4;
}


//  Construct an unsignalized traffic control.
TNetUnsignalizedControl::TNetUnsignalizedControl()
  : TNetTrafficControl(),
    fSigns()
{
    fSigns.SetHashFunction(LinkHashValue);
}

TNetUnsignalizedControl::TNetUnsignalizedControl(TNetNode& node)
  : TNetTrafficControl(node),
    fSigns()
{
    fSigns.SetHashFunction(LinkHashValue);
}

TNetUnsignalizedControl::TNetUnsignalizedControl(const TNetUnsignalizedControl&
        u)
  : TNetTrafficControl(u),
    fSigns()
{
    fSigns.SetHashFunction(LinkHashValue);
    fSigns = u.fSigns;
}


//  Construct an unsignalized traffic control using the reader.
TNetUnsignalizedControl::TNetUnsignalizedControl(TNetUnsignalizedControlReader&
        reader, TNetNetwork& network)
  : TNetTrafficControl(**network.GetNodes().ValueOf(reader.GetNode())),
    fSigns()
{
    fSigns.SetHashFunction(LinkHashValue);
}


//  Destroy an unsignalized traffic control.
TNetUnsignalizedControl::~TNetUnsignalizedControl()
{
}


//  Assign an unsignalized traffic control.
TNetUnsignalizedControl& TNetUnsignalizedControl::operator=(const
        TNetUnsignalizedControl& u)
{
```

```
    if (this == &u)
        return *this;
    TNetTrafficControl::operator=(u);
    fSigns = u.fSigns;
    return *this;
}


//  Return whether two unsignalized controls are the same.
bool TNetUnsignalizedControl::operator==(const TNetUnsignalizedControl& u) const
{
    return (this == &u);
}

bool TNetUnsignalizedControl::operator!=(const TNetUnsignalizedControl& u) const
{
    return !(this == &u);
}


//  Return the vehicle control signal for the specified lane.
TNetTrafficControl::ETrafficControl
        TNetUnsignalizedControl::GetVehicleControl(const TNetLane& lane) const
{
    return *fSigns.ValueOf((TNetLink*)&lane.GetLink());
}


//  Define the vehicle control sign.
void TNetUnsignalizedControl::SetVehicleControl(TNetLink& link,
        TNetTrafficControl::ETrafficControl sign)
{
    fSigns.Bind(&link, sign);
}


//  Define the vehicle control sign using the reader.
void TNetUnsignalizedControl::SetVehicleControl(TNetUnsignalizedControlReader&
        reader, TNetNetwork& network)
{
    TNetNetwork::LinkMap& links = network.GetLinks();
    NetLinkId inlink = reader.GetInlink();
    string s = reader.GetSign();
    if (s == "S" && links.IsBound(inlink))
        fSigns.Bind (*links.ValueOf(inlink), TNetTrafficControl::kStop);
    else if (s == "Y" && links.IsBound(inlink))
        fSigns.Bind (*links.ValueOf(inlink), TNetTrafficControl::kYield);
    else if (s == "N" && links.IsBound(inlink))
        fSigns.Bind (*links.ValueOf(inlink), TNetTrafficControl::kNone);
    else
        throw TNetNotFound ("The link is not in the network or the sign type is"
                " invalid.");
}
```

OO. TNetUnsignalizedControlReader Class

1. UnsignalizedControlReader.h

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: UnsignalizedControlReader.h,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 19:29:51 $
//  $State: Rel $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


#ifndef TRANSIMS_NET_UNSIGREADER
#define TRANSIMS_NET_UNSIGREADER


//  Include TRANSIMS header files.
```

```
#include <GBL/Globals.h>
#include <DBS/Accessor.h>
#include <NET/Id.h>


//  Forward declarations.
class TNetReader;


//  This reader reads unsignalized control values from the database.
class TNetUnsignalizedControlReader
{
  public:

    //  Construct an unsignalized control reader for a given network.
    TNetUnsignalizedControlReader(TNetReader& reader);

    //  Construct a copy of the given unsignalized control reader.
//  TNetUnsignalizedControlReader(const TNetUnsignalizedControlReader& reader);

    //  Make the reader a copy of the given unsignalized control reader.
//  TNetUnsignalizedControlReader& operator=(const
//          TNetUnsignalizedControlReader& reader);

    //  Reset the iteration over the table.
    void Reset();

    //  Get the next node in the table.
    void GetNextNode();

    //  Return whether there are any more nodes in the table.
    bool MoreNodes();

    //  Return the id for the current node.
    NetNodeId GetNode() const;

    //  Return the id for incoming link.
    NetLinkId GetInlink() const;

    //  Return the sign control indication on incoming link.
    string GetSign() const;

  private:

    //  Each unsignalized control reader has a database table accessor.
    TDbAccessor fAccessor;

    //  Each record has a node field.
    const TDbField fNodeField;

    //  Each record has an incoming link field.
    const TDbField fInlinkField;

    //  Each record has a sign control field.
    const TDbField fSignField;
};


#endif // TRANSIMS_NET_UNSIGREADER
```

2. UnsignalizedControlReader.C

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: UnsignalizedControlReader.C,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 19:29:51 $
//  $State: Rel $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


//  Include TRANSIMS header files.
#include <NET/UnsignalizedControlReader.h>
```

```
#include <NET/Reader.h>


//  Construct an unsignalized control reader for a given network.
TNetUnsignalizedControlReader::TNetUnsignalizedControlReader(TNetReader& reader)
  : fAccessor(reader.GetUnsignalizedControlTable()),
    fNodeField(reader.GetUnsignalizedControlTable().GetField("NODE")),
    fInlinkField(reader.GetUnsignalizedControlTable().GetField("INLINK")),
    fSignField(reader.GetUnsignalizedControlTable().GetField("SIGN"))
{
}


//  Reset the iteration over the table.
void TNetUnsignalizedControlReader::Reset()
{
    fAccessor.GotoFirst();
}


//  Get the next node in the table.
void TNetUnsignalizedControlReader::GetNextNode()
{
    fAccessor.GotoNext();
}


//  Return whether there are any more nodes in the table.
bool TNetUnsignalizedControlReader::MoreNodes()
{
    return fAccessor.IsAtRecord();
}


//  Return the id for the current node.
NetNodeId TNetUnsignalizedControlReader::GetNode() const
{
    NetNodeId node;
    fAccessor.GetField(fNodeField, node);
    return node;
}


//  Return the id for incoming link.
NetLinkId TNetUnsignalizedControlReader::GetInlink() const
{
    NetLinkId inlink;
    fAccessor.GetField(fInlinkField, inlink);
    return inlink;
}


//  Return the sign control indication on incoming link.
string TNetUnsignalizedControlReader::GetSign() const
{
    string sign;
    fAccessor.GetField(fSignField, sign);
    return sign;
}
```

PP. Constants

1. Id.h

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: Id.h,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 19:29:51 $
//  $State: Rel $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved
```

```
#ifndef TRANSIMS_NET_ID
#define TRANSIMS_NET_ID


//  Include TRANSIMS header files.
#include <GBL/Globals.h>


//  A network node id is four bytes long.
typedef DWORD NetNodeId;


//  A network link id is four bytes long.
typedef DWORD NetLinkId;


//  A network accessory id is four bytes long.
typedef DWORD NetAccessoryId;


//  A network lane number is one byte long.
typedef BYTE NetLaneNumber;


//  A network plan id is one byte long.
typedef BYTE NetPlanId;


//  A network phase number is four bytes long.
//ISSUE(kpb): Do not use BYTE for NetPhaseNumber--causes errors in fNextPhases.
typedef DWORD NetPhaseNumber;


//  A network coordinator id is four bytes long.
typedef DWORD NetCoordinatorId;


#endif // TRANSIMS_NET_ID
```

IX. APPENDIX: Test Program

This appendix contains the complete C++ source code for the network subsystem test program. In order for the tests to run properly, the TRANSIMS database must contain the sample network data in the Appendix.

A. Test.C

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: Test.C,v $
//  $Revision: 2.1 $
//  $Date: 1996/02/14 21:33:43 $
//  $State: Exp $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


//  Include Standard C++ header files.
#include <iostream.h>


//  Include TRANSIMS header files.
#include <DBS/Exception.h>
#include <NET/Exception.h>
#include <NET/Network.h>
#include <NET/Subnetwork.h>


//  Function prototypes.
extern bool TestReader(TNetNetwork&, TNetSubnetwork*&);
```

```
extern bool TestNetwork(TNetNetwork&);
extern bool TestSubnetwork(TNetSubnetwork&);
extern bool TestNode(TNetSubnetwork&);
extern bool TestLink(TNetSubnetwork&);
extern bool TestLane(TNetSubnetwork&);
extern bool TestPocket(TNetSubnetwork&);
extern bool TestParking(TNetSubnetwork&);
extern bool TestGeography();
extern bool TestFilterFunction();
extern bool TestTrafficControl(TNetNetwork&);
extern bool TestUnsignalizedControl(TNetNetwork&);
extern bool TestNullControl(TNetNetwork&);
extern bool TestSignalizedControl(TNetNetwork&);
extern bool TestTimedControl(TNetNetwork&);
extern bool TestSignalCoordinator(TNetNetwork&);
extern bool TestIsolatedControl(TNetNetwork&);
extern bool TestPhasingPlan(TNetNetwork&);
extern bool TestPhaseDescription(TNetNetwork&);
extern bool TestPhase(TNetNetwork&);


//  Main program.
int main(int, char*[])
{
    try {

        bool fail = FALSE;

        cout << "Network Subsystem Tests "
            << "[$Revision: 2.1 $]"
            << endl;

        TNetNetwork network;
        TNetSubnetwork* subnetwork = NULL;

        fail |= !TestReader(network, subnetwork);
        fail |= !TestNetwork(network);
        fail |= !TestSubnetwork(*subnetwork);
        fail |= !TestNode(*subnetwork);
        fail |= !TestLink(*subnetwork);
        fail |= !TestLane(*subnetwork);
        fail |= !TestPocket(*subnetwork);
        fail |= !TestParking(*subnetwork);
        fail |= !TestGeography();
        fail |= !TestFilterFunction();
        fail |= !TestTrafficControl (network);
        fail |= !TestUnsignalizedControl (network);
        fail |= !TestNullControl (network);
        fail |= !TestSignalizedControl (network);
        fail |= !TestTimedControl (network);
        fail |= !TestPhase (network);
        fail |= !TestPhasingPlan (network);
        fail |= !TestPhaseDescription (network);
        fail |= !TestSignalCoordinator (network);
        fail |= !TestIsolatedControl (network);

        delete subnetwork;

        cout << (fail ? "  F" : "  No f") << "ailures occured." << endl;

    } catch(const TNetException& exception) {

        cout << endl;
        cout << "Unexpected network subsystem exception occurred--aborting." <<
                endl;
        cout << "(" << exception.GetMessage() << ")" << endl;

    } catch(const TDbException& exception) {

        cout << endl;
        cout << "Unexpected database subsystem exception occurred--aborting." <<
                endl;
        cout << "(" << exception.GetMessage() << ")" << endl;

    } catch(...) {
```

```
        cout << endl;
        cout << "Unexpected unknown exception occurred--aborting." << endl;

    }

    return 0;
}
```

B. TestFilterFunction.C

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: TestFilterFunction.C,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 19:29:51 $
//  $State: Rel $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


//  Include Standard C++ header files.
#include <iostream.h>


//  Include TRANSIMS header files.
#include <NET/FilterFunction.h>
#include <NET/FilterNone.h>
#include <NET/FilterRectangle.h>


//  Test filter functions.
bool TestFilterFunction()
{
    cout << "  Filter Function Classes Tests "
         << "[$Revision: 2.0 $]"
         << endl;

    bool anyFail = FALSE;
    bool fail;

    cout << "    NET-FF-010: Null filter.";
    fail = !TGeoFilterNone()(TGeoPoint());
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    TGeoFilterRectangle rectFilter(TGeoRectangle(TGeoPoint(3, 2), TGeoPoint(1,
            4)));

    cout << "    NET-FF-020: Point inside rectangular filter.";
    fail = !rectFilter(TGeoPoint(2, 3));
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    NET-FF-030: Point outside rectangular filter.";
    fail = rectFilter(TGeoPoint(2, 5)) || rectFilter(TGeoPoint(2, -1)) ||
            rectFilter(TGeoPoint(5, 3)) || rectFilter(TGeoPoint(-1, 3));
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << (anyFail ? "    F" : "    No f") << "ailures occurred." << endl;
    return !anyFail;
}
```

C. TestGeography.C

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: TestGeography.C,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 19:29:51 $
//  $State: Rel $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved
```

```
//  Include Standard C header files.
#include <math.h>


//  Include Standard C++ header files.
#include <iostream.h>


//  Include TRANSIMS header files.
#include <NET/Point.h>
#include <NET/Rectangle.h>


//  Return whether two real numbers are the same.
static bool Equal(REAL a, REAL b)
{
    return fabs(a - b) < 1e-5;
}


//  Test geography classes.
bool TestGeography()
{
    cout << "  Geography Classes Tests "
         << "[$Revision: 2.0 $]"
         << endl;

    bool anyFail = FALSE;
    bool fail;

    cout << "    NET-GE-010: Point class access.";
    fail = TGeoPoint(1, 4).GetX() != 1 || TGeoPoint(2, 5).GetY() != 5;
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    NET-GE-020: Point class angle calculation.";
    const REAL pi = fabs(atan2(0, -1));
    fail = !Equal(TGeoPoint(1, 2).GetAngleTo(TGeoPoint(3, 4)), pi / 4) ||
            !Equal(TGeoPoint(-1, 5).GetAngleTo(TGeoPoint(-2, 4)), - pi * 3 / 4)
            || !Equal(TGeoPoint(2, 3).GetAngleTo(TGeoPoint(2, 1)), - pi / 2) ||
            !Equal(TGeoPoint(2, 3).GetAngleTo(TGeoPoint(3, 2)), - pi / 4) ||
            !Equal(TGeoPoint(4, 5).GetAngleTo(TGeoPoint(3, 6)), pi * 3 / 4);
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    TGeoRectangle rectangle(TGeoPoint(3, 2), TGeoPoint(1, 4));

    cout << "    NET-GE-030: Rectangle class access.";
    {
        TGeoPoint corner1, corner2;
        rectangle.GetCorners(corner1, corner2);
        fail = corner1.GetX() != 1 || corner1.GetY() != 2 || corner2.GetX() != 3
                || corner2.GetY() != 4;
    }
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    NET-GE-040: Rectangle class containment.";
    fail = !rectangle.Contains(TGeoPoint(2, 3)) ||
            rectangle.Contains(TGeoPoint(2, 5)) ||
            rectangle.Contains(TGeoPoint(2, -1)) ||
            rectangle.Contains(TGeoPoint(5, 3)) ||
            rectangle.Contains(TGeoPoint(-1, 3));
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << (anyFail ? "    F" : "    No f") << "ailures occurred." << endl;
    return !anyFail;
}
```

D. TestIsolatedControl.C

```
//  Project: TRANSIMS
```

```
//  Subsystem: Network
//  $RCSfile: TestIsolatedControl.C,v $
//  $Revision: 2.1 $
//  $Date: 1996/02/14 21:39:13 $
//  $State: Exp $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


//  Include standard C++ header files
#include <iostream.h>

//  Include TRANSIMS header files.
#include "NET/Network.h"
#include "NET/Node.h"
#include "NET/Link.h"
#include "NET/IsolatedControl.h"
#include "NET/SignalizedControl.h"
#include "NET/Phase.h"


//  Test isolated control class.
bool TestIsolatedControl(TNetNetwork& network)
{
    cout << "  Isolated Control Class Tests "
         << endl;

    bool anyFail = FALSE;
    bool fail;

    TNetNetwork::NodeMap& nodes = network.GetNodes();
    TNetNode* n1 = *nodes.ValueOf(14141);
    TNetNetwork::LinkMap& links = network.GetLinks();
    TNetLink* l1 = *links.ValueOf(11487);
    TNetLink* l2 = *links.ValueOf(11486);
    TNetLink* l3 = *links.ValueOf(11495);
    TNetLink* l4 = *links.ValueOf(28800);
    TNetSignalizedControl *s = (TNetSignalizedControl*)&n1->GetTrafficControl();
    TNetSignalCoordinator *c = (TNetSignalCoordinator*)&s->GetCoordinator();

    cout << "    NET-IC-010: Update signalized control to time=130 (phase 1).";
    c->UpdateSignalizedControl(130.);
    //cout << "Current interval = " << s->GetPhase().GetInterval() << endl;
    //cout << "Traffic control on link " << l4->GetId() << " lane 1 = "
    //     << n1->GetTrafficControl().GetVehicleControl(*(l4
    //             ->GetLanesTowards(*n1,TRUE)[0])) << endl;
    //cout << "Traffic control on link " << l4->GetId() << " lane 3 = "
    //     << n1->GetTrafficControl().GetVehicleControl(*(l4
    //             ->GetLanesTowards(*n1,TRUE)[2])) << endl;
    //cout << "Traffic control on link " << l4->GetId() << " lane 6 = "
    //     << n1->GetTrafficControl().GetVehicleControl(*(l4
    //             ->GetLanesTowards(*n1,TRUE)[5])) << endl;
    fail = s->GetPhase().GetInterval() != 0;
    fail |= l4->GetId() != 28800
             || n1->GetTrafficControl().GetVehicleControl(*(l4
             ->GetLanesTowards(*n1,TRUE)[0])) != 3
             || n1->GetTrafficControl().GetVehicleControl(*(l4
             ->GetLanesTowards(*n1,TRUE)[2])) != 3
             || n1->GetTrafficControl().GetVehicleControl(*(l4
             ->GetLanesTowards(*n1,TRUE)[5])) != 3;
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << (anyFail ? "    F" : "    No f") << "ailures occurred." << endl;
    return !anyFail;
}
```

E. TestLane.C

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: TestLane.C,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 19:29:51 $
```

```
//  $State: Rel $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


//  Include Standard C++ header files.
#include <iostream.h>


//  Include TRANSIMS header files.
#include <NET/Network.h>
#include <NET/Subnetwork.h>
#include <NET/Node.h>
#include <NET/Link.h>
#include <NET/Lane.h>


//  Test lane class.
bool TestLane(TNetSubnetwork& subnetwork)
{
    cout << "  Lane Class Tests "
         << "[$Revision: 2.0 $]"
         << endl;

    bool anyFail = FALSE;
    bool fail;

    TNetNetwork::NodeMap& nodes = subnetwork.GetNetwork().GetNodes();
    TNetNetwork::LinkMap& links = subnetwork.GetNetwork().GetLinks();

    TNetNode& node = **nodes.ValueOf(8520);
    TNetLink& link = **links.ValueOf(28800);
    TNetLink::LaneCollection& lanes = link.GetLanesFrom(node, TRUE);

    cout << "    NET-LA-010: Lane construction.";
    fail = lanes[2]->GetLink() != link;
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    NET-LA-020: Starting node access.";
    fail = lanes[3]->GetStartNode().GetId() != 8520;
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    NET-LA-030: Ending node access.";
    fail = lanes[1]->GetEndNode().GetId() != 14141;
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    NET-LA-040: Pocket access.";
    fail = lanes[0]->GetPockets().Length() != 1 ||
            lanes[1]->GetPockets().Length() != 0 ||
            lanes[5]->GetPockets().Length() != 1;
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    NET-LA-050: Left adjacent lane access.";
    fail = lanes[1]->GetLeftAdjacentLane() != lanes[0];
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    NET-LA-060: Left adjacent lane access failure.";
    fail = lanes[0]->GetLeftAdjacentLane() != NULL;
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    NET-LA-070: Right adjacent lane access.";
    fail = lanes[2]->GetRightAdjacentLane() != lanes[3];
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    NET-LA-080: Right adjacent lane access failure.";
    fail = lanes[5]->GetRightAdjacentLane() != NULL;
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;
```

```
    cout << "    NET-LA-090: Lane numbering.";
    fail = lanes[3]->GetNumber() != 4;
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << (anyFail ? "    F" : "    No f") << "ailures occurred." << endl;
    return !anyFail;
}
```

F. TestLink.C

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: TestLink.C,v $
//  $Revision: 2.1 $
//  $Date: 1996/05/02 19:50:38 $
//  $State: Rel $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


//  Include Standard C header files.
#include <math.h>


//  Include Standard C++ header files.
#include <iostream.h>


//  Include TRANSIMS header files.
#include <NET/Exception.h>
#include <NET/Network.h>
#include <NET/Subnetwork.h>
#include <NET/Node.h>
#include <NET/Link.h>


//  Return whether two real numbers are the same.
static bool Equal(REAL a, REAL b)
{
    return fabs(a - b) < 1e-5;
}


//  Test link class.
bool TestLink(TNetSubnetwork& subnetwork)
{
    cout << "  Link Class Tests "
         << "[$Revision: 2.1 $]"
         << endl;

    bool anyFail = FALSE;
    bool fail;

    TNetNetwork::NodeMap& nodes = subnetwork.GetNetwork().GetNodes();
    TNetNetwork::LinkMap& links = subnetwork.GetNetwork().GetLinks();

    cout << "    NET-LI-010: Link construction.";
    TNetLink& link = **links.ValueOf(12407);
    fail = link.GetId() != 12407;
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    NET-LI-020: Link node access.";
    NetNodeId idA, idB;
    TNetNode *nodeA, *nodeB;
    link.GetNodes(idA, idB);
    link.GetNodes(nodeA, nodeB);
    fail = idA != 8521 || idB != 14136 || nodeA != *nodes.ValueOf(8521) ||
        nodeB != *nodes.ValueOf(14136);
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;
```

```
cout << "    NET-LI-030: Accessory access.";
fail = link.GetAccessories().Length() != 2;
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    NET-LI-040: Lanes going away from link w/o pockets.";
fail = link.GetLanesFrom(*nodeA).Length() != 2 ||
        link.GetLanesFrom(*nodeB).Length() != 2;
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    NET-LI-050: Lanes going away from link including pockets.";
fail = link.GetLanesFrom(*nodeA, TRUE).Length() != 2 ||
        link.GetLanesFrom(*nodeB, TRUE).Length() != 3;
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    NET-LI-060: Lanes going away from link failure.";
try {
    link.GetLanesFrom(**nodes.ValueOf(14141));
    fail = TRUE;
} catch(const TNetNotFound& exception) {
    fail = FALSE;
}
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    NET-LI-070: Lanes going toward link w/o pockets.";
fail = link.GetLanesTowards(*nodeA).Length() != 2 ||
        link.GetLanesTowards(*nodeB).Length() != 2;
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    NET-LI-080: Lanes going toward link including pockets.";
fail = link.GetLanesTowards(*nodeA, TRUE).Length() != 3 ||
        link.GetLanesTowards(*nodeB, TRUE).Length() != 2;
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    NET-LI-090: Lanes going toward link failure.";
try {
    link.GetLanesTowards(**nodes.ValueOf(14141));
    fail = TRUE;
} catch(const TNetNotFound& exception) {
    fail = FALSE;
}
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    NET-LI-100: Length w/o setbacks.";
fail = link.GetLength() != 1200;
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    NET-LI-110: Length including setbacks.";
fail = link.GetLength(TRUE) != 1175;
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    NET-LI-120: Setback lengths.";
fail = link.GetSetback(*nodeA) != 10 || link.GetSetback(*nodeB) != 15;
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    NET-LI-130: Setback length failure.";
try {
    link.GetSetback(**nodes.ValueOf(14141));
    fail = TRUE;
} catch(const TNetNotFound& exception) {
    fail = FALSE;
}
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    NET-LI-140: Through links.";
fail = link.GetThroughLink(*nodeA) != 9704 || link.GetThroughLink(*nodeB)
```

```
            != 0;
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    NET-LI-150: Speed limits.";
    fail = link.GetSpeedLimitTowards(*nodeA) != 30 ||
            link.GetSpeedLimitTowards(*nodeB) != 35;
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    NET-LI-160: Link angle.";
    const REAL pi = fabs(atan2(0, -1));
    fail = !Equal(link.GetAngle(*nodeA), 0) || !Equal(link.GetAngle(*nodeB),
            pi);
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    NET-LI-170: Link angle failure.";
    try {
        link.GetAngle(**nodes.ValueOf(14141));
        fail = TRUE;
    } catch(const TNetNotFound& exception) {
        fail = FALSE;
    }
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    NET-LI-180: Equality operator.";
    fail = !(link == TNetLink(12407)) && link == TNetLink(12345);
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    NET-LI-190: Inequality operator.";
    fail = link != TNetLink(12407) && !(link != TNetLink(12345));
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << (anyFail ? "    F" : "    No f") << "ailures occurred." << endl;
    return !anyFail;
}
```

G. TestNetwork.C

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: TestNetwork.C,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 19:29:51 $
//  $State: Rel $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


//  Include Standard C++ header files.
#include <iostream.h>


//  Include TRANSIMS header files.
#include <NET/Network.h>


//  Test network class.
bool TestNetwork(TNetNetwork& network)
{
    cout << "  Network Class Tests "
         << "[$Revision: 2.0 $]"
         << endl;

    bool anyFail = FALSE;
    bool fail;

    cout << "    NET-NE-010: Network node retrieval.";
    TNetNetwork::NodeMap& nodes = network.GetNodes();
    fail = nodes.Extent() != 11 || !nodes.IsBound(8520) ||
```

```
            !nodes.IsBound(8521) || !nodes.IsBound(14136) ||
            !nodes.IsBound(14141);
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    NET-NE-020: Network link retrieval.";
    TNetNetwork::LinkMap& links = network.GetLinks();
    fail = links.Extent() != 10 || !links.IsBound(12384) ||
            !links.IsBound(12407) || !links.IsBound(28800);
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << (anyFail ? "    F" : "    No f") << "ailures occurred." << endl;
    return !anyFail;
}
```

H. TestNode.C

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: TestNode.C,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 19:29:51 $
//  $State: Rel $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


//  Include Standard C++ header files.
#include <iostream.h>


//  Include TRANSIMS header files.
#include <NET/Network.h>
#include <NET/Subnetwork.h>
#include <NET/Node.h>
#include <NET/Link.h>


//  Test node class.
bool TestNode(TNetSubnetwork& subnetwork)
{
    cout << "  Node Class Tests "
         << "[$Revision: 2.0 $]"
         << endl;

    bool anyFail = FALSE;
    bool fail;

    TNetNetwork::NodeMap& nodes = subnetwork.GetNetwork().GetNodes();

    cout << "    NET-NO-010: Node construction.";
    TNetNode& node = **nodes.ValueOf(14136);
    fail = node.GetId() != 14136;
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    NET-NO-020: Node coordinate access.";
    fail = node.GetGeographicPosition().GetX() != 1.0 ||
            node.GetGeographicPosition().GetY() != 0.0;
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    NET-NO-030: Node link ring access.";
    TNetNode::LinkRingIterator i(node.GetLinks());
    fail = (*i.CurrentItem())->GetId() != 12407;
    i.Next();
    fail |= (*i.CurrentItem())->GetId() != 12384;
    i.Next();
    fail |= (*i.CurrentItem())->GetId() != 28804;
    i.Next();
    fail |= !i.IsDone();
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;
```

```
    cout << "    NET-NO-040: Equality operator.";
    fail = !(node == TNetNode(14136)) || node == TNetNode(12345);
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    NET-NO-050: Inequality operator.";
    fail = node != TNetNode(14136) || !(node != TNetNode(12345));
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << (anyFail ? "    F" : "    No f") << "ailures occurred." << endl;
    return !anyFail;
}
```

I. TestNullControl.C

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: TestNullControl.C,v $
//  $Revision: 2.1 $
//  $Date: 1996/02/14 21:36:42 $
//  $State: Exp $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


//  Include standard C++ header files
#include <iostream.h>


//  Include TRANSIMS header files.
#include "NET/Network.h"
#include "NET/Node.h"
#include "NET/Link.h"
#include "NET/NullControl.h"
#include "NET/Exception.h"


//  Test null control class.
bool TestNullControl(TNetNetwork& network)
{
    cout << "  Null Control Class Tests "
         << endl;

    bool anyFail = FALSE;
    bool fail;

    TNetNetwork::NodeMap& nodes = network.GetNodes();
    TNetNode* n4 = *nodes.ValueOf(14062);
    TNetNetwork::LinkMap& links = network.GetLinks();
    TNetLink* l1 = *links.ValueOf(11487);

    cout << "    NET-NC-010: Retrieve vehicle control for the specified lane.";
    try {
        n4->GetTrafficControl().GetVehicleControl(*(l1->GetLanesTowards(*n4,
                TRUE)[0]));
        fail = TRUE;
    } catch (const TNetUndefinedControl& exception) {
        //cout << "(" << exception.GetMessage() << ")" << endl;
        fail = FALSE;
    }
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << (anyFail ? "    F" : "    No f") << "ailures occurred." << endl;
    return !anyFail;
}
```

J. TestParking.C

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: TestParking.C,v $
```

```
//  $Revision: 2.0 $
//  $Date: 1995/08/04 19:29:51 $
//  $State: Rel $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


//  Include Standard C++ header files.
#include <iostream.h>


//  Include TRANSIMS header files.
#include <NET/Network.h>
#include <NET/Subnetwork.h>
#include <NET/Node.h>
#include <NET/Link.h>
#include <NET/Lane.h>
#include <NET/Parking.h>
#include <NET/Location.h>


//  Test parking class.
bool TestParking(TNetSubnetwork& subnetwork)
{
    cout << "  Parking Class Tests "
         << "[$Revision: 2.0 $]"
         << endl;

    bool anyFail = FALSE;
    bool fail;

    TNetNetwork::NodeMap& nodes = subnetwork.GetNetwork().GetNodes();
    TNetNetwork::LinkMap& links = subnetwork.GetNetwork().GetLinks();

    TNetNode& node = **nodes.ValueOf(8521);
    TNetLink& link = **links.ValueOf(12407);
    TNetParking& parking = (TNetParking&) *link.GetAccessories()[0];

    cout << "    NET-PA-010: Parking construction.";
    fail = parking.GetId() != 1003;
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    NET-PA-020: Type access.";
    fail = parking.GetType() != TNetAccessory::kParking;
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    NET-PA-030: Location access.";
    const TNetLocation& location = parking.GetLocation();
    fail = location.GetLink() != link || location.GetOffsetFrom(node) != 1000;
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    NET-PA-040: Coordinate location.";
    fail = location.GetGeographicPosition().GetX() != 5.0 / 6.0 ||
           location.GetGeographicPosition().GetY() != 0;
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    NET-PA-050: Style access.";
    fail = parking.GetStyle() != TNetParking::kHeadInOnStreet;
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    NET-PA-060: Capacity access.";
    fail = parking.GetCapacity() != 10;
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    NET-PA-070: Genericity access.";
    fail = !parking.IsGeneric();
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;
```

```
    //ISSUE(bwb): Not all styles of parking are tested.

    cout << (anyFail ? "    F" : "    No f") << "ailures occurred." << endl;
    return !anyFail;
}
```

K. TestPhase.C

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: TestPhase.C,v $
//  $Revision: 2.1 $
//  $Date: 1996/02/14 21:39:13 $
//  $State: Exp $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


//  Include standard C++ header files
#include <iostream.h>


//  Include TRANSIMS header files.
#include "NET/Network.h"
#include "NET/Node.h"
#include "NET/SignalizedControl.h"
#include "NET/Phase.h"


//  Test phase class.
bool TestPhase(TNetNetwork& network)
{
    cout << "  Phase Class Tests "
         << endl;

    bool anyFail = FALSE;
    bool fail;

    TNetNetwork::NodeMap& nodes = network.GetNodes();
    TNetNode* n1 = *nodes.ValueOf(14141);
    TNetNetwork::LinkMap& links = network.GetLinks();
    TNetLink* l1 = *links.ValueOf(11487);
    TNetSignalizedControl *s = (TNetSignalizedControl*)&n1->GetTrafficControl();
    TNetPhase& p = s->GetPhase();

    cout << "    NET-PH-010: Retrieve current phase's phase description.";
    //cout << "Current phase's phase description's green interval length = "
    fail = p.GetPhaseDescription().GetPhaseNumber() != 5;
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    NET-PH-020: Retrieve current interval.";
    //cout << "Current interval = " << p.GetInterval() << endl;
    fail = p.GetInterval() != 0;
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    NET-PH-030: Retrieve current phase's next phase.";
    //cout << "Current phases's next phase = " << p.GetNextPhases().First() <<
    //        endl;
    //cout << "Next phase's green interval length = "
    //     << p.GetNextPhases().First()->GetPhaseDescription().GetGreenLength()
    //        << endl;
    TNetPhase* addr = p.GetNextPhases().First();
    fail = p.GetNextPhases().First()->GetPhaseDescription().GetPhaseNumber()
            != 6;
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    NET-PH-040: Update to next phase and retrieve current phase.";
    s->UpdateSignalizedControl(129.);
    //cout << "New current phase = " << &s->GetPhase() << endl;
    //cout << "Current phase's green interval length = "
    //     << s->GetPhase().GetPhaseDescription().GetGreenLength() << endl;
```

```
    fail = &s->GetPhase() != addr
             || s->GetPhase().GetPhaseDescription().GetPhaseNumber() != 6;
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << (anyFail ? "    F" : "    No f") << "ailures occurred." << endl;
    return !anyFail;
}
```

L. TestPhaseDescription.C

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: TestPhaseDescription.C,v $
//  $Revision: 2.1 $
//  $Date: 1996/02/14 21:39:13 $
//  $State: Exp $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


//  Include standard C++ header files
#include <iostream.h>


//  Include TRANSIMS header files.
#include "NET/Network.h"
#include "NET/Node.h"
#include "NET/Link.h"
#include "NET/SignalizedControl.h"
#include "NET/PhasingPlan.h"


//  Test phase description class.
bool TestPhaseDescription(TNetNetwork& network)
{
    cout << "  Phase Description Class Tests "
         << endl;

    bool anyFail = FALSE;
    bool fail;

    TNetNetwork::NodeMap& nodes = network.GetNodes();
    TNetNode* n1 = *nodes.ValueOf(14141);
    TNetNetwork::LinkMap& links = network.GetLinks();
    TNetLink* l1 = *links.ValueOf(11487);
    TNetSignalizedControl *s = (TNetSignalizedControl*)&n1->GetTrafficControl();
    TNetPhaseDescription *d = s->GetPhasingPlan().GetPhaseDescriptions()[0];

    cout << "    NET-PD-010: Retrieve green interval length.";
    //cout << "Green interval length = " << d->GetGreenLength() << endl;
    fail = d->GetGreenLength() != 35;
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    NET-PD-020: Retrieve minimum green interval length.";
    //cout << "Minimum green interval length = " << d->GetMinGreenLength() <<
    //        endl;
    fail = d->GetMinGreenLength() != 35;
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    NET-PD-030: Retrieve maximum green interval length.";
    //cout << "Maximum green interval length = " << d->GetMaxGreenLength() <<
    //        endl;
    fail = d->GetMaxGreenLength() != 0;
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    NET-PD-040: Retrieve extension green interval length.";
    //cout << "Extension green interval length = " << d->GetExtGreenLength()
            << endl;
    fail = d->GetExtGreenLength() != 0;
    cout << (fail ? " [failed]" : " [passed]") << endl;
```

```
    anyFail |= fail;

    cout << "    NET-PD-050: Retrieve yellow interval length.";
    //cout << "Yellow interval length = " << d->GetYellowLength() << endl;
    fail = d->GetYellowLength() != 4;
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    NET-PD-060: Retrieve red clearance interval length.";
    //cout << "Red clearance interval length = " << d->GetRedLength() << endl;
    fail = d->GetRedLength() != 0;
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    NET-PD-070: Retrieve all link movements.";
    //cout << "Number of link movements = " << d->GetLinkMovements().Extent()
    //        << endl;
    fail = d->GetLinkMovements().Extent() != 4;
    for (TNetPhaseDescription::LinkMovementMapIterator
            i1(d->GetLinkMovements()); i1.Next(); !i1.IsDone())
        //cout << "Movement exists for link = " << (*i1.CurrentItem())->GetId()
        //        << endl;
        switch ((*i1.CurrentItem())->GetId()) {
            case 11495:
            case 11487:
            case 11486:
            case 28800:
                fail |= FALSE;
                break;
            default:
                fail |= TRUE;
                break;
        }
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    NET-PD-080: Retrieve link movements for the specified link.";
    //cout << "Number of link movements  for link " << l1->GetId() << " = " <<
    //        d->GetLinkMovements(*l1).Extent() << endl;
    fail = l1->GetId() != 11487
         || d->GetLinkMovements(*l1).Extent() != 2;
    for (TNetPhaseDescription::LinkProtectionMapIterator
            i2(d->GetLinkMovements(*l1)); i2.Next(); !i2.IsDone())
        //cout << "Protection for link " << (*i2.CurrentItem())->GetId() <<
        //        " = " << *i2.CurrentValue() << endl;
        switch ((*i2.CurrentItem())->GetId()) {
            case 11495:
                fail |= *i2.CurrentValue() != 0;
                break;
            case 28800:
                fail |= *i2.CurrentValue() != 1;
                break;
            default:
                fail |= FALSE;
                break;
        }
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    NET-PD-090: Retrieve phase number.";
    //cout << "Phase number = " << d->GetPhaseNumber() << endl;
    fail = d->GetPhaseNumber() != 1;
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << (anyFail ? "    F" : "    No f") << "ailures occurred." << endl;
    return !anyFail;
}
```

M. TestPhasingPlan.C

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: TestPhasingPlan.C,v $
//  $Revision: 2.1 $
```

```
//  $Date: 1996/02/14 21:36:42 $
//  $State: Exp $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


//  Include standard C++ header files
#include <iostream.h>


//  Include TRANSIMS header files.
#include "NET/Network.h"
#include "NET/Node.h"
#include "NET/SignalizedControl.h"
#include "NET/PhasingPlan.h"


//  Test phasing plan class.
bool TestPhasingPlan(TNetNetwork& network)
{
    cout << "  Phasing Plan Class Tests "
         << endl;

    bool anyFail = FALSE;
    bool fail;

    TNetNetwork::NodeMap& nodes = network.GetNodes();
    TNetNode* n1 = *nodes.ValueOf(14141);
    TNetSignalizedControl *s = (TNetSignalizedControl*)&n1->GetTrafficControl();

    cout << "    NET-PP-010: Retrieve phasing plan id.";
    //cout << "Phasing plan id = " << s->GetPhasingPlan().GetId() << endl;
    fail = s->GetPhasingPlan().GetId() != 1;
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    NET-PP-020: Retrieve phase descriptions.";
    //cout << "Phasing plan length = " <<
    //        s->GetPhasingPlan().GetPhaseDescriptions().Length() << endl;
    fail = s->GetPhasingPlan().GetPhaseDescriptions().Length() != 6;
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << (anyFail ? "    F" : "    No f") << "ailures occurred." << endl;
    return !anyFail;
}
```

N. TestPocket.C

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: TestPocket.C,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 19:29:51 $
//  $State: Rel $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


//  Include Standard C header files.
#include <math.h>


//  Include Standard C++ header files.
#include <iostream.h>


//  Include TRANSIMS header files.
#include <NET/Network.h>
#include <NET/Subnetwork.h>
#include <NET/Node.h>
#include <NET/Link.h>
#include <NET/Lane.h>
```

```
#include <NET/Pocket.h>
#include <NET/LaneLocation.h>


//  Return whether two real numbers are the same.
static bool Equal(REAL a, REAL b)
{
    return fabs(a - b) < 1e-5;
}


//  Test pocket class.
bool TestPocket(TNetSubnetwork& subnetwork)
{
    cout << "  Pocket Class Tests "
         << "[$Revision: 2.0 $]"
         << endl;

    bool anyFail = FALSE;
    bool fail;

    TNetNetwork::NodeMap& nodes = subnetwork.GetNetwork().GetNodes();
    TNetNetwork::LinkMap& links = subnetwork.GetNetwork().GetLinks();

    TNetNode& node = **nodes.ValueOf(8520);
    TNetLink& link = **links.ValueOf(12384);
    TNetLink::LaneCollection& lanes = link.GetLanesTowards(node, TRUE);
    TNetPocket& pocket = *lanes[0]->GetPockets()[0];

    cout << "    NET-PO-010: Pocket construction.";
    fail = pocket.GetId() != 85201;
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    NET-PO-020: Type access.";
    fail = pocket.GetType() != TNetAccessory::kPocket;
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    NET-PO-030: Location access.";
    const TNetLocation& location = pocket.GetLocation();
    fail = location.GetLink() != link || !Equal(location.GetOffsetFrom(node),
            1000);
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    NET-PO-040: Coordinate location.";
    fail = !Equal(location.GetGeographicPosition().GetX(), 1) ||
            !Equal(location.GetGeographicPosition().GetY(), 0);
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    NET-PO-050: Style access.";
    fail = pocket.GetStyle() != TNetPocket::kMerge;
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    NET-PO-060: Length access.";
    fail = pocket.GetLength() != 100;
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    NET-PO-070: Lane access.";
    fail = &pocket.GetLane() != lanes[0];
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    //ISSUE(bwb): Not all styles of pockets are tested.

    cout << (anyFail ? "    F" : "    No f") << "ailures occurred." << endl;
    return !anyFail;
}
```

O. TestReader.C

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: TestReader.C,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 19:29:51 $
//  $State: Rel $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


//  Include Standard C++ header files.
#include <iostream.h>


//  Include TRANSIMS header files.
#include <DBS/Directory.h>
#include <DBS/Source.h>
#include <DBS/Table.h>
#include <NET/Reader.h>
#include <NET/Network.h>
#include <NET/Subnetwork.h>


//  Test reader classes.
bool TestReader(TNetNetwork& network, TNetSubnetwork*& subnetwork)
{
    cout << "  Reader Classes Tests "
         << "[$Revision: 2.0 $]"
         << endl;

    bool anyFail = FALSE;
    bool fail;

    TDbDirectory directory(TDbDirectoryDescription("IOC-1"));

    TDbSource nodeSource(directory, directory.GetSource("Node"));
    TDbSource linkSource(directory, directory.GetSource("Link"));
    TDbSource pocketSource(directory, directory.GetSource("Pocket Lane"));
    TDbSource parkingSource(directory, directory.GetSource("Parking"));
    TDbSource laneSource(directory, directory.GetSource("Lane Connectivity"));
    TDbSource ucontrolSource(directory, directory.GetSource(
            "Unsignalized Node"));
    TDbSource scontrolSource(directory, directory.GetSource(
            "Signalized Node"));
    TDbSource phasingSource(directory, directory.GetSource(
            "Phasing Plan"));
    TDbSource timingSource(directory, directory.GetSource(
            "Timing Plan"));

    TDbTable nodeTable(nodeSource, nodeSource.GetTable("Sample Node Table II"));
    TDbTable linkTable(linkSource, linkSource.GetTable("Sample Link Table II"));
    TDbTable pocketTable(pocketSource,
            pocketSource.GetTable("Sample Pocket Lane Table II"));
    TDbTable parkingTable(parkingSource,
            parkingSource.GetTable("Sample Parking Table II"));
    TDbTable laneTable(laneSource,
            laneSource.GetTable("Sample Lane Connectivity Table II"));
    TDbTable ucontrolTable(ucontrolSource,
            ucontrolSource.GetTable("Sample Unsignalized Node Table II"));
    TDbTable scontrolTable(scontrolSource,
            scontrolSource.GetTable("Sample Signalized Node Table II"));
    TDbTable phasingTable(phasingSource,
            phasingSource.GetTable("Sample Phasing Plan Table II"));
    TDbTable timingTable(timingSource,
            timingSource.GetTable("Sample Timing Plan Table II"));

    TNetReader reader(nodeTable, linkTable, pocketTable, parkingTable,
        laneTable, ucontrolTable, scontrolTable, phasingTable, timingTable);

    cout << "    NET-RE-010: Reader construction.";
    fail = reader.GetNodeTable().GetDescription().GetName() !=
            "Sample Node Table II" ||
            reader.GetLinkTable().GetDescription().GetName() !=
```

```
            "Sample Link Table II" |
            reader.GetPocketTable().GetDescription().GetName() !=
            "Sample Pocket Lane Table II" ||
            reader.GetParkingTable().GetDescription().GetName() !=
            "Sample Parking Table II";
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    NET-RE-020: Subnetwork construction.";
    try {
        subnetwork = new TNetSubnetwork(reader, network);
        fail = FALSE;
    } catch(...) {
        fail = TRUE;
    }
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    //ISSUE(bwb): Complex subnetwork construction has not been tested.

    cout << (anyFail ? "    F" : "    No f") << "ailures occurred." << endl;
    return !anyFail;
}
```

P. TestSignalCoordinator.C

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: TestSignalCoordinator.C,v $
//  $Revision: 2.1 $
//  $Date: 1996/02/14 21:36:42 $
//  $State: Exp $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


//  Include standard C++ header files
#include <iostream.h>


//  Include TRANSIMS header files.
#include "NET/Network.h"
#include "NET/Node.h"
#include "NET/SignalCoordinator.h"
#include "NET/SignalizedControl.h"
#include "NET/PhasingPlan.h"


//  Test signal coordinator class.
bool TestSignalCoordinator(TNetNetwork& network)
{
    cout << "  Signal Coordinator Class Tests "
         << endl;

    bool anyFail = FALSE;
    bool fail;

    TNetNetwork::NodeMap& nodes = network.GetNodes();
    TNetNode* n1 = *nodes.ValueOf(14141);
    TNetSignalizedControl *s = (TNetSignalizedControl*)&n1->GetTrafficControl();
    TNetSignalCoordinator *c = (TNetSignalCoordinator*)&s->GetCoordinator();

    cout << "    NET-CO-010: Retrieve signal coordinator's id.";
    //cout << "Coordinator's id = " << c->GetId() << " for node = " <<
    //        s->GetNode().GetId() << endl;
    fail = c->GetId() != 14141 || s->GetNode().GetId() != 14141;
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    NET-CO-020: Retrieve coordinator's signal controllers.";
    //cout << "Coordinator has " << c->GetControllers().Length() <<
    //        " signal controllers" << endl;
    //cout << "Node id of first controller = " << c->GetControllers().First()
    //        ->GetNode().GetId() << endl;
```

```
    fail = c->GetControllers().Length() != 1
         || c->GetControllers().First()->GetNode().GetId() != 14141;
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    NET-CO-030: Retrieve phasing plans.";
    //cout << "Coordinator has " << c->GetPhasingPlans().Extent() <<
    //        " phasing plans" << endl;
    fail = c->GetPhasingPlans().Extent() != 1;
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    NET-CO-040: Retrieve phasing plan with specific id.";
    //cout << "First phasing plan id = " << c->GetPhasingPlan(1).GetId() <<
    //        endl;
    fail = c->GetPhasingPlan(1).GetId() != 1;
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << (anyFail ? "    F" : "    No f") << "ailures occurred." << endl;
    return !anyFail;
}
```

Q. TestSignalizedControl.C

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: TestSignalizedControl.C,v $
//  $Revision: 2.1 $
//  $Date: 1996/02/14 21:39:13 $
//  $State: Exp $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


//  Include standard C++ header files
#include <iostream.h>


//  Include TRANSIMS header files.
#include "NET/Network.h"
#include "NET/Node.h"
#include "NET/Link.h"
#include "NET/SignalizedControl.h"
#include "NET/SignalCoordinator.h"
#include "NET/PhasingPlan.h"
#include "NET/Phase.h"


//  Test signalized control class.
bool TestSignalizedControl(TNetNetwork& network)
{
    cout << "  Signalized Control Class Tests "
         << endl;

    bool anyFail = FALSE;
    bool fail;

    TNetNetwork::NodeMap& nodes = network.GetNodes();
    TNetNode* n1 = *nodes.ValueOf(14141);
    TNetNetwork::LinkMap& links = network.GetLinks();
    TNetLink* l1 = *links.ValueOf(11487);
    TNetLink* l2 = *links.ValueOf(11486);
    TNetLink* l3 = *links.ValueOf(11495);
    TNetLink* l4 = *links.ValueOf(28800);
    TNetSignalizedControl *s = (TNetSignalizedControl*)&n1->GetTrafficControl();

    cout << "    NET-SC-010: Retrieve signal coordinator.";
    //cout << "Signal's node id = " << s->GetNode().GetId()
    //     << " and coordinator's id = " << s->GetCoordinator().GetId() << endl;
    fail = s->GetNode().GetId() != 14141
             || s->GetCoordinator().GetId() != 14141;
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;
```

```
cout << "    NET-SC-020: Retrieve phasing plan.";
//cout << "Signal's phasing plan id = " << s->GetPhasingPlan().GetId() <<
//        endl;
fail = s->GetPhasingPlan().GetId() != 1;
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    NET-SC-030: Retrieve phasing plan offset.";
//cout << "Signal's phasing plan offset = " << s->GetPhasingPlanOffset() <<
//        endl;
fail = s->GetPhasingPlanOffset() != 19;
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    NET-SC-040: Retrieve phasing plan phases.";
//cout << "Number of phases in plan = " << s->GetPhases().Length() << endl;
fail = s->GetPhases().Length() != 6;
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    NET-SC-050: Initialize phasing plan to its first phase.";
s->InitPhase(*(s->GetPhases().First()));
//cout << "First phase green length = " <<
//        s->GetPhase().GetPhaseDescription().GetGreenLength() << endl;
fail = s->GetPhase().GetPhaseDescription().GetPhaseNumber() != 1;
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    NET-SC-060: Set phasing plan to its second phase.";
s->SetPhase(*(s->GetPhase().GetNextPhases().First()));
//cout << "Second phase green length = " <<
//        s->GetPhase().GetPhaseDescription().GetGreenLength() << endl;
fail = s->GetPhase().GetPhaseDescription().GetPhaseNumber() != 2;
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    NET-SC-070: Retrieve vehicle control for the specified lane "
        "during first phase green.";
s->InitPhase(*(s->GetPhases().First()));
//cout << "Traffic control on link " << l4->GetId() << " lane 1 = "
//     << n1->GetTrafficControl().GetVehicleControl(*(l4
//     ->GetLanesTowards(*n1,TRUE)[0])) << endl;
//cout << "Traffic control on link " << l4->GetId() << " lane 3 = "
//     << n1->GetTrafficControl().GetVehicleControl(*(l4
//     ->GetLanesTowards(*n1,TRUE)[2])) << endl;
//cout << "Traffic control on link " << l4->GetId() << " lane 6 = "
//     << n1->GetTrafficControl().GetVehicleControl(*(l4
//     ->GetLanesTowards(*n1,TRUE)[5])) << endl;
fail = l4->GetId() != 28800
         || n1->GetTrafficControl().GetVehicleControl(*(l4
         ->GetLanesTowards(*n1,TRUE)[0])) != 3
         || n1->GetTrafficControl().GetVehicleControl(*(l4
         ->GetLanesTowards(*n1,TRUE)[2])) != 3
         || n1->GetTrafficControl().GetVehicleControl(*(l4
         ->GetLanesTowards(*n1,TRUE)[5])) != 3;
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    NET-SC-080: Retrieve vehicle control for the specified lane "
        "during second phase green.";
s->UpdateSignalizedControl(60.);
//cout << "Traffic control on link " << l4->GetId() << " lane 1 = "
//     << n1->GetTrafficControl().GetVehicleControl(*(l4
//     ->GetLanesTowards(*n1,TRUE)[0])) << endl;
//cout << "Traffic control on link " << l4->GetId() << " lane 3 = "
//     << n1->GetTrafficControl().GetVehicleControl(*(l4
//     ->GetLanesTowards(*n1,TRUE)[2])) << endl;
//cout << "Traffic control on link " << l4->GetId() << " lane 6 = "
//     << n1->GetTrafficControl().GetVehicleControl(*(l4
//     ->GetLanesTowards(*n1,TRUE)[5])) << endl;
fail = l4->GetId() != 28800
         || n1->GetTrafficControl().GetVehicleControl(*(l4
         ->GetLanesTowards(*n1,TRUE)[0])) != 6
         || n1->GetTrafficControl().GetVehicleControl(*(l4
         ->GetLanesTowards(*n1,TRUE)[2])) != 3
```

```
            || n1->GetTrafficControl().GetVehicleControl(*(l4
            ->GetLanesTowards(*n1,TRUE)[5])) != 3;
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    NET-SC-090: Retrieve vehicle control for the specified lane "
            "during second phase yellow.";
    s->UpdateSignalizedControl(65.);
    //cout << "Traffic control on link " << l4->GetId() << " lane 1 = "
    //     << n1->GetTrafficControl().GetVehicleControl(*(l4
    //     ->GetLanesTowards(*n1,TRUE)[0])) << endl;
    //cout << "Traffic control on link " << l4->GetId() << " lane 3 = "
    //     << n1->GetTrafficControl().GetVehicleControl(*(l4
    //     ->GetLanesTowards(*n1,TRUE)[2])) << endl;
    //cout << "Traffic control on link " << l4->GetId() << " lane 6 = "
    //     << n1->GetTrafficControl().GetVehicleControl(*(l4
    //     ->GetLanesTowards(*n1,TRUE)[5])) << endl;
    fail = l4->GetId() != 28800
            || n1->GetTrafficControl().GetVehicleControl(*(l4
            ->GetLanesTowards(*n1,TRUE)[0])) != 6
            || n1->GetTrafficControl().GetVehicleControl(*(l4
            ->GetLanesTowards(*n1,TRUE)[2])) != 3
            || n1->GetTrafficControl().GetVehicleControl(*(l4
            ->GetLanesTowards(*n1,TRUE)[5])) != 3;
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    NET-SC-100: Retrieve vehicle control for the specified lane "
            "during third phase green.";
    s->UpdateSignalizedControl(70.);
    //cout << "Traffic control on link " << l4->GetId() << " lane 1 = "
    //     << n1->GetTrafficControl().GetVehicleControl(*(l4
    //     ->GetLanesTowards(*n1,TRUE)[0])) << endl;
    //cout << "Traffic control on link " << l4->GetId() << " lane 3 = "
    //     << n1->GetTrafficControl().GetVehicleControl(*(l4
    /     ->GetLanesTowards(*n1,TRUE)[2])) << endl;
    //cout << "Traffic control on link " << l4->GetId() << " lane 6 = "
    //     << n1->GetTrafficControl().GetVehicleControl(*(l4
    //     ->GetLanesTowards(*n1,TRUE)[5])) << endl;
    fail = l4->GetId() != 28800
            || n1->GetTrafficControl().GetVehicleControl(*(l4
            ->GetLanesTowards(*n1,TRUE)[0])) != 6
            || n1->GetTrafficControl().GetVehicleControl(*(l4
            ->GetLanesTowards(*n1,TRUE)[2])) != 5
            || n1->GetTrafficControl().GetVehicleControl(*(l4
            ->GetLanesTowards(*n1,TRUE)[5])) != 6;
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << (anyFail ? "    F" : "    No f") << "ailures occurred." << endl;
    return !anyFail;
}
```

R. TestSubnetwork.C

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: TestSubnetwork.C,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 19:29:51 $
//  $State: Rel $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


//  Include Standard C++ header files.
#include <iostream.h>


//  Include TRANSIMS header files.
#include <NET/Network.h>
#include <NET/Subnetwork.h>
#include <NET/Node.h>
#include <NET/Link.h>
```

```
//  Test subnetwork class.
bool TestSubnetwork(TNetSubnetwork& subnetwork)
{
    cout << "  Subnetwork Class Tests "
         << "[$Revision: 2.0 $]"
         << endl;

    bool anyFail = FALSE;
    bool fail;

    cout << "    NET-SN-010: Subnetwork node retrieval.";
    TNetNetwork::NodeMap& netNodes = subnetwork.GetNetwork().GetNodes();
    TNetSubnetwork::NodeSet& nodes = subnetwork.GetNodes();
    fail = nodes.Extent() != 11 || !nodes.IsMember(*netNodes.ValueOf(8520)) ||
            !nodes.IsMember(*netNodes.ValueOf(8521)) ||
            !nodes.IsMember(*netNodes.ValueOf(14136)) ||
            !nodes.IsMember(*netNodes.ValueOf(14141));
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    NET-SN-020: Subnetwork link retrieval.";
    TNetNetwork::LinkMap& netLinks = subnetwork.GetNetwork().GetLinks();
    TNetSubnetwork::LinkSet& links = subnetwork.GetLinks();
    fail = links.Extent() != 10 || !links.IsMember(*netLinks.ValueOf(12384)) ||
            !links.IsMember(*netLinks.ValueOf(12407)) ||
            !links.IsMember(*netLinks.ValueOf(28800));
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << (anyFail ? "    F" : "    No f") << "ailures occurred." << endl;
    return !anyFail;
}
```

S. TestTimedControl.C

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: TestTimedControl.C,v $
//  $Revision: 2.1 $
//  $Date: 1996/02/14 21:39:13 $
//  $State: Exp $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


//  Include standard C++ header files
#include <iostream.h>


//  Include TRANSIMS header files.
#include "NET/Network.h"
#include "NET/Node.h"
#include "NET/Link.h"
#include "NET/SignalizedControl.h"
#include "NET/TimedControl.h"
#include "NET/Phase.h"


//  Test timed control class.
bool TestTimedControl(TNetNetwork& network)
{
    cout << "  Timed Control Class Tests "
         << endl;

    bool anyFail = FALSE;
    bool fail;

    TNetNetwork::NodeMap& nodes = network.GetNodes();
    TNetNode* n1 = *nodes.ValueOf(14141);
    TNetNetwork::LinkMap& links = network.GetLinks();
    TNetLink* l1 = *links.ValueOf(11487);
    TNetLink* l2 = *links.ValueOf(11486);
```

```
    TNetLink* l3 = *links.ValueOf(11495);
    TNetLink* l4 = *links.ValueOf(28800);
    TNetSignalizedControl *s = (TNetSignalizedControl*)&n1->GetTrafficControl();

    cout << "    NET-TI-010: Update signalized control to time=109.1 "
            "(phase 4).";
    s->UpdateSignalizedControl(109.1);
    //cout << "Current interval = " << s->GetPhase().GetInterval() << endl;
    //cout << "Traffic control on link " << l4->GetId() << " lane 1 = "
    //     << n1->GetTrafficControl().GetVehicleControl(*(l4
    //     ->GetLanesTowards(*n1,TRUE)[0])) << endl;
    //cout << "Traffic control on link " << l4->GetId() << " lane 3 = "
    //     << n1->GetTrafficControl().GetVehicleControl(*(l4
    //     ->GetLanesTowards(*n1,TRUE)[2])) << endl;
    //cout << "Traffic control on link " << l4->GetId() << " lane 6 = "
    //     << n1->GetTrafficControl().GetVehicleControl(*(l4
    //     ->GetLanesTowards(*n1,TRUE)[5])) << endl;
    fail = s->GetPhase().GetInterval() != 1;
    fail |= l4->GetId() != 28800
             || n1->GetTrafficControl().GetVehicleControl(*(l4
             ->GetLanesTowards(*n1,TRUE)[0])) != 3
             || n1->GetTrafficControl().GetVehicleControl(*(l4
             ->GetLanesTowards(*n1,TRUE)[2])) != 4
             || n1->GetTrafficControl().GetVehicleControl(*(l4
             ->GetLanesTowards(*n1,TRUE)[5])) != 6;
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    NET-TI-020: Update signalized control to time=114 (phase 5).";
    s->UpdateSignalizedControl(114.);
    //cout << "Current interval = " << s->GetPhase().GetInterval() << endl;
    //cout << "Traffic control on link " << l4->GetId() << " lane 1 = "
    //     << n1->GetTrafficControl().GetVehicleControl(*(l4
    //     ->GetLanesTowards(*n1,TRUE)[0])) << endl;
    //cout << "Traffic control on link " << l4->GetId() << " lane 3 = "
    //     << n1->GetTrafficControl().GetVehicleControl(*(l4
    //     ->GetLanesTowards(*n1,TRUE)[2])) << endl;
    //cout << "Traffic control on link " << l4->GetId() << " lane 6 = "
    //     << n1->GetTrafficControl().GetVehicleControl(*(l4
    //     ->GetLanesTowards(*n1,TRUE)[5])) << endl;
    fail = s->GetPhase().GetInterval() != 0;
    fail |= l4->GetId() != 28800
             || n1->GetTrafficControl().GetVehicleControl(*(l4
             ->GetLanesTowards(*n1,TRUE)[0])) != 3
             || n1->GetTrafficControl().GetVehicleControl(*(l4
             ->GetLanesTowards(*n1,TRUE)[2])) != 3
             || n1->GetTrafficControl().GetVehicleControl(*(l4
             ->GetLanesTowards(*n1,TRUE)[5])) != 6;
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << (anyFail ? "    F" : "    No f") << "ailures occurred." << endl;
    return !anyFail;
}
```

T. TestTrafficControl.C

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: TestTrafficControl.C,v $
//  $Revision: 2.2 $
//  $Date: 1996/08/29 15:23:34 $
//  $State: Stab $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


//  Include standard C++ header files
#include <iostream.h>


//  Include TRANSIMS header files.
#include "NET/Network.h"
#include "NET/Node.h"
```

```
#include "NET/Link.h"
#include "NET/Lane.h"
#include "NET/TrafficControl.h"
#include "NET/SignalizedControl.h"
#include "NET/Exception.h"


//  Test traffic control class.
bool TestTrafficControl(TNetNetwork& network)
{
    cout << "  Traffic Control Class Tests "
         << endl;

    bool anyFail = FALSE;
    bool fail;

    TNetNetwork::NodeMap& nodes = network.GetNodes();
    TNetNode* n1 = *nodes.ValueOf(14141);
    TNetNode* n2 = *nodes.ValueOf(14136);
    TNetNode* n3 = *nodes.ValueOf(8520);
    TNetNode* n4 = *nodes.ValueOf(14062);
    TNetNode* n5 = *nodes.ValueOf(8521);
    TNetNetwork::LinkMap& links = network.GetLinks();
    TNetLink* l1 = *links.ValueOf(11487);
    TNetLink* l2 = *links.ValueOf(11486);
    TNetLink* l3 = *links.ValueOf(11495);
    TNetLink* l4 = *links.ValueOf(28800);
    TNetLink* l5 = *links.ValueOf(12384);
    TNetLink* l6 = *links.ValueOf(12407);
    TNetLink* l7 = *links.ValueOf(28804);
    TNetLink* l8 = *links.ValueOf(9704);
    TNetTrafficControl::LaneCollection fromlanes;
    TNetTrafficControl::LaneCollection tolanes;

    cout << "    NET-TC-010: Retrieve traffic control's node.";
    //cout << "Node Id = " << n1->GetId() << endl;
    //cout << "Traffic control's node id = " <<
    //        n1->GetTrafficControl().GetNode().GetId() << endl;
    fail = n1->GetId() != 14141
            || n1->GetTrafficControl().GetNode().GetId() != 14141;
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    NET-TC-020: Retrieve lane connectivity.";
    //cout << "Lane connectivity map size = " << n1
    //        ->GetTrafficControl().GetConnectivity().Extent() << endl;
    fail = n1->GetTrafficControl().GetConnectivity().Extent() != 21;
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    NET-TC-030: Retrieve lane connectivity for specific lane.";
    TNetLink::LaneCollectionIterator ita(n1->GetTrafficControl()
            .GetConnectivity(*(l1->GetLanesTowards(*n1, TRUE)[0])));
    //cout << "Link 11487 lane "
    //     << (int)(l1->GetLanesTowards(*n1,TRUE)[0])->GetNumber() <<
    //     " connected to lane "
    //     << (int)(*ita.CurrentItem())->GetNumber()
    //     << " on link " << (*ita.CurrentItem())->GetLink().GetId() <<
    //     endl;
    fail =  (l1->GetLanesTowards(*n1,TRUE)[0])->GetNumber() != 1
         || (*ita.CurrentItem())->GetNumber() != 1
            || (*ita.CurrentItem())->GetLink().GetId() != 11486;
    ita.Next();
    fail |= !ita.IsDone();
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "   Test Allowed Movements method returning collection of "
            "destination lanes." << endl;

    cout << "    NET-TC-040: Left turn through unsignalized control.";
    n2->GetTrafficControl().AllowedMovements (tolanes, *l6,
            *l6->GetLanesTowards(*n2,TRUE)[0], *l5);
    //cout << "# allowed lanes going from 12407 lane 1 to 12384 = " <<
    //        tolanes.Length() << endl;
    fail = tolanes.Length() != 1;
```

```
TNetTrafficControl::LaneCollectionIterator itg(tolanes);
//cout << "To Lane " << (int)(*itg.CurrentItem())->GetNumber() <<
//     " on link "
//     << (*itg.CurrentItem())->GetLink().GetId() << endl;
fail |= (*itg.CurrentItem())->GetNumber() != 1
         || (*itg.CurrentItem())->GetLink().GetId() != 12384;
itg.Next();
fail |= !itg.IsDone();
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    NET-TC-050: Right turn through unsignalized control.";
n2->GetTrafficControl().AllowedMovements (tolanes, *l6,
        *l6->GetLanesTowards(*n2,TRUE)[1], *l7);
//cout << "# allowed lanes going from 12407 lane 2 to 28804 = " <<
//    tolanes.Length() << endl;
fail = tolanes.Length() != 1;
TNetTrafficControl::LaneCollectionIterator ith(tolanes);
//cout << "To Lane " << (int)(*ith.CurrentItem())->GetNumber() <<
//     " on link "
//     << (*ith.CurrentItem())->GetLink().GetId() << endl;
fail |= (*ith.CurrentItem())->GetNumber() != 4
     || (*ith.CurrentItem())->GetLink().GetId() != 28804;
ith.Next();
fail |= !ith.IsDone();
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    NET-TC-060: Straight through unsignalized control.";
n2->GetTrafficControl().AllowedMovements (tolanes, *l7,
          *l7->GetLanesTowards(*n2,TRUE)[0], *l5);
//cout << "# allowed lanes going from 28804 lane 1 to 12384 = " <<
//        tolanes.Length() << endl;
fail = tolanes.Length() != 1;
TNetTrafficControl::LaneCollectionIterator it1(tolanes);
//cout << "To Lane " << (int)(*it1.CurrentItem())->GetNumber() <<
//     " on link "
//     << (*it1.CurrentItem())->GetLink().GetId() << endl;
fail |= (*it1.CurrentItem())->GetNumber() != 2
     || (*it1.CurrentItem())->GetLink().GetId() != 12384;
it1.Next();
fail |= !it1.IsDone();
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    NET-TC-070: Left turn through signalized control.";
n1->GetTrafficControl().AllowedMovements (tolanes, *l1,
        *l1->GetLanesTowards(*n1,TRUE)[1], *l2);
//cout << "# allowed lanes going from 11487 lane 2 to 11486 = " <<
//        tolanes.Length() << endl;
fail = tolanes.Length() != 1;
TNetTrafficControl::LaneCollectionIterator itd(tolanes);
//cout << "To Lane " << (int)(*itd.CurrentItem())->GetNumber() <<
//     " on link "
//     << (*itd.CurrentItem())->GetLink().GetId() << endl;
fail |= (*itd.CurrentItem())->GetNumber() != 2
     || (*itd.CurrentItem())->GetLink().GetId() != 11486;
itd.Next();
fail |= !itd.IsDone();
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    NET-TC-080: Right turn through signalized control.";
n1->GetTrafficControl().AllowedMovements (tolanes, *l2,
        *l2->GetLanesTowards(*n1,TRUE)[2], *l1);
//cout << "# allowed lanes going from 11486 lane 3 to 11487 = " <<
//       tolanes.Length() << endl;
fail = tolanes.Length() != 1;
TNetTrafficControl::LaneCollectionIterator it2(tolanes);
//cout << "To Lane " << (int)(*it2.CurrentItem())->GetNumber() <<
//    " on link "
//     << (*it2.CurrentItem())->GetLink().GetId() << endl;
fail |= (*it2.CurrentItem())->GetNumber() != 3
     || (*it2.CurrentItem())->GetLink().GetId() != 11487;
it2.Next();
fail |= !it2.IsDone();
```

```
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    NET-TC-090: Straight through signalized control.";
TNetSignalizedControl *t = (TNetSignalizedControl*)&n1->GetTrafficControl();
t->AllowedMovements (tolanes, *l1, *l1->GetLanesTowards(*n1,TRUE)[3], *l3);
//cout << "# allowed lanes going from 11487 lane 4 to 11495 = " <<
//    tolanes.Length() << endl;
fail = tolanes.Length() != 1;
TNetTrafficControl::LaneCollectionIterator iti(tolanes);
//cout << "To Lane " << (int)(*iti.CurrentItem())->GetNumber() <<
//     " on link "
//     << (*iti.CurrentItem())->GetLink().GetId() << endl;
fail |= (*iti.CurrentItem())->GetNumber() != 2
     || (*iti.CurrentItem())->GetLink().GetId() != 11495;
iti.Next();
fail |= !iti.IsDone();
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    NET-TC-100: Across lane change node.";
n3->GetTrafficControl().AllowedMovements (tolanes, *l4,
        *l4->GetLanesTowards(*n3,TRUE)[2], *l5);
//cout << "# allowed lanes going from 28800 lane 3 to 12384 = " <<
//        tolanes.Length() << endl;
fail = tolanes.Length() != 2;
TNetTrafficControl::LaneCollectionIterator itf(tolanes);
//cout << "To Lane " << (int)(*itf.CurrentItem())->GetNumber() <<
//     " on link "
//     << (*itf.CurrentItem())->GetLink().GetId() << endl;
fail |= (*itf.CurrentItem())->GetNumber() != 3
     || (*itf.CurrentItem())->GetLink().GetId() != 12384;
itf.Next();
fail |= (*itf.CurrentItem())->GetNumber() != 4
     || (*itf.CurrentItem())->GetLink().GetId() != 12384;
itf.Next();
fail |= !itf.IsDone();
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    NET-TC-110: Between lanes that are not connected.";
n1->GetTrafficControl().AllowedMovements (tolanes, *l1,
        *l1->GetLanesTowards(*n1,TRUE)[1], *l3);
//cout << "# allowed lanes going from 11487 lane 2 to 11495 = " <<
//        tolanes.Length() << endl;
fail = tolanes.Length() != 0;
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    NET-TC-120: Through node with null control.";
try {
    n4->GetTrafficControl().AllowedMovements (tolanes, *l1,
            *l1->GetLanesTowards(*n4,TRUE)[1], *l2);
    fail = TRUE;
} catch (const TNetUndefinedControl& exception) {
    //cout << "(" << exception.GetMessage() << ")" << endl;
    fail = FALSE;
}
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "   Test Allowed Movements method returning collection "
        "of approaching lanes." << endl;

cout << "    NET-TC-130: Left turn through unsignalized control with "
        "phase=false.";
n2->GetTrafficControl().AllowedMovements (fromlanes, *l6, *l5, FALSE);
//cout << "# allowed lanes going from 12407 to 12384 = " <<
//        fromlanes.Length() << endl;
fail = fromlanes.Length() != 1;
TNetTrafficControl::LaneCollectionIterator it3(fromlanes);
//cout << "From Lane " << (int)(*it3.CurrentItem())->GetNumber() <<
//        " on link " << (*it3.CurrentItem())->GetLink().GetId() << endl;
fail |= (*it3.CurrentItem())->GetNumber() != 1
     || (*it3.CurrentItem())->GetLink().GetId() != 12407;
it3.Next();
```

```
fail |= !it3.IsDone();
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    NET-TC-140: Left turn through unsignalized control with "
        "phase=true.";
n2->GetTrafficControl().AllowedMovements (fromlanes, *l6, *l5, TRUE);
//cout << "# allowed lanes going from 12407 to 12384 = " <<
//        fromlanes.Length() << endl;
fail = fromlanes.Length() != 1;
TNetTrafficControl::LaneCollectionIterator it4(fromlanes);
//cout << "From Lane " << (int)(*it4.CurrentItem())->GetNumber() <<
//        " on link " << (*it4.CurrentItem())->GetLink().GetId() << endl;
fail |= (*it4.CurrentItem())->GetNumber() != 1
     || (*it4.CurrentItem())->GetLink().GetId() != 12407;
it4.Next();
fail |= !it4.IsDone();
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    NET-TC-150: Right turn through unsignalized control with "
        "phase=false.";
n2->GetTrafficControl().AllowedMovements (fromlanes, *l6, *l7, FALSE);
//cout << "# allowed lanes going from 12407 to 28804 = " <<
//        fromlanes.Length() << endl;
fail = fromlanes.Length() != 1;
TNetTrafficControl::LaneCollectionIterator ito(fromlanes);
//cout << "From Lane " << (int)(*ito.CurrentItem())->GetNumber() <<
//        " on link " << (*ito.CurrentItem())->GetLink().GetId() << endl;
fail |= (*ito.CurrentItem())->GetNumber() != 2
     || (*ito.CurrentItem())->GetLink().GetId() != 12407;
ito.Next();
fail |= !ito.IsDone();
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    NET-TC-160: Straight through unsignalized control with "
        "phase=default.";
n2->GetTrafficControl().AllowedMovements (fromlanes, *l7, *l5);
//cout << "# allowed lanes going from 28804 to 12384 = " <<
//        fromlanes.Length() << endl;
fail = fromlanes.Length() != 5;
TNetTrafficControl::LaneCollectionIterator itq(fromlanes);
//cout << "From Lane " << (int)(*itq.CurrentItem())->GetNumber() <<
//        " on link " << (*itq.CurrentItem())->GetLink().GetId() << endl;
fail |= (*itq.CurrentItem())->GetNumber() != 1
     || (*itq.CurrentItem())->GetLink().GetId() != 28804;
itq.Next();
fail |= (*itq.CurrentItem())->GetNumber() != 2
     || (*itq.CurrentItem())->GetLink().GetId() != 28804;
itq.Next();
fail |= (*itq.CurrentItem())->GetNumber() != 3
     || (*itq.CurrentItem())->GetLink().GetId() != 28804;
itq.Next();
fail |= (*itq.CurrentItem())->GetNumber() != 4
     || (*itq.CurrentItem())->GetLink().GetId() != 28804;
itq.Next();
fail |= (*itq.CurrentItem())->GetNumber() != 5
     || (*itq.CurrentItem())->GetLink().GetId() != 28804;
itq.Next();
fail |= !itq.IsDone();
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

//  initialize phase for signalized intersection
TNetSignalizedControl *s = (TNetSignalizedControl*)&n5->GetTrafficControl();
s->InitPhase(*(s->GetPhases().First()));
s = (TNetSignalizedControl*)&n1->GetTrafficControl();
s->InitPhase(*(s->GetPhases().First()));
cout << "   The following traffic control tests are performed with "
        "phase=1." << endl;

cout << "    NET-TC-170: Left turn through signalized control with "
        "phase=false.";
n1->GetTrafficControl().AllowedMovements (fromlanes, *l1, *l2, FALSE);
//cout << "# allowed lanes going from 11487 to 11486 = " <<
```

```
//        fromlanes.Length() << endl;
fail = fromlanes.Length() != 2;
TNetTrafficControl::LaneCollectionIterator itn(fromlanes);
//cout << "From Lane " << (int)(*itn.CurrentItem())->GetNumber() <<
//        " on link " << (*itn.CurrentItem())->GetLink().GetId() << endl;
fail |= (*itn.CurrentItem())->GetNumber() != 1
     || (*itn.CurrentItem())->GetLink().GetId() != 11487;
itn.Next();
fail |= (*itn.CurrentItem())->GetNumber() != 2
     || (*itn.CurrentItem())->GetLink().GetId() != 11487;
itn.Next();
fail |= !itn.IsDone();
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    NET-TC-180: Left turn through signalized control with "
        "phase=true.";
n1->GetTrafficControl().AllowedMovements (fromlanes, *l1, *l2, TRUE);
//cout << "# allowed lanes going from 11487 to 11486 = " <<
//        fromlanes.Length() << endl;
fail = fromlanes.Length() != 0;
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    NET-TC-190: Right turn through signalized control with "
        "phase=false.";
n1->GetTrafficControl().AllowedMovements (fromlanes, *l2, *l1, FALSE);
//cout << "# allowed lanes going from 11486 to 11487 = " <<
//        fromlanes.Length() << endl;
fail = fromlanes.Length() != 1;
TNetTrafficControl::LaneCollectionIterator it6(fromlanes);
//cout << "From Lane " << (int)(*it6.CurrentItem())->GetNumber() <<
//    " on link " << (*it6.CurrentItem())->GetLink().GetId() << endl;
fail |= (*it6.CurrentItem())->GetNumber() != 3
     || (*it6.CurrentItem())->GetLink().GetId() != 11486;
it6.Next();
fail |= !it6.IsDone();
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    NET-TC-200: Straight through signalized control with "
        "phase=default.";
n1->GetTrafficControl().AllowedMovements (fromlanes, *l1, *l3);
//cout << "# allowed lanes going from 11487 to 11495 = " <<
//        fromlanes.Length() << endl;
fail = fromlanes.Length() != 3;
TNetTrafficControl::LaneCollectionIterator it7(fromlanes);
//cout << "From Lane " << (int)(*it7.CurrentItem())->GetNumber() <<
//        " on link " << (*it7.CurrentItem())->GetLink().GetId() << endl;
fail |= (*it7.CurrentItem())->GetNumber() != 3
     || (*it7.CurrentItem())->GetLink().GetId() != 11487;
it7.Next();
fail |= (*it7.CurrentItem())->GetNumber() != 4
     || (*it7.CurrentItem())->GetLink().GetId() != 11487;
it7.Next();
fail |= (*it7.CurrentItem())->GetNumber() != 5
     || (*it7.CurrentItem())->GetLink().GetId() != 11487;
it7.Next();
fail |= !it7.IsDone();
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    NET-TC-210: Straight through signalized control with "
        "phase=true.";
n5->GetTrafficControl().AllowedMovements (fromlanes, *l6, *l8, TRUE);
//cout << "# allowed lanes going from 12407 to 9704 = " <<
//        fromlanes.Length() << endl;
fail = fromlanes.Length() != 0;
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    NET-TC-220: Across lane change node with phase=false.";
n3->GetTrafficControl().AllowedMovements (fromlanes, *l5, *l4, FALSE);
//cout << "# allowed lanes going from 12384 to 28800 = " <<
//        fromlanes.Length() << endl;
fail = fromlanes.Length() != 4;
```

```
TNetTrafficControl::LaneCollectionIterator itp(fromlanes);
//cout << "From Lane " << (int)(*itp.CurrentItem())->GetNumber() <<
//        " on link " << (*itp.CurrentItem())->GetLink().GetId() << endl;
fail |= (*itp.CurrentItem())->GetNumber() != 2
     || (*itp.CurrentItem())->GetLink().GetId() != 12384;
itp.Next();
fail |= (*itp.CurrentItem())->GetNumber() != 3
     || (*itp.CurrentItem())->GetLink().GetId() != 12384;
itp.Next();
fail |= (*itp.CurrentItem())->GetNumber() != 4
     || (*itp.CurrentItem())->GetLink().GetId() != 12384;
itp.Next();
fail |= (*itp.CurrentItem())->GetNumber() != 5
     || (*itp.CurrentItem())->GetLink().GetId() != 12384;
itp.Next();
fail |= !itp.IsDone();
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    NET-TC-230: Across lane change node with phase=true.";
n3->GetTrafficControl().AllowedMovements (fromlanes, *l5, *l4, TRUE);
//cout << "# allowed lanes going from 12384 to 28800 = " <<
//        fromlanes.Length() << endl;
fail = fromlanes.Length() != 4;
TNetTrafficControl::LaneCollectionIterator it8(fromlanes);
//cout << "From Lane " << (int)(*it8.CurrentItem())->GetNumber() <<
//        " on link " << (*it8.CurrentItem())->GetLink().GetId() << endl;
fail |= (*it8.CurrentItem())->GetNumber() != 2
     || (*it8.CurrentItem())->GetLink().GetId() != 12384;
it8.Next();
fail |= (*it8.CurrentItem())->GetNumber() != 3
     || (*it8.CurrentItem())->GetLink().GetId() != 12384;
it8.Next();
fail |= (*it8.CurrentItem())->GetNumber() != 4
     || (*it8.CurrentItem())->GetLink().GetId() != 12384;
it8.Next();
fail |= (*it8.CurrentItem())->GetNumber() != 5
     || (*it8.CurrentItem())->GetLink().GetId() != 12384;
it8.Next();
fail |= !it8.IsDone();
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    NET-TC-240: Between links that are not connected.";
try {
    n1->GetTrafficControl().AllowedMovements (fromlanes, *l1, *l5, FALSE);
    fail = TRUE;
} catch (const TNetNotFound& exception) {
    //cout << "(" << exception.GetMessage() << ")" << endl;
    fail = FALSE;
}
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    NET-TC-250: Through node with null control.";
try {
    n4->GetTrafficControl().AllowedMovements (fromlanes, *l1, *l2, FALSE);
    fail = TRUE;
} catch (const TNetUndefinedControl& exception) {
//cout << "(" << exception.GetMessage() << ")" << endl;
    fail = FALSE;
}
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "   Test Interfering lanes method returning collection of "
        "approaching lanes." << endl;

cout << "    NET-TC-260: Left turn through unsignalized control with "
        "phase=false.";
n2->GetTrafficControl().InterferingLanes (fromlanes,
        *(l6->GetLanesTowards(*n2,TRUE)[0]), *(l5->GetLanesFrom(*n2,
        TRUE)[0]), FALSE);
//cout << "# interfering lanes going from 12407 lane 1 to 12384 lane"
//        " 1 = " <<
fail = fromlanes.Length() != 5;
```

```
TNetTrafficControl::LaneCollectionIterator its(fromlanes);
//cout << "Lane " << (int)(*its.CurrentItem())->GetNumber() <<
//        " on link " << (*its.CurrentItem())->GetLink().GetId() << endl;
fail |= (*its.CurrentItem())->GetNumber() != 1
     || (*its.CurrentItem())->GetLink().GetId() != 12384;
its.Next();
fail |= (*its.CurrentItem())->GetNumber() != 2
     || (*its.CurrentItem())->GetLink().GetId() != 12384;
its.Next();
fail |= (*its.CurrentItem())->GetNumber() != 3
     || (*its.CurrentItem())->GetLink().GetId() != 12384;
its.Next();
fail |= (*its.CurrentItem())->GetNumber() != 4
     || (*its.CurrentItem())->GetLink().GetId() != 12384;
its.Next();
fail |= (*its.CurrentItem())->GetNumber() != 1
     || (*its.CurrentItem())->GetLink().GetId() != 28804;
its.Next();
fail |= !its.IsDone();
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    NET-TC-270: Left turn through unsignalized control with "
        "phase=true.";
n2->GetTrafficControl().InterferingLanes (fromlanes,
        *(l6->GetLanesTowards(*n2,TRUE)[0]), *(l5->GetLanesFrom(*n2,
        TRUE)[0]), TRUE);
//cout << "# interfering lanes going from 12407 lane 1 to 12384 lane "
//        "1 = " <<
fail = fromlanes.Length() != 5;
TNetTrafficControl::LaneCollectionIterator it9(fromlanes);
//cout << "Lane " << (int)(*it9.CurrentItem())->GetNumber() <<
//        " on link " << (*it9.CurrentItem())->GetLink().GetId() << endl;
fail |= (*it9.CurrentItem())->GetNumber() != 1
     || (*it9.CurrentItem())->GetLink().GetId() != 12384;
it9.Next();
fail |= (*it9.CurrentItem())->GetNumber() != 2
     || (*it9.CurrentItem())->GetLink().GetId() != 12384;
it9.Next();
fail |= (*it9.CurrentItem())->GetNumber() != 3
     || (*it9.CurrentItem())->GetLink().GetId() != 12384;
it9.Next();
fail |= (*it9.CurrentItem())->GetNumber() != 4
     || (*it9.CurrentItem())->GetLink().GetId() != 12384;
it9.Next();
fail |= (*it9.CurrentItem())->GetNumber() != 1
     || (*it9.CurrentItem())->GetLink().GetId() != 28804;
it9.Next();
fail |= !it9.IsDone();
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    NET-TC-280: Right turn through unsignalized control "
        "with phase=false.";
n2->GetTrafficControl().InterferingLanes (fromlanes,
        *(l6->GetLanesTowards(*n2,TRUE)[1]), *(l7->GetLanesFrom(*n2,
        TRUE)[3]), FALSE);
//cout << "# interfering lanes going from 12407 lane 2 to 28804 "
//        "lane 4 = " <<
fail = fromlanes.Length() != 1;
TNetTrafficControl::LaneCollectionIterator itx(fromlanes);
//cout << "Lane " << (int)(*itx.CurrentItem())->GetNumber() <<
//        " on link " << (*itx.CurrentItem())->GetLink().GetId() << endl;
fail |= (*itx.CurrentItem())->GetNumber() != 4
     || (*itx.CurrentItem())->GetLink().GetId() != 12384;
itx.Next();
fail |= !itx.IsDone();
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    NET-TC-290: Straight through unsignalized control with "
        "phase=default.";
n2->GetTrafficControl().InterferingLanes (fromlanes,
        *(l5->GetLanesTowards(*n2,TRUE)[3]), *(l7->GetLanesFrom(*n2,
        TRUE)[3]));
//cout << "# interfering lanes going from 12384 lane 4 to 28804 lane 4 = "
```

```
//     <<
fail = fromlanes.Length() != 0;
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    NET-TC-300: Left turn through signalized control with "
        "phase=false.";
n1->GetTrafficControl().InterferingLanes (fromlanes,
        *(l1->GetLanesTowards(*n1,TRUE)[0]), *(l2->GetLanesFrom(*n1,
        TRUE)[0]), FALSE);
//cout << "# interfering lanes going from 11487 lane 1 to 11486 "
//        "lane 1 = " <<
fail = fromlanes.Length() != 9;
TNetTrafficControl::LaneCollectionIterator itt(fromlanes);
//cout << "Lane " << (int)(*itt.CurrentItem())->GetNumber() <<
//        " on link " << (*itt.CurrentItem())->GetLink().GetId() << endl;
fail |= (*itt.CurrentItem())->GetNumber() != 1
     || (*itt.CurrentItem())->GetLink().GetId() != 11486;
itt.Next();
fail |= (*itt.CurrentItem())->GetNumber() != 2
     || (*itt.CurrentItem())->GetLink().GetId() != 11486;
itt.Next();
fail |= (*itt.CurrentItem())->GetNumber() != 3
     || (*itt.CurrentItem())->GetLink().GetId() != 11486;
itt.Next();
fail |= (*itt.CurrentItem())->GetNumber() != 1
     || (*itt.CurrentItem())->GetLink().GetId() != 28800;
itt.Next();
fail |= (*itt.CurrentItem())->GetNumber() != 2
     || (*itt.CurrentItem())->GetLink().GetId() != 28800;
itt.Next();
fail |= (*itt.CurrentItem())->GetNumber() != 3
     || (*itt.CurrentItem())->GetLink().GetId() != 28800;
itt.Next();
fail |= (*itt.CurrentItem())->GetNumber() != 3
     || (*itt.CurrentItem())->GetLink().GetId() != 11495;
itt.Next();
fail |= (*itt.CurrentItem())->GetNumber() != 4
     || (*itt.CurrentItem())->GetLink().GetId() != 11495;
itt.Next();
fail |= (*itt.CurrentItem())->GetNumber() != 5
     || (*itt.CurrentItem())->GetLink().GetId() != 11495;
itt.Next();
fail |= !itt.IsDone();
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    NET-TC-310: Left turn through signalized control with "
        "phase=true.";
n1->GetTrafficControl().InterferingLanes (fromlanes,
        *(l1->GetLanesTowards(*n1,TRUE)[0]), *(l2->GetLanesFrom(*n1,
        TRUE)[0]), TRUE);
//cout << "# interfering lanes going from 11487 lane 1 to 11486 lane "
//        "1 = " <<
fail = fromlanes.Length() != 0;
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    NET-TC-320: Right turn through signalized control with "
        "phase=true.";
n1->GetTrafficControl().InterferingLanes (fromlanes,
        *(l2->GetLanesTowards(*n1,TRUE)[2]), *(l1->GetLanesFrom(*n1,
        TRUE)[2]), TRUE);
//cout << "# interfering lanes going from 11486 lane 3 to 11487 lane 3 = "
fail = fromlanes.Length() != 1;
TNetTrafficControl::LaneCollectionIterator it12(fromlanes);
//cout << "Lane " << (int)(*it12.CurrentItem())->GetNumber() << " on link "
//        << (*it12.CurrentItem())->GetLink().GetId() << endl;
fail |= (*it12.CurrentItem())->GetNumber() != 5
     || (*it12.CurrentItem())->GetLink().GetId() != 11495;
it12.Next();
fail |= !it12.IsDone();
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    NET-TC-330: Straight through signalized control with "
```

```
        "phase=default.";
n1->GetTrafficControl().InterferingLanes (fromlanes,
        *(l1->GetLanesTowards(*n1,TRUE)[3]), *(l3->GetLanesFrom(*n1,
        TRUE)[1]));
//cout << "# interfering lanes going from 11487 lane 4 to 11495 lane 2 = "
fail = fromlanes.Length() != 9;
TNetTrafficControl::LaneCollectionIterator itu(fromlanes);
//cout << "Lane " << (int)(*itu.CurrentItem())->GetNumber() <<
//    " on link " << (*itu.CurrentItem())->GetLink().GetId() << endl;
fail |= (*itu.CurrentItem())->GetNumber() != 1
     || (*itu.CurrentItem())->GetLink().GetId() != 28800;
itu.Next();
fail |= (*itu.CurrentItem())->GetNumber() != 2
     || (*itu.CurrentItem())->GetLink().GetId() != 28800;
itu.Next();
fail |= (*itu.CurrentItem())->GetNumber() != 3
     || (*itu.CurrentItem())->GetLink().GetId() != 28800;
itu.Next();
fail |= (*itu.CurrentItem())->GetNumber() != 4
     || (*itu.CurrentItem())->GetLink().GetId() != 28800;
itu.Next();
fail |= (*itu.CurrentItem())->GetNumber() != 5
     || (*itu.CurrentItem())->GetLink().GetId() != 28800;
itu.Next();
fail |= (*itu.CurrentItem())->GetNumber() != 2
     || (*itu.CurrentItem())->GetLink().GetId() != 11486;
itu.Next();
fail |= (*itu.CurrentItem())->GetNumber() != 3
     || (*itu.CurrentItem())->GetLink().GetId() != 11486;
itu.Next();
fail |= (*itu.CurrentItem())->GetNumber() != 1
     || (*itu.CurrentItem())->GetLink().GetId() != 11495;
itu.Next();
fail |= (*itu.CurrentItem())->GetNumber() != 2
     || (*itu.CurrentItem())->GetLink().GetId() != 11495;
itu.Next();
fail |= !itu.IsDone();
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    NET-TC-340: Straight through signalized control with "
        "phase=true.";
n5->GetTrafficControl().InterferingLanes (fromlanes,
        *(l6->GetLanesTowards(*n5,TRUE)[0]), *(l8->GetLanesFrom(*n5,
        TRUE)[0]), TRUE);
//cout << "# interfering lanes going from 12407 lane 1 to 9704 lane 1 = "
fail = fromlanes.Length() != 0;
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    NET-TC-350: Across lane change node with phase=false.";
n3->GetTrafficControl().InterferingLanes (fromlanes,
        *(l4->GetLanesTowards(*n3,TRUE)[2]), *(l5->GetLanesFrom(*n3,
        TRUE)[2]));
//cout << "# interfering lanes going from 28800 lane 3 to 12384 lane 3 = "
fail = fromlanes.Length() != 0;
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    NET-TC-360: Across lane change node with phase=true.";
n3->GetTrafficControl().InterferingLanes (fromlanes,
        *(l4->GetLanesTowards(*n3,TRUE)[2]), *(l5->GetLanesFrom(*n3,
        TRUE)[2]));
//cout << "# interfering lanes going from 28800 lane 3 to 12384 lane 3 = "
fail = fromlanes.Length() != 0;
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    NET-TC-370: Between lanes that are not connected.";
n1->GetTrafficControl().InterferingLanes (fromlanes,
        *(l1->GetLanesTowards(*n1,TRUE)[5]), *(l3->GetLanesFrom(*n1,
        TRUE)[2]));
//cout << "# interfering lanes going from 11487 lane 6 to 11495 lane 3 = "
fail = fromlanes.Length() != 0;
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;
```

```
    cout << "    NET-TC-380: Through node with null control.";
    try {
        n4->GetTrafficControl().InterferingLanes (fromlanes,
                *(l1->GetLanesTowards(*n4,TRUE)[0]), *(l2->GetLanesFrom(*n4,
                TRUE)[0]));
        fail = TRUE;
    } catch (const TNetNotFound& exception) {
        //cout << "(" << exception.GetMessage() << ")" << endl;
        fail = FALSE;
    }
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << (anyFail ? "    F" : "    No f") << "ailures occurred." << endl;
    return !anyFail;
}
```

U. TestUnsignalizedControl.C

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: TestUnsignalizedControl.C,v $
//  $Revision: 2.1 $
//  $Date: 1996/02/14 21:36:42 $
//  $State: Exp $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


//  Include standard C++ header files
#include <iostream.h>


//  Include TRANSIMS header files.
#include "NET/Network.h"
#include "NET/Node.h"
#include "NET/Link.h"
#include "NET/UnsignalizedControl.h"


//  Test Unsignalized control class.
bool TestUnsignalizedControl(TNetNetwork& network)
{
    cout << "  Unsignalized Control Class Tests "
         << endl;

    bool anyFail = FALSE;
    bool fail;

    TNetNetwork::NodeMap& nodes = network.GetNodes();
    TNetNode* n2 = *nodes.ValueOf(14136);
    TNetNetwork::LinkMap& links = network.GetLinks();
    TNetLink* l6 = *links.ValueOf(12407);
    TNetLink* l7 = *links.ValueOf(28804);

    cout << "    NET-UC-010: Retrieve vehicle control for the specified lane.";
    //cout << "Traffic control on link " << l6->GetId() << " = " <<
    //    n2->GetTrafficControl().GetVehicleControl(*(l6->GetLanesTowards(*n2,
    //    TRUE)[0])) << endl;
    //cout << "Traffic control on link " << l7->GetId() << " = " <<
    //        n2->GetTrafficControl().GetVehicleControl(*(l7
    //        ->GetLanesTowards(*n2,TRUE)[0])) << endl;
    fail =  l6->GetId() != 12407
         || n2->GetTrafficControl().GetVehicleControl(*(l6
          ->GetLanesTowards(*n2,TRUE)[0])) != 1
         || l7->GetId() != 28804
         || n2->GetTrafficControl().GetVehicleControl(*(l7
         ->GetLanesTowards(*n2,TRUE)[0])) != 0;
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << (anyFail ? "    F" : "    No f") << "ailures occurred." << endl;
    return !anyFail;
```

```
}
```

X. APPENDIX: Data Source Creation Utility

This appendix contains the complete C++ source code for the network data source creation utility.

A. CreateSources.C

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: CreateSources.C,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 19:29:51 $
//  $State: Rel $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


//  Include Standard C++ header files.
#include <iostream.h>


//  Include TRANSIMS header files.
#include <GBL/Globals.h>
#include <DBS/Directory.h>
#include <DBS/Exception.h>


//  Main program.
int main(int, char*[])
{
    try {

        TDbDirectory directory(TDbDirectoryDescription("IOC-1"));

        if (!directory.HasSource("Node"))
            directory.CreateSource(TDbSourceDescription("Node",
                    "This data source contains node data."));

        if (!directory.HasSource("Link"))
            directory.CreateSource(TDbSourceDescription("Link",
                    "This data source contains link data."));

        if (!directory.HasSource("Pocket Lane"))
            directory.CreateSource(TDbSourceDescription("Pocket Lane",
                    "This data source contains pocket lane data."));

        if (!directory.HasSource("Lane Use"))
            directory.CreateSource(TDbSourceDescription("Lane Use",
                    "This data source contains lane use data."));

        if (!directory.HasSource("Parking"))
            directory.CreateSource(TDbSourceDescription("Parking",
                    "This data source contains parking data."));

        if (!directory.HasSource("Lane Connectivity"))
            directory.CreateSource(TDbSourceDescription("Lane Connectivity",
                    "This data source contains lane connectivity data."));

        if (!directory.HasSource("Unsignalized Node"))
            directory.CreateSource(TDbSourceDescription("Unsignalized Node",
                    "This data source contains unsignalized node data."));

        if (!directory.HasSource("Signalized Node"))
            directory.CreateSource(TDbSourceDescription("Signalized Node",
                    "This data source contains signalized node data."));

        if (!directory.HasSource("Phasing Plan"))
            directory.CreateSource(TDbSourceDescription("Phasing Plan",
                    "This data source contains signalized phasing plan data."));
```

```
        if (!directory.HasSource("Timing Plan"))
            directory.CreateSource(TDbSourceDescription("Timing Plan",
                    "This data source contains signalized timing plan data."));

    } catch(const TDbException& dbException) {

        cerr << "Database exception: " << dbException.GetMessage() << endl;

    } catch(...) {

        cerr << "Unknown exception." << endl;

    }

    return 0;
}
```

XI. APPENDIX: Network Dump Utility

This appendix contains the complete C++ source code for the network data dump utility.

A. DumpNetwork.C

```
//  Project: TRANSIMS
//  Subsystem: Network
//  $RCSfile: DumpNetwork.C,v $
//  $Revision: 2.1 $
//  $Date: 1996/05/02 19:50:38 $
//  $State: Rel $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


//  Include Standard C++ header files.
#include <iostream.h>


//  Include TRANSIMS header files.
#include <DBS/Exception.h>
#include <DBS/Directory.h>
#include <DBS/Source.h>
#include <DBS/Table.h>
#include <NET/Exception.h>
#include <NET/Reader.h>
#include <NET/Network.h>
#include <NET/Subnetwork.h>
#include <NET/Node.h>
#include <NET/Link.h>
#include <NET/Lane.h>
#include <NET/Accessory.h>
#include <NET/Pocket.h>
#include <NET/Parking.h>
#include <NET/Location.h>
#include <NET/LaneLocation.h>


//  Put a description of the specified node on the specified stream.
ostream& operator<<(ostream& os, const TNetNode& node)
{
    os << "N#" << node.GetId() << "@" << (void*) &node;

    return os;
}


//  Put a description of the specified link on the specified stream.
ostream& operator<<(ostream& os, const TNetLink& link)
{
    os << "L#" << link.GetId() << "@" << (void*) &link;

    return os;
}
```

```
//  Put a description of the specified lane on the specified stream.
ostream& operator<<(ostream& os, const TNetLane& lane)
{
    os << " l#" << lane.GetNumber() << "@" << (void*) &lane;

    return os;
}


//  Put a description of the specified accessory on the specified stream.
ostream& operator<<(ostream& os, const TNetAccessory& accessory)
{
    os << "A#" << accessory.GetId() << "@" << (void*) &accessory;

    return os;
}


//  Put a representation of the specified point on the specified stream.
ostream& operator<<(ostream& os, const TGeoPoint& point)
{
    os << "(" << point.GetX() << ", " << point.GetY() << ")";

    return os;
}


//  Put a representation of the specified location on the specified stream.
ostream& operator<<(ostream& os, const TNetLocation& location)
{
    const TNetLink& link = location.GetLink();
    const TNetNode* node[2];
    link.GetNodes(node[0], node[1]);
    os << location.GetOffsetFrom(*node[0]) << " from " << *node[0] << " and " <<
            location.GetOffsetFrom(*node[1]) << " from " << *node[1] << " on "
            << link;
    if (location.IsOnSpecificLane())
        os << " on the lane " << ((const TNetLaneLocation&)
                location).GetLane().GetNumber();

    return os;
}


//  Dump a complete description of the specified node on the specified stream.
void DumpNode(const TNetNode& node, ostream& os)
{
    os << "Node: " << node << endl;
    os << "  Position: " << node.GetGeographicPosition() << endl;
    os << "  Links:";
    for (TNetNode::LinkRingIterator i(node.GetLinks()); !i.IsDone(); i.Next())
        os << " " << **i.CurrentItem();
    os << endl;
}


//  Dump a complete description of the specified link on the specified stream.
void DumpLink(const TNetLink& link, ostream& os)
{
    os << "Link: " << link << endl;
    const TNetNode* nodeA;
    const TNetNode* nodeB;
    link.GetNodes(nodeA, nodeB);
    os << "  Nodes: " << *nodeA << " " << *nodeB << endl;
    os << "  Accessories:";
    for (TNetLink::AccessoryCollectionIterator i(link.GetAccessories());
            !i.IsDone(); i.Next())
        os << " " << **i.CurrentItem();
    os << endl;
    os << "  Length:" << endl;
    os << "    w/o setbacks: " << link.GetLength() << endl;
    os << "    with setbacks: " << link.GetLength(TRUE) << endl;
    os << "  Setbacks:" << endl;
    os << "    at " << *nodeA << ": " << link.GetSetback(*nodeA) << endl;
    os << "    at " << *nodeB << ": " << link.GetSetback(*nodeB) << endl;
```

```
    os << "  Through links:" << endl;
    os << "    at " << *nodeA << ": " << link.GetThroughLink(*nodeA) << endl;
    os << "    at " << *nodeB << ": " << link.GetThroughLink(*nodeB) << endl;
    os << "  Speed Limits:" << endl;
    os << "    at " << *nodeA << ": " << link.GetSpeedLimitTowards(*nodeA) <<
            endl;
    os << "    at " << *nodeB << ": " << link.GetSpeedLimitTowards(*nodeB) <<
            endl;
}


//  Dump a complete description of the specified pocket on the specified stream.
void DumpPocket(const TNetPocket& pocket, ostream& os)
{
    os << "  Style: ";
    switch (pocket.GetStyle()) {
        case TNetPocket::kTurn:
            os << "turn" << endl;
            break;
        case TNetPocket::kPullout:
            os << "pullout" << endl;
            break;
        case TNetPocket::kMerge:
            os << "merge" << endl;
            break;
    }
    os << "  Length: " << pocket.GetLength() << endl;
    os << "  Lane: " << pocket.GetLane() << endl;
}


//  Dump a complete description of the specified parking on the specified
//  stream.
void DumpParking(const TNetParking& parking, ostream& os)
{
    os << "  Style: ";
    switch (parking.GetStyle()) {
        case TNetParking::kParallelOnStreet:
            os << "parallel on street" << endl;
            break;
        case TNetParking::kHeadInOnStreet:
            os << "head-in on street" << endl;
            break;
        case TNetParking::kDriveway:
            os << "driveway" << endl;
            break;
        case TNetParking::kLot:
            os << "lot" << endl;
            break;
        case TNetParking::kBoundary:
            os << "boundary" << endl;
            break;
    }
    os << "  Capacity: " << parking.GetCapacity() << endl;
    os << "  Generic: " << (parking.IsGeneric() ? "yes" : "no") << endl;
}


//  Dump a complete description of the specified accessory on the specified
//  stream.
void DumpAccessory(const TNetAccessory& accessory, ostream& os)
{
    os << "Accessory: " << accessory << endl;
    os << "  Location: " << accessory.GetLocation() << endl;
    os << "  Type: ";
    switch (accessory.GetType()) {
        case TNetAccessory::kPocket:
            os << "pocket" << endl;
            DumpPocket((const TNetPocket&) accessory, os);
            break;
        case TNetAccessory::kParking:
            os << "parking" << endl;
            DumpParking((const TNetParking&) accessory, os);
            break;
    }
}
```

```
//  Dump a complete description of the specified lane on the specified stream.
void DumpLane(const TNetLane& lane, ostream& os)
{
    os << "Lane: " << lane << endl;
    os << "  Link: " << lane.GetLink() << endl;
    os << "  Start Node: " << lane.GetStartNode() << endl;
    os << "  End Node: " << lane.GetEndNode() << endl;
    bool perm = FALSE;
    for (TNetLink::LaneCollectionIterator
            i(lane.GetLink().GetLanesFrom(lane.GetStartNode())); !i.IsDone();
            i.Next())
        if (*i.CurrentItem() == &lane)
            perm = TRUE;
    os << "  Permanent: " << (perm ? "yes" : "no") << endl;
}


//  Dump a network.
int main(int, char*[])
{
    try {

        TDbDirectory directory(TDbDirectoryDescription("IOC-1"));

        TDbSource nodeSource(directory, directory.GetSource("Node"));
        TDbSource linkSource(directory, directory.GetSource("Link"));
        TDbSource pocketSource(directory, directory.GetSource("Pocket Lane"));
        TDbSource parkingSource(directory, directory.GetSource("Parking"));
        TDbSource laneSource(directory,
                directory.GetSource("Lane Connectivity"));
        TDbSource ucontrolSource(directory,
                directory.GetSource("Unsignalized Node"));
        TDbSource scontrolSource(directory,
                directory.GetSource("Signalized Node"));
        TDbSource phasingSource(directory,
                directory.GetSource("Phasing Plan"));
        TDbSource timingSource(directory,
                directory.GetSource("Timing Plan"));

        char line[80];
        cin.getline(line, 80);
        TDbTable nodeTable(nodeSource, nodeSource.GetTable(line));
        cin.getline(line, 80);
        TDbTable linkTable(linkSource, linkSource.GetTable(line));
        cin.getline(line, 80);
        TDbTable pocketTable(pocketSource, pocketSource.GetTable(line));
        cin.getline(line, 80);
        TDbTable parkingTable(parkingSource, parkingSource.GetTable(line));
        cin.getline(line, 80);
        TDbTable laneTable(laneSource, laneSource.GetTable(line));
        cin.getline(line, 80);
        TDbTable ucontrolTable(ucontrolSource, ucontrolSource.GetTable(line));
        cin.getline(line, 80);
        TDbTable scontrolTable(scontrolSource, scontrolSource.GetTable(line));
        cin.getline(line, 80);
        TDbTable phasingTable(phasingSource, phasingSource.GetTable(line));
        cin.getline(line, 80);
        TDbTable timingTable(timingSource, timingSource.GetTable(line));

        TNetReader reader(nodeTable, linkTable, pocketTable, parkingTable,
                laneTable, ucontrolTable, scontrolTable, phasingTable,
                timingTable);

        TNetNetwork network;
        TNetSubnetwork subnetwork(reader, network);

        TNetSubnetwork::NodeSet& nodes = subnetwork.GetNodes();
        TNetSubnetwork::LinkSet& links = subnetwork.GetLinks();

        cout << "***** NODES *****" << endl;
        for (TNetSubnetwork::NodeSetIterator n(nodes); !n.IsDone(); n.Next())
            DumpNode(**n.CurrentItem(), cout);
        cout << endl;
```

```
        cout << "***** LINKS *****" << endl;
        for (TNetSubnetwork::LinkSetIterator l(links); !l.IsDone(); l.Next())
            DumpLink(**l.CurrentItem(), cout);
        cout << endl;

        cout << "***** ACCESSORIES ON LINKS *****" << endl;
        for (l.Reset(); !l.IsDone(); l.Next()) {
            const TNetLink& link = **l.CurrentItem();
            cout << ". . . on " << link << " . . ." << endl;
            for (TNetLink::AccessoryCollectionIterator a(link.GetAccessories());
                    !a.IsDone(); a.Next())
                DumpAccessory(**a.CurrentItem(), cout);
        }
        cout << endl;

        cout << "***** LANES *****" << endl;
        for (l.Reset(); !l.IsDone(); l.Next()) {
            const TNetLink& link = **l.CurrentItem();
            const TNetNode* node[2];
            link.GetNodes((TNetNode*&) node[0], (TNetNode*&) node[1]);
            for (size_t side = 0; side < 2; ++side) {
                cout << ". . . on " << link << " toward " << *node[side] <<
                        " . . ." << endl;
                for (TNetLink::LaneCollectionIterator
                        ll(link.GetLanesTowards(*node[side], TRUE));
                        !ll.IsDone(); ll.Next())
                    DumpLane(**ll.CurrentItem(), cout);
            }
        }
        cout << endl;

        cout << "***** POCKETS ON LANES *****" << endl;
        for (l.Reset(); !l.IsDone(); l.Next()) {
            const TNetLink& link = **l.CurrentItem();
            const TNetNode* node[2];
            link.GetNodes((TNetNode*&) node[0], (TNetNode*&) node[1]);
            for (size_t side = 0; side < 2; ++side) {
                cout << ". . . on " << link << " toward " << *node[side] <<
                        " . . ." << endl;
                for (TNetLink::LaneCollectionIterator
                        ll(link.GetLanesTowards(*node[side], TRUE));
                        !ll.IsDone(); ll.Next())
                    for (TNetLane::PocketCollectionIterator
                            p((*ll.CurrentItem())->GetPockets()); !p.IsDone();
                            p.Next())
                        DumpAccessory(**p.CurrentItem(), cout);
            }
        }
        cout << endl;

    } catch(const TNetException& netException) {

        cerr << "Network exception: " << netException.GetMessage() << endl;

    } catch(const TDbException& dbException) {

        cerr << "Database exception: " << dbException.GetMessage() << endl;

    } catch(...) {

        cerr << "Unknown exception." << endl;

    }

    return 0;
}
```

XII. APPENDIX: Network Data Tables

This appendix contains database subsystem import files for a variety of network data tables.

A. *Empty Network*

The following empty network tables can be used as prototypes for constructing networks.

1. EmptyNetwork.Dat

```
Empty Node Table
This is an empty node table.
NODEEMPT
Node
CREATE TABLE NODEEMPT (
  ID NUMBER(10),
  ABSCISSA FLOAT,
  ORDINATE FLOAT,
  PRIMARY KEY (ID)
);
ID, ABSCISSA, ORDINATE

Empty Link Table
This is an empty link table.
LINKEMPT
Link
CREATE TABLE LINKEMPT (
  ID  NUMBER(10),
  NODEA NUMBER(10),
  NODEB NUMBER(10),
  PERMLANESA NUMBER(3),
  PERMLANESB NUMBER(3),
  LEFTPOCKETSA NUMBER(3),
  LEFTPOCKETSB NUMBER(3),
  RGHTPOCKETSA NUMBER(3),
  RGHTPOCKETSB NUMBER(3),
  TWOWAYTURN CHAR(1),
  LENGTH FLOAT,
  GRADE FLOAT,
  SETBACKA FLOAT,
  SETBACKB FLOAT,
  CAPACITYA FLOAT,
  CAPACITYB FLOAT,
  SPEEDLIMITA FLOAT,
  SPEEDLIMITB FLOAT,
  FREESPEEDA FLOAT,
  FREESPEEDB FLOAT,
  CRAWLSPEEDA FLOAT,
  CRAWLSPEEDB FLOAT,
  THRUA NUMBER(10),
  THRUB NUMBER(10),
  COSTA NUMBER(10),
  COSTB NUMBER(10),
  FUNCTCLASS VARCHAR(10),
  PRIMARY KEY (ID)
);
ID, NODEA, NODEB, PERMLANESA, PERMLANESB, LEFTPOCKETSA, LEFTPOCKETSB, RGHTPOCKETSA,
RGHTPOCKETSB, TWOWAYTURN, LENGTH, GRADE, SETBACKA, SETBACKB, CAPACITYA, CAPACITYB,
SPEEDLIMITA, SPEEDLIMITB, FREESPEEDA, FREESPEEDB, CRAWLSPEEDA, CRAWLSPEEDB, THRUA, THRUB,
COSTA, COSTB, FUNCTCLASS

Empty Pocket Lane Table
This is an empty pocket lane table.
POCKEMPT
Pocket Lane
CREATE TABLE POCKEMPT (
  ID NUMBER(10),
  NODE NUMBER(10),
  LINK NUMBER(10),
  OFFSET FLOAT,
  LANE NUMBER(3),
  STYLE CHAR(1),
  LENGTH FLOAT,
  PRIMARY KEY (ID)
);
ID, NODE, LINK, OFFSET, LANE, STYLE, LENGTH

Empty Lane Use Table
This is an empty lane use table.
```

```
LANEEMPT
Lane Use
CREATE TABLE LANEEMPT (
  NODE NUMBER(10),
  LINK NUMBER(10),
  LANE NUMBER(3),
  USE VARCHAR(10)
);
NODE, LINK, LANE, USE

Empty Parking Table
This is an empty parking table.
PARKEMPT
Parking
CREATE TABLE PARKEMPT (
  ID NUMBER(10),
  LINK NUMBER(10),
  NODE NUMBER(10),
  OFFSET FLOAT,
  STYLE CHAR(5),
  CAPACITY NUMBER(5),
  GENERIC CHAR(1),
  PRIMARY KEY (ID)
);
ID, LINK, NODE, OFFSET, STYLE, CAPACITY, GENERIC

Empty Lane Connectivity Table
This is an empty lane connectivity table.
CONNEMPT
Lane Connectivity
CREATE TABLE CONNEMPT (
  NODE NUMBER(10),
  INLINK NUMBER(10),
  INLANE NUMBER(3),
  OUTLINK NUMBER(10),
  OUTLANE NUMBER(3)
);
NODE, INLINK, INLANE, OUTLINK, OUTLANE

Empty Unsignalized Node Table
This is an empty unsignalized node table.
UNSIEMPT
Unsignalized Node
CREATE TABLE UNSIEMPT (
  NODE NUMBER(10),
  INLINK NUMBER(10),
  SIGN CHAR(1)
);
NODE, INLINK, SIGN

Empty Signalized Node Table
This is an empty signalized node table.
SIGNEMPT
Signalized Node
CREATE TABLE SIGNEMPT (
  NODE NUMBER(10),
  TYPE CHAR(1),
  PLAN NUMBER(3),
  OFFSET FLOAT,
  STARTTIME CHAR(8)
);
NODE, TYPE, PLAN, OFFSET, STARTTIME

Empty Phasing Plan Table
This is an empty phasing plan table.
PHASEMPT
Phasing Plan
CREATE TABLE PHASEMPT (
  NODE NUMBER(10),
  PLAN NUMBER(3),
  PHASE NUMBER(3),
  INLINK NUMBER(10),
  OUTLINK NUMBER(10),
  PROTECTION CHAR(1)
);
NODE, PLAN, PHASE, INLINK, OUTLINK, PROTECTION
```

```
Empty Timing Plan Table
This is an empty timing plan table.
TIMEEMPT
Timing Plan
CREATE TABLE TIMEEMPT (
  PLAN NUMBER(3),
  PHASE NUMBER(3),
  NEXTPHASES VARCHAR(20),
  GREENMIN FLOAT,
  GREENMAX FLOAT,
  GREENEXT FLOAT,
  YELLOW FLOAT,
  REDCLEAR FLOAT
);
PLAN, PHASE, NEXTPHASES, GREENMIN, GREENMAX, GREENEXT, YELLOW, REDCLEAR
```

B. Sample Networks

The following sample network tables are required for the network test program. They correspond closely to the example network in the text.

1. SampleNetwork2.dat

```
Sample Node Table II
This is a sample node table.
NODESAMP2
Node
CREATE TABLE NODESAMP2 (
  ID NUMBER(10),
  ABSCISSA FLOAT,
  ORDINATE FLOAT,
  PRIMARY KEY (ID)
);
ID, ABSCISSA, ORDINATE
8520, 1.0, 0.5
8521, 0.0, 0.0
14136, 1.0, 0.0
14141, 1.0, 1.0
14062, 0.5, 1.0
14142, 1.0, 1.5
14340, 1.5, 1.0
8525, 1.0, -0.5
8522, 0.0, 0.5
8523, -0.5, 0.0
8524, 0.0, -0.5

Sample Link Table II
This is a sample link table.
LINKSAMP2
Link
CREATE TABLE LINKSAMP2 (
  ID  NUMBER(10),
  NODEA NUMBER(10),
  NODEB NUMBER(10),
  PERMLANESA NUMBER(3),
  PERMLANESB NUMBER(3),
  LEFTPOCKETSA NUMBER(3),
  LEFTPOCKETSB NUMBER(3),
  RGHTPOCKETSA NUMBER(3),
  RGHTPOCKETSB NUMBER(3),
  TWOWAYTURN CHAR(1),
  LENGTH FLOAT,
  GRADE FLOAT,
  SETBACKA FLOAT,
  SETBACKB FLOAT,
  CAPACITYA FLOAT,
  CAPACITYB FLOAT,
  SPEEDLIMITA FLOAT,
  SPEEDLIMITB FLOAT,
  FREESPEEDA FLOAT,
  FREESPEEDB FLOAT,
  CRAWLSPEEDA FLOAT,
```

```
  CRAWLSPEEDB FLOAT,
  THRUA NUMBER(10),
  THRUB NUMBER(10),
  COSTA NUMBER(10),
  COSTB NUMBER(10),
  FUNCTCLASS VARCHAR(10),
  PRIMARY KEY (ID)
);
ID, NODEA, NODEB, PERMLANESA, PERMLANESB, LEFTPOCKETSA, LEFTPOCKETSB, RGHTPOCKETSA,
RGHTPOCKETSB, TWOWAYTURN, LENGTH, GRADE, SETBACKA, SETBACKB, CAPACITYA, CAPACITYB,
SPEEDLIMITA, SPEEDLIMITB, FREESPEEDA, FREESPEEDB, CRAWLSPEEDA, CRAWLSPEEDB, THRUA, THRUB,
COSTA, COSTB, FUNCTCLASS
12384, 14136, 8520, 4, 4, 0, 1, 0, 1, 'T', 1000.0, 1.0, 15.0, 0.0, 800, 1000, 45.0, 45.0,
50.0, 50.0, 15.0, 15.0, 28804, 28800, 1, 1, 'OTHER'
12407, 8521, 14136, 2, 2, 0, 0, 1, 0, 'F', 1200.0, 0.0, 10.0, 15.0, 500, 500, 35.0, 35.0,
40.0, 40.0, 10.0, 10.0, 9704, 0, 1, 1, 'OTHER'
28800, 8520, 14141, 3, 4, 0, 1, 0, 1, 'T', 1500.0, 1.0, 0.0, 15.0, 800, 1000, 45.0, 45.0,
50.0, 50.0, 15.0, 15.0, 12384, 11486, 1, 1, 'OTHER'
11487, 14062, 14141, 3, 3, 0, 2, 0, 1, 'F', 1000.0, 1.0, 0.0, 15.0, 800, 1000, 45.0, 45.0,
50.0, 50.0, 15.0, 15.0, 0, 11495, 1, 1, 'OTHER'
11486, 14141, 14142, 2, 3, 1, 0, 0, 0, 'F', 1000.0, 1.0, 0.0, 15.0, 800, 1000, 45.0, 45.0,
50.0, 50.0, 15.0, 15.0, 28800, 0, 1, 1, 'OTHER'
11495, 14141, 14340, 3, 3, 2, 0, 1, 0, 'F', 1000.0, 1.0, 0.0, 15.0, 800, 1000, 45.0, 45.0,
50.0, 50.0, 15.0, 15.0, 11487, 0, 1, 1, 'OTHER'
28804, 14136, 8525, 5, 4, 0, 0, 0, 0, 'F', 1000.0, 1.0, 0.0, 15.0, 800, 1000, 45.0, 45.0,
50.0, 50.0, 15.0, 15.0, 12384, 0, 1, 1, 'OTHER'
9706, 8521, 8524, 1, 1, 0, 0, 0, 0, 'F', 1000.0, 1.0, 0.0, 15.0, 800, 1000, 45.0, 45.0,
50.0, 50.0, 15.0, 15.0, 9705, 0, 1, 1, 'OTHER'
9705, 8521, 8522, 1, 1, 0, 0, 0, 0, 'F', 1000.0, 1.0, 0.0, 15.0, 800, 1000, 45.0, 45.0,
50.0, 50.0, 15.0, 15.0, 9706, 0, 1, 1, 'OTHER'
9704, 8521, 8523, 2, 2, 0, 0, 0, 0, 'F', 1000.0, 1.0, 0.0, 15.0, 800, 1000, 45.0, 45.0,
50.0, 50.0, 15.0, 15.0, 9704, 0, 1, 1, 'OTHER'

Sample Pocket Lane Table II
This is a sample pocket lane table.
POCKSAMP2
Pocket Lane
CREATE TABLE POCKSAMP2 (
  ID NUMBER(10),
  NODE NUMBER(10),
  LINK NUMBER(10),
  OFFSET FLOAT,
  LANE NUMBER(3),
  STYLE CHAR(1),
  LENGTH FLOAT,
  PRIMARY KEY (ID)
);
ID, NODE, LINK, OFFSET, LANE, STYLE, LENGTH
85201, 8520, 12384, 0, 1, 'M', 100.0
85206, 8520, 12384, 0, 6, 'M', 200.0
85213, 8521, 12407, 600.0, 3, 'P', 30.0
141411, 14141, 28800, 0, 1, 'T', 200.0
141416, 14141, 28800, 0, 6, 'T', 300.0

Sample Lane Use Table II
This is a sample lane use table.
LANESAMP2
Lane Use
CREATE TABLE LANESAMP2 (
  NODE NUMBER(10),
  LINK NUMBER(10),
  LANE NUMBER(3),
  USE VARCHAR(10)
);
NODE, LINK, LANE, USE
14136, 12407, 2, 'PARKING'

Sample Parking Table II
This is a sample parking table.
PARKSAMP2
Parking
CREATE TABLE PARKSAMP2 (
  ID NUMBER(10),
  LINK NUMBER(10),
  NODE NUMBER(10),
  OFFSET FLOAT,
```

```
  STYLE CHAR(5),
  CAPACITY NUMBER(5),
  GENERIC CHAR(1),
  PRIMARY KEY (ID)
);
ID, LINK, NODE, OFFSET, STYLE, CAPACITY, GENERIC
1001, 28800, 8520, 400, 'LOT', 50, 'T'
1002, 12384, 14136, 300, 'PRSTR', 10, 'T'
1003, 12407, 14136, 200, 'HISTR', 10, 'T'
1004, 12407, 8521, 100, 'DRVWY', 1, 'F'

Sample Lane Connectivity Table II
This is a sample lane connectivity table.
CONNSAMP2
Lane Connectivity
CREATE TABLE CONNSAMP2 (
  NODE NUMBER(10),
  INLINK NUMBER(10),
  INLANE NUMBER(3),
  OUTLINK NUMBER(10),
  OUTLANE NUMBER(3)
);
NODE, INLINK, INLANE, OUTLINK, OUTLANE
14141, 11487, 1, 11486, 1
14141, 11487, 2, 11486, 2
14141, 11487, 3, 11495, 1
14141, 11487, 4, 11495, 2
14141, 11487, 5, 11495, 3
14141, 11487, 6, 28800, 3
14141, 11486, 1, 11495, 1
14141, 11486, 2, 28800, 1
14141, 11486, 3, 28800, 2
14141, 11486, 3, 11487, 3
14141, 11495, 1, 28800, 1
14141, 11495, 2, 28800, 2
14141, 11495, 3, 11487, 1
14141, 11495, 4, 11487, 2
14141, 11495, 5, 11487, 3
14141, 11495, 6, 11486, 3
14141, 28800, 1, 11487, 1
14141, 28800, 2, 11487, 2
14141, 28800, 3, 11486, 1
14141, 28800, 4, 11486, 2
14141, 28800, 5, 11486, 3
14141, 28800, 6, 11495, 3
8520, 12384, 2, 28800, 2
8520, 12384, 3, 28800, 3
8520, 12384, 4, 28800, 4
8520, 12384, 5, 28800, 5
8520, 28800, 1, 12384, 1
8520, 28800, 2, 12384, 2
8520, 28800, 3, 12384, 3
8520, 28800, 3, 12384, 4
14136, 12407, 1, 12384, 1
14136, 12407, 2, 28804, 4
14136, 12384, 1, 28804, 1
14136, 12384, 2, 28804, 2
14136, 12384, 3, 28804, 3
14136, 12384, 4, 28804, 4
14136, 12384, 4, 12407, 2
14136, 28804, 1, 12407, 1
14136, 28804, 1, 12384, 2
14136, 28804, 2, 12384, 3
14136, 28804, 3, 12384, 4
14136, 28804, 4, 12384, 5
14136, 28804, 5, 12384, 6
8521, 12407, 1, 9704, 1
8521, 12407, 1, 9706, 1
8521, 12407, 2, 9704, 2
8521, 12407, 2, 9705, 1
8521, 9704, 1, 12407, 1
8521, 9704, 1, 9705, 1
8521, 9704, 2, 12407, 2
8521, 9704, 2, 9706, 1
8521, 9705, 1, 9706, 1
8521, 9705, 1, 9704, 2
```

```
8521, 9705, 1, 12407, 1
8521, 9706, 1, 9705, 1
8521, 9706, 1, 12407, 2
8521, 9706, 1, 9704, 1
```

Sample Unsignalized Node Table II
This is a sample unsignalized node table.
UNSISAMP2
Unsignalized Node

```
CREATE TABLE UNSISAMP2 (
  NODE NUMBER(10),
  INLINK NUMBER(10),
  SIGN CHAR(1)
);
NODE, INLINK, SIGN
8520, 12384, 'N'
8520, 28800, 'N'
14136, 12407, 'S'
14136, 12384, 'N'
14136, 28804, 'N'
```

Sample Signalized Node Table II
This is a sample signalized node table.
SIGNSAMP2
Signalized Node

```
CREATE TABLE SIGNSAMP2 (
  NODE NUMBER(10),
  TYPE CHAR(1),
  PLAN NUMBER(3),
  OFFSET FLOAT,
  STARTTIME CHAR(8)
);
NODE, TYPE, PLAN, OFFSET, STARTTIME
14141, 'T', 1, 19.0, 'ALL0:00'
8521, 'A', 2, 0.0, 'ALL18:00'
8521, 'A', 3, 0.0, 'WKD7:00'
```

Sample Phasing Plan Table II
This is a sample phasing plan table.
PHASSAMP2
Phasing Plan

```
CREATE TABLE PHASSAMP2 (
  NODE NUMBER(10),
  PLAN NUMBER(3),
  PHASE NUMBER(3),
  INLINK NUMBER(10),
  OUTLINK NUMBER(10),
  PROTECTION CHAR(1)
);
NODE, PLAN, PHASE, INLINK, OUTLINK, PROTECTION
14141, 1, 1, 11487, 11495, 'U'
14141, 1, 1, 11487, 28800, 'P'
14141, 1, 1, 11495, 11487, 'U'
14141, 1, 1, 11495, 11486, 'P'
14141, 1, 1, 11486, 11487, 'U'
14141, 1, 1, 28800, 11495, 'U'
14141, 1, 2, 11487, 28800, 'P'
14141, 1, 2, 11495, 11486, 'P'
14141, 1, 2, 11486, 11495, 'P'
14141, 1, 2, 28800, 11487, 'P'
14141, 1, 2, 28800, 11495, 'U'
14141, 1, 2, 11486, 11487, 'U'
14141, 1, 3, 11487, 28800, 'P'
14141, 1, 3, 28800, 11487, 'P'
14141, 1, 3, 28800, 11486, 'U'
14141, 1, 3, 28800, 11495, 'P'
14141, 1, 3, 11495, 11486, 'U'
14141, 1, 3, 11486, 11487, 'U'
14141, 1, 4, 11487, 28800, 'P'
14141, 1, 4, 11486, 11495, 'U'
14141, 1, 4, 11486, 28800, 'U'
14141, 1, 4, 11486, 11487, 'P'
14141, 1, 4, 28800, 11486, 'U'
14141, 1, 4, 28800, 11495, 'P'
14141, 1, 4, 11495, 11486, 'U'
14141, 1, 5, 11487, 11486, 'P'
```

```
14141, 1, 5, 11487, 28800, 'P'
14141, 1, 5, 11495, 28800, 'P'
14141, 1, 5, 11495, 11486, 'U'
14141, 1, 5, 11486, 11487, 'P'
14141, 1, 5, 28800, 11495, 'P'
14141, 1, 6, 11487, 28800, 'P'
14141, 1, 6, 11495, 28800, 'P'
14141, 1, 6, 11495, 11487, 'U'
14141, 1, 6, 11495, 11486, 'P'
14141, 1, 6, 11486, 11487, 'U'
14141, 1, 6, 28800, 11495, 'P'
8521, 2, 1, 9705, 9704, 'U'
8521, 2, 1, 9705, 9706, 'U'
8521, 2, 1, 9705, 12407, 'U'
8521, 2, 1, 9706, 9705, 'U'
8521, 2, 1, 9706, 12407, 'U'
8521, 2, 1, 9706, 9704, 'U'
8521, 2, 2, 12407, 9704, 'U'
8521, 2, 2, 12407, 9705, 'U'
8521, 2, 2, 12407, 9706, 'U'
8521, 2, 2, 9704, 12407, 'U'
8521, 2, 2, 9704, 9705, 'U'
8521, 2, 2, 9704, 9706, 'U'
8521, 3, 1, 9705, 9704, 'U'
8521, 3, 1, 9705, 9706, 'U'
8521, 3, 1, 9705, 12407, 'U'
8521, 3, 1, 9706, 9705, 'U'
8521, 3, 1, 9706, 12407, 'U'
8521, 3, 1, 9706, 9704, 'U'
8521, 3, 2, 12407, 9706, 'P'
8521, 3, 2, 9704, 9705, 'P'
8521, 3, 3, 12407, 9704, 'U'
8521, 3, 3, 12407, 9705, 'U'
8521, 3, 3, 12407, 9706, 'U'
8521, 3, 3, 9704, 12407, 'U'
8521, 3, 3, 9704, 9705, 'U'
8521, 3, 3, 9704, 9706, 'U'

Sample Timing Plan Table II
This is a sample timing plan table.
TIMESAMP2
Timing Plan
CREATE TABLE TIMESAMP2 (
  PLAN NUMBER(3),
  PHASE NUMBER(3),
  NEXTPHASES VARCHAR(20),
  GREENMIN FLOAT,
  GREENMAX FLOAT,
  GREENEXT FLOAT,
  YELLOW FLOAT,
  REDCLEAR FLOAT
);
PLAN, PHASE, NEXTPHASES, GREENMIN, GREENMAX, GREENEXT, YELLOW, REDCLEAR
1, 1, '2', 35.0, 0.0, 0.0, 4.0, 0.0
1, 2, '3', 5.0, 0.0, 0.0, 3.0, 0.0
1, 3, '4', 8.0, 0.0, 0.0, 3.0, 0.0
1, 4, '5', 32.0, 0.0, 0.0, 4.0, 0.0
1, 5, '6', 9.0, 0.0, 0.0, 3.0, 0.0
1, 6, '1', 1.0, 0.0, 0.0, 3.0, 0.0
2, 1, '2', 12.0, 30.0, 4.0, 3.0, 0.0
2, 2, '1', 10.0, 40.0, 4.0, 3.0, 0.0
3, 1, '2', 12.0, 30.0, 4.0, 3.0, 1.0
3, 2, '3', 4.0, 8.0, 2.0, 3.0, 0.0
3, 3, '1', 10.0, 20.0, 4.0, 3.0, 1.0
```

2. SampleNetwork2U.dat

```
Sample Unsignalized Node Table II-U
This is a sample unsignalized node table.
UNSISAMP2U
Unsignalized Node
CREATE TABLE UNSISAMP2U (
  NODE NUMBER(10),
  INLINK NUMBER(10),
  SIGN CHAR(1)
```

```
);
NODE, INLINK, SIGN
8520, 12384, 'N'
8520, 28800, 'N'
14136, 12407, 'S'
14136, 12384, 'N'
14136, 28804, 'N'
14141, 11487, 'S'
14141, 11486, 'S'
14141, 11495, 'S'
14141, 28800, 'S'
8521, 12407, 'S'
8521, 9705, 'S'
8521, 9704, 'S'
8521, 9706, 'S'
```