TRANSPORTATION ANALYSIS SIMULATION SYSTEM (TRANSIMS)

Version: TRANSIMS-LANL-1.0

VOLUME 2 – SOFTWARE
PART 5 – LIBRARIES

28 May 1999

LA-UR 99-2578

COPYRIGHT, 1999, THE REGENTS OF THE UNIVERSITY OF CALIFORNIA. THIS SOFTWARE WAS PRODUCED UNDER A U.S. GOVERNMENT CONTRACT (W-7405-ENG-36) BY LOS ALAMOS NATIONAL LABORATORY, WHICH IS OPERATED BY THE UNIVERSITY OF CALIFORNIA FOR THE U.S. DEPARTMENT OF ENERGY. THE U.S. GOVERNMENT IS LICENSED TO USE, REPRODUCE, AND DISTRIBUTE THIS SOFTWARE. NEITHER THE GOVERNMENT NOR THE UNIVERSITY MAKES ANY WARRANTY, EXPRESS OR IMPLIED, OR ASSUMES ANY LIABILITY OR RESPONSIBILITY FOR THE USE OF THIS SOFTWARE.

.

TRANSIMS

Version: TRANSIMS-LANL-1.0

VOLUME 2 – SOFTWARE
PART 5 – LIBRARIES

28 May 1999

LA-UR-99-2578

The following persons contributed to this document:
C. L. Barrett*
R. J. Beckman*
K. P. Berkbigler*
K. R. Bisset*
B. W. Bush*
S. Eubank*
J. M. Hurford*
G. Konjevod*
D. A. Kubicek*
M. V. Marathe*
J. D. Morgeson*
M. Rickert*
P. R. Romero*
L. L. Smith*
M. P. Speckman**
P. L. Speckman**
P. E. Stretz*
G. L. Thayer*
M. D. Williams*

* Los Alamos National Laboratory, Los Alamos, NM 87545
** National Institute of Statistical Sciences, Research Triangle Park, NC

Acknowledgments

This work was supported by the U. S. Department of Transportation (Assistant Secretary for Transportation Policy, Federal Highway Administration, Federal Transit Administration), the U. S. Environmental Protection Agency, and the U. S. Department of Energy as part of the Travel Model Improvement Program.

CONTENTS

1. NETWORK SUBSYSTEM 7

1.1 OVERVIEW 7
1.2 CLASSES AND STRUCTURES 7
1.3 IMPLEMENTATION 75
1.4 EXAMPLES 76
1.5 FUTURE WORK 77
1.6 REFERENCES 77

2. OUTPUT SUBSYSTEM 78

2.1 OVERVIEW 78
2.2 DESIGN 80
2.3 IMPLEMENTATION 119
2.4 INTEGRATION INTO THE MICROSIMULATION 120
2.5 USAGE 121
2.6 FUTURE WORK 122
2.7 REFERENCES 122

3. PARALLEL TOOLBOX 123

3.1 CONFIGURATION GROUP MPI 123
3.2 CONFIGURATION GROUP PVM 123
3.3 CONFIGURATION GROUP PARALLEL 123
3.4 TENCODINGMODE 124
3.5 TTOPOLOGY TYPE 124
3.6 TCPNID 124
3.7 THOSTID 124
3.8 TMESSAGEID 124
3.9 TMESSAGETAG 125
3.10 TMESSAGETYPE 125
3.11 TMESSAGEFLAGS 125
3.12 TARCHCODE 125
3.13 NULL_EVENT_ID 125
3.14 MSG_FLAG_EVENT 125
3.15 MSG_FLAG_AUTO 125
3.16 MSG_FLAG_DELETE 126
3.17 TBASEMESSAGE 126
3.18 TBUSTOPOLOGY 128
3.19 TCPNINFO 128
3.20 TCOMMUNICATION 128
3.21 TCOMMUNICATIONCONTROL 130
3.22 DEST_ALL 131
3.23 DEST_ALL_BUT_MYSELF 131
3.24 TDESTINATION 131
3.25 TEVENTHANDLERPRIMITIVE 132
3.26 THOSTINFO 132
3.27 TMPIBUFFER 132

3.28 TMPIMESSAGE 132
3.29 TMPIMESSAGECONTROL 133
3.30 TMPIPARALLELCONTROL 133
3.31 TMESSAGERECIPIENT 133
3.32 TMESSAGESELECTOR 133
3.33 TPVMMESSAGE 134
3.34 TPVMMESSAGECONTROL 134
3.35 TPVMPARALLELCONTROL 135
3.36 TPARALLEL 135
3.37 TPARALLELCONTROL 136
3.38 CONFIGURATION GROUP PARALLELDEMO 136
3.39 TPARALLELDEMO 136
3.40 TPROFILEEVENT 138
3.41 TPROFILE 138
3.42 TROMIOBUFFERINFO 138
3.43 TOPOLOGY_MY_RANK 138
3.44 TTOPOLOGY 138

1. NETWORK SUBSYSTEM

1.1 Overview

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.

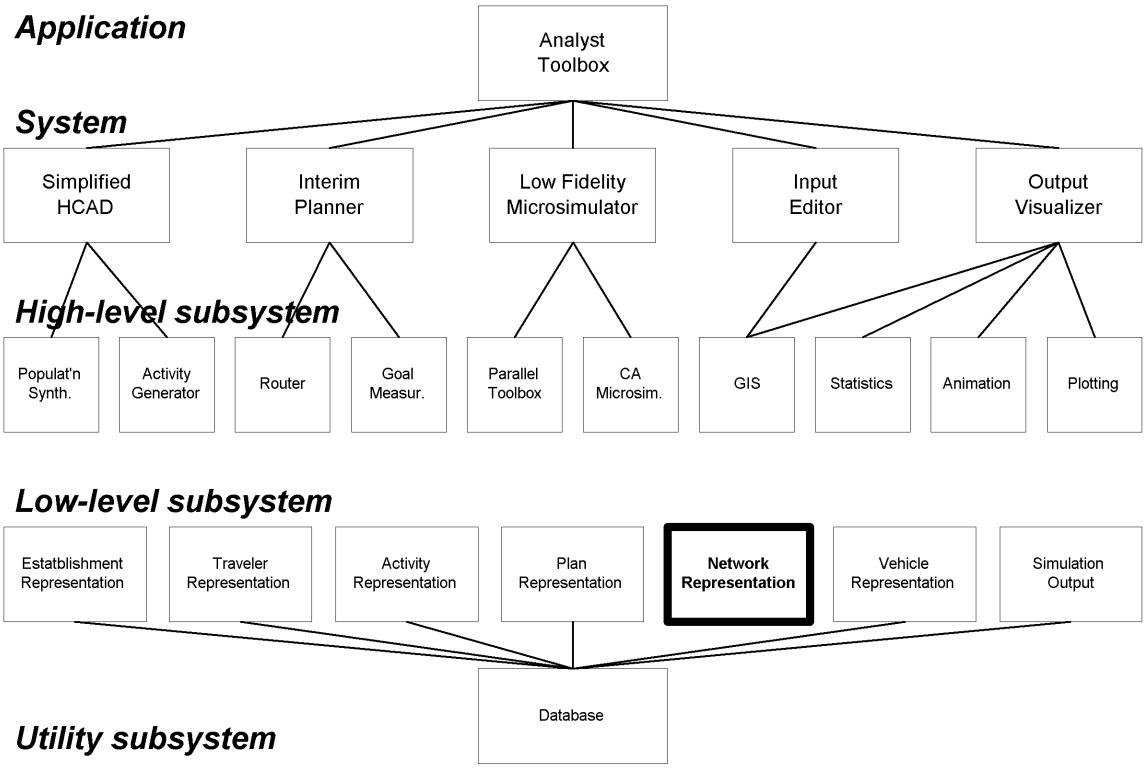

Figure 1: Location of the network subsystem in the TRANSIMS software architecture.

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. Transit (e.g., bus and light rail) stops and stations may also be present.

1.2 Classes and Structures

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 show the relationships between the classes.

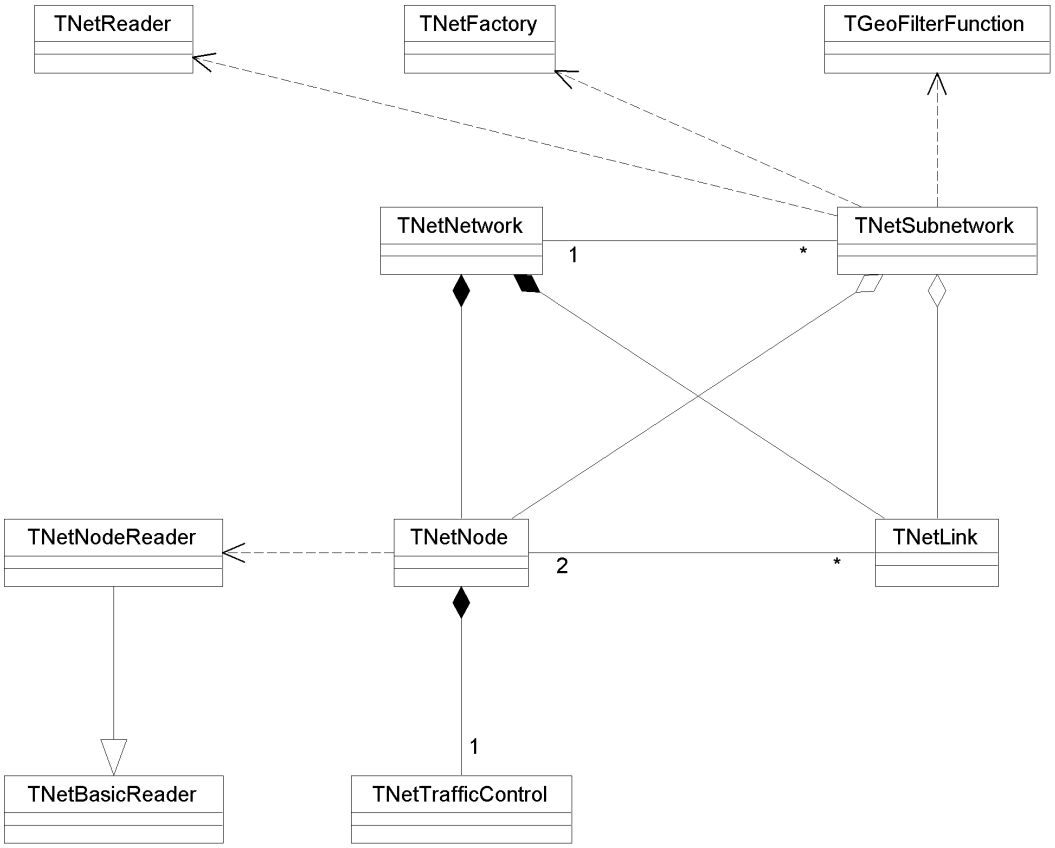

Figure 2: Class diagram for the node-related classes in the TRANSIMS network representation subsystem (UML notation).

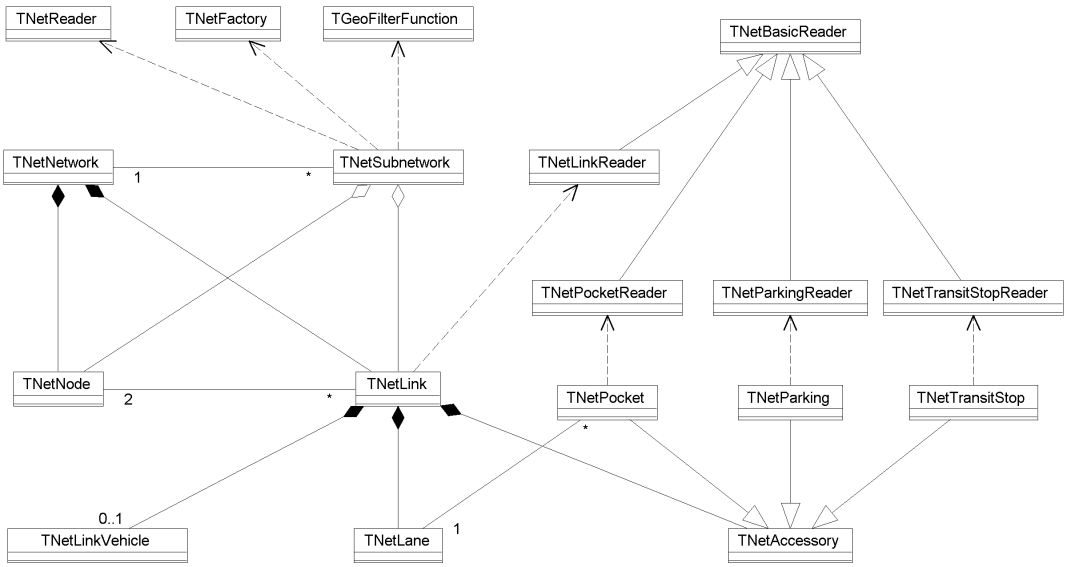

Figure 3: Class diagram for the link-related classes in the TRANSIMS network representation subsystem (UML notation).

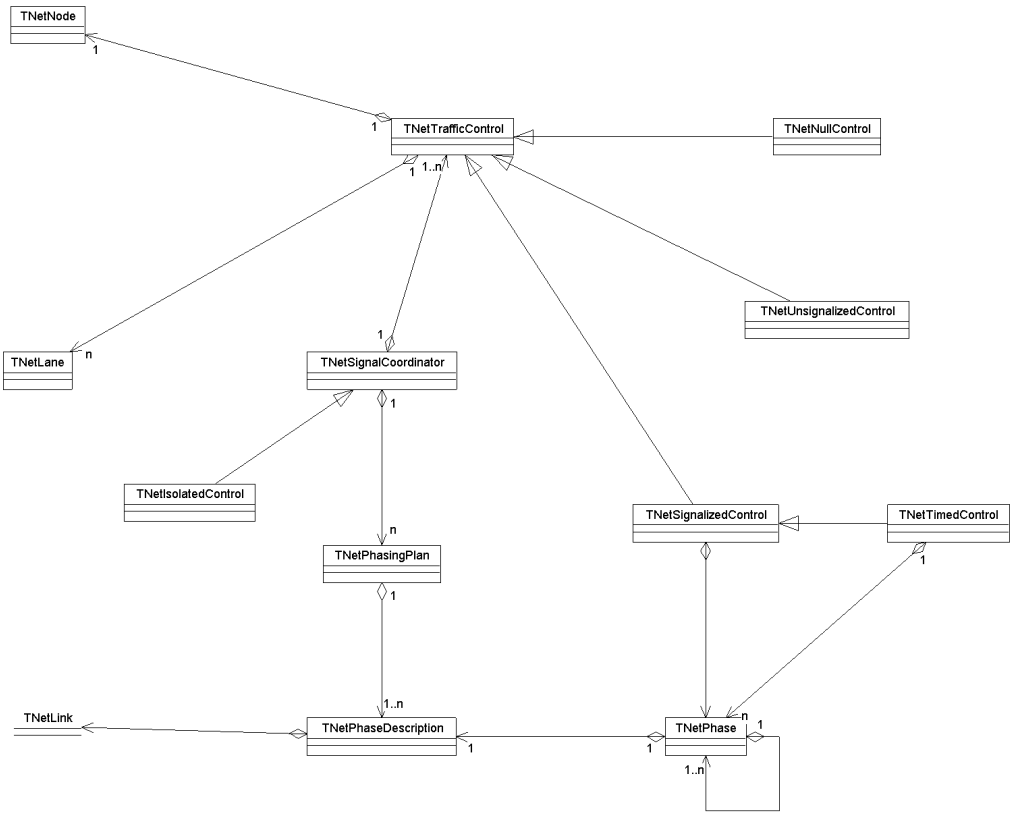

Figure 4: Class diagram for the control-related classes in the TRANSIMS network representation subsystem (UML notation).

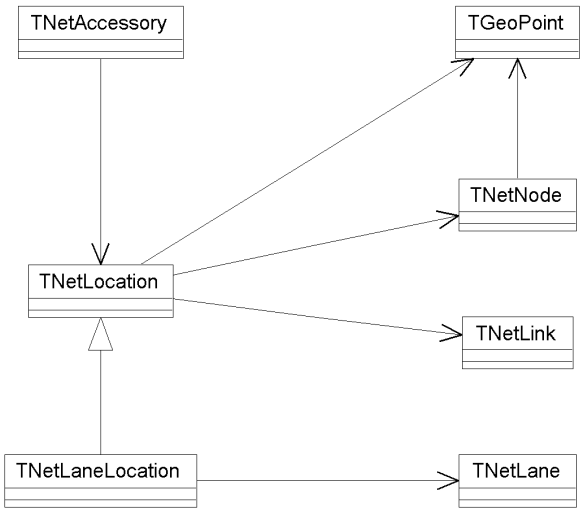

Figure 5: Class diagram for the location-related classes in the TRANSIMS network representation subsystem (UML notation).

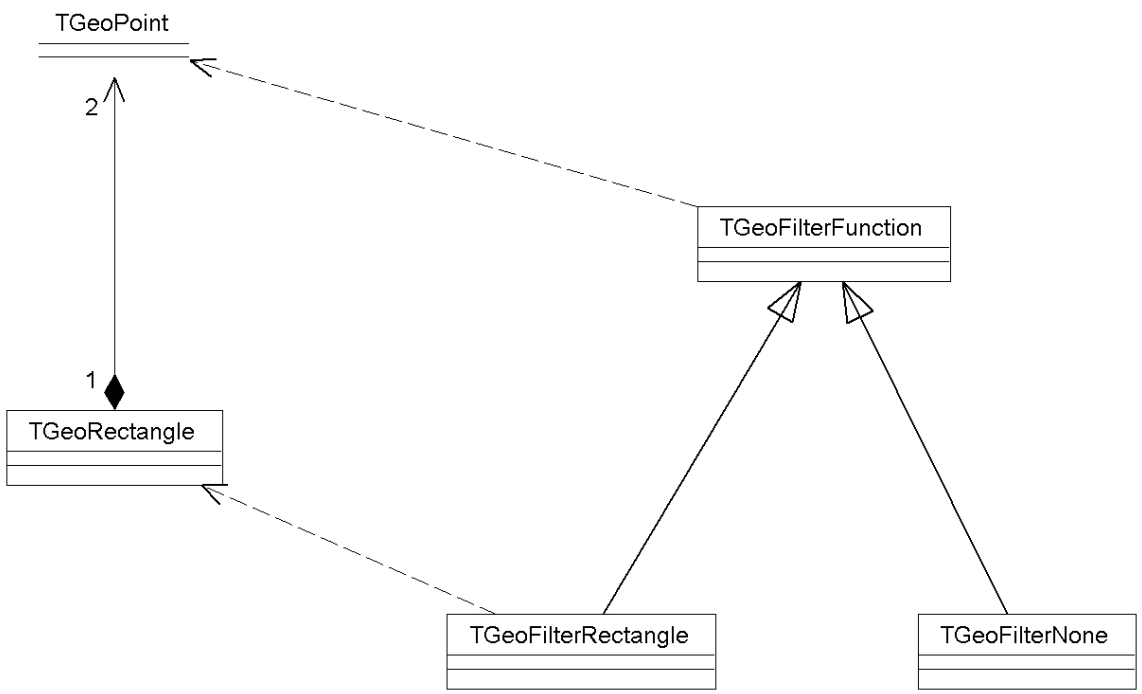

Figure 6: Class diagram for the geography-related classes in the TRANSIMS network representation subsystem (UML notation).

1.2.1 TNetAccessory

An accessory is where something happens along a link.

Public access:

```
enum EType {kPocket, kActivityLocation, kParking,
      kTransitStop, kDetector, kBarrier}
```
There are several types of accessories.

```
typedef vector<TNetAccessory*> Collection
typedef Collection::iterator CollectionIterator
typedef Collection::const_iterator CollectionConstIterator
```
Type definitions.

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

```
virtual ~TNetAccessory()
```
Destroy the accessory.

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

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

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

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

Private access:

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

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

```
NetAccessoryId fId
```
Each accessory has a unique id.

```
EType fType
```
Each accessory has a type.

```
TNetLocation* fLocation
```
Each accessory has a location.

1.2.2 TNetActivityLocation

An activity location is a source or sink of travelers along a link.

Public access:

```
typedef map<NetAccessoryId, TNetActivityLocation*,
      less<NetAccessoryId> > Map
typedef Map::iterator MapIterator
typedef Map::const_iterator MapConstIterator
edef map<const string, REAL, less<string> > UserDataMap
typedef UserDataMap::iterator UserDataMapIterator
typedef UserDataMap::const_iterator UserDataMapConstIterator
```
Type definitions.

```
TNetActivityLocation(TNetActivityLocationReader& reader)
```
Construct the activity location from the specified reader.

```
inline EVehType GetLayer() const
```
Return the modal “layer” on which the activity location lies.

```
inline const TGeoPoint& GetGeographicPosition() const
```
Return the geographic position of the activity location.

Private access:

`EVehType fLayer`
Each activity location is associated with a layer.

`TGeoPoint fPosition`
Each activity location has a geographic position.

```
Return the modal "layer" on which the activity location
      lies. ne EVehType TNetActivityLocation::GetLayer()
      const ne EVehType TNetActivityLocation::GetLayer()
      const return fLayer
```
Each activity location has a geographic position.

1.2.3 TNetActivityLocationReader

An activity location reader reads activity location values from the database.

Public access:

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

`void GetNextRecord()`
Get the next record in the table.

`NetAccessoryId GetId() const`
Return the id for the current accessory. A `TNetIllegalValue` exception is thrown if the data is not valid.

```
void GetLocation(NetLinkId& link, NetNodeId& node, REAL&
      offset) const
```
Return the location of the current accessory. A `TNetIllegalValue` exception is thrown if the data is not valid.

`EVehType GetLayer() const`
Return the layer of the current activty location. A `TNetIllegalValue` exception is thrown if the data is not valid.

`TGeoPoint GetGeographicPosition() const`
Return the position of the current activity location. A `TNetIllegalValue` exception is thrown if the data is not valid.

`string GetNotes() const`
Return the notes.

1.2.4 TNetBarrierReader

A barrier reader reads barrier values from the database.

Public access:

```
enum EStyle {kCurb, kBarrier, kGradeSeparation, kStripe,
      kTemporary}
```
There are several barrier styles.

```
TNetBarrierReader(TNetReader& reader)
```
Construct a barrier reader for a given network.

```
void GetNextRecord()
```
Get the next record in the table.

```
NetAccessoryId GetId() const
```
Return the id for the current accessory. A `TNetIllegalValue` exception is thrown if the data is not valid.

```
void GetLocation(NetLinkId& link, NetNodeId& node, REAL&
      offset) const
```
Return the location of the current accessory. A `TNetIllegalValue` exception is thrown if the data is not valid.

```
NetLaneNumber GetLane() const
```
Return the lane number to the left of the barrier. A `TNetIllegalValue` exception is thrown if the data is not valid.

```
EStyle GetStyle() const
```
Return the style of the current barrier. A `TNetIllegalValue` exception is thrown if the data is not valid.

```
REAL GetLength() const
```
Return the length of the current barrier. A `TNetIllegalValue` exception is thrown if the data is not valid.

```
string GetNotes() const
```
Return the notes.

1.2.5 TNetBasicReader

A basic reader is the base class for readers accessing values from the database.

Public access:

`virtual ~TNetBasicReader()`
Destroy a basic reader.

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

`virtual void GetNextRecord()`
Get the next record in the table.

`inline bool MoreRecords() const`
Return whether there are any more records in the table.

`virtual string GetNotes() const = 0`
Return the notes.

Protected access:

`TNetBasicReader(const string& name, const string& table)`
Construct a reader for a given table.

```
void ReportIllegalValue(const string& field, const string&
      value) const
void ReportIllegalValue(const string& field, int value)
      const
```
Report an illegal value in a data field.

`const string fName`
Each reader has a file name.

`FILE* fFile`
Each reader has a file handle.

`long fLength`
Each file has a length.

`bool fAtEnd`
Each file has an end indicator.

1.2.6 TNetDetectorReader

A detector reader reads detector values from the database.

Public access:

`enum EStyle {kPresence, kPassage}`
There are several detector styles.

`TNetDetectorReader(TNetReader& reader)`
Construct a detector reader for a given network.

`void GetNextRecord()`
Get the next record in the table.

`NetAccessoryId GetId() const`
Return the id for the current accessory. A `TNetIllegalValue` exception is thrown if the data is not valid.

```
void GetLocation(NetLinkId& link, NetNodeId& node, REAL&
      offset) const
```
Return the location of the current accessory. A `TNetIllegalValue` exception is thrown if the data is not valid.

`NetLaneNumber GetBeginLane() const`
Return the lane number for the beginning of the detector. A `TNetIllegalValue` exception is thrown if the data is not valid.

`NetLaneNumber GetEndLane() const`
Return the lane number for the ending of the detector. A `TNetIllegalValue` exception is thrown if the data is not valid.

`REAL GetLength() const`
Return the length of the current detector. A `TNetIllegalValue` exception is thrown if the data is not valid.

`EStyle GetStyle() const`
Return the style of the current detector. A `TNetIllegalValue` exception is thrown if the data is not valid.

`NetCoordinatorIdSet GetCoordinators() const`
Return the coordinators for the detector. A `TNetIllegalValue` exception is thrown if the data is not valid.

`string GetNotes() const`
Return the notes.

1.2.7 TNetFactory

A network factory allocates and constructs new network objects.

Public access:

`TNetFactory()`
Construct a factory

`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 TNetTransitStop*
      NewTransitStop(TNetTransitStopReader& reader)
```
Return a new transit stop from the specified reader.

```
virtual TNetActivityLocation*
      NewActivityLocation(TNetActivityLocationReader&
      reader)
```
Return a new activity location from the specified reader.

```
virtual TNetProcessLink*
      NewProcessLink(TNetProcessLinkReader& reader)
```
Return a new process link 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.

`inline bool StoreNames() const;`
Return whether the link and transit stop names should be stored.

`inline bool StoreTrafficControls() const;`
Return whether the traffic controls should be stored.

`inline bool StoreActivityUserData() const;`
Return whether the activity location user data should be stored.

`inline void SetStoreNames(bool b);`
Define whether the link and transit stop names should be stored.

`inline void SetStoreTrafficControls(bool b);`
Define whether the traffic controls should be stored.

`inline void SetStoreActivityUserData(bool b);`
Define whether the activity location user data should be stored.

Private access:

`bool fStoreNames`
Factory keeps track of whether to store link and transit stop names. Default is false.

`bool fStoreTrafficControls`
Factory keeps track of whether to create and store traffic controls. Default is true.

`bool fStoreActivityUserData`
Factory keeps track of whether to store activity location user data. Default is false.

1.2.8 TGeoFilterFunction

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

Public access:

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

1.2.9 TGeoFilterNone

This geographic filter accepts all points.

Public access:

`TGeoFilterNone()`
Construct an instance.

`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.

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

1.2.10 TGeoFilterRectangle

This geographic filter accepts all points within a rectangle.

Public access:

`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.

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

Private access:

1.2.11 TGeoRectangle fRectangle

Each rectangular filter has a rectangle.

1.2.12 TNetIsolatedControl

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

Public access:

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

```
~TNetIsolatedControl()
```
Destroy an isolated control.

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

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

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

1.2.13 TNetLane

A lane is where traffic flows on a link.

Public access:

```
typedef vector<TNetLane*> Collection
typedef Collection::iterator CollectionIterator
typedef Collection::const_iterator CollectionConstIterator
```
Type definitions.

```
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.

```
inline TNetLink& GetLink()
inline 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 ending node for the lane.

`inline TNetPocket::Collection& GetPockets()`
`inline const TNetPocket::Collection& 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.

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

`VehTypeSet& GetVehicleTypes()`
`const VehTypeSet& GetVehicleTypes() const`
Return the vehicle types allowed on the lane. (All vehicle types allowed for the link are also allowed for the lane if this set is empty.) Note that it is preferable to use the TNetLane::IsAllowed member function instead of this one whenever possible.

`bool IsAllowed(EVehType veh) const`
Return whether a particular vehicle type is allowed on the lane.

`inline bool IsHOV(int n) const`
Return whether the lane is an HOV lane for the specified number of occupants.

`inline void SetOccupantMin(UINT8 n)`
Set the minimum number of occupants required to use the lane.

`inline UINT8 GetOccupantMin() const`
Return the minimum number of occupants required to use the lane.

`inline static REAL GetDefaultWidth()`
Return the default lane width.

`inline static void SetDefaultWidth(REAL width)`
Define the default lane width.

Private access:

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

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

`TNetLink* fLink`
Each lane belongs on a link.

`NetNodeId fNode`
Each lane has a starting node.

`NetLaneNumber fNumber`
Each lane has a number.

`TNetPocket::Collection fPockets`
Each lane has zero or more pockets.

`VehTypeSet fVehicleTypes`
Each restricted lane has a set of vehicle types allowed to use the lane.

`UINT8 fOccupantMin`
Each lane has a minimum occupancy for vehicles.

`static REAL fgDefaultWidth`
Default lane width.

1.2.14 TNetLaneConnectivityReader

This reader reads lane connectivity values from the database.

Public access:

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

`void GetNextRecord()`
Get the next record in the table.

`NetNodeId GetNode() const`
Return the id for the current node. A `TNetIllegalValue` exception is thrown if the data is not valid.

`NetLinkId GetInlink() const`
Return the id for incoming link. A `TNetIllegalValue` exception is thrown if the data is not valid.

`NetLaneNumber GetInlane() const`
Return the id for incoming lane. A `TNetIllegalValue` exception is thrown if the data is not valid.

`NetLinkId GetOutlink() const`
Return the id for outgoing link. A `TNetIllegalValue` exception is thrown if the data is not valid.

`NetLaneNumber GetOutlane() const`
Return the id for outgoing lane. A `TNetIllegalValue` exception is thrown if the data is not valid.

`string GetNotes() const`
Return the notes.

1.2.15 TNetLaneLocation

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

Public access:

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

```
TNetLaneLocation(TNetLink& link, TNetNode& node, REAL
      offset,  NetLaneNumber n)
```
Construct a location along the link, with the given offset from its end node and in the specified lane.

```
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.

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

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

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

Private access:

```
TNetLane* fLane
```
Each lane location is on a lane.

1.2.16 TNetLaneUseReader

A lane use reader reads lane use values from the database.

Public access:

`enum ERestriction {kOnly, kRequired, kForbidden}`
There are several restriction types.

`TNetLaneUseReader(TNetReader& reader)`
Construct a lane use reader for a given network.

`void GetNextRecord()`
Get the next record in the table.

`NetLinkId GetLink() const`
Return the link id for the link the lane is on. A `TNetIllegalValue` exception is thrown if the data is not valid.

`NetNodeId GetNode() const`
Return the node toward which the lane is headed. A `TNetIllegalValue` exception is thrown if the data is not valid.

`NetLaneNumber GetLane() const`
Return the lane number for the current lane. A `TNetIllegalValue` exception is thrown if the data is not valid.

`VehTypeSet GetVehicleTypes() const`
Return the vehicle types to which the restriction applies. A `TNetIllegalValue` exception is thrown if the data is not valid.

`ERestriction GetRestriction() const`
Return the lane restriction indicator. A `TNetIllegalValue` exception is thrown if the data is not valid.

`string GetNotes() const`
Return the notes.

1.2.17 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).

```
Allow TNetLinkVehicleSpeeds access to GetNodeIndex() iend
      class TNetLinkVehicleSpeeds
```

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).

Public access:

```
enum EFunctionalClass {kFreeway, kExpressway, kPrimary,
      kSecondary, kFrontage, kCollector, kLocal, kRamp,
      kZonal, kOther, kWalkway, kBikeway, kBusway,
      kLightRail, kHeavyRail, kFerry}
```

There are several functional classes for a link.

```
typedef map<NetLinkId, TNetLink*, less<NetLinkId> > Map
typedef Map::iterator MapIterator
typedef Map::const_iterator MapConstIterator
typedef ring< vector<TNetLink*> > Ring
typedef Ring::iterator RingIterator
typedef Ring::const_iterator RingConstIterator
typedef set<TNetLink*, TLess> Set
typedef Set::iterator SetIterator
typedef Set::const_iterator SetConstIterator
```

Type definitions.

```
TNetLink(TNetLinkReader& reader)
```

Construct a link using the reader.

```
TNetLink(NetLinkId id)
```

Construct a dummy link with the specified id.

```
virtual ~TNetLink()
```

Destroy a link.

```
inline NetLinkId GetId() const
```

Return the id of the link.

```
inline void GetNodes(NetNodeId& nodeA, NetNodeId& nodeB)
      const
inline void GetNodes(TNetNode*& nodeA, TNetNode*& nodeB)
      const
inline 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.

```
inline TNetAccessory::Collection& GetAccessories()
inline const TNetAccessory::Collection& GetAccessories()
      const
```
Return the accessories on the link.

```
TNetAccessory::Collection GetAccessoriesTowards(const
      TNetNode& node, TNetAccessory::EType type) const
```
Return the accessories of the specified type on the side of the link heading toward the specified node. A `TNetNotFound` exception is thrown if the node is not at one of the link's ends.

```
inline TNetLane::Collection& GetLanesFrom(const TNetNode&
      node, bool pockets = false)
inline const TNetLane::Collection& GetLanesFrom(const
      TNetNode& node, bool pockets = false) const
```
Return the lanes going away from the specified node. The lanes are ordered 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.

```
inline TNetLane::Collection& GetLanesTowards(const TNetNode&
      node, bool pockets = false)
inline const TNetLane::Collection& GetLanesTowards(const
      TNetNode& node, bool pockets = false) const
```
Return the lanes going toward the specified node. The lanes are ordered 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.

```
inline REAL GetLength(bool setback = false) const
```
Return the length of the link in meters. The length is measured from one node to the other, unless setback distance subtraction is requested.

```
inline REAL GetGradeTowards(const TNetNode& node) const
```
Return the percent grade of the link heading toward the given node, with uphill being a positive number. A `TNetNotFound` exception is thrown if the node is not at one of the link's ends.

```
inline REAL GetSetback(const TNetNode& node) const
```
Return the setback distance in meters at the specified node. A `TNetNotFound` exception is thrown if the node is not at one of the link's ends.

```
inline REAL GetCapacityTowards(const TNetNode& node) const
```
Return the capacity in vehicles per hour for lanes heading toward the specified node. A `TNetNotFound` exception is thrown if the node is not at one of the link's ends.

```
TNetLink& GetThroughLink(const TNetNode& node)
const TNetLink& 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.

`inline 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.

```
inline REAL GetFreeFlowSpeedTowards(const TNetNode& node)
      const
```
Return the free flow speed of the vehicles 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, EVehType
      veh) const
```
Return the speed limit for a particular vehicle type 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(const TNetNode& node, EVehType
      veh) const
```
Return the free flow speed for a particular vehicle type 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.

`inline EFunctionalClass GetFunctionalClass() const`
Return the functional class for the link.

`inline 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.

```
TNetLink& GetLinkFromTurn(const TNetNode& node,
      ENetTurnLabel turn)
const TNetLink& GetLinkFromTurn(const TNetNode& node,
      ENetTurnLabel turn) const
```
Return the link for the specified turn. A `TNetNotFound` exception is thrown if there is no link for the specified turn.

`ENetTurnLabel GetTurnForLink(const TNetLink& link) const`
Return the turn number for the specified link. A `TNetNotFound` exception is thrown if the specified link is not adjacent to this link.

```
ENetTurnLabel GetTurnForLink(const TNetLink& link, const
      TNetNode& node) const
```
Return the turn number for the specified link at the specified node. A `TNetNotFound` exception is thrown if the specified link is not adjacent to this link.

`inline NetColor GetColor() const`
Return the color of the link.

`inline VehTypeSet& GetVehicleTypes()`
`inline const VehTypeSet& GetVehicleTypes() const`
Return the vehicle types allowed on the link.

`inline bool IsAllowed(EVehType veh) const`
Return whether a particular vehicle type is allowed on the link.

`inline const string& GetName() const`
Return the name of the link.

`void SetName(string name)`
Define the name of the link.

`inline TNetLinkVehicleSpeeds* GetLinkVehicleSpeedsPtr()`
Return the TNetLinkVehicleSpeeds object pointer.

```
inline void SetLinkVehicleSpeedsPtr(TNetLinkVehicleSpeeds*
      p)
```
Define the TNetLinkVehicleSpeeds object pointer.

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

`inline bool operator!=(const TNetLink& link) const`
Return whether the link has a different id from the given link.

Protected access:

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

Private access:

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

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

```
ENetTurnLabel GetTurnForLink(const TNetLink& link, size_t
      iNode) const
```
Return the turn number for the specified link at the specified node. A `TNetNotFound` exception is thrown if the specified link is not adjacent to this link.

`NetLinkId fId`
Each link has a unique id.

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

`TNetAccessory::Collection fAccessories`
Each link has zero or more accessories.

`TNetLane::Collection fLanes[2]`
Each link has zero or more lanes on each side.

`TNetLane::Collection fPermanentLanes[2]`
Each link has zero or more permanent lanes on each side.

`REAL fLength`
Each link has a length.

`REAL fGrade`
Each link has a grade.

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

`REAL fCapacities[2]`
Each link has capacities (vehicles/hour) in either direction.

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

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

`REAL fFreeSpeeds[2]`
Each link has free flow speed in either direction.

`EFunctionalClass fFunctionalClass`
Each link has a functional class.

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

`NetColor fColor`
Each link has a color.

`VehTypeSet fVehicleTypes`
Each link has a set of vehicle types allowed to use the link.

`string* fName`
Each link may have a name.

`TNetLinkVehicleSpeeds* fVehicleSpeeds`
Each link has a pointer to a network object that contains the speed information when speeds on the link differ by vehicle type.

`static const string fNoName`
Default name when name is not defined.

1.2.18 TNetLinkReader

A link reader reads link values from the database.

Public access:

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

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

`NetLinkId GetId() const`
Return the id for the current link. A `TNetIllegalValue` exception is thrown if the data is not valid.

`string GetName() const`
Return the name of the link.

`void GetNodeIds(NetNodeId& nodeA, NetNodeId& nodeB) const`
Return the node ids at the ends of the current link. A `TNetIllegalValue` exception is thrown if the data is not valid.

`UINT8 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. A `TNetIllegalValue` exception is thrown if the data is not valid.

`UINT8 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. A `TNetIllegalValue` exception is thrown if the data is not valid.

`UINT8 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. A `TNetIllegalValue` exception is thrown if the data is not valid.

`bool HasTwoWayLeftTurnLane() const`
Return whether there is a two-way left turn lane on the current link. A `TNetIllegalValue` exception is thrown if the data is not valid.

`REAL GetLength() const`
Return the length of the current link. A `TNetIllegalValue` exception is thrown if the data is not valid.

`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. A `TNetIllegalValue` exception is thrown if the data is not valide.

`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. A `TNetIllegalValue` exception is thrown if the data is not valid.

`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. A `TNetIllegalValue` exception is thrown if the data is not valid.

`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. A `TNetIllegalValue` exception is thrown if the data is not valid.

`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. A `TNetIllegalValue` exception is thrown if the data is not valid.

`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. A `TNetIllegalValue` exception is thrown if the data is not valid.

`TNetLink::EFunctionalClass GetFunctionalClass() const`
Return the functional class for the link. A `TNetIllegalValue` exception is thrown if the data is not valid.

`NetColor GetColor() const`
Return the color of the link. A `TNetIllegalValue` exception is thrown if the data is not valid.

```
VehTypeSet GetVehicles() const
```
Return the vehicles allowed on the link. A `TNetIllegalValue` exception is thrown if the data is not valid.

```
string GetNotes() const
```
Return the notes.

Private access:

```
void ReportInvalidNode(NetNodeId id) const
```
Report that an invalid node id has been passed as an argument.

1.2.19 TNetLinkVehicleSpeeds

On some links, the vehicle speeds and limits may vary by vehicle type. This auxilliary class contains the per vehicle type speed information. Values are normally retrieved through the Link class which uses this class.

Public access:

```
typedef map<EVehType, REAL, less<EVehType> > VehSpeedMap
typedef VehSpeedMap::iterator VehSpeedMapIterator
typedef VehSpeedMap::const_iterator VehSpeedMapConstIterator
```
Type definitions.

```
TNetLinkVehicleSpeeds(TNetSpeedReader& reader, TNetNetwork&
      network)
```
Construct link speeds using the reader.

```
~TNetLinkVehicleSpeeds()
```
Destroy link speeds.

```
REAL GetSpeedLimitTowards(const TNetNode& node, EVehType
      veh) const
```
Return the speed limit for a particular vehicle type heading toward the given node. A TNetNodeNotFound exception is thrown if the node is not at one of the link's ends.

```
REAL GetFreeFlowSpeedTowards(const TNetNode& node, EVehType
      veh) const
```
Return the free speed for a particular vehicle type heading toward the given node. A TNetNodeNotFound exception is thrown if the node is not at one of the link's ends.

```
void SetSpeedLimitTowards(const TNetNode& node, EVehType
      veh, REAL speed)
```
Define the speed limit for a particular vehicle type heading toward the given node. A TNetNodeNotFound exception is thrown if the node is not at one of the link's ends.

```
void SetFreeFlowSpeedTowards(const TNetNode& node, EVehType
      veh, REAL speed)
```
Define the free speed for a particular vehicle type heading toward the given node. A TNetNodeNotFound exception is thrown if the node is not at one of the link's ends.

Private access:

`VehSpeedMap fVehSpeedLimits[2]`
Each link may have speed limits per vehicle type.

`VehSpeedMap fVehFreeSpeeds[2]`
Each link may have free speeds per vehicle type.

`TNetLink* fLink`
Object knows its associated link

1.2.20 TNetLocation

A location is a point along a link.

Public access:

`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.

`virtual ~TNetLocation() {`
Destroy an instance.

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

```
inline TNetLink& GetLink()
inline const TNetLink& GetLink() const
```
Return the link on which the location lies.

```
inline TNetNode& GetNode()
inline const TNetNode& GetNode() const
```
Return the node toward which vehicles travel.

`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.

`virtual TGeoPoint GetGeographicPosition() const`
Return the geographic position of the location.

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

Private access:

`void ReportInvalidNode(NetNodeId id) const`
Report that an invalid node id has been passed as an argument.

`TNetLink* fLink`
Each location is on a link.

`TNetNode* fNode`
Each location is measured from the node toward which vehicles travel. The value of fNode thus specifies the side of the link that the location is on.

`REAL fOffset`
Each location is specified as an offset relative to fNode.

1.2.21 TNetNetwork

A network represents all of the network database that has been instantiated.

Public access:

`TNetNetwork()`
Construct a network.

`~TNetNetwork()`
`Destroy a network.`

```
inline TNetNode::Map& GetNodes()
inline const TNetNode::Map& GetNodes() const
```
Return the set of nodes in the network.

```
inline TNetLink::Map& GetLinks()
inline const TNetLink::Map& GetLinks() const
```
Return the set of links in the network.

```
inline TNetParking::Map& GetParkings()
inline const TNetParking::Map& GetParkings() const
```
Return the set of parking places in the network.

```
inline TNetTransitStop::Map& GetTransitStops()
inline const TNetTransitStop::Map& GetTransitStops() const
```
Return the set of transit stops in the network.

```
inline TNetActivityLocation::Map& GetActivityLocations()
inline const TNetActivityLocation::Map&
      GetActivityLocations() const
```
Return the set of activity locations in the network.

```
inline TNetProcessLink::Map& GetProcessLinks()
inline const TNetProcessLink::Map& GetProcessLinks() const
```
Return the set of virtual (process) links in the network.

```
inline TNetSignalCoordinator::Map& GetSignalCoordinators()
inline const TNetSignalCoordinator::Map&
      GetSignalCoordinators() const
```
Return the set of signal coordinators in the network.

Private access:

`TNetNetwork(const TNetNetwork&) {`
Do not allow networks to be copied.

`TNetNetwork& operator=(const TNetNetwork&) {return *this;`
Do not allow networks to be assigned.

`TNetNode::Map fNodes`
Each network manages/owns the nodes it contains.

`TNetLink::Map fLinks`
Each network manages/owns the links it contains.

`TNetParking::Map fParkings`
Each network has a map for parking places.

`TNetTransitStop::Map fTransitStops`
Each network has a map for transit stops.

`TNetActivityLocation::Map fActivityLocations`
Each network has a map for activity locations.

`TNetProcessLink::Map fProcessLinks`
Each network has a map for process links;

`TNetSignalCoordinator::Map fCoordinators`
Each network manages/owns the signal coordinators it contains.

1.2.22 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.)

Public access:

```
typedef map<NetNodeId, TNetNode*, less<NetNodeId> > Map
typedef Map::iterator MapIterator
typedef Map::const_iterator MapConstIterator
typedef set<TNetNode*, TLess> Set
typedef Set::iterator SetIterator
typedef Set::const_iterator SetConstIterator
```
Type definitions.

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

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

`virtual ~TNetNode()`
Destroy a node.

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

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

```
inline TNetLink::Ring& GetLinks()
inline const TNetLink::Ring& GetLinks() const
```
Return the ring of links in order.

```
TNetLink& GetLink(NetColor color)
const TNetLink& GetLink(NetColor color) const
```
Return the link of the specified color, if any. A `TNetNotFound` exception is thrown if there is no link of the specified color.

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

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

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

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

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

Private access:

`TNetNode(const TNetNode&) {`
Do not allow nodes to be copied.

`TNetNode& operator=(const TNetNode&) {return *this;`
Do not allows nodes to be assigned.

`NetNodeId fId`
Each node has a unique id.

`TGeoPoint fPosition`
Each node has a geographic position.

`TNetLink::Ring fLinks`
Each node is connected to links.

`TNetTrafficControl* fTrafficControl`
Each node has an associated traffic control

1.2.23 TNetNodeLink

Node-link data type.

Public access:

```
typedef map<TNetNodeLink, NetLinkId, less<TNetNodeLink> >
      LinkMap
typedef LinkMap::iterator LinkMapIterator
typedef LinkMap::const_iterator LinkMapConstIterator
```
Type defintions.

```
TNetNodeLink(NetNodeId node, NetLinkId link) : fNode(node),
      fLink(link) {
```
Construct an instance.

`NetNodeId fNode`
Each node-link combination has a node id.

`NetLinkId fLink`
Each node-link combination has a link id.

1.2.24 TNetNodeReader

A node reader reads node values from the database.

Public access:

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

`void GetNextRecord()`
Get the next record in the table.

`NetNodeId GetId() const`
Return the id for the current node. A `TNetIllegalValue` exception is thrown if the data is not valid.

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

`string GetNotes() const`
Return the notes.

1.2.25 TNetNullControl

Null controls are used by nodes just outside the network boundary.

Public access:

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

`~TNetNullControl()`
Destroy a null traffic control.

`TNetNullControl& operator=(const TNetNullControl& control)`
Assign a null traffic control.

```
inline bool operator==(const TNetNullControl& control) const
inline bool operator!=(const TNetNullControl& control) const
```
Return whether two null controls are the same.

```
void AllowedMovements(TNetLane::Collection& lanes, const
      TNetLink& fromlink, const TNetLane& fromlane, const
      TNetLink& tolink)
void AllowedMovements(TNetLane::Collection& lanes, const
      TNetLink& fromlink, const TNetLink& tolink, const
      TNetLane& tolane)
void AllowedMovements(TNetLane::Collection& lanes, const
      TNetLink& fromlink, const TNetLink& tolink, bool phase
      = false)
void InterferingLanes(TNetLane::Collection& 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.

1.2.26 TNetParking

A parking place is a source or sink of vehicles along a link.

Public access:

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

```
typedef map<NetAccessoryId, TNetParking*,
      less<NetAccessoryId> > Map
typedef Map::iterator MapIterator
typedef Map::const_iterator MapConstIterator
```
Type definitions.

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

```
TNetParking (NetAccessoryId, TNetLink&, TNetNode&, REAL
      offset, EStyle style, UINT32 capacity, bool generic)
```
Construct a parking place with specified values.

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

```
inline UINT32 GetCapacity() const
```
Return the capacity of the parking. A capacity of zero represents unlimited capacity.

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

```
inline VehTypeSet& GetVehicleTypes()
inline const VehTypeSet& GetVehicleTypes() const
```
Return the vehicle types allowed to park.

```
bool IsAllowed(EVehType veh) const
```
Return whether a particular vehicle type is allowed to park.

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

Private access:

```
EStyle fStyle
```
A parking place has a style.

```
UINT32 fCapacity
```
A parking place has a capacity.

`bool fGeneric`
A parking place may be generic.

`VehTypeSet fVehicleTypes`
A parking place has a set of vehicle types allowed to park there.

1.2.27 TNetParkingReader

A parking reader reads parking values from the database.

Public access:

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

`void GetNextRecord()`
Get the next record in the table.

`NetAccessoryId GetId() const`
Return the id for the current accessory. A `TNetIllegalValue` exception is thrown if the data is not valid.

```
void GetLocation(NetLinkId& link, NetNodeId& node, REAL&
      offset) const
```
Return the location of the current accessory. A `TNetIllegalValue` exception is thrown if the data is not valid.

`TNetParking::EStyle GetStyle() const`
Return the style of the current parking. A `TNetIllegalValue` exception is thrown if the data is not valid.

`UINT16 GetCapacity() const`
Return the capacity of the current parking. A `TNetIllegalValue` exception is thrown if the data is not valid.

`bool IsGeneric() const`
Return whether the current parking is generic. A `TNetIllegalValue` exception is thrown if the data is not valid.

`VehTypeSet GetVehicleTypes() const`
Return the vehicle types allowed in the parking. A `TNetIllegalValue` exception is thrown if the data is not valid.

`string GetNotes() const`
Return the notes.

1.2.28 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.

Public access:

```
enum EInterval {kGreen, kYellow, kRed}
```
Each phase has three intervals.

```
typedef ring<vector<EInterval> > IntervalRing
typedef IntervalRing::iterator IntervalRingIterator
typedef IntervalRing::const_iterator
      IntervalRingConstIterator
typedef vector<TNetPhase*> Collection
typedef Collection::iterator CollectionIterator
typedef Collection::const_iterator CollectionConstIterator
typedef vector<TNetPhase*> OrderedCollection
typedef OrderedCollection::iterator
      OrderedCollectionIterator
typedef OrderedCollection::const_iterator
      OrderedCollectionConstIterator
```
Type definitions.

```
TNetPhase()
TNetPhase(TNetPhaseDescription& description)
TNetPhase(const TNetPhase& phase)
```
Construct a phase.

```
~TNetPhase()
```
Destroy a phase.

```
TNetPhase& operator=(const TNetPhase& phase)
```
Assign a phase.

```
inline bool operator==(const TNetPhase&) const
inline bool operator!=(const TNetPhase&) const
```
Return whether two phases are the same.

```
inline TNetPhaseDescription& GetPhaseDescription()
inline const TNetPhaseDescription& GetPhaseDescription()
      const
```
Return phase description associated with this phase.

```
inline TNetPhase::EInterval GetInterval() const
```
Return current interval.

```
void SetInterval(EInterval interval)
```
Update signal interval.

```
inline TNetPhase::Collection& GetNextPhases()
inline const TNetPhase::Collection& GetNextPhases() const
```
Return phases to which this phase can transition.

```
inline void SetNextPhase(TNetPhase& phase)
```
Define the next phase.

Private access:

`IntervalRing  fIntervals`
Each phase has a sequence of intervals.

`TNetPhaseDescription* fPhaseDescription`
Each phase has an associated phase description.

`TNetPhase::Collection fNextPhases`
A phase has one or more next phases that it can transition to.

1.2.29 TNetPhaseDescription

A phase description specifies the interval lengths and allowed movements and associated turn protections during a phase.

Public access:

```
enum EProtection {kPermitted, kProtected,
      kPermittedAfterStop}
```
Turn protections are a subset of ETrafficControl

```
typedef vector<TNetPhaseDescription*> Collection
typedef Collection::iterator CollectionIterator
typedef Collection::const_iterator CollectionConstIterator
typedef map<TNetLink*, EProtection, less<TNetLink*> >
      LinkProtectionMap
typedef LinkProtectionMap::iterator
      LinkProtectionMapIterator
typedef LinkProtectionMap::const_iterator
      LinkProtectionMapConstIterator
typedef map<TNetLink*, LinkProtectionMap*, less<TNetLink*> >
      LinkMovementMap
typedef LinkMovementMap::iterator LinkMovementMapIterator
typedef LinkMovementMap::const_iterator
      LinkMovementMapConstIterator
```
Type definitions.

```
TNetPhaseDescription(NetPhaseNumber phase)
TNetPhaseDescription(const TNetPhaseDescription&
      description)
```
Construct a phasing plan description.

`~TNetPhaseDescription()`
Destroy a phasing plan description.

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

```
inline bool operator==(const TNetPhaseDescription&
      description) const
inline bool operator!=(const TNetPhaseDescription&
      description) const
```
Return whether two phase descriptions are the same.

```
inline NetPhaseNumber GetPhaseNumber() const
```
Return the phase number.

```
inline REAL GetGreenLength() const
```
Return the green interval length.

```
inline REAL GetMinGreenLength() const
```
Return the minimum green interval length.

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

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

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

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

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

```
LinkProtectionMap& GetLinkMovements(const TNetLink& link)
const LinkProtectionMap& GetLinkMovements(const TNetLink&
      link) const
```
Return the link movements for the specified link.

```
void SetLengths(REAL min, REAL max, REAL ext, REAL yellow,
      REAL red)
```
Set the interval lengths.

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

Private access:

```
NetPhaseNumber fPhaseNumber
```
Phase number.

```
union { REAL fMinGreenLength
REAL fGreenLength
}
```
Minimum green interval length, or green interval length for fixed time.

```
REAL fMaxGreenLength
```
Maximum green interval length, undefined for fixed time.

```
REAL fExtGreenLength
```
Green interval extension increment, equals zero for fixed time.

```
REAL fYellowLength
```
Yellow interval length.

```
REAL fRedLength
```
Red clearance interval length.

```
LinkMovementMap fLinkMovements
```
Movements allowed during this phase.

```
static LinkProtectionMap fNull
static LinkProtectionMap fNull
```
Null map is returned when no link movement binding exists.

```
Return whether two phase descriptions are the same. ne bool
      TNetPhaseDescription::operator==(const
      TNetPhaseDescription& p)  const const return (this ==
      &p)
```
REAL fWalkLength; REAL fWalkLength; Flashing don't walk interval length. REAL fFDWLength; REAL fFDWLength; REAL fFDWLength; REAL fFDWLength;

1.2.30 TNetPhasingPlan

A phasing plan is composed of a series of phase descriptions.

Public access:

```
typedef set<TNetPhasingPlan*, less<TNetPhasingPlan*> > Set
typedef Set::iterator SetIterator
typedef Set::const_iterator SetConstIterator
typedef map<NetPhaseGroup, NetPhaseNumber,
      less<NetPhaseGroup> >  GroupFirstPhaseMap
typedef GroupFirstPhaseMap::iterator
      GroupFirstPhaseMapIterator
typedef GroupFirstPhaseMap::const_iterator
      GroupFirstPhaseMapConstIterator
```
Type definitions.

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

```
~TNetPhasingPlan()
```
Destroy a phasing plan.

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

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

```
inline TNetPhaseDescription::Collection&
      GetPhaseDescriptions()
inline const TNetPhaseDescription::Collection&
      GetPhaseDescriptions() const
```
Return this phasing plan.

```
TNetPhaseDescription& CreatePhaseDescription(NetPhaseNumber
      phase)
```
Create a phase description.

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

```
inline NetPhaseNumber GetFirstPhase(NetPhaseGroup group)
      const
```
Return the first phase in the specified phase group.

```
inline TNetPhasingPlan::GroupFirstPhaseMap& GetFirstPhases()
inline const TNetPhasingPlan::GroupFirstPhaseMap&
      GetFirstPhases() const
```
Return the map of first phases per group.

```
inline void SetFirstPhase(NetPhaseGroup group,
      NetPhaseNumber phase)
```
Define the first phase for a group.

Private access:

```
NetPlanId fId
```
Phasing plan has an id.

```
TNetPhaseDescription::Collection  fPhasingPlan
```
A phasing plan is a sequence of interval length descriptions, one for each phase.

```
GroupFirstPhaseMap fFirstPhases
```
A phasing plan has a first phase in each phase group.

1.2.31 TNetPhasingPlanReader

This reader reads phasing plan values from the database.

Public access:

```
struct TRecord { NetNodeId fNode
NetPlanId fPlan
NetPhaseNumber fPhase
NetLinkId fInlink
NetLinkId fOutlink
TNetPhaseDescription::EProtection fProtection
}
```

Phasing plan records can be cached.

```
typedef list<TRecord*> RecordCache
typedef RecordCache::iterator RecordCacheIterator
typedef RecordCache::const_iterator RecordCacheConstIterator
```

Type definitions.

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

`void GetNextRecord()`
Get the next record in the table.

`NetNodeId GetNode() const`
Return the id for the current node. A `TNetIllegalValue` exception is thrown if the data is not valid.

`NetPlanId GetPlan() const`
Return the timing plan id. A `TNetIllegalValue` exception is thrown if the data is not valid.

`NetPhaseNumber GetPhase() const`
Return the phase number. A `TNetIllegalValue` exception is thrown if the data is not valid.

`NetLinkId GetInlink() const`
Return the incoming link id. A `TNetIllegalValue` exception is thrown if the data is not valid.

`NetLinkId GetOutlink() const`
Return the outgoing link id. A `TNetIllegalValue` exception is thrown if the data is not valid.

`TNetPhaseDescription::EProtection GetProtection() const`
Return the turn protection indicator. A `TNetIllegalValue` exception is thrown if the data is not valid.

`string GetNotes() const`
Return the notes.

1.2.32 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.

Public access:

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

```
typedef vector<TNetPocket*> Collection
typedef Collection::iterator CollectionIterator
typedef Collection::const_iterator CollectionConstIterator
```
Type definitions.

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

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

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

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

Private access:

`EStyle fStyle`
Each pocket has a style.

`REAL fLength`
Each pocket has a length.

1.2.33 TNetPocketReader

A pocket reader reads pocket values from the database.

Public access:

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

`void GetNextRecord()`
Get the next record in the table.

`NetAccessoryId GetId() const`
Return the id for the current accessory. A `TNetIllegalValue` exception is thrown if the data is not valid.

```
void GetLocation(NetLinkId& link, NetNodeId& node, REAL&
      offset) const
```
Return the location of the current accessory. A `TNetIllegalValue` exception is thrown if the data is not valid.

`NetLaneNumber GetLaneNumber() const`
Return the lane number of the current pocket. A `TNetIllegalValue` exception is thrown if the data is not valid.

`TNetPocket::EStyle GetStyle() const`
Return the style of the current pocket. A `TNetIllegalValue` exception is thrown if the data is not valid.

`REAL GetLength() const`
Return the length of the current pocket. A `TNetIllegalValue` exception is thrown if the data is not valid.

`string GetNotes() const`
Return the notes.

1.2.34 TGeoPoint

A geographic point contains the coordinates of a position on a map.

Public access:

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

`inline REAL GetX() const`
Return the x-coordinate.

`inline REAL GetY() const`
Return the y-coordinate.

`inline REAL GetZ() const`
Return the z-coordinate.

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

Private access:

`REAL fX`
Each point has an x-coordinate.

`REAL fY`
Each point has a y-coordinate.

`REAL fZ`
Each point has a z-coordinate.

1.2.35 TNetProcessLink

Class for a virtual (process) link.

Public access:

```
typedef map<NetLinkId, TNetProcessLink*, less<NetLinkId> >
      Map
typedef Map::iterator MapIterator
typedef Map::const_iterator MapConstIterator
```
Type definitions.

`TNetProcessLink(TNetProcessLinkReader& reader)`
Construct the virtual link from the specified reader.

`inline NetLinkId GetId() const`
Return the ID of the virtual link.

```
inline TNetAccessory& GetFromAccessory()
inline const TNetAccessory& GetFromAccessory() const
```
Return the “from” accessory.

```
inline TNetAccessory& GetToAccessory()
inline const TNetAccessory& GetToAccessory() const
```
Return the “to” accessory.

`inline REAL GetDelay() const`
Return the delay (in seconds) associated with traversing the link.

`inline REAL GetCost() const`
Return the cost (in arbitrary units) associated with traversing the link.

```
inline void SetAccessories(TNetAccessory& from,
      TNetAccessory& to)
```
Set the “from” and “to” accessories.

Private access:

`NetLinkId fId`
Each process link has an ID.

`TNetAccessory* fFrom`
Each process link has a “from” accessory connection.

`TNetAccessory* fTo`
Each process link has a “to” accessory connection.

`REAL fDelay`
Each process link has a delay.

`REAL fCost`
Each process link has a cost.

1.2.36 TNetProcessLinkReader

Class for reading virtual links from the database.

Public access:

`TNetProcessLinkReader(TNetReader& reader)`
Construct a virtual link reader for a given network.

`void GetNextRecord()`
Get the next record in the table.

`NetLinkId GetId() const`
Return the id for the current link. A `TNetIllegalValue` exception is thrown if the data is not valid.

```
void GetFromAccessory(NetAccessoryId& id,
      TNetAccessory::EType& type) const
```
Return the "from" accessory.

```
void GetToAccessory(NetAccessoryId& id,
      TNetAccessory::EType& type) const
```
Return the "to" accessory.

`REAL GetDelay() const`
Return the delay (in seconds) associated with traversing the link.

`REAL GetCost() const`
Return the cost (in arbitrary units) associated with traversing the link.

`string GetNotes() const`
Return the nodes.

1.2.37 TNetReader

A network reader reads a network from the database.

Public access:

```
enum ELevel {kBare = 0, kActivity = 1, kProcess = 2, kLane =
      3, kControl = 4}
```
Levels of network construction:

	Level				
Table	*Bare*	*Activity*	*Process*	*Lane*	*Control*
Node	*x*	*x*	*x*	*x*	*x*
Link	*x*	*x*	*x*	*x*	*x*
Activity Location		*x*	*x*	*x*	*x*
Parking			*x*	*x*	*x*
Transit Stop			*x*	*x*	*x*

Process Link	*x*	*x*	*x*
Pocket Lane		*x*	*x*
Lane Connectivity		*x*	*x*
Turn Prohibition		*x*	*x*
Lane Use		*x*	*x*
Speed		*x*	*x*
Barrier		*x*	*x*
Unsignalized Node			*x*
Signalized Node			*x*
Phasing Plan			*x*
Timing Plan			*x*
Signal Coordinator			*x*
Detector			*x*

```
TNetReader(const TConfiguration& configuration, ELevel level
      = kControl)
```

Construct a reader for the specified configuration.

```
TNetReader(const string& nodeTable, const string& linkTable,
      const string& pocketLaneTable, const string&
      parkingTable, const string& laneConnectivityTable,
      const string& unsignalizedNodeTable, const string&
      signalizedNodeTable, const string& phasingPlanTable,
      const string& timingPlanTable, const string&
      speedTable, const string& laneUseTable, const string&
      transitStopTable, const string&
      signalCoordinatorTable, const string& detectorTable,
      const string& turnProhibitionTable, const string&
      barrierTable, const string& activityLocationTable,
      const string& processLinkTable)
```

Construct a reader for the specified tables.

```
TNetReader(const TNetReader& reader)
```

Construct a copy of the specified reader.

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

Make the reader a copy of the specified reader.

```
static TNetReader GetReader(const string& directory, const
      string& node = “”, const string& link = “”, const
      string& pocketLane = “”, const string& parking = “”,
      const string& laneConnectivity = “”, const string&
      unsignalizedNode = “”, const string& signalizedNode =
      , const string& phasingPlan = “”, const string&
      timingPlan = “”, const string& speed = “”, const
      string& laneUse = “”, const string& transitStop = “”,
      const string& signalCoordinator = “””, const string&
      detector = “”, const string& turnProhibition = “”,
      const string& barrier = “”, const string& activity =
      “”, const string& process = “”)
```

Return a reader with the specified table names.

`inline const string& GetNodeTable()`
Return the node table.

`inline const string& GetLinkTable()`
Return the link table.

`inline const string& GetPocketLaneTable()`
Return the pocket lane table.

`inline const string& GetParkingTable()`
Return the parking table.

`inline const string& GetLaneConnectivityTable()`
Return the lane connectivity table.

`inline const string& GetUnsignalizedControlTable()`
Return the unsignalized control table.

`inline const string& GetSignalizedControlTable()`
Return the signalized control table.

`inline const string& GetPhasingPlanTable()`
Return the phasing plan table.

`inline const string& GetTimingPlanTable()`
Return the timing plan table.

`inline const string& GetSpeedTable()`
Return the speed table.

`inline const string& GetLaneUseTable()`
Return the lane use table.

`inline const string& GetTransitStopTable()`
Return the transit stop table.

`inline const string& GetSignalCoordinatorTable()`
Return the signal coordinator table.

`inline const string& GetDetectorTable()`
Return the detector table.

`inline const string& GetTurnProhibitionTable()`
Return the turn prohibition table.

`inline const string& GetBarrierTable()`
Return the barrier table.

`inline const string& GetActivityLocationTable()`
Return the activity location table.

`inline const string& GetProcessLinkTable()`
Return the process link table.

`static const string kgDirectory`
Configuration key for directory where the network files reside.

`static const string kgNodeTable`
Configuration key for node table name.

`static const string kgLinkTable`
Configuration key for link table name.

`static const string kgPocketLaneTable`
Configuration key for pocket lane table name.

`static const string kgParkingTable`
Configuration key for parking table name.

`static const string kgLaneConnectivityTable`
Configuration key for lane connectivity table name.

`static const string kgUnsignalizedNodeTable`
Configuration key for unsignalized node table name.

`static const string kgSignalizedNodeTable`
Configuration key for signalized node table name.

`static const string kgPhasingPlanTable`
Configuration key for phasing plan table name.

`static const string kgTimingPlanTable`
Configuration key for timing plan table name.

`static const string kgSpeedTable`
Configuration key for speed table name.

`static const string kgLaneUseTable`
Configuration key for lane use table name.

`static const string kgTransitStopTable`
Configuration key for transit stop table name.

`static const string kgSignalCoordinatorTable`
Configuration key for signal coordinator table name.

`static const string kgDetectorTable`
Configuration key for detector table name.

`static const string kgTurnProhibitionTable`
Configuration key for turn prohibition table name.

`static const string kgBarrierTable`
Configuration key for barrier table name.

`static const string kgActivityLocationTable`
Configuration key for activity location table.

`static const string kgProcessLinkTable`
Configuration key for process link table.

Private access:

`string fNodeTable`
Each reader has a node table.

`string fLinkTable`
Each reader has a link table.

`string fPocketLaneTable`
Each reader has a pocket lane table.

`string fParkingTable`
Each reader has a parking table.

`string fLaneConnectivityTable`
Each reader has a lane connectivity table.

`string fUnsignalizedControlTable`
Each reader has an unsignalized control table.

`string fSignalizedControlTable`
Each reader has a signalized control table.

`string fPhasingPlanTable`
Each reader has a phasing plan table.

`string fTimingPlanTable`
Each reader has a timing plan table.

`string fSpeedTable`
Each reader has a speed table.

`string fLaneUseTable`
Each reader has a lane use table.

`string fTransitStopTable`
Each reader has a transit stop table.

`string fSignalCoordinatorTable`
Each reader has a signal coordinator table.

`string fDetectorTable`
Each reader has a detector table.

`string fTurnProhibitionTable`
Each reader has a turn prohibition table.

`string fBarrierTable`
Each reader has a barrier table.

`string fActivityLocationTable`
Each reader has an activity location table.

`string fProcessLinkTable`
Each reader has a process link table.

1.2.38 TGeoRectangle

A geographic rectangle is a rectangle in a map coordinate system.

Public access:

```
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.

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

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

Private access:

`TGeoPoint fMinimum`
Each rectangle has a minimum corner.

`TGeoPoint fMaximum`
Each rectangle has a maximum corner.

1.2.39 TNetSignalCoordinator

A signal coordinator coordinates the operation of several traffic signals.

Public access:

```
typedef map<NetCoordinatorId, TNetSignalCoordinator*,
      less<NetCoordinatorId> > Map
typedef Map::iterator MapIterator
typedef Map::const_iterator MapConstIterator
```

Type definitions.

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

Construct a signalized control coordinator.

```
virtual ~TNetSignalCoordinator()
```

Destroy a coordinator.

```
virtual void UpdateSignalizedControl(REAL sim_time) = 0
```

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

```
TNetPhasingPlan::Set& GetPhasingPlans()
const TNetPhasingPlan::Set& GetPhasingPlans() const
```

Return the coordinator's phasing plans.

```
TNetPhasingPlan& CreatePhasingPlan(NetNodeId node, NetPlanId
      phasing, const TNetTimingPlanReader::RecordCache&
      timingRecordCache, const
      TNetPhasingPlanReader::RecordCache&
      phasingRecordCache, TNetNetwork& network,
      NetPhaseNumberMap& numbers)
```

Define the phasing plan.

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

Return the phasing plan with specified id.

```
inline TNetSignalizedControl::Collection& GetControllers()
inline const TNetSignalizedControl::Collection&
      GetControllers() const
```

Return the signalized controls coordinated by this coordinator.

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

Define a controller.

`inline NetCoordinatorId GetId() const`
Return the coordinator's id.

Protected access:

```
TNetSignalCoordinator& operator=(const
      TNetSignalCoordinator&)
```
Assign a signal coordinator.

Private access:

`NetCoordinatorId fId`
Each coordinator has a unique id.

`TNetSignalizedControl::Collection fControllers`
A signalized control coordinator has a group of associated sensors. A signalized control coordinator has a group of associated sensors. A signalized control coordinator controls one or more signalized controllers.

`TNetPhasingPlan::Set fPhasingPlans`
A signalized control coordinator has a group of phasing plans that it can tell its controllers to use.

1.2.40 TNetSignalCoordinatorReader

This reader reads signal coordinator values from the database.

Public access:

`TNetSignalCoordinatorReader(TNetReader& reader)`
Construct a signal coordinator reader for a given network.

`void GetNextRecord()`
Get the next record in the table.

`NetCoordinatorId GetId() const`
Return the id for the coordinator. A `TNetIllegalValue` exception is thrown if the data is not valid.

`string GetNotes() const`
Return the notes.

1.2.41 TNetSignalizedControl

A signalized control specifies the signal phases and phasing plan at a node.

Public access:

```
typedef vector<TNetSignalizedControl*> Collection
typedef Collection::iterator CollectionIterator
typedef Collection::const_iterator CollectionConstIterator
```
Type definitions.

```
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 ~TNetSignalizedControl()
```
Destroy a signalized traffic control.

```
virtual void AllowedMovements(TNetLane::Collection& 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(TNetLane::Collection& 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(TNetLane::Collection& 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(TNetLane::Collection& 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) = 0
```
Update the traffic control state based on current simulation time.

```
inline virtual TNetPhasingPlan& GetPhasingPlan()
inline virtual const TNetPhasingPlan& GetPhasingPlan() const
```
Return the current phasing plan.

```
inline virtual REAL GetPhasingPlanOffset() const
```
Return the current phasing plan offset.

```
inline virtual void SetPhasingPlan(TNetPhasingPlan& plan)
```
Define the phasing plan.

```
inline virtual void SetPhasingPlanOffset(REAL offset)
```
Define the phasing plan offset.

```
inline virtual TNetSignalCoordinator& GetCoordinator()
inline 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.

```
inline virtual TNetPhase::OrderedCollection& GetPhases()
inline virtual const TNetPhase::OrderedCollection&
      GetPhases() const
```
Return the phases for this signal.

```
inline virtual TNetPhase& GetPhase()
inline virtual const TNetPhase& GetPhase() const
```
Return the current phase.

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

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

Protected access:

```
typedef set<TNetPhaseDescription::EProtection,
      less<TNetPhaseDescription::EProtection> >
      ProtectionSet
```
Type definitions.

`TNetSignalizedControl& operator=(const TNetSignalizedControl&)`
Assign a signalized traffic control.

Private access:

`TNetPhase::OrderedCollection fPhases`
A signalized control has a sequence of phases.

`TNetPhasingPlan* fPhasingPlan`
A signalized controller has a phasing plan.

`REAL fPhasingPlanOffset`
A signalized controller may have a non-zero phasing plan offset.

`TNetSignalCoordinator* fCoordinator`
A signalized controller knows its coordinator.

`TNetPhase* fCurrentPhase`
Current phase.

1.2.42 TNetSignalizedControlReader

This reader reads signalized control values from the database.

Public access:

`enum EType {kTimed, kActuated}`
There are several types.

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

`void GetNextRecord()`
Get the next record in the table.

`NetNodeId GetNode() const`
Return the id for the current node. A `TNetIllegalValue` exception is thrown if the data is not valid.

`EType GetType() const`
Return the type of the signal. A `TNetIllegalValue` exception is thrown if the data is not valid.

`NetPlanId GetPlan() const`
Return the timing plan id. A `TNetIllegalValue` exception is thrown if the data is not valid.

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

`NetCoordinatorId GetCoordinator() const`
Return the coordinator id. A `TNetIllegalValue` exception is thrown if the data is not valid.

`bool IsSingleRing() const`
Return the ring for actuated signals. A `TNetIllegalValue` exception is / thrown if the data is not valid.

`bool IsSingleEntry() const`
Return the entry for dual ring actuated signals. A `TNetIllegalValue` exception is thrown if the data is not valid.

`string GetNotes() const`
Return the notes.

1.2.43 TNetSimulationArea

The simulation area describes the simulation region of interest.

Public access:

```
enum EType {kStudy, kBuffer}
```
Link type indicates whether the link is in the buffer or study area.

```
typedef map<NetLinkId, EType, less<NetLinkId> > LinkIdMap
typedef LinkIdMap::iterator LinkIdMapIterator
typedef LinkIdMap::const_iterator LinkIdMapConstIterator
typedef map<NetNodeId, EType, less<NetNodeId> > NodeIdMap
typedef NodeIdMap::iterator NodeIdMapIterator
typedef NodeIdMap::const_iterator NodeIdMapConstIterator
```
Type definitions.

```
TNetSimulationArea(TNetSimulationAreaReader& areaReader,
      TNetReader& reader)
TNetSimulationArea(const TNetSimulationArea& area)
```
Construct a simulation area.

```
~TNetSimulationArea()
```
Destroy a simulation area.

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

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

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

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

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

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

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

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

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

`inline NodeIdMap& GetNodes()`
`inline const NodeIdMap& GetNodes() const`
Return all nodes in simulation area.

`inline bool IsNodeInSimulationArea(NetNodeId id) const`
Return whether a node is in the simulation area (study area + buffer).

`inline bool IsNodeInStudyArea(NetNodeId id) const`
Return whether a node is in the study area.

`inline bool IsNodeInBufferArea(NetNodeId id) const`
Return whether a node is in the simulation area buffer region.

Private access:

`LinkIdMap fLinks`
A simulation area is described by links and whether the link is in the buffer portion of the simulation area.

`NodeIdMap fNodes`
All nodes at the ends of links in the simulation area are in the simulation area. A node is in the study area if any of its links are in the study area, and in the buffer area otherwise.

1.2.44 TNetSimulationAreaLinkReader

A simulation area link reader reads link values from the database.

Public access:

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

```
void GetNextRecord()
```
Get the next record in the table.

```
NetLinkId GetLink() const
```
Return the link id. A `TNetIllegalValue` exception is thrown if the data is not valid.

```
bool IsInBufferArea() const
```
Return whether the link is in the buffer area. A `TNetIllegalValue` exception is thrown if the data is not valid.

```
string GetNotes() const
```
Return the notes.

1.2.45 TNetSimulationAreaReader

A simulation area reader reads a simulation area from the database.

Public access:

```
TNetSimulationAreaReader(const TConfiguration&
      configuration)
```
Construct a reader for the specified configuration.

```
TNetSimulationAreaReader(const string& areaTable)
```
Construct a reader for the specified tables.

```
const string& GetSimulationAreaTable()
```
Return the simulation area table.

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

```
static const string kgStudyAreaLinksTable
```
Configuration key for study area links table name.

Private access:

`string fSimulationAreaTable`
Each reader has a simulation area table.

1.2.46 TNetSpeedReader

A speed reader reads speed values from the database.

Public access:

`TNetSpeedReader(TNetReader& reader)`
Construct a link reader for a given network.

`void GetNextRecord()`
Get the next record in the table.

`NetLinkId GetLink() const`
Return the id for the current speed. A `TNetIllegalValue` exception is thrown if the data is not valid.

`NetNodeId GetNode() const`
Return the node toward which lanes are headed. A `TNetIllegalValue` exception is thrown if the data is not valid.

`REAL GetSpeedLimit() const`
Return the speed limit of the lanes. A `TNetIllegalValue` exception is thrown if the data is not valid.

`REAL GetFreeFlowSpeed() const`
Return the free-flow speed of the lanes. A `TNetIllegalValue` exception is thrown if the data is not valid.

`VehTypeSet GetVehicleTypes() const`
Return the vehicle types to which speeds apply. A `TNetIllegalValue` exception is thrown if the data is not valid.

`string GetNotes() const`
Return the notes.

1.2.47 TNetSubnetwork

A subnetwork represents a subset of the instantiated network.

Public access:

```
TNetSubnetwork(TNetReader& reader, TNetNetwork& network,
      const  TGeoFilterFunction& filter = TGeoFilterNone(),
      TNetFactory& factory = fgDefaultFactory)
```
Construct a subnetwork with geographic filtering using the reader.

```
TNetSubnetwork(TNetReader& reader, TNetNetwork& network,
      NetNodeIdSet& nodesRequested, NetLinkIdSet&
      linksRequested,  TNetNode::Set& nodesProvided,
      TNetLink::Set& linksProvided,  TNetFactory& factory =
      fgDefaultFactory)
```

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

```
TNetSubnetwork(TNetReader& reader, TNetNetwork& network,
      TNetSimulationArea& simArea, TNetNode::Set&
      nodesProvided,  TNetLink::Set& linksProvided,
      TNetFactory& factory = fgDefaultFactory)
```

Construct a subnetwork based on the specified simulation area.

```
inline TNetNode::Set& GetNodes()
inline const TNetNode::Set& GetNodes() const
```

Return the set of nodes in the subnetwork.

```
inline TNetLink::Set& GetLinks()
inline const TNetLink::Set& GetLinks() const
```

Return the set of links in the subnetwork.

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

Return the network.

Protected access:

```
void CreateNodes(TNetReader&, TNetFactory&, NetNodeIdSet&)
```

Create the network nodes.

```
void CreateLinks(TNetReader&, TNetFactory&, NetLinkIdSet&,
      NetLinkIdSet&)
```

Create the network links.

```
void CreateAccessories(TNetReader&, TNetFactory&,
      NetLinkIdSet&)
```

Create the network accessories.

```
void CreateTrafficControls(TNetReader&, TNetFactory&,
      NetNodeIdSet&)
```

Create the network traffic controls.

```
static TNetFactory fgDefaultFactory
```

The default factory.

Private access:

```
TNetSubnetwork(const TNetSubnetwork&) : fNetwork(0){
```

Do not allow subnetworks to be copied.

```
TNetSubnetwork& operator=(const TNetSubnetwork&) {return
      *this;
```
Do not allow subnetworks to be assigned.

```
TNetNetwork* const fNetwork
```
Each subnetwork is part of a network.

```
TNetNode::Set fNodes
```
Each subnetwork has references to the nodes it contains.

```
TNetLink::Set fLinks
```
Each subnetwork has references to the links it contains.

1.2.48 TNetTimedControl

A timed control specifies the performance of a pre-timed signal.

Public access:

```
typedef map<TNetPhase*, REAL, less<TNetPhase*> > PhaseEndMap
typedef PhaseEndMap::iterator PhaseEndMapIterator
typedef PhaseEndMap::const_iterator PhaseEndMapConstIterator
```
Type definitions.

```
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()
```
Destroy a timed signalized traffic control.

```
TNetTimedControl& operator=(const TNetTimedControl& control)
```
Assign a timed control.

```
inline bool operator==(const TNetTimedControl& control)
      const
inline 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.

Private access:

`REAL fCycleLength`
Cycle length.

`PhaseEndMap  fPhaseEnd`
Times at which each phase ends.

1.2.49 TNetTimingPlanReader

This reader reads timing plan values from the database.

Public access:

```
struct TRecord { NetPlanId fPlan
NetPhaseNumber fPhase
NetPhaseNumberCollection fNext
REAL fGreenMin
REAL fGreenMax
REAL fGreenExt
REAL fYellow
REAL fRed
NetPhaseGroup fGroup
}
```
Timing plan records can be cached.

```
typedef list<TRecord*> RecordCache
typedef RecordCache::iterator RecordCacheIterator
typedef RecordCache::const_iterator RecordCacheConstIterator
```
Type definitions.

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

`void GetNextRecord()`
Get the next record in the table.

`NetPlanId GetPlan() const`
Return the timing plan id. A `TNetIllegalValue` exception is thrown if the data is not valid.

`NetPhaseNumber GetPhase() const`
Return the phase number. A `TNetIllegalValue` exception is thrown if the data is not valid.

`NetPhaseNumberCollection GetNextPhases() const`
Return the phase numbers of the next phases. A `TNetIllegalValue` exception is thrown if the data is not valid.

`REAL GetGreenMin() const`
Return the minimum length of green interval. A `TNetIllegalValue` exception is thrown if the data is not valid.

`REAL GetGreenMax() const`
Return the maximum length of green interval. A `TNetIllegalValue` exception is thrown if the data is not valid.

`REAL GetGreenExt() const`
Return the length of green interval extension. A `TNetIllegalValue` exception is thrown if the data is not valid.

`REAL GetYellow() const`
Return the length of yellow interval. A `TNetIllegalValue` exception is thrown if the data is not valid.

`REAL GetRedClear() const`
Return the length of red clearance interval. A `TNetIllegalValue` exception is thrown if the data is not valid.

`NetPhaseGroup GetGroupFirst() const`
Return the phase group for which this phase is the first phase. A `TNetIllegalValue` exception is thrown if the data is not valid.

`string GetNotes() const`
Return the notes.

1.2.50 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.

Public access:

```
typedef map<TNetLane*, TNetLane::Collection, less<TNetLane*>
      >  ConnectivityMap
typedef ConnectivityMap::iterator ConnectivityMapIterator
typedef ConnectivityMap::const_iterator
      ConnectivityMapConstIterator
```
Type definitions.

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

```
TNetTrafficControl()
TNetTrafficControl(TNetNode& node)
TNetTrafficControl(const TNetTrafficControl& control)
```
Pedestrian signals display a walk indicator enum EPedestrianControl { ... }; enum EPedestrianControl { ... }; Construct a traffic control .

```
virtual ~TNetTrafficControl()
```
Destroy a traffic control.

```
virtual void AllowedMovements (TNetLane::Collection& 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 (TNetLane::Collection& 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 (TNetLane::Collection& 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 (TNetLane::Collection& 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 = 0
```
Return the vehicle control signal for the specified lane.

```
inline TNetNode& GetNode()
inline 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.

```
inline TNetLane::Collection& GetConnectivity(const TNetLane&
      lane)
inline const TNetLane::Collection& GetConnectivity(const
      TNetLane& lane) const
```
Return the lane connectivity for the specified lane.

```
inline TNetTrafficControl::ConnectivityMap&
      GetConnectivity()
inline const TNetTrafficControl::ConnectivityMap&
      GetConnectivity() const
```
Return the lane connectivity map.

Protected access:

```
TNetTrafficControl& operator=(const TNetTrafficControl&
      control)
```
Assign a traffic control.

Private access:

```
ConnectivityMap fLaneConnectivity
```
A traffic control contains a table that describes how lanes are connected across the node.

```
TNetNode* fNode
```
Traffic control knows its associated node

1.2.51 TNetTransitStop

A transit stop is a place along a link where passengers board/alight transit vehicles.

Public access:

```
enum EStyle {kStop, kStation, kYard}
```
There are several styles of transit stops.

```
typedef map<NetAccessoryId, TNetTransitStop*,
      less<NetAccessoryId> > Map
typedef Map::iterator MapIterator
typedef Map::const_iterator MapConstIterator
```
Type definitions.

```
TNetTransitStop(TNetTransitStopReader& reader)
```
Construct the transit stop from the specified reader.

```
~TNetTransitStop()
```
Destroy the transit stop.

```
inline EStyle GetStyle() const
```
Return the style of the transit stop.

`inline UINT16 GetCapacity() const`
Return the capacity of the transit stop. A capacity of zero represents unlimited capacity.

`inline const string& GetName() const`
Return the name of the transit stop.

`void SetName(string name)`
Define the name of the transit stop.

`inline VehTypeSet& GetVehicleTypes()`
`inline const VehTypeSet& GetVehicleTypes() const`
Return the vehicle types allowed to stop.

`inline bool IsAllowed(EVehType veh) const`
Return whether a particular vehicle type is allowed to stop.

Private access:

`EStyle fStyle`
A transit stop has a style.

`UINT16 fCapacity`
A transit stop has a vehicular capacity.

`string* fName`
A transit stop may have a name.

`VehTypeSet fVehicleTypes`
A transit stop has a set of vehicle types allowed to stop there.

`static const string fNoName`
Default name when name is not defined.

1.2.52 TNetTransitStopReader

A transit stop reader reads transit stop values from the database.

Public access:

`TNetTransitStopReader(TNetReader& reader)`
Construct a transit stop reader for a given network.

`void GetNextRecord()`
Get the next record in the table.

`NetAccessoryId GetId() const`
Return the id for the current accessory. A `TNetIllegalValue` exception is thrown if the data is not valid.

`void GetLocation(NetLinkId& link, NetNodeId& node, REAL& offset) const`
Return the location of the current accessory. A `TNetIllegalValue` exception is thrown if the data is not valid.

`string GetName() const`
Return the name of the transit stop.

`TNetTransitStop::EStyle GetStyle() const`
Return the style of the current transit stop. A `TNetIllegalValue` exception is thrown if the data is not valid.

`UINT16 GetCapacity() const`
Return the capacity of the current transit stop. A `TNetIllegalValue` exception is thrown if the data is not valid.

`VehTypeSet GetVehicleTypes() const`
Return the vehicle types allowed to stop. A `TNetIllegalValue` exception is thrown if the data is not valid.

`string GetNotes() const`
Return the notes.

1.2.53 TNetTurnProhibitionReader

This reader reads turn prohibition values from the database.

Public access:

`TNetTurnProhibitionReader(TNetReader& reader)`
Construct a turn prohibition reader for a given network.

`void GetNextRecord()`
Get the next record in the table.

`NetNodeId GetNode() const`
Return the id for the current node. A `TNetIllegalValue` exception is thrown if the data is not valid.

`NetLinkId GetInlink() const`
Return the id for incoming link. A `TNetIllegalValue` exception is thrown if the data is not valid.

`NetLinkId GetOutlink() const`
Return the id for outgoing link. A `TNetIllegalValue` exception is thrown if the data is not valid.

`string GetNotes() const`
Return the notes.

1.2.54 TNetUnsignalizedControl

An unsignalized control specifies the sign control at a node.

Public access:

```
typedef map<TNetLink*, TNetTrafficControl::ETrafficControl,
      less<TNetLink*> > SignMap
typedef SignMap::iterator SignMapIterator
typedef SignMap::const_iterator SignMapConstIterator
```

Type definitions.

```
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()
```

Destroy an unsignalized traffic control.

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

Assign an unsignalized traffic control.

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

Return whether two unsignalized controls are the same.

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

Return the vehicle control signal for the specified lane.

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

Define the vehicle control sign using the reader.

Private access:

```
SignMap fSigns
```

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.

1.2.55 TNetUnsignalizedControlReader

This reader reads unsignalized control values from the database.

Public access:

`TNetUnsignalizedControlReader(TNetReader& reader)`
Construct an unsignalized control reader for a given network.

`void GetNextRecord()`
Get the next record in the table.

`NetNodeId GetNode() const`
Return the id for the current node. A `TNetIllegalValue` exception is thrown if the data is not valid.

`NetLinkId GetInlink() const`
Return the id for incoming link. A `TNetIllegalValue` exception is thrown if the data is not valid.

`TNetTrafficControl::ETrafficControl GetSign() const`
Return the sign control indication on incoming link. A `TNetIllegalValue` exception is thrown if the data is not valid.

`string GetNotes() const`
Return the notes.

1.2.56 TNetException

A network exception signals the failure of a member function.

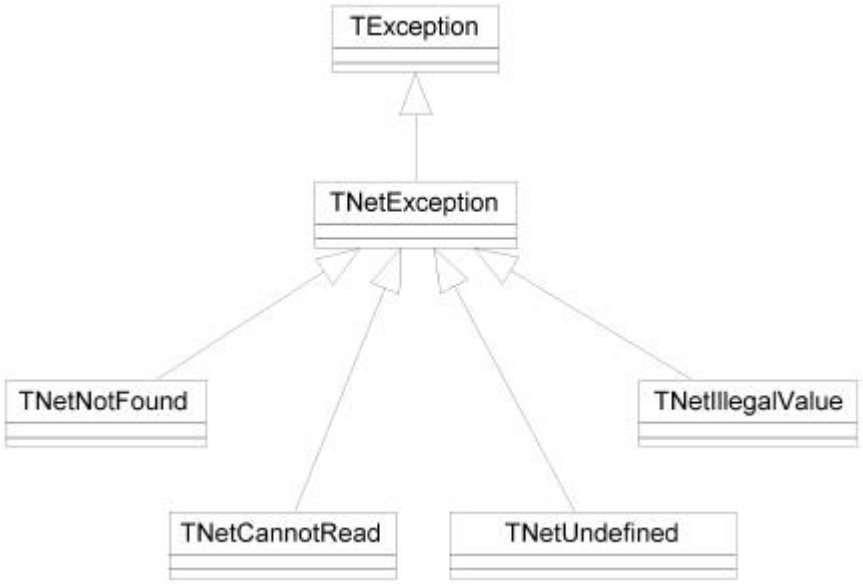

Figure 7: Exception hierarchy for the TRANSIMS network subsystem (UML notation).

`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.

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

`class TNetIllegalValue`
This exception is thrown when a database table contains an illegal value.

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

`class TNetCannotRead`
This exception is thrown when network data cannot be read from a file.

1.3 Implementation

1.3.1 C++ Libraries

The network subsystem uses the Standard C++ Library [Pl 95] (including the Standard Template Library), 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).

1.3.2 Sources for Traffic Engineering Information

References [GHA 88], [ITE 85], [ITE], [MM 84], [Or 93], and [PP 93] provide an overview of traffic engineering practice incorporated into the TRANSIMS network representation.

1.3.3 Data Table Formats

Seventeen data tables are required to describe a TRANSIMS format road network. The tables must be in tab-delimited ISO text format where the first line of the table is the header.

1.3.4 Limitations

1.3.4.1 Boundary Accessories

Boundary accessories have not been implemented for the IOC-2 version of the network representation subsystem.

1.4 Examples

1.4.1 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 data directory.
const string directory =
        "/home/transims/database/";

//  Get the table names.
char line[80];
cin.getline(line, 80);
const string nodeName = line;
cin.getline(line, 80);
const string linkName = line;
cin.getline(line, 80);
const string pocketName = line;
cin.getline(line, 80);
const string parkingName = line;
cin.getline(line, 80);
const string laneName = line;
cin.getline(line, 80);
const string ucontrolName = line;
cin.getline(line, 80);
const string scontrolName = line;
cin.getline(line, 80);
const string phasingName = line;
cin.getline(line, 80);
const string timingName = line;
cin.getline(line, 80);
const string speedName = line;
cin.getline(line, 80);
const string useName = line;
cin.getline(line, 80);
const string transitName = line;
cin.getline(line, 80);
const string coordName = line;
cin.getline(line, 80);
const string detectorName = line;
cin.getline(line, 80);
const string turnName = line;
cin.getline(line, 80);
const string barrierName = line;

//  Read network.
TNetReader reader =
        TNetReader::GetReader(directory,
        nodeName, linkName,
        pocketName, parkingName, laneName,
        ucontrolName, scontrolName, phasingName,
        timingName, speedName, useName,
```

```
        transitName, coordName, detectorName,
        turnName, barrierName);
TNetNetwork network;
TNetSubnetwork subnetwork(reader, network);
TNetNode::Set& nodes = subnetwork.GetNodes();
TNetLink::Set& links = subnetwork.GetLinks();
```

1.5 Future Work

Future work planned for the TRANSIMS network representation subsystem will focus on the inclusion of more complicated traffic controls (actuated signals and signals coordinated over a wide area) and time-of-day variation in network properties.

1.6 References

[BBS 98] B. W. Bush, K. P. Berkbigler, and L. L. Smith, *TRANSIMS Data Preparation Guide*, Revision 2.1, (Los Alamos, New Mexico: Los Alamos National Laboratory Report LA-UR-98-1411, 1998).

[GHA 88] Federal Highway Administration, *Manual on Uniform Traffic Control Devices*, (Washington, D.C.: U.S. Government Printing Office, 1988).

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

[ITE 85] Institute of Transportation Engineers, *Traffic Control Systems Handbook*, (Washington, D.C.: ITE Publications, 1985).

[ITE] Institute of Transportation Engineers, *Traffic Detector Handbook*, (Washington, D.C.: ITE Publications, n.d.).

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

[Or 93] F. L. Orcutt, Jr., *The Traffic Signal Book*, (Englewood Cliffs, New Jersey: Prentice Hall, 1993).

[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).

2. OUTPUT SUBSYSTEM

2.1 Overview

The output subsystem collects data from a running microsimulation, stores the data for future use, and manages the subsequent retrieval of the data. It forms a layer separating the other subsystems from the actual data files so that the other subsystems do not need to access the data files at the physical level or deal with the physical location and organization of the files. Figure 8 shows the position of the simulation output subsystem within the TRANSIMS software architecture. This subsystem only depends on the database subsystem (strongly) and the network subsystem (weakly) and is not tied to the specific design used for the IOC-1 microsimulation; this opens the possibility to reuse it in other TRANSIMS traffic simulations.

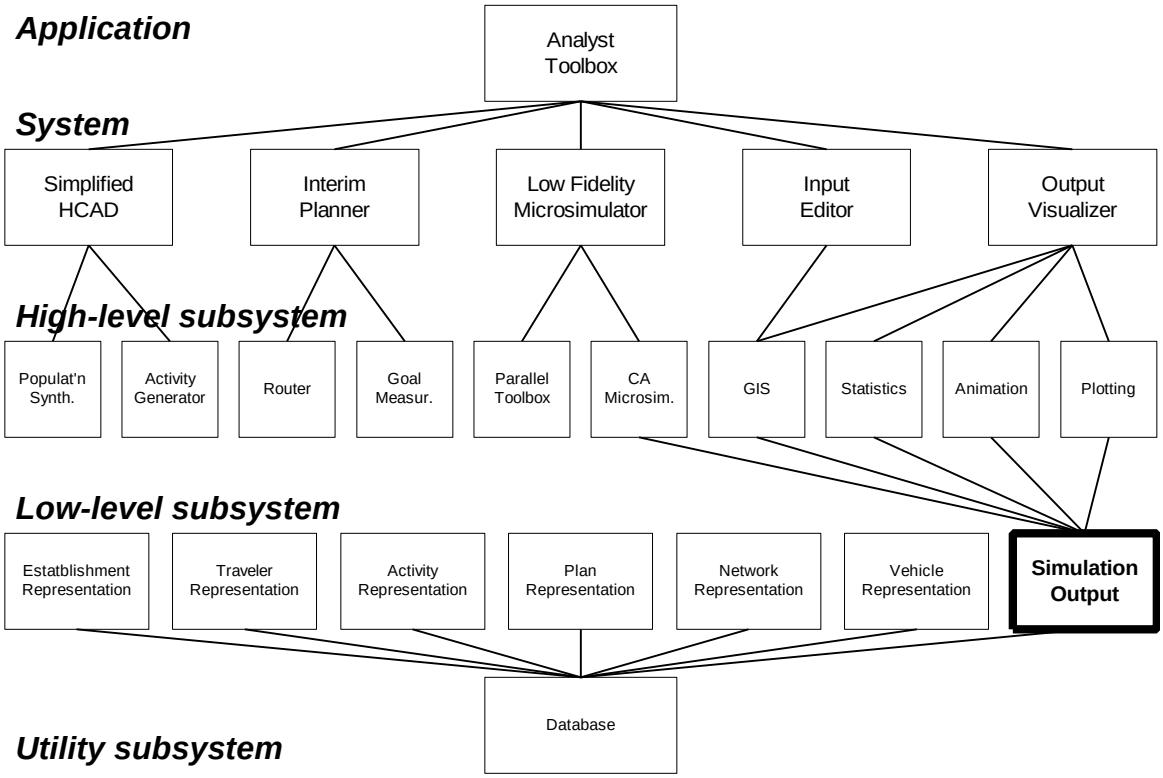

Figure 8: Location of the simulation output subsystem in the TRANSIMS software architecture.

This subsystem also allows the user to specify what data is collected and retrieved, and to filter it by space and time. Users can configure the subsystem to collect a wide variety of trajectory, event, and summary data from the simulation. Figure 9 shows an example of how data collection can be configured. The data can be accessed in binary format via a direct connection to the subsystem or in a delimited-text format for off-line postprocessing.

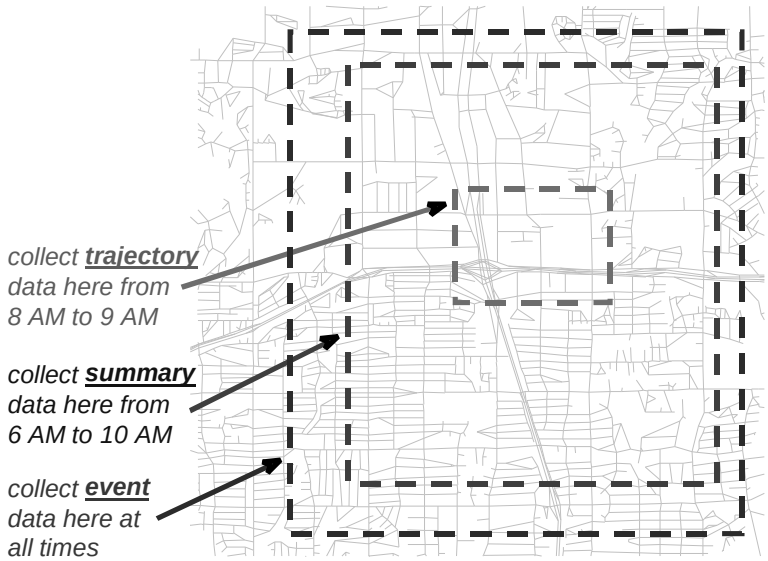

Figure 9: Example of how data collection can be filtered by space and by time.

The subsystem has been put to a wide variety of uses in the first TRANSIMS case study. Evolution data was used for animating vehicle movement, making periodic snapshots of the traffic, understanding the traffic behavior induced by CA (cellular automaton) microsimulation rules, refining the driving logic, and deriving fundamental diagrams. Event data has helped to locate problems with network data, driver logic, and plans, and to record the entry and exit times of vehicles in and out of the study area. Summary data provided a means to animate vehicle densities, identify congestion and deadlocks, and replan trips using observed link travel times.

The collection occurs in a distributed manner such that the impact of the subsystem on the microsimulation performance is minimized: Although the simulation output subsystem runs on multiple computational nodes (CPNs) during data collection, it does not require any communication between the CPNs. This leaves the full communication bandwidth available for use by the simulation proper. The retrieval, on the other hand, provides a unified view of the distributed data by coordinating the retrieval of data from remote file systems. Figure 10 illustrates the dual uses of the subsystem.

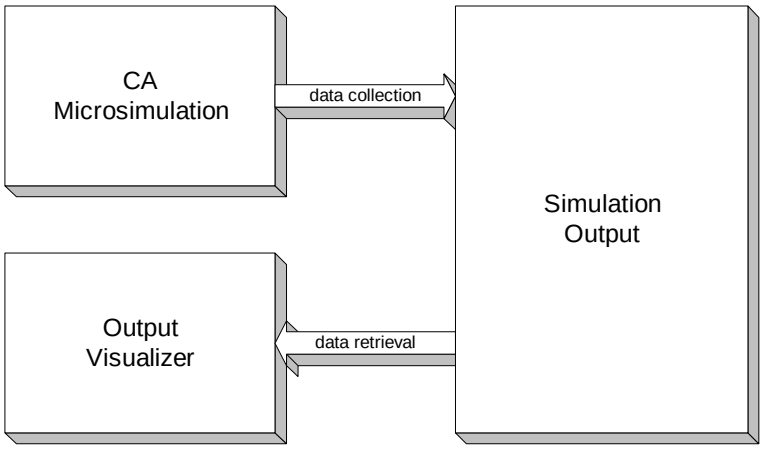

Figure 10: Dual uses of the simulation output subsystem.

2.2 Design

2.2.1 Concepts

2.2.1.1 Types of Data Collection

The output subsystem currently can collect three types of data: evolution (trajectory) data, event data, and summary data. Any number of each of these may be collected simultaneously in a simulation.

Evolution data provides the most detailed information about how the state of the microsimulation evolves in time. The vehicle data for links consists of the location, velocity, and status of each vehicle; this provides a complete "trajectory" for each vehicle in the simulation. The vehicle data for intersections consists of the location of the vehicle within the intersection buffer. The traffic control data simply reports the current phase and allowed movements at the traffic control. Evolution data may be collected for each timestep; the data is not summarized (i.e., totaled or averaged) in any way.

Event data supplies information on exceptional conditions of traveler status. Examples include when a traveler becomes lost (unable to follow its plan), when the plan for a traveler is invalid, and when the traveler enters or exits the study area. Event data is collected only when an event occurs.

Summary data reports aggregate data about the simulation. The link travel time data consists of counts of vehicles exiting links and means and variances of the vehicle traversal times for those links. Link density data provides counts and mean velocities of vehicles in variably-sized boxes that partition links. Summary data is sampled and reported periodically throughout the simulation.

2.2.1.2 Filtering

The simulation output subsystem has the capability to collect data on any subset of nodes and links in the road network (Figure 9). It is also possible to set the starting and ending times within which data collection occurs (Figure 10). A user can also specify the frequency of reporting for evolution and summary data and the sampling frequency (i.e., the frequency at which the data is observed) for summary data. The space and time filtering can be different for each type of data.

2.2.2 Classes

The simulation output subsystem has classes containing domain knowledge and classes forming a domain-independent data management layer (Figure 11). Figure 12, Figure 13, and Figure 14 show the relationships between classes used for evolution, event, and summary data collection, respectively; Figure 15 shows the relationships between classes used for data retrieval.

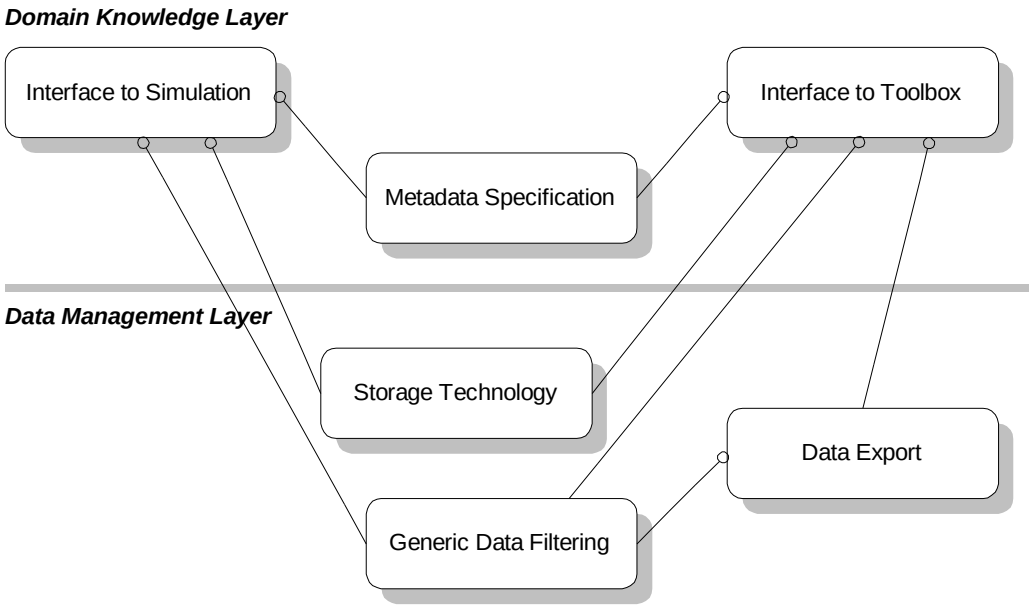

Figure 11: Categories of classes in the TRANSIMS simulation output subsystem.

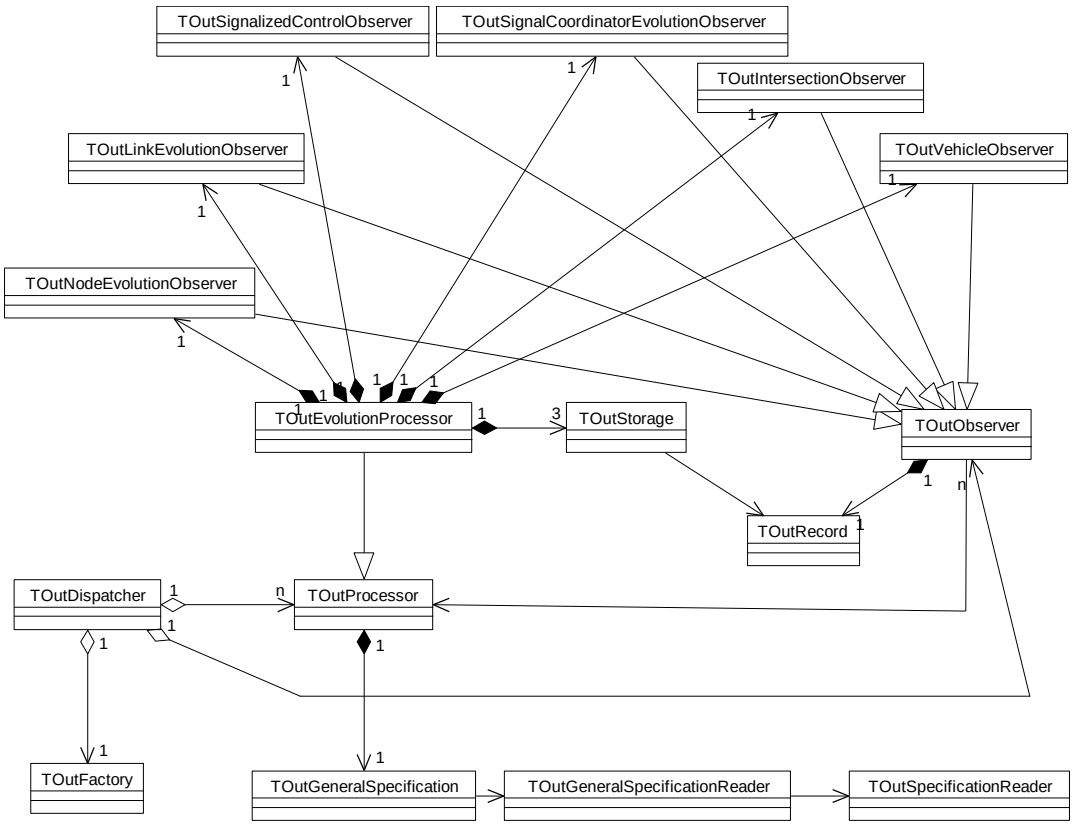

Figure 12: Class diagram for the TRANSIMS simulation output subsystem classes involved in evolution data collection(unified notation).

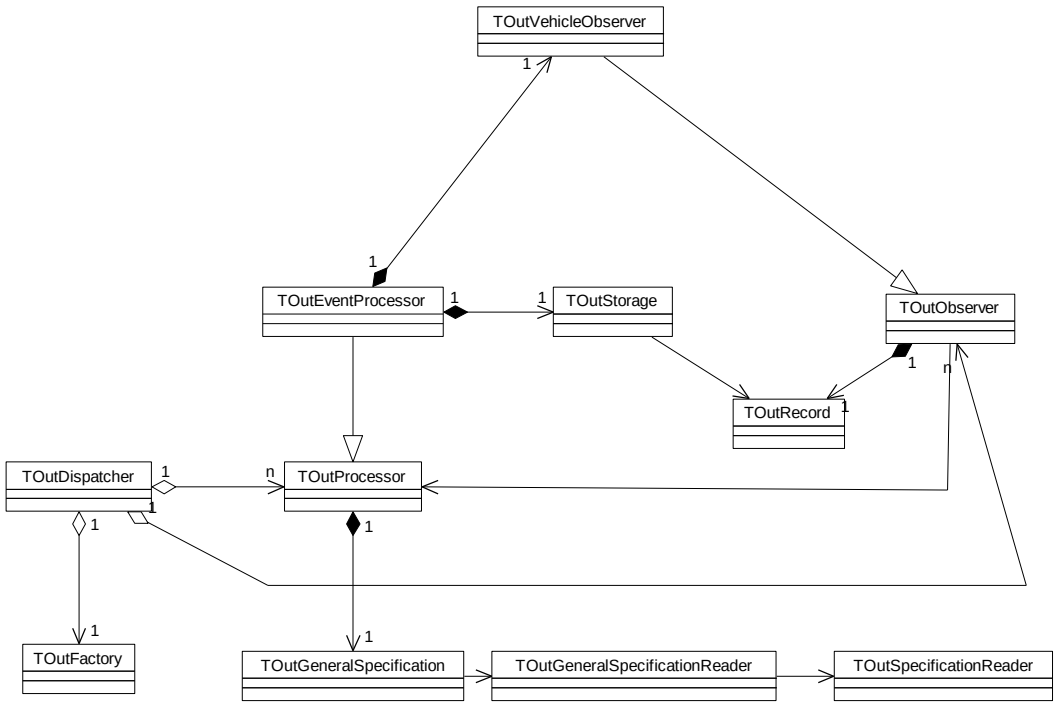

Figure 13: Class diagram for the TRANSIMS simulation output subsystem classes involved in event data collection (unified notation).

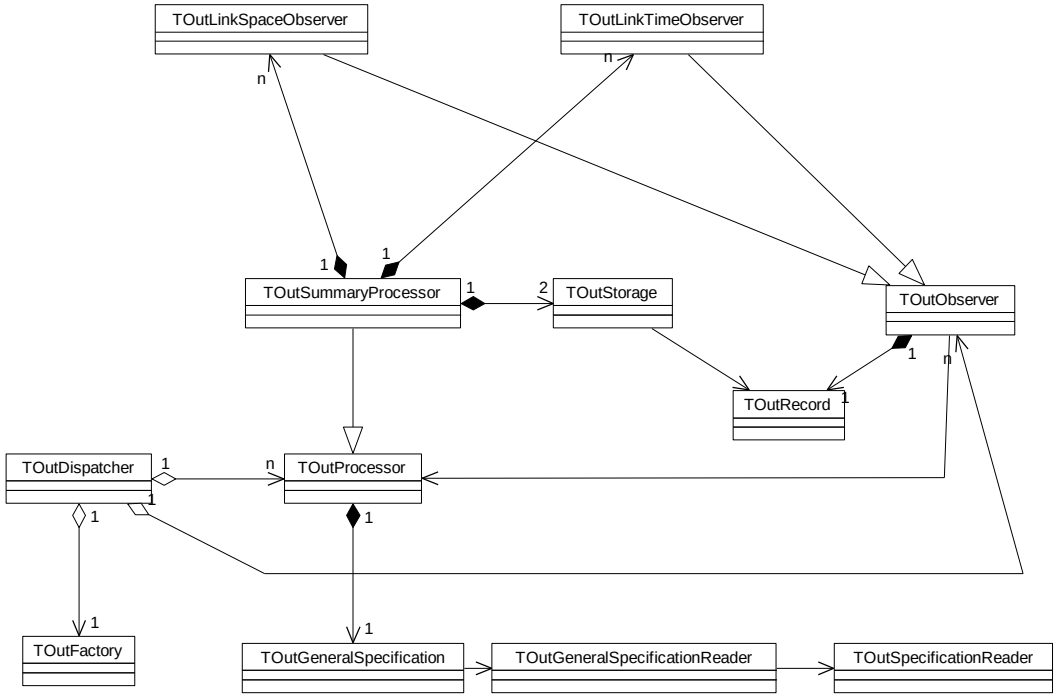

Figure 14: Class diagram for the TRANSIMS simulation output subsystem classes involved in summary data collection (unified notation).

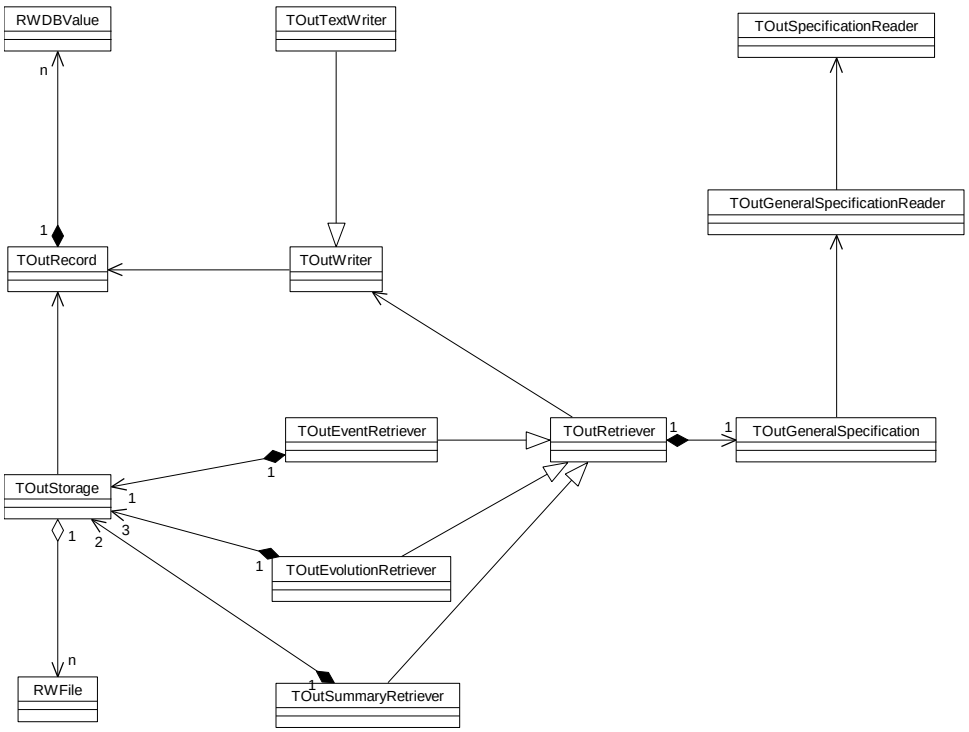

Figure 15: Class diagram for the TRANSIMS simulation output subsystem classes involved in data retrieval (unified notation).

2.2.2.1 TOutDispatcher

An output dispatcher coordinates the construction of simulation output objects and supervises the transfer of data.

Public access:

```
typedef map<TOutProcessor*, TOutObserver*,
      less<TOutProcessor*> > Map
typedef Map::iterator MapIterator
typedef Map::const_iterator MapConstIterator
```

Type definitions.

```
TOutDispatcher(const TConfiguration& config, TOutFactory&
      factory)
```

Construct an output dispatcher.

```
~TOutDispatcher()
```

Destroy an output dispatcher.

```
bool operator==(const TOutDispatcher& dispatcher) const
bool operator!=(const TOutDispatcher& dispatcher) const
```

Return whether two output dispatchers are the same.

```
void RecordOutput(REAL time, bool IsLastTimeStep)
```
Begin output recording for this timestep.

```
bool TerminateOutput(void)
```
Must be called after final FinishOutput, but before destruction

```
inline TOutProcessor::Map& GetProcessors()
inline const TOutProcessor::Map& GetProcessors() const
```
Return the processors.

```
inline TOutObserver::Map& GetObservers()
inline const TOutObserver::Map& GetObservers() const
```
Return the observers.

```
inline TOutEvolutionProcessor::Set& GetEvolutionProcessors()
inline const TOutEvolutionProcessor::Set&
      GetEvolutionProcessors() const
```
Return the evolution processors.

```
inline TOutEventProcessor::Set& GetEventProcessors()
inline const TOutEventProcessor::Set& GetEventProcessors()
      const
```
Return the event processors.

```
inline TOutSummaryProcessor::Set& GetSummaryProcessors()
inline const TOutSummaryProcessor::Set&
      GetSummaryProcessors() const
```
Return the summary processors.

```
void SetNodeObservers(Map& observers)
```
Define the node observers.

```
void SetLinkObservers(Map& observers)
```
Define the link observers.

```
void SetVehicleObservers(Map& observers)
```
Define the vehicle observers.

```
void SetIntersectionObservers(Map& observers)
```
Define the intersection observers.

```
void SetSignalCoordinatorObservers(Map& observers)
```
Define the signal coordinator observers.

```
void SetSignalizedControlObservers(Map& observers)
```
Define the signalized control observers.

```
void SetTravelerObservers(Map& observers)
```
Define the traveler observers.

`void SetLinkSpaceObservers(Map& observers)`
Define the link space observers.

`void SetLinkTimeObservers(Map& observers)`
Define the link time observers.

`void SetLinkVelocityObservers(Map& observers)`
Define the link velocity observers.

`void SetLinkEnergyObservers(Map& observers)`
Define the link energy observers.

`void ClearLinkSpaceObservers(Map& observers)`
Undefine the link space observers.

`void ClearLinkTimeObservers(Map& observers)`
Undefine the link time observers.

`void ClearLinkVelocityObservers(Map& observers)`
Undefine the link velocity observers.

`void ClearLinkEnergyObservers(Map& observers)`
Undefine the link energy observers.

Private access:

`TOutDispatcher(const TOutDispatcher&) {`
Do not allow dispatchers to be copied.

```
TOutDispatcher& operator=(const TOutDispatcher&) {return
      *this;
```
Do not allow dispatchers to be assigned.

`TOutProcessor::Map fProcessors`
Each dispatcher has a processor map.

`TOutObserver::Map fObservers`
Each dispatcher has an observer map.

`TOutEvolutionProcessor::Set fEvolutionProcessors`
Each dispatcher has evolution processors.

`TOutEventProcessor::Set fEventProcessors`
Each dispatcher has event processors.

`TOutSummaryProcessor::Set fSummaryProcessors`
Each dispatcher has summary processors.

`TOutFactory* fFactory`
Each dispatcher remembers the factory it uses.

2.2.2.2 TOutEventProcessor

An output event processor deals with conditions occuring in the simulation

Public access:

```
typedef set<TOutEventProcessor*, TOutProcessor::TLess> Set
typedef Set::iterator SetIterator
typedef Set::const_iterator SetConstIterator
```
Type definitions.

```
TOutEventProcessor(OutProcessorId id, const
      TOutGeneralSpecification& specification)
```
Construct an event processor.

```
virtual ~TOutEventProcessor()
```
Destroy an event processor

```
inline EProcessorType GetProcessorType() const
```
Return the processor type.

```
virtual void RecordOutput(REAL time, bool IsFinalTimeStep)
```
Begin recording output for this timestep.

```
virtual void RecordTraveler()
```
Finish recording output for a traveler.

```
inline TOutObserver& GetTravelerObserver()
inline const TOutObserver& GetTravelerObserver() const
```
Return the traveler observer.

```
inline void SetTravelerObserver(TOutObserver& observer)
```
Define the traveler observer.

```
TOutProcessor::StorageSet GetStorages()
```
Return the processor's storages.

Private access:

```
TOutObserver* fTravelerObserver
```
An event processor has a traveler observer.

```
TOutStorage fTravelerStorage
```
An event processor is connected to a traveler storage.

```
TOutRecord fTravelerRecordCache
```
Cache for unsuppressed fields in traveler output record

2.2.2.3 TOutEvolutionProcessor

An output evolution processor deals with evolving data such as that needed for animation, waterfall plots, etc.

Public access:

```
typedef set<TOutEvolutionProcessor*, TOutProcessor::TLess >
      Set
typedef Set::iterator SetIterator
typedef Set::const_iterator SetConstIterator
```
Type definitions.

```
TOutEvolutionProcessor(OutProcessorId, const
      TOutGeneralSpecification&)
```
Construct an evolution processor .

```
virtual ~TOutEvolutionProcessor()
```
Destroy an evolution processor.

```
inline EProcessorType GetProcessorType() const
```
Return the processor type.

```
virtual void RecordOutput(REAL time, bool IsFinalTimeStep)
```
Begin recording output for this timestep.

```
virtual void RecordNode()
```
Finish recording output for a node.

```
virtual void RecordLink()
```
Finish recording output for a link.

```
virtual void RecordVehicle()
```
Finish recording output for a vehicle.

```
virtual void RecordIntersection()
```
Finish recording output for an intersection.

```
virtual void RecordSignalCoordinator()
```
Finish recording output for a signal coordinator.

```
virtual void RecordSignalizedControl()
```
Finish recording output for a signalized control.

```
inline TOutObserver& GetNodeObserver()
inline const TOutObserver& GetNodeObserver() const
```
Return the node observer.

```
inline void SetNodeObserver(TOutObserver& observer)
```
Define the node observer.

```
inline TOutObserver& GetLinkObserver()
inline const TOutObserver& GetLinkObserver() const
```
Return the link observer.

```
inline void SetLinkObserver(TOutObserver& observer)
```
Define the link observer.

```
inline TOutObserver& GetVehicleObserver()
inline const TOutObserver& GetVehicleObserver() const
```
Return the vehicle observer.

```
inline void SetVehicleObserver(TOutObserver& observer)
```
Define the vehicle observer.

```
inline TOutObserver& GetIntersectionObserver()
inline const TOutObserver& GetIntersectionObserver() const
```
Return the intersection observer.

```
inline void SetIntersectionObserver(TOutObserver& observer)
```
Define the intersection observer.

```
inline TOutObserver& GetSignalCoordinatorObserver()
inline const TOutObserver& GetSignalCoordinatorObserver()
      const
```
Return the signal coordinator observer.

```
inline void SetSignalCoordinatorObserver(TOutObserver&
      observer)
```
Define the signal coordinator observer.

```
inline TOutObserver& GetSignalizedControlObserver()
inline const TOutObserver& GetSignalizedControlObserver()
      const
```
Return the signalized control observer.

```
inline void SetSignalizedControlObserver(TOutObserver&
      observer)
```
Define the signalized control observer.

```
TOutProcessor::StorageSet GetStorages()
```
Return the processor's storages.

Private access:

```
TOutObserver* fNodeObserver
```
An evolution processor has a node observer.

```
TOutObserver* fLinkObserver
```
An evolution processor has a link observer.

`TOutObserver* fVehicleObserver`
An evolution processor has a vehicle observer.

`TOutObserver* fIntersectionObserver`
An evolution processor has an intersection observer.

`TOutObserver* fSignalCoordinatorObserver`
An evolution processor has a signal coordinator observer.

`TOutObserver* fSignalizedControlObserver`
An evolution processor has a signalized control observer.

`TOutStorage fVehicleStorage`
An evolution processor is connected to a vehicle storage.

`TOutStorage fIntersectionStorage`
An evolution processor is connected to an intersection storage.

`TOutStorage fSignalStorage`
An evolution processor is connected to a signalized control storage.

`TOutRecord fVehicleRecordCache`
Cache for unsuppressed fields in vehicle output record

`TOutRecord fIntersectionRecordCache`
Cache for unsuppressed fields in intersection output record

`TOutRecord fSignalRecordCache`
Cache for unsuppressed fields in signal output record

2.2.2.4 TOutFactory

An output factory allocates and constructs new output objects.

Public access:

```
TOutFactory ()
```
Construct a factory.

```
virtual ~TOutFactory();
```
Destroy a factory.

```
virtual TOutEvolutionProcessor*
      NewEvolutionProcessor(OutProcessorId id,  const
      TOutGeneralSpecification& specification) = 0
```
Return a new evolution processor from the specification.

```
virtual TOutEventProcessor* NewEventProcessor(OutProcessorId
      id, const TOutGeneralSpecification& specification) = 0
```
Return a new event processor from the specification.

```
virtual TOutSummaryProcessor*
      NewSummaryProcessor(OutProcessorId id, const
      TOutGeneralSpecification& specification) = 0
```

Return a new summary processor from the specification.

```
virtual TOutLinkEvolutionObserver*
      NewLinkEvolutionObserver(OutObserverId id) = 0
```

Return a new link evolution observer.

```
virtual TOutVehicleObserver*
      NewVehicleObserver(OutObserverId id) = 0
```

Return a new vehicle observer.

```
virtual TOutNodeEvolutionObserver*
      NewNodeEvolutionObserver(OutObserverId id) = 0
```

Return a new node evolution observer.

```
virtual TOutIntersectionObserver*
      NewIntersectionObserver(OutObserverId id) = 0
```

Return a new intersection observer.

```
virtual TOutSignalCoordinatorEvolutionObserver*
      NewSignalCoordinatorEvolutionObserver(OutObserverId
      id) = 0
```

Return a new signal coordinator evolution observer.

```
virtual TOutSignalizedControlObserver*
      NewSignalizedControlObserver(OutObserverId id) = 0
```

Return a new signalized control observer.

```
virtual TOutTravelerObserver*
      NewTravelerObserver(OutObserverId id) = 0
```

Return a new traveler observer.

```
virtual TOutLinkSpaceObserver*
      NewLinkSpaceObserver(OutObserverId id) = 0
```

Return a new link space observer.

```
virtual TOutLinkTimeObserver*
      NewLinkTimeObserver(OutObserverId id) = 0
```

Return a new link time observer.

```
virtual TOutLinkVelocityObserver*
      NewLinkVelocityObserver(OutObserverId id, const
      TOutHistogramSpecification& specification) = 0
```

Return a new link velocity observer.

```
virtual TOutLinkEnergyObserver*
      NewLinkEnergyObserver(OutObserverId id, const
      TOutHistogramSpecification& specification) = 0
```

Return a new link energy observer.

2.2.2.5 TOutGeneralSpecification

The general specification defines the frequency and extent of data to be collected or retrieved in both space and time.

Public access:

```
static const REAL kMinusInfinity
static const REAL kPlusInfinity
```
Time constants.

```
TOutGeneralSpecification(const TConfiguration& config,  ring
      type, int index)
```
Construct a general specification from a configuration.

```
TOutGeneralSpecification(const TOutGeneralSpecification&
      specification)
```
Construct a copy of the specified specification.

```
TOutGeneralSpecification& operator=(const
      TOutGeneralSpecification& specification)
```
Make the specification a copy of the specified specification.

```
inline const string& GetRoot() const
```
Return the root for the specification.

```
inline const string& GetName() const
```
Return the name for the specification.

```
inline bool CollectForTime(REAL time) const
```
Return whether data should be collected for the specified time.

```
inline bool SampleForTime(REAL time) const
```
Return whether data should be sampled at the specified time.

```
inline bool IsAtStartTime(REAL time) const
```
Return whether the specified time is the start time.

```
inline REAL GetBoxLength() const
```
Return the box length.

```
inline bool CollectForPoint(const TGeoPoint& point) const
```
Return whether data should be collected for the specified point in space.

```
inline bool CollectForNode(NetNodeId id) const
```
Return whether data should be collected for the specified node.

```
inline bool CollectForLink(NetLinkId id) const
```
Return whether data should be collected for the specified link.

```
inline bool CollectForObserver(TOutObserver::EObserverType
      obs) const
```
Return whether data should be collected for the specified observer type.

```
inline const StringSet& GetSuppressed() const
```
Return the suppressed fields.

```
bool CollectFilteredRecord(const TOutRecord& record) const
```
Return whether the record passes the value filters.

Private access:

```
NetNodeIdSet ReadNodes(const string path) const
```
Return the nodes in the specification.

```
NetLinkIdSet ReadLinks(const string path) const
```
Return the links in the specification.

```
StringSet ParseSuppressed(const string value) const
```
Return the suppressed fields in the specification.

```
TOutValueFilter::Collection ParseFilters(const string value)
      const
```
Return the filters in the specification.

```
TOutGeneralSpecification::HistogramMap
      ParseObserverTypes(const string value, nst
      TConfiguration& config, string index) const
```
Return the observer types in the specification.

```
string fRoot
```
Each specification has a root.

```
string fName
```
Each specification has a name.

```
REAL fTimeMinimum
```
Each specification has a minumum time.

```
REAL fTimeMaximum
```
Each specification has a maximum time.

```
REAL fTimeStep
```
Each specification has a timestep.

```
REAL fTimeSample
```
Each specification has a time sample.

```
REAL fBoxLength
```
Each specification has a box length.

`TGeoRectangle fRegion`
Each specification has a collection region.

`NetNodeIdSet fNodes`
Each specification has a set of node ids.

`NetLinkIdSet fLinks`
Each specification has a set of link ids.

`StringSet fSuppressed`
Each specification has a set of suppressed fields.

`TOutValueFilter::Collection fFilters`
Each specification has a collection of filters for values.

`HistogramMap fHistogramSpecs`
Each specification has a map of histogram specifications.

2.2.2.6 TOutHistogram

A histogram records the number of occurences within intervals of the domain.

Public access:

`TOutHistogram()`
Default histogram constructor.

`TOutHistogram(const TOutHistogramSpecification& spec)`
Construct a histogram with a specification.

`TOutHistogram(const TOutHistogram& histogram)`
Construct a copy of the specified histogram.

`TOutHistogram& operator=(const TOutHistogram& histogram)`
Make the histogram a copy of the specified histogram.

Private access:

```
Return the histogram specification. ne const
      TOutHistogramSpecification&
      Histogram::GetHistogramSpecification() const
      Histogram::GetHistogramSpecification() const urn
      *fSpec
```
Make the histogram a copy of the specified histogram.

2.2.2.7 TOutHistogramSpecification

The histogram specification defines the parameters for a histogram.

```
Allow histogram quick access to specification parameters.
      iend class TOutHistogram
```

The histogram specification defines the parameters for a histogram.

Public access:

```
TOutHistogramSpecification(int bins, REAL maximum, REAL
      minimum = 0., ol underflow = false, bool overflow =
      false)
```

Construct a histogram specification.

```
TOutHistogramSpecification(const TOutHistogramSpecification&
      specification)
```

Construct a copy of the specified specification.

```
TOutHistogramSpecification& operator=(const
      TOutHistogramSpecification& specification)
```

Make the specification a copy of the specified specification.

```
ine int GetNumberBins() const
```

Return the number of bins.

Private access:

```
Return the number of bins. ne int
      TOutHistogramSpecification::GetNumberBins() const ne
      int TOutHistogramSpecification::GetNumberBins() const
      return fNumberBins
```

Return the number of bins.

2.2.2.8 TOutIntersectionObserver

An intersection observer observes data related to intersections.

Public access:

```
TOutIntersectionObserver(OutObserverId id)
```

Construct an intersection observer

```
virtual ~TOutIntersectionObserver()
```

Destroy an intersection observer.

Protected access:

`Define values for the fields being observed.  The Observe method will use these methods`
Destroy an intersection observer.

`inline void SetId(INT32 id)`
Define the vehicle's id.

`inline void SetLink(NetLinkId link)`
Define the id of the link the vehicle entered from.

`inline void SetLane(NetLaneNumber lane)`
Define the lane number the vehicle entered from.

`inline void SetNode(NetNodeId node)`
Define the id of the node the intersection is associated with.

`inline void SetQIndex(UINT8 index)`
Define the index of the vehicle's position in the queue.

2.2.2.9 TOutLinkEnergyObserver

A link energy observer summarizes vehicle energy data on a link.

Public access:

`TOutLinkEnergyObserver(OutObserverId id, const TOutHistogramSpecification& ec)`
Construct a link energy observer.

`virtual ~TOutLinkEnergyObserver()`
Destroy a link energy observer.

`inline void AddVehicle(REAL energy)`
Add a vehicle.

Protected access:

`inline bool IsInitialized() const`
Return whether the observer has been initialized.

`void ClearData(TOutHistogram& data)`
Clear the data for the observer.

`inline void SetLink(NetLinkId id)`
Set the link id.

`inline void SetNode(NetNodeId id)`
Set the departure node id.

```
void ReportObservations(TOutProcessor& processor,
      TOutHistogram& data)
```
Report the observations to the specified processor.

```
TOutHistogram fData
TOutHistogram fData; ivate
Each observer has a histogram specification. st
      TOutHistogramSpecification* fHistogramSpec
```
Each observer has histogram data.

2.2.2.10 TOutLinkEvolutionObserver

A link evolution observer observes evolving data related to links.

Public access:

```
TOutLinkEvolutionObserver(OutObserverId id)
```
Construct a link evolution observer.

```
virtual ~TOutLinkEvolutionObserver()
```
Destroy a link evolution observer.

2.2.2.11 TOutLinkSpaceObserver

A link space observer summarizes vehicle spatial data on a link.

Public access:

```
TOutLinkSpaceObserver(OutObserverId id)
```
Construct a link space observer.

```
virtual ~TOutLinkSpaceObserver()
```
Destroy a link space observer.

Protected access:

```
typedef vector<TOutTally> DatumCollection
typedef DatumCollection::iterator DatumCollectionIterator
typedef DatumCollection::const_iterator
      DatumCollectionConstIterator
```
Type definitions.

```
inline bool IsInitialized() const
```
Return whether the observer has been initialized.

```
void ClearData(DatumCollection &data)
```
Clear the data for the observer.

```
inline void SetLink(NetLinkId id)
```
Set the link id.

```
inline void SetNode(NetNodeId id)
```
Set the departure node id.

```
void SetLengths(UINT8 lanes, REAL linkLength, REAL
      boxLength, REAL cellLength = 0)
```
Set the number of lanes and link, box, and cell lengths.

```
inline void AddVehicle(NetLaneNumber lane, REAL distance,
      REAL velocity)
```
Add a vehicle.

```
void ReportObservations(TOutProcessor& processor,
      DatumCollection &data)
```
Report the observations to the specified processor.

```
DatumCollection fData
```
Each observer has box data.

Private access:

```
UINT8 fLaneCount
```
Each observer has a lane count.

```
REAL fLinkLength
```
Each observer has a link length.

```
REAL fBoxLength
```
Each observer has a box length.

```
Return whether the observer has been initialized. ne bool
      TOutLinkSpaceObserver::IsInitialized() const ne bool
      TOutLinkSpaceObserver::IsInitialized() const return
      !fData.empty()
```
Each observer has a box length.

2.2.2.12 TOutLinkTimeObserver

A link time observer records vehicle travel times on a link.

Public access:

```
TOutLinkTimeObserver(OutObserverId id)
```
Construct a link time observer.

```
virtual ~TOutLinkTimeObserver()
```
Destroy a link time observer.

```
inline void AddVehicle(NetLaneNumber lane, ENetTurnLabel
      turn, REAL time)
```
Add a vehicle.

Protected access:

```
class TMovement { public
```
A movement consists of an exit lane and a turn.

```
TMovement(NetLaneNumber lane = 0, ENetTurnLabel turn =
      kStraight) : fLane(lane), fTurn(turn) {
```
Construct an instance.

```
NetLaneNumber GetLane() const { return fLane
```
Return the exit lane number.

```
ENetTurnLabel GetTurn() const { return fTurn
```
Return the exit turn label.

```
bool operator<(const TMovement& movement) const { return
      fLane < movement.fLane || fLane == movement.fLane &&
      fTurn < movement.fTurn
ol  code(TBaseStream *Stream) const code(TBaseStream
      *Stream) const eturn (Stream->Encode((int &) fLane) &&
      Stream->Encode((int &) fTurn))
```
Compare movements.

Private access:

```
NetLaneNumber fLane
```
Each movement has an exit lane number.

```
ENetTurnLabel fTurn
}
```
Each movement has an exit turn label.

```
typedef map<TMovement, TOutTally, less<TMovement> > DatumMap
typedef DatumMap::iterator DatumMapIterator
typedef DatumMap::const_iterator DatumMapConstIterator
```
Type definitions.

```
inline bool IsInitialized() const
```
Return whether the observer has been initialized.

```
void ClearData(DatumMap &Data, TOutTally& veh)
```
Clear the data for the observer.

```
inline void SetLink(NetLinkId id)
```
Set the link id.

```
inline void SetNode(NetNodeId id)
```
Set the departure node id.

```
inline void SetMovement(NetLaneNumber lane, ENetTurnLabel
      turn)
```
Initialize a possible movement.

```
void ReportObservations(TOutProcessor& processor, DatumMap
      &Data, OutTally& veh)
```
Report the observations to the specified processor.

```
DatumMap fData
```
Each observer has movement data.

```
Return whether the observer has been initialized. ne bool
      TOutLinkTimeObserver::IsInitialized() const ne bool
      TOutLinkTimeObserver::IsInitialized() const return
      !fData.empty()
```
Each observer has movement data.

2.2.2.13 TOutLinkVelocityObserver

A link velocity observer summarizes vehicle velocity data on a link.

Public access:

```
TOutLinkVelocityObserver(OutObserverId id, const
      TOutHistogramSpecification& ec)
```
Construct a link velocity observer.

```
virtual ~TOutLinkVelocityObserver()
```
Destroy a link velocity observer.

Protected access:

```
typedef vector<TOutHistogram> DatumCollection
typedef DatumCollection::iterator DatumCollectionIterator
typedef DatumCollection::const_iterator
      DatumCollectionConstIterator
```
Type definitions.

```
inline bool IsInitialized() const
```
Return whether the observer has been initialized.

```
void ClearData(DatumCollection& data)
```
Clear the data for the observer.

```
inline void SetLink(NetLinkId id)
```
Set the link id.

```
inline void SetNode(NetNodeId id)
```
Set the departure node id.

```
void SetLengths(REAL linkLength, REAL boxLength, REAL
      cellLength = 0)
```
Set the link, box, and cell lengths.

```
inline void AddVehicle(REAL distance, REAL velocity)
```
Add a vehicle.

```
void ReportObservations(TOutProcessor& processor,
      DatumCollection& data)
```
Report the observations to the specified processor.

```
DatumCollection fData
```
Each observer has box data.

Private access:

```
REAL fLinkLength
```
Each observer has a link length.

```
REAL fBoxLength
```
Each observer has a box length.

```
Return whether the observer has been initialized. ne bool
      TOutLinkVelocityObserver::IsInitialized() const ne
      bool TOutLinkVelocityObserver::IsInitialized() const
      return !fData.empty()
```
Each observer has a box length.

2.2.2.14 TOutNodeEvolutionObserver

A node evolution observer observes evolving data related to nodes.

Public access:

```
TOutNodeEvolutionObserver(OutObserverId id)
```
Construct a node evolution observer.

```
virtual ~TOutNodeEvolutionObserver()
```
Destroy a node evolution observer.

2.2.2.15 TOutObserver

An observer converts data from the object to which it is attached into the generic form understood by the output subsystem.

Public access:

```
class TLess : public binary_function<TOutObserver*,
      TOutObserver*, bool> { public: bool
      operator()(TOutObserver *const& first, TOutObserver
      *const& second) { return first->GetId() < second-
      >GetId()
} }
```

Return whether the id of the first observer is less than the id of the second one.

```
typedef map<OutObserverId, TOutObserver*,
      less<OutObserverId> > Map
typedef Map::iterator MapIterator
typedef Map::const_iterator MapConstIterator
typedef set<TOutObserver*, TLess> Set
typedef Set::iterator SetIterator
typedef Set::const_iterator ConstIterator
```

Type definitions

```
m EObserverType {kNodeEvolutionObserver,
      kLinkEvolutionObserver, VehicleObserver,
      kIntersectionObserver,
      SignalCoordinatorEvolutionObserver,
      kSignalizedControlObserver, TravelerObserver,
      kLinkDensityObserver, kLinkTimeObserver,
      LinkVelocityObserver, kLinkEnergyObserver}
```

Observer types.

```
TOutObserver(OutObserverId id)
```

Construct an observer.

```
virtual ~TOutObserver()
```

Destroy an observer.

```
virtual void Observe(const void* object, const
      TOutProcessor& processor) = 0
```

An observer has an observe function for noting the values of data members of interest in the observed object. This virtual function in the basic representation must be overridden in the view.

```
inline OutObserverId GetId() const
```

Return the observer's id.

```
inline TOutRecord& GetRecord()
inline const TOutRecord& GetRecord() const
```
Return the associated record.

```
inline static OutObserverId GetNextId()
```
Return the next unused observer id.

Protected access:

```
inline void SetTime(REAL time)
```
Define the record's time.

Private access:

```
TOutObserver(const TOutObserver&) {} TOutObserver(const
      TOutObserver&) {
TOutObserver& operator=(const TOutObserver&) {return *this;
```
Do not allow observers to be copied.

```
OutObserverId fId
```
An observer has an id.

```
TOutRecord fRecord
```
An observer has an output record.

```
static OutObserverId fNextId
```
Keep track of the next observer id.

2.2.2.16 TOutProcessor

An output processor coordinates the processing of domain information into a domain-independent representation that is filtered and summarized before storage.

Public access:

```
class TLess : public binary_function<TOutProcessor*,
      TOutProcessor*, bool> { public: bool
      operator()(TOutProcessor *const& first, TOutProcessor
      *const& second) { return first->GetId() < second-
      >GetId()
} }
}
typedef map<OutProcessorId, TOutProcessor*,
      less<OutProcessorId> > Map
typedef Map::iterator MapIterator
typedef Map::const_iterator MapConstIterator
typedef set<TOutStorage*, less<TOutStorage*> > StorageSet
typedef StorageSet::iterator StorageSetIterator
typedef StorageSet::const_iterator StorageSetConstIterator
```
Return whether the id of the first processor is less than the id of the second one.

```
enum EProcessorType {kEvolutionProcessor, kEventProcessor,
      kSummaryProcessor}
kSummaryProcessor}
TOutProcessor(OutProcessorId id, const
      TOutGeneralSpecification& specification)
```
Processor types.

```
virtual ~TOutProcessor()
```
Destroy a processor.

```
virtual EProcessorType GetProcessorType() const = 0
```
Return the processor type.

```
virtual void RecordOutput(REAL time, bool IsFinalTimeStep) = 0
```
Begin recording output for this time.

```
virtual bool TerminateOutput(void)
```
Handle pending messages between of split summary records

```
virtual void FinishOutput(REAL time)
```
Finish recording output for this time.

```
inline OutProcessorId GetId() const
```
Return the processor's id.

```
static OutProcessorId GetNextId()
```
Return the next unused processor id.

```
inline TOutGeneralSpecification& GetGeneralSpecification ()
inline const TOutGeneralSpecification&
      GetGeneralSpecification() const
```
Return general specification.

```
virtual TOutProcessor::StorageSet GetStorages() = 0
```
Return the processor's storages.

Private access:

```
TOutProcessor& operator=(const TOutProcessor&) {return
      *this;
```
Do not allow processors to be copied. TOutProcessor(const TOutProcessor&) {} TOutProcessor(const TOutProcessor&) {} Do not allow processors to be assigned.

```
OutProcessorId  fId
```
A processor has an id.

```
TOutGeneralSpecification fGeneralSpecification
```
A processor has a general output specification.

```
static OutProcessorId fNextId
```
Keep track of next processor id.

```
Return the processor's id. ne OutProcessorId
      TOutProcessor::GetId() const ne OutProcessorId
      TOutProcessor::GetId() const return fId
```
Keep track of next processor id.

2.2.2.17 TOutRecord

This class is used for storing the values of a collection of fields.

Public access:

```
typedef TValue::EType Type
```
Field types.

```
TOutRecord()
```
Construct a record.

```
TOutRecord(const TOutRecord& record)
```
Make a copy of the given record.

```
TOutRecord& operator=(const TOutRecord& record)
```
Make the record a copy of the given record.

```
inline void SetField(const string& field, const TValue&
      value)
inline void SetField(const string& field, signed char value)
inline void SetField(const string& field, unsigned char
      value)
inline void SetField(const string& field, short value)
inline void SetField(const string& field, unsigned short
      value)
inline void SetField(const string& field, int value)
inline void SetField(const string& field, unsigned int
      value)
inline void SetField(const string& field, long value)
inline void SetField(const string& field, unsigned long
      value)
inline void SetField(const string& field, float value)
inline void SetField(const string& field, double value)
inline void SetField(const string& field, const string&
      value)
inline void SetField(const string& field)
```
Set the value of the specified field.

```
inline void GetField(const string& field, TValue& value)
      const
inline void GetField(const string& field, signed char&
      value) const
```

```
inline void GetField(const string& field, unsigned char&
      value) const
inline void GetField(const string& field, short& value)
      const
inline void GetField(const string& field, unsigned short&
      value) const
inline void GetField(const string& field, int& value) const
inline void GetField(const string& field, unsigned int&
      value) const
inline void GetField(const string& field, long& value) const
inline void GetField(const string& field, unsigned long&
      value) const
inline void GetField(const string& field, float& value)
      const
inline void GetField(const string& field, double& value)
      const
inline void GetField(const string& field, string& value)
      const
```

Get the value of the specified field.

```
inline Type GetType(const string& field) const
```

Return the type of the specified field.

```
inline TValue::Map& GetFields()
inline const TValue::Map& GetFields() const
```

Return the field map for the record.

```
inline TValue::MapConstIterator GetFieldIterator() const
```

Return an iterator for the record's map.

Private access:

```
TValue::Map fFields
```

Each record has a field map holding the current values of the fields.

2.2.2.18 TOutSignalCoordinatorEvolutionObserver

A signal coordinator evolution observer observes evolving data related to signal coordinators.

Public access:

```
TOutSignalCoordinatorEvolutionObserver(OutObserverId id)
```

Construct a signal coordinator evolution observer.

```
virtual ~TOutSignalCoordinatorEvolutionObserver()
```

Destroy a signal coordinator evolution observer.

2.2.2.19 TOutSignalizedControlObserver

A signalized control observer observes data related to signals.

Public access:

`TOutSignalizedControlObserver(OutObserverId id)`
Construct a signalized control observer.

`virtual ~TOutSignalizedControlObserver()`
Destroy a signalized control observer.

Protected access:

```
Define values for the fields being observed.   The Observe
      method will use these methods
```
Destroy a signalized control observer.

`inline void SetLink(NetLinkId link)`
Define the id of the link entering the intersection.

`inline void SetLane(NetLaneNumber lane)`
Define the lane number of the lane entering the intersection.

`inline void SetNode(NetNodeId node)`
Define the id of the node associated with the intersection.

```
inline void SetSignal(TNetTrafficControl::ETrafficControl
      control)
```
Define the current signal state.

2.2.2.20 TOutStorage

An output storage manages the distributed file system and isolates the rest of the simulation output objects from the details of the physical storage. Member functions throw the exception `TOutStorageFailure` when errors occur.

Public access:

`typedef FILE* HostHandle`
Type definitions.

```
static const long kBegin
static const long kEnd
```
Constants for seek positions.

`enum Mode {kRead, kWrite, kDelete}`
Storage modes: Use read mode for opening existing files, write mode for creating a new file, and delete mode for deleting existing files.

```
TOutStorage(const string& root, const string& name, Mode
      mode = kRead)
TOutStorage(const StringSet& hosts, const string& root,
      const string& name, Mode mode = kRead)
```
Connect the storage with the given root and basic name to the specified hosts.

```
~TOutStorage()
```
Disconnect the storage.

```
bool FinishOutputForTime(REAL Time)
```
Tell storage that all future outputs will have a time stamp > Time.

```
inline const string& GetRoot() const
```
Return the root name.

```
inline const string& GetName() const
```
Return the basic name.

```
StringSet GetHosts() const
```
Return the host names.

```
HostHandle GetHostHandle(const string& host) const
```
Return the host handle.

```
inline long GetOffset() const
inline long GetOffset(const string& host) const
long GetOffset(HostHandle host) const
```
Return the current offset for the specified host file.

```
inline bool AtEnd() const
inline bool AtEnd(const string& host) const
inline bool AtEnd(HostHandle host) const
```
Return whether an end-of-file has occured for the specified host file.

```
inline void Seek(long position = kBegin)
inline void Seek(const string& host, long position = kBegin)
void Seek(HostHandle host, long position = kBegin)
```
Position the specified host file to the given location.

```
inline void Write(const TOutRecord& record)
inline void Write(const string& host, const TOutRecord&
      record)
void Write(HostHandle host, const TOutRecord& record)
void Write(int id, const TOutRecord& record)
```
Write the given record on the specified host.

```
inline bool Read(TOutRecord& record)
inline bool Read(const string& host, TOutRecord& record)
bool Read(HostHandle host, TOutRecord& record)
```
Read the given record on the specified host. Return whether a record was available for reading.

```
inline void WriteHeader(const TOutRecord& record)
inline void WriteHeader(const string& host, const
      TOutRecord& record)
void WriteHeader(HostHandle host, const TOutRecord& record)
void WriteHeader(int id, const TOutRecord& record)
```
Write the given record header on the specified host.

```
inline bool ReadHeader(TOutRecord& record)
inline bool ReadHeader(const string& host, TOutRecord&
      record)
bool ReadHeader(HostHandle host, TOutRecord& record)
```
Read the given record header on the specified host. Return whether a record was available for reading.

```
void Flush()
```
Flush any pending operations.

```
inline static const string& GetDefaultHost()
```
Get the default host name.

```
inline static void SetDefaultHost(const string& host)
```
Set the default host name.

```
inline static void SetParallelIO(TParallelIO* pio)
```
Define parallel IO object.

```
inline void SetParallelIOId(int id)
```
Define parallel IO identifier.

```
inline static void SetTimeResolution(REAL time)
```
Define parallel IO time resolution.

Private access:

```
typedef map<const string, HostHandle, less<const string> >
      FileMap
typedef FileMap::iterator FileMapIterator
typedef FileMap::const_iterator FileMapConstIterator
```
Type definitions.

```
static string fgLocal
```
Local host name.

```
static const string fgSuffix
```
File suffix.

```
TOutStorage(const TOutStorage&) : fRoot(), fName() {
```
Do not allow storages to be copied.

```
TOutStorage& operator=(const TOutStorage&) {return *this;
```
Do not allow storages to be assigned.

`const string fRoot`
A storage has a root location.

`const string fName`
A storage has a basic name.

`FileMap fFiles`
A storage has a file on each host.

`static TParallelIO* fgParallelIO`
Pointer to object for doing parallel IO.

`int fParallelIOId`
Identifier for parallel IO.

`long fParallelCount`
Count of bytes written to parallel IO.

`static UINT32 fgTimeResolution`
Time resolution for parallel IO. 1/fgTimeResolution is the smallest time value that can be distinguished in the DataID sorting field for parallel IO

`static string fgFieldDelimeter`
Field delimeter for parallel IO.

2.2.2.21 TOutSummaryProcessor

An output summary processor collects statistics on the simulation.

Public access:

```
typedef set<TOutSummaryProcessor*, TOutProcessor::TLess> Set
typedef Set::iterator SetIterator
typedef Set::const_iterator SetConstIterator
```
Type definitions.

```
TOutSummaryProcessor(OutProcessorId id, const
      TOutGeneralSpecification& specification)
```
Construct a summary processor.

`virtual ~TOutSummaryProcessor()`
Destroy a summary processor.

`inline EProcessorType GetProcessorType() const`
Return processor type.

`virtual void RecordOutput(REAL time, bool IsFinalTimeStep)`
Begin recording output for this timestep.

`virtual void RecordSpace(const TOutObserver& observer)`
Finish recording output for link space.

`virtual void RecordTime(const TOutObserver& observer)`
Finish recording output for link time.

`virtual void RecordVelocity(const TOutObserver& observer)`
Finish recording output for link velocity.

`virtual void RecordEnergy(const TOutObserver& observer)`
Finish recording output for link energy.

```
inline TOutObserver::Set& GetSpaceObservers()
inline const TOutObserver::Set& GetSpaceObservers() const
```
Return the space observers.

```
inline TOutObserver::Set& GetTimeObservers()
inline const TOutObserver::Set& GetTimeObservers() const
```
Return the time observers.

```
inline TOutObserver::Set& GetVelocityObservers()
inline const TOutObserver::Set& GetVelocityObservers() const
```
Return the velocity observers.

```
inline TOutObserver::Set& GetEnergyObservers()
inline const TOutObserver::Set& GetEnergyObservers() const
```
Return the energy observers.

`inline void AddSpaceObserver(TOutObserver& observer)`
Define a space observer.

`inline void AddTimeObserver(TOutObserver& observer)`
Define a time observer.

`inline void AddVelocityObserver(TOutObserver& observer)`
Define a velocity observer.

`inline void AddEnergyObserver(TOutObserver& observer)`
Define a energy observer.

`inline void RemoveSpaceObserver(TOutObserver& observer)`
Undefine a space observer.

`inline void RemoveTimeObserver(TOutObserver& observer)`
Undefine a time observer.

`inline void RemoveVelocityObserver(TOutObserver& observer)`
Undefine a velocity observer.

`inline void RemoveEnergyObserver(TOutObserver& observer)`
Undefine a energy observer.

`TOutProcessor::StorageSet GetStorages()`
Return the processor's storages.

Private access:

`TOutObserver::Set fSpaceObservers`
Do not allow summary processors to be copied. TOutSummaryProcessor(const TOutSummaryProcessor&) {} TOutSummaryProcessor(const TOutSummaryProcessor&) {} Do not allow summary processors to be assigned. TOutSummaryProcessor& operator=(const TOutSummaryProcessor&) {return *this;} *this;} A summary processor has space observers.

`TOutObserver::Set fTimeObservers`
A summary processor has time observers;

`TOutObserver::Set fVelocityObservers`
A summary processor has velocity observers;

`TOutObserver::Set fEnergyObservers`
A summary processor has energy observers;

`TOutStorage fSpaceStorage`
An summary processor is connected to a space storage.

`TOutStorage fTimeStorage`
A summary processor is connected to a time storage.

`TOutStorage fVelocityStorage`
A summary processor is connected to a velocity storage.

`TOutStorage fEnergyStorage`
A summary processor is connected to a energy storage.

`TOutRecord fSpaceRecordCache`
Cache for unsuppressed fields in space output record

`TOutRecord fTimeRecordCache`
Cache for unsuppressed fields in time output record

`TOutRecord fVelocityRecordCache`
Cache for unsuppressed fields in velocity output record

`TOutRecord fEnergyRecordCache`
Cache for unsuppressed fields in energy output record

2.2.2.22 TOutTally

A tally keeps track of the statistics of a floating-point observation.

Public access:

`inline TOutTally()`
Construct an instance.

`inline TOutTally(const TOutTally& tally)`
Construct a copy.

`inline TOutTally& operator=(const TOutTally& tally)`
Copy an instance.

`inline TOutTally& operator+=(const TOutTally& tally)`
Add the values of another instance.

`inline void Clear()`
Clear the observations.

`inline void Add(REAL value)`
Add an observation.

`inline int GetCount() const`
Return the number of observations.

`inline REAL GetSum() const`
Return the sum of the observations.

`inline REAL GetSquares() const`
Return the sum of the squares of the observations.

Private access:

`int fCount`
Each tally keeps track of the number of observations.

`REAL fSum`
Each tally keeps track of the sum of the observations.

`REAL fSquares`
Each tally keeps track of the sum of the squares of the observations.

2.2.2.23 TOutTextWriter

A text writer puts simulation output data into a formatted text file. The exception `TOutWriterFailure` is thrown if an operation fails.

Public access:

```
TOutTextWriter(const string& file, const string& delimiter =
      "\t", bool includeHeader = false)
```

Open the specified file for writing, including the record header if requested.

```
void Write(const TOutRecord& record)
void Write(const TOutRecord& record, const StringCollection&
      fields)
```

Write the specified record with fields in the given order.

Private access:

```
ofstream fOut
```

Each text writer is connected to a file.

```
string fDelimiter
```

Each text writer has a delimiter.

```
bool fIncludeHeader
```

A text writer may have to include header information.

2.2.2.24 TOutTravelerObserver

A traveler observer observes data related to travelers.

Public access:

```
enum EAnomaly {kNone, kLost, kNoLink, kNoParking,
      kNoVehicle, kNoTransitStop, kNoBoard, kSkipStop,
      kNoLaneChange}
```

Anomaly types.

```
TOutTravelerObserver(OutObserverId id)
```

Construct a traveler observer.

```
virtual ~TOutTravelerObserver()
```

Destroy a traveler observer.

Protected access:

```
Define values for the fields being observed.   The Observe
      method will use these methods
```

Destroy a traveler observer.

`inline void SetId(TravelerId id)`
Define the traveler's id.

`inline void SetTrip(TripId trip)`
Define the id of the trip the traveler is on.

`inline void SetLeg(LegId leg)`
Define the id of the trip leg the traveler is on.

`inline void SetLocation(NetLocationId loc)`
Define the location (link or accessory id) the traveler is on.

`inline void SetVehicle(VehicleId id)`
Define the traveler's vehicle id. ISSUE(kpb) type should come from vehicle rep., currently in Plan.h

`inline void SetVehicleType(EVehType type)`
Define the traveler's vehicle type. ISSUE(kpb) type should come from vehicle rep., currently in NET/Id.h

`inline void SetVehicleSubtype(UINT8 type)`
Define the traveler's vehicle subtype ISSUE(kpb) type should come from vehicle rep.

`inline void SetRoute(RouteId route)`
Define the route id of a transit trip

`inline void SetStops(UINT8 n)`
Define the number of stop signs encountered so far on current trip leg.

`inline void SetYields(UINT8 n)`
Define the number of yield signs encountered so far on current trip leg.

`inline void SetSignals(UINT8 n)`
Define the number of signals encountered so far on current trip leg.

`inline void SetTurn(ENetTurnLabel turn)`
Define the type of turn made upon exiting the link.

`inline void SetStoppedTime(REAL time)`
Define the amount of time the traveler has spent stopped so far on current trip leg.

`inline void SetAccelTime(REAL time)`
Define the amount of time the traveler has spent accelerating from zero so far on current trip leg.

`inline void SetTotalTime(REAL time)`
Define the traveler's total time traveled.

```
inline void SetTotalDistance(REAL distance)
```
Define the traveler's total distance traveled.

```
inline void SetUser(UINT32 user)
```
Define the traveler's user defined field.

```
inline void SetStatus(UINT32 status)
```
Define the traveler's status.

```
inline void SetAnomaly(EAnomaly anomaly)
```
Define any traveler anomalies.

2.2.2.25 TOutValueFilter

A value filter tests whether a value meets a specified condition.

Public access:

```
enum EOperation {kEqual = 0, kNotEqual = 1, kLess = 2,
      kLessOrEqual = 3, kGreater = 4, kGreaterOrEqual = 5,
      kMultipleOf = 6,  kNotMultipleOf = 7, kIncludedIn = 8,
      kNotIncludedIn = 9,  kBitsSet = 10, kBitsCleared = 11}
```
There are several supported operations.

```
typedef list<TOutValueFilter> Collection
typedef Collection::iterator CollectionIterator
typedef Collection::const_iterator CollectionConstIterator
```
Type definitions.

```
TOutValueFilter(const string& field, EOperation operation,
      const TValue& object)
TOutValueFilter(const string& field, const string&
      operation, const TValue& object)
```
Construct an instance.

```
TOutValueFilter(const TOutValueFilter& filter)
```
Construct a copy of the specified filter.

```
TOutValueFilter& operator=(const TOutValueFilter& filter)
```
Make a copy of the specified filter.

```
bool Accept(const TValue& value) const
```
Return whether the specified value is accepted by the filter.

```
inline bool Accept(const TOutRecord& record) const
```
Return whether the specified record is accepted by the filter.

```
static EOperation GetOperation(const string& operation)
```
Return the operation corresponding to the specified string.

`static const string& GetOperation(EOperation operation)`
Return the string corresponding to the specified operation.

Private access:

`static void Initialize()`
Initialize the operation strings.

`static string fOperations[12]`
There are strings corresponding to each operation.

`string fField`
Each filter is applied to a field.

`EOperation fOperation`
Each filter applies an operation.

`TValue fObject`
Each filter has an object of comparison.

2.2.2.26 TOutVehicleObserver

A vehicle observer observes data related to vehicles.

Public access:

`TOutVehicleObserver(OutObserverId id)`
Construct a vehicle observer.

`virtual ~TOutVehicleObserver()`
Destroy a vehicle observer.

Protected access:

```
Define values for the fields being observed.   The Observe
      method will use these methods
```
Destroy a vehicle observer.

`inline void SetId(VehicleId id)`
Define the vehicle's id. ISSUE(kpb) type should come from vehicle rep., currently in Plan.h

`inline void SetLink(NetLinkId link)`
Define the id of the link the vehicle is on.

`inline void SetLane(NetLaneNumber lane)`
Define the lane number the vehicle is on.

`inline void SetDistance(REAL distance)`
Define the vehicle's distance from a node.

`inline void SetNode(NetNodeId node)`
Define the id of the node the distance is measured from.

`inline void SetVelocity(REAL velocity)`
Define the vehicle's velocity.

`inline void SetAcceleration(REAL acceleration)`
Define the vehicle's acceleration.

`inline void SetVehicleType(EVehType type)`
Define the vehicle's type. ISSUE(kpb) type should come from vehicle rep., currently in NET/Id.h

`inline void SetDriver(TravelerId id)`
Define the vehicle's driver.

`inline void SetPassengers(UINT32 count)`
Define the number of passengers in the vehicle.

`inline void SetEasting(REAL easting)`
Define the vehicle's easting field.

`inline void SetNorthing(REAL northing)`
Define the vehicle's northing field.

`inline void SetElevation(REAL elevation)`
Define the vehicle's elevation field.

`inline void SetAzimuth(REAL azimuth)`
Define the vehicle's orientation angle (degrees from east in counterclockwise direction).

`inline void SetUser(UINT32 user)`
Define the vehicle's user defined field.

2.2.2.27 TOutWriter

An output writer provides an interface for the external writing of data from simulation output. The exception `TOutWriterFailure` is thrown if an operation fails.

Public access:

```
virtual void Write(const TOutRecord& record) = 0
virtual void Write(const TOutRecord& record, const
      StringCollection& fields) = 0
```
Write the specified record.

Protected access:

`TOutWriter() {`
`ivate`
Construct a writer.

`TOutWriter(const TOutWriter&) {`
Do not allow writers to be copied.

2.2.2.28 TOutException

An output exception signals the failure of a simulation output subsystem function. Each exception has a message. Figure 16 shows the hierarchy of exception classes.

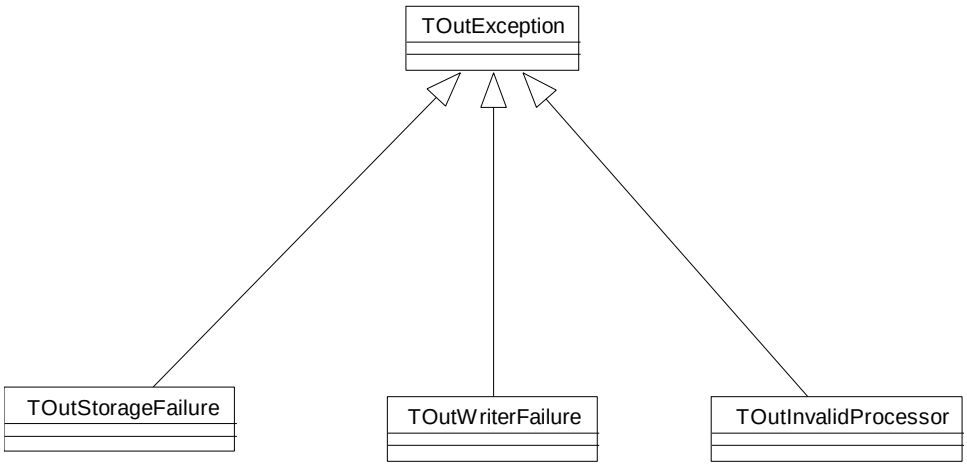

Figure 16: Exception hierarchy for the TRANSIMS simulation output subsystem (unified notation).

`TOutException(const string& message = “Simulation output error.”)`
Construct an exception with the specified message text.

`TOutException(const TOutException& exception)`
Construct a copy of the specified exception.

`TOutException& operator=(const TOutException& exception)`
Make the exception a copy of the specified exception.

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

`class TOutStorageFailure`
This exception is thrown when a storage operation fails.

`class TOutWriterFailure`
This exception is thrown when a writer operation fails.

`class TOutInvalidProcessor`
This exception is thrown when the processor type is invalid.

2.3 Implementation

2.3.1 C++ Libraries

The Booch Components [RW 94] provide C++ container classes that the simulation output subsystem uses extensively. The DBtools.h++ [SL 95; Su 95] and Tools.h++ [Ke 94] libraries provide platform-independent data type and file system support, respectively. The subsystem also uses the standard C++ library [Pl 95], the standard C library [Pl 92], and the POSIX library [Ga 95]. All of these libraries compile on a wide variety of platforms (UNIX and otherwise).

2.3.2 File System

Although the simulation output subsystem runs on multiple computational nodes (CPNs), it stores output data locally (Figure 17) and thus does not require any communication between the CPNs. A unified view of the data is provided during data retrieval by accessing and collating the data on multiple remote disks (Figure 18).

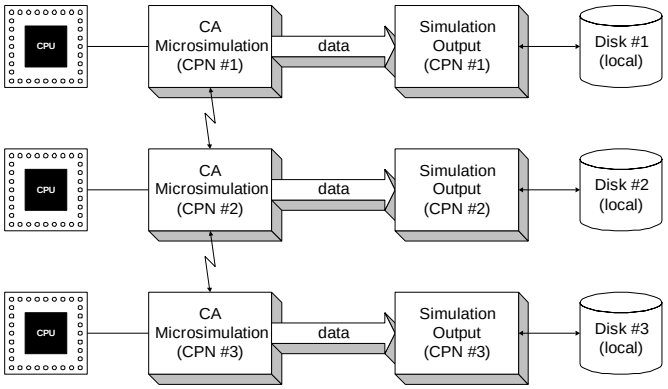

Figure 17: Simulation output subsystem configuration for the simultaneous data collection on multiple CPNs.

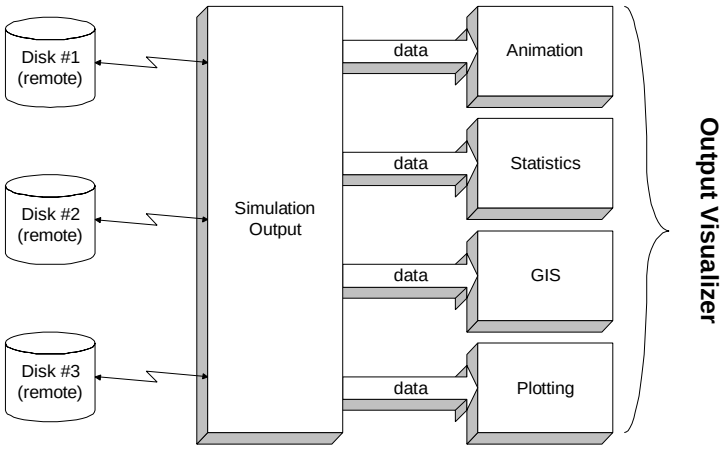

Figure 18: Simulation output subsystem configuration for data retrieval from multiple disks.

The local-storage/remote-retrieval paradigm requires coordination between the microsimulation software and the postprocessing software. The simplest way to accomplish this is to have each CPN store data locally into a directory named "local" on each CPN—this will, of course, be a different physical disk for each CPN. Each of these local directories is given a different NFS (network file system) name (typically the name of the CPN) so that it can be accessed remotely. Figure 19 illustrates this scheme. Several other workable arrangements are possible, however.

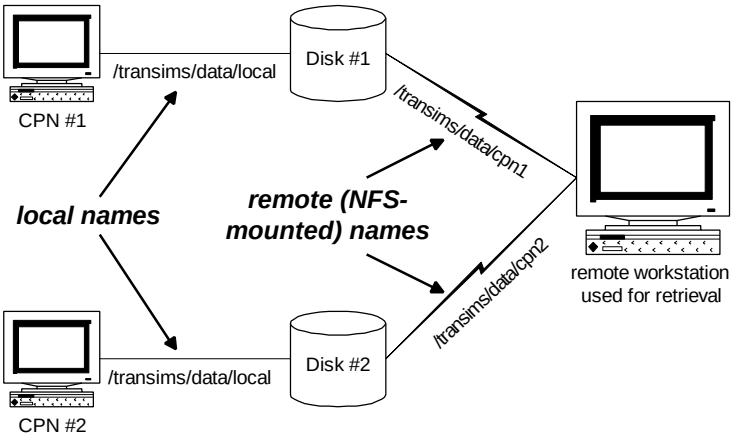

Figure 19: Typical naming and NFS mounting scheme for simulation output directories.

2.4 Integration into the Microsimulation

The simulation output subsystem must be integrated into the microsimulation by subclassing the appropriate processor (`TOut…Processor`) and observer (`TOut…Observer`) classes and by providing a subclass of `TOutFactory` for creating instances of these subclasses. This mechanism allows a flexible, yet tight, coupling between the two subsystems without requiring the simulation output subsystem to be tailored to the specifics of the microsimulation. Figure 20 illustrates an example of how this subclassing can be implemented.

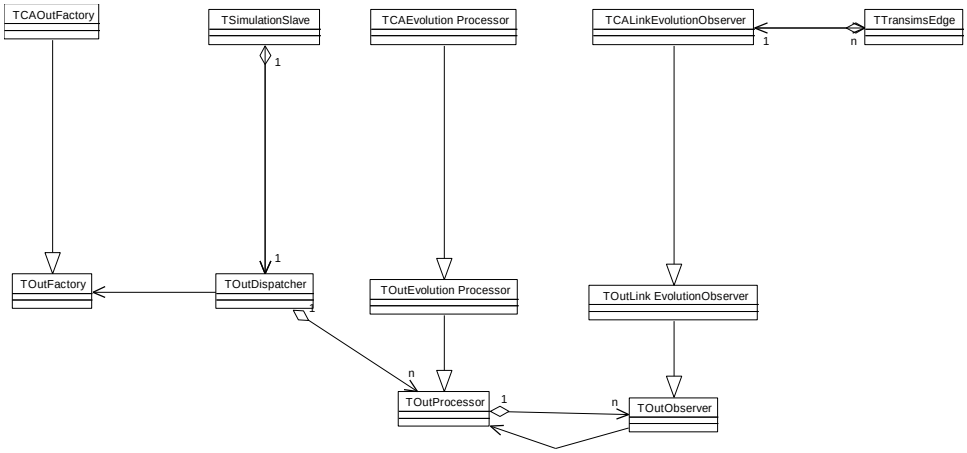

Figure 20: Example of subclassing simulation output classes in a simulation (unified notation).

2.5 Usage

2.5.1 Example of Retrieval Using C++

The following example shows how to access and retrieve evolution data using C++ calls to the simulation output subsystem.

```
//  Create a character buffer for lines from standard input.
char line[1000];

//  Get the file name for the vehicle text output file.
cin.getline(line, 1000);
TOutWriter* vehicleWriter = new TOutTextWriter(line, “\t”, TRUE);

//  Get the file name for the output text files.
cin.getline(line, 1000);
TOutWriter* intersectionWriter = new TOutTextWriter(line, “\t”,
        TRUE);
cin.getline(line, 1000);
TOutWriter* signalWriter = new TOutTextWriter(line, “\t”, TRUE);
TDbDirectory directory(TDbDirectoryDescription(“IOC-1”));

//  Open the data sources.
TDbSource generalSource(directory,
        directory.GetSource(“Output Specification”));
TDbSource nodeSource(directory,
            directory.GetSource(“Output Node Specification”));
TDbSource linkSource(directory,
            directory.GetSource(“Output Link Specification”));

//  Get the data table names and open the data tables.
cin.getline(line, 1000);
TDbTable generalTable(generalSource,
        generalSource.GetTable(line));
cin.getline(line, 1000);
TDbTable nodeTable(nodeSource, nodeSource.GetTable(line));
cin.getline(line, 1000);
TDbTable linkTable(linkSource, linkSource.GetTable(line));

//  Create the specification.
TOutGeneralSpecificationReader
        reader(TOutSpecificationReader(generalTable,
        nodeTable, linkTable));
reader.Reset();

//  Get the set of hosts.
TOutStorage::HostSet hosts(HashValue);
for (cin.getline(line, 1000); !cin.eof(); cin.getline(line, 1000))
    hosts.Add(line);

//  Create the retriever.
TOutEvolutionRetriever retriever(hosts,
        TOutGeneralSpecification(reader));

//  Retrieve the data.
retriever.Retrieve(*vehicleWriter, *intersectionWriter,
        *signalWriter, hosts.Extent() > 1);
```

```
//  Destroy the writers.
delete vehicleWriter;
delete intersectionWriter;
delete signalWriter;
```

2.6 Future Work

Future work planned for the TRANSIMS simulation output subsystem will focus on several areas:

- collecting new types of data such as turn counts, fundamental diagrams, and velocity-acceleration histograms,
- enhancing the performance of the subsystem by supporting data compression, indexing, sorting, and filtering,
- improving the DumpStorage utility to provide more flexibility in exporting data, and
- eliminating the dependence on commercial products such as DBtools.h++ and Tools.h++.

2.7 References

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

[Ke 94] T. Keffer, *Tools.h++ Introduction and Reference Manual*, Version 6, (Corvallis, Oregon: Rogue Wave Software, 1994).

[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).

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

[SL 95] S. Sulsky and K. L. Lohn, *DBtools.h++ User's Guide and Tutorial*, Version 1, (Corvallis, Oregon: Rogue Wave Software, 1995).

[Su 95] S. Sulsky, *DBtools.h++ Class Reference*, Version 1, (Corvallis, Oregon: Rogue Wave Software, 1995).

3. PARALLEL TOOLBOX

3.1 Configuration Group MPI

`#define` **`Configuration Group MPI`**

The configuration group [MPI] understands the following names:

- **SPIN=N** – Activate (N=1) or deactivate (N=0) spinning while waiting for messages. When spinning is active, the processes waiting for messages will not sleep in system interrupts (using `MPI_Probe`) but call `MPI_Iprobe` repeatedly until a message arrives.
- **MaxBuffer=N** – Use N message buffers. This is the maximum number of active messages that can be handled simultaneously.
- **BufferSize=N** – Set size of each buffer to N bytes.
- **SendBufferSize=N** – Set send buffer size to N bytes. The buffer is used for all messages when `SendBufferSize>0`. Messages will be sent using `MPI_BSend` instead of `MPI_Send`.
- **ImmediateSend=N** – Activate (N=1) or deactivate (N=0) immediate sending of messages. Messages will be sent using `MPI_Isend` instead of `MPI_Send`.

3.2 Configuration Group PVM

`#define` **`Configuration Group PVM`**

The configuration group [PVM] understands the following names:

- **RoutingMode=N** – Set the PVM routing mode to the specified value. See pvm3.h for values. Warning: this value should not be set unless it is clear that modes other than the default `PvmAllowDirect` work.
- **Hostfilename="filename"** – Use hostfile `filename`.
- **Encoding=N** – Set encoding mode. Choose "0" for default encoding and "1" for raw encoding. When raw encoding is chosen, there is no way to communicate between architectures that have different representations of base types.
- **DebugSlaves=N** – If active, (N=1) starts the slaves in debug mode using a terminal and a `dbx` (or `gdb`) process.

3.3 Configuration Group Parallel

`#define` **`Configuration Group Parallel`**

The configuration group [Parallel] understands the following names:

- **BroadcastConfig** – Triggers broadcast of complete configuration to all slaves during `TParallel::Activate`.

3.4 TEncodingMode

`(ID, Bit, Mask) enum` **`TEncodingMode`**

Coding modes.

3.4.1 EncodingRaw

Do not perform any coding or decoding. This should be the fastest way to encode messages. However, message will not be able to be sent between architectures having different representations of base types.

3.4.2 EncodingXDR

Code values using XDR. This allows communication between architectures with different representations of base types.

3.5 TTopology Type

`enum` **`TTopologyType`**

Typology types.

3.5.1 SimpleTopology

Topology of the `TTopology` base class. Not really used since the default topology assumed is the `BusTopology`.

3.5.2 BusTopology

Bus topology. Only a linear ranking is provided.

3.6 TCPNID

`typedef long` **`TCPNID`**

Type for storing a CPN ID. Each process will have a unique CPN ID.

3.7 THostID

`typedef long` **`THostID`**

Type for storing a host ID. All processes running on the same host will have the same host ID.

3.8 TMessageID

`typedef long` **`TMessageID`**

Type for storing a message ID. Unique ID (on each CPN) generated for each outgoing message.

3.9 TMessageTag

`typedef long` **`TMessageTag`**

Type for storing a message tag. This is usually a bit-coded value with a single bit set designating the class by which the message should be received.

3.10 TMessageType

`typedef long` **`TMessageType`**

Type for storing a message type. User-defined value that can be used to select a subset of messages.

3.11 TMessageFlags

`typedef long` **`TMessageFlags`**

Type for storing message flags. These flags define how flags are handled by `TCommunication`.

3.12 TArchCode

`typedef long` **`TArchCode`**

Type for storing the architecture code of a computer system. Architectures that are binary-compatible, that is, they can exchange binary data without conversion, will have the same code.

3.13 NULL_EVENT_ID

`#define` **`NULL_EVENT_ID`**

Predefined event ID that is used to mark the last event of a message.

3.14 MSG_FLAG_EVENT

`#define` **`MSG_FLAG_EVENT`**

Message flag: the message contains a series of events (default).

3.15 MSG_FLAG_AUTO

`#define` **`MSG_FLAG_AUTO`**

Message flag: the methods `BeginSend` and `EndSend` are called automatically for an outgoing message (default).

3.16 MSG_FLAG_DELETE

`#define` **`MSG_FLAG_DELETE`**

Message flag: the message is deleted automatically after it has been committed (delete).

3.17 TBaseMessage

`class` **`TBaseMessage`** `: public TBaseStream, public TDLNodePrimitive`

Base class to all message classes.

3.17.1 TBaseMessage

`TBaseMessage` `(TCommunicationControl *theControl, TCPNID theSender)`

Default constructor for an incoming message. Note: this constructor should never be called directly because the class `TCommunication` handles all incoming messages. Use the method `TCommunication::GetNewMessage` to create a new message instance.

3.17.2 TBaseMessage

`TBaseMessage` `(TCommunicationControl *theControl, TMessageID theID, TCPNID theSender, TDestination *theDestination, TMessageTab theTag, TMessageType theType, TMessageFlags theFlags)`

Default constructor for an outgoing message. Note: this constructor should never be called directly because the class `TCommunication` handles all outgoing messages.

3.17.3 Flush

`virtual bool` **`Flush`** `(void)`

Discard all remaining data in an incoming message. If this method is not called for messages that are completely decoded, there will be a warning when the message is destroyed.

3.17.4 EventPending

`virtual bool` **`EventPending`** `(TEventID &anEventID)`

Decode an event ID from the message. If there are no event IDs left, this method sets `anEventID` to `NULL_EVENT-ID`.

Return Value: TRUE, if successful; FALSE, otherwise.

3.17.5 GetTag

`inline TMessageTag` **`GetTag`** `(void) const`

Returns the tag of the message. This value was given in `TCommunication::GetNewMessage`.

3.17.6 GetType

`inline TMessageType` **`GetType`** `(void) const`

Returns the type of the message. This value was given in `TCommunication::GetNewMessage`.

3.17.7 BeginReceive

`virtual bool` **`BeginReceive`** `(void)`

Decode the header of an incoming message. This method decodes the members `ID, Tag, Type, Flags`. It may be overwritten by descendent classes, which may require additional information to be decoded. It should be called at the very beginning of the overwritten method.

Return Value: TRUE, if successful; FALSE, otherwise.

3.17.8 BeginSend

`virtual bool` **`BeginSend`** `(void)`

Encode the header of an outgoing message. This method encodes the members `ID, Tag, Type, Flags`. It may be overwritten by descendent classes, which may require additional information to be encoded. It should be called at the very beginning of the overwritten method.

Return Value: TRUE, if successful; FALSE, otherwise.

3.17.9 GetClassName

`virtual TString` **`GetClassName`** `(void) const`

Returns the logical class name. Should be overwritten by descendent class.

3.17.10 Deselect

`virtual bool` **`Deselect`** `(void)`

Temporarily deactivate a message. PVM has problems handling multiple message buffers without explicitly switching between them. `Deselect` should be called whenever another message is received or sent while still processing an old one. Use `Select` to activate a message again. This is done automatically for event messages before the `HandleEvent` method is called for each event.

3.17.11 EndReceive

`virtual bool` **`EndReceive`** `(void)`

Clean up after receiving a message. May be overwritten by descendent class.

3.18 TBusTopology

`class` **`TBusTopology`** `: public TTopology`

Topology class representing clusters coupled by a fixed bandwidth bus.

3.19 TCPNInfo

`class` **`TCPNInfo`** `: public TBaseObject, public TDLNodePrimitive`

Information about one CPN.

3.19.1 ID

`TCPNID` **`ID`**

Unique ID of CPN in parallel topology. For MPI, this is a number of 0 through the number of CPNs minus one. For PVM, it is the ID returned by the PVM daemon.

3.20 TCommunication

`(MSG_FLAG_EVENT – MSG_FLAG_AUTO – MSG_FLAG_DELETE) class` **`TCommunication`**
`: public TBaseObject`

Class handling messages. All messages are sent and received through using this class.

3.20.1 Activate

`bool` **`Activate`** `(void)`

Activate the communication class. This method must be called before a message can be sent.

Return Values: TRUE, if successful.

3.20.2 ActivateTags

`void` **`ActivateTags`** `(TMessageTag ActivateMask)`

Activates a subset of message tags. Only messages containing tags that have been activated through this method will be processed.

Parameters: **ActivateMask** – bit mask of tags to be activated.

3.20.3 CheckCapabilities

`bool` **`CheckCapabilities`** `(long RequiredCapabilities) const`

Check capabilities of communication subsystem.

Return Value: TRUE, if all required capabilities are available.

Parameters: `RequiredCapabilities` – bit mask of required capabilities. The masks are defined in file ParallelTypes.h.

3.20.4 Commit

`bool` **`Commit`** `(TBaseMessage *&aMessage)`

Finish encoding and send a message. Sends a message that had been created using the `GetNewMessage` method.

Return Value: TRUE, if successful.

3.20.5 DeactivateTags

`void` **`DeactivateTags`** `(TMessageTag DeactivateMask)`

Deactivates a subset of message tags. Only messages containing tags that have been activated through this method will be processed.

Parameters: `DeactivateMask` – bit mask of tags to be deactivated.

3.20.6 DestroyMessage

`bool` **`DestroyMessage`** `(TBaseMessage *&aMessage) const`

Dispose of a message that was received through `ReceiveMessage`. Be sure to call this method always. Otherwise, some message buffers may never be deallocated! Note: Messages that are handled by a call to `MessageLoop` or similar methods need not be destroyed by this method!

3.20.7 GetNewMessage

```
bool GetNewMessage (TBaseMessage *&aMessage, TDestination
theDestination, TMessageTag theTag, TMessageType theType, TMessageFlags
theFlags = MSG_FLAG_DEFAULT)
```

Creates and returns a new outgoing message.

Return Value: TRUE, if successful.

Parameters: `aMessage` – reference to pointer that will hold the message. This message will have to be sent using the Commit method.

`theTag` – selects a specific object on the receiving side that will receive the message.

`theType` – can be used on the receiving side to select messages of this type.

`theFlags` – a bit-coded integer that will define how the message is handled by TCommunication.

3.20.8 MessageLoop

```
bool MessageLoop (TMessageSelector &Selector, bool Block = TRUE)
```

Wait for incoming messages. This method will wait for incoming messages and pass them to the registered message recipients.

Return Value: TRUE, if successful.

Parameters: `Selector` – can be used to select a subset of messages to be received.

`Block` – if TRUE, return only if message has been received.

3.20.9 ReceiveMessage

```
bool ReceiveMessage (TBaseMessage *&Message, TDestination theSender,
TMessageTag theTag, TMessageType theType)
```

Receive a message explicitly. This method waits for a message matching the given specifications. Note that all other (non-matching) messages will be queued, which may result in an overflow. If this happens, you may want to change the setting of [PVM].MaxBuffer.

3.20.10 RegisterRecipient

```
bool RegisterRecipient (TMessageTag theTagMask, TMessageRecipient
*theRecipient)
```

Register a class instance that will receive messages. Each class instance that will receive messages has to be registered using this method.

Parameters: `theTagMask` – bit mask defining which message tags should be passed to this class.

`theRecipient` – pointer to the class instance that will receive the message.

3.20.11 SendEvent

```
bool SendEvent (TEventID EventID, TDestination Destination, TMessageTag
Tag, TMessageType Type, TMessageFlags Flags = MSG_FLAG_DEFAULT)
```

Shortcut to send a message consisting of a single event. There is no additional data enclosed.

3.21 TCommunicationControl

```
class TCommunicationControl : public TBaseObject
```

Base class for the low-level communication drivers. Most of its methods are virtual hooks for its dependent classes. None of its methods should be called directly. Use the equivalent methods of `TCommunication` instead.

3.21.1 Activate

```
virtual bool Activate (void)
```

Activate the low-level communication protocol. Must be overloaded.

3.21.2 GetCapabilities

```
virtual long GetCapabilities (void) const
```

Return a bit-coded integer of the capabilities of the low-level driver. Must be overloaded.

3.21.3 GetNewMessage

```
virtual TBaseMessage* GetNewMessage (TMessageID theID, TCPNID theSender,
TDestination *theDestination, TMessageTag theTag, TMessageType theType,
TMessageFlags theFlags)
```

Return a new instance of the message class of the low-level driver for sending. Must be overloaded. The message class must be derived from `TBaseMessage`. In case of an error, the value 0 must be returned.

3.21.4 ReceiveMessage

```
virtual bool ReceiveMessage (TBaseMessage *&theMessage, long TimeOut)
```

Receive a message and return a pointer to a message class in `theMessage`. If no message is available, this method will block until a message is received. If `TimeOut>0`, the low-level driver will try to return after as many milliseconds, even if no message is available.

3.22 DEST_ALL

```
#define DEST_ALL (-3)
```

```
Predefined destination: send to all CPNs (including sender)
```

3.23 DEST_ALL_BUT_MYSELF

```
#define DEST_ALL_BUT_MYSELF (-4)
```

Predefined destination: send to all CPNs (excluding sender)

3.24 TDestination

```
class TDestination : public TBaseObject
```

```
Class defining the recipients of an outgoing message.
```

3.24.1 TDestination

`TDestination (TCPNID theRecipient)`

Create a destination with a single recipient.

Parameters: `theRecipient` – can be any valid TCPNID found in a `TCPNInfo` or one of the predefined destination values.

3.25 TEventHandlerPrimitive

`class TEventHandlerPrimitive : public TBaseObject`

Class for optimized event handling (without 'case statement').

3.26 THostInfo

`class THostInfo : public TBaseObject, public TDLNodePrimitive`

Class containing information about a host.

3.26.1 ArchCode

`long ArchCode`

Code identifying the binary data coding. If the ArchCodes of two hosts are equal, they can exchange binary data without having to convert.

3.26.2 Performance

`double Performance`

Integer defining the performance of the host. Larger values designate better performance.

3.27 TMPIBuffer

`class TMPIBuffer : public TBaseObject, publicTDLNodePrimitive`

Class holding the buffer for one MPI message.

3.28 TMPIMessage

`class TMPIMessage : public TBaseMessage`

Contains the MPI interface for a message.

3.29 TMPIMessageControl

`class` **`TMPIMessageControl`** `: public TCommunicationControl`

Class providing the low-level functionality of MPI for `sendinfo` and receiving messages.

3.30 TMPIParallelControl

`class` **`TMPIParallelControl`** `: public TParallelControl`

Class providing the low-level functionality of MPI for application control.

3.31 TMessageRecipient

`class` **`TMessageRecipient`**

Base class for classes that are to receive messages.

3.31.1 HandleEvent

`virtual bool` **`HandleEvent`** `(TEventID theEventID, TBaseMessage *theMessage, int &Handled)`

Stub to handle an event encoded in the message. Each event found in a message received by `TCommunication` is passed to this method, if the message tags match those used when registering this recipient.

Return Value: TRUE, if successful.

Parameters: `theEventID` – event ID of the current event that is to be decoded from the message.

`aMessage` – reference to an incoming message.

`Handled` – must be set to FALSE if an error has occurred while processing the message; TRUE if the processing was successful; or EXIT_LOOP if the processing was successful and the calling message loop is to be exited.

3.32 TMessageSelector

`(-2) class` **`TMessageSelector`**

Class defining a subset of messages to be received

3.32.1 TMessageSelector

```
inline TMessageSelector (TCPNID Sender = SOURCE_ALL, TMessageType
FirstType = MTYPE_ALL, TMessageType SecondType – MTYPE_NONE)
```

Constructor.

Parameters: `Sender` – if given, restricts the message to that CPN.

`FirstType` – if given, restricts the messages to that type.

`SecondType` – if given, restricts the messages to those of the first type or the second type.

3.32.2 IsMessageActive

```
inline bool IsMessageActive (TBaseMessage *Msg) const
```

Check if message fulfills current selector requirements.

Parameters: TRUE – if it does.

3.32.3 SetActiveTagMask

```
inline void SetActiveTagMask (TMessageTag theTagMask)
```

Restrict messages to a certain tag mask.

Parameters: `theTagMask` – tags to be accepted.

3.32.4 SetMaxNrOfNonPendingMessages

```
inline void SetMaxNrOfNonPendingMessages (int MaxNrOfMessages)
```

Receive, at most, a certain number of pending messages. Pending messages are those that were queued before a receiving method (e.g., `TCommunication::MessageLoop`) was called.

3.33 TPVMMessage

```
class TPVMMessage : public TBaseMessage
```

Contains the PVM interface for a message.

3.34 TPVMMessageControl

```
class TPVMMessageControl : public TCommunicationControl
```

Class providing the low-level functionality of PVM for sending and receiving messages.

3.35 TPVMParallelControl

`class` **`TPVMParallelControl`** `: public TParallelControl`

Class providing the low-level functionality of PVM for application control.

3.36 TParallel

`class` **`TParallel`** `: public TBaseObject, pubic TMessageRecipient`

High-level parallel control class.

3.36.1 AddAllCPN

`bool` **`AddAllCPN`** `(void)`

Add all available CPNs on all hosts to the parallel topology.

Return Value: TRUE, if successful.

3.36.2 AddCPN

`bool` **`AddCPN`** `(THostInfo *aHostInfo, int NrOfInstances)`

Add NrOfInstances CPNs of a specific host to the parallel topology.

Return Value: TRUE, if successful.

3.36.3 AddHostInfo

`bool` **`AddHostInfo`** `(THostInfo *HostInfo)`

Add information about a specific host. Note: all host information should be supplied before the first call to either `AddAllCPN` or `AddCPN` since otherwise, default values will be used for the hosts.

Return Value: TRUE, if successful.

3.36.4 AddMaxCPN

`bool` **`AddMaxCPN`** `(THostInfo *aHostInfo)`

Add all CPNs of a specific host to the parallel topology.

Return Value: TRUE, if successful.

3.36.5 AddSomeCPN

`bool` **`AddSomeCPN`** `(int NrOfInstances)`

Add a certain number of CPNs to the parallel topology using the hosts, one after another.

Return Value: TRUE, if successful.

3.36.6 Activate

`bool` **`Activate`** `(void)`

Prepare the activation of the parallel environment.

Return Value: TRUE, if successful.

3.36.7 GetUniqueID

`inline long` **`GetUniqueID`** `(long OtherID)`

Return a unique systemwise wide ID giving the locally unique ID `OtherID`. `OtherID` may use only its 19 lower-significant bits.

3.37 TParallelControl

`class` **`TParallelControl`** `: public TBaseObject, public TMessageRecipient`

Low-level parallel control class.

3.38 Configuration Group ParallelDemo

`#define` **`Configuration Group ParallelDemo`**

The Parallel Demo handles the following configuration items:

- RandomValues = N – sets the number of random values used in part 4.
- ShowValues – will print all values on each CPN and their sums on the master.

3.39 TParallelDemo

`class` **`TParallelDemo`** `: public TBaseObject,public TMessageRecipient`

Main class of the ParallelDemo. The Parallel Demo is meant as a small test program for the new release of the Parallel Toolbox. It does the following:

1) Allocate as many CPNs as possible.

2) The first CPN serves as master (and a slave); all others as slaves.

3) The master sends out a broadcast message that is acknowledged by all slaves (part 1)

4) The master sends out a message in a clockwise loop through all CPNs until it ends up on the master again. The master sends out a message in an anti-clockwise loop through all CPNs until it ends up on the master again (part 2).

5) If the demo detects a two-dimensional grid as underlying communication topology, the master sends out a broadcast to all row masters (all CPNs that have the same row index as the master). The row masters forward the broadcast to all CPNs in their column, which return an acknowledgment. The row masters collect the acknowledgments and send an acknowledgment to the master.

3.39.1 TParallelDemo

`TParallelDemo (int argc, char *argv[])`

Constructor. Creates instances of classes `TEnvironment`, `TCommunication`, and `TParallel`.

3.39.2 ~TParallelDemo

`virtual ~TParallelDemo (void)`

Destructor. Destroy instances of classes `TEnvironment`, `TCommunication`, and `TParallel`.

3.39.3 Run

`int Run (void)`

Main method of the class (called by main()). This method does the following:

1) Activate the local instances of `TEnvironment, TCommuniation`, and `TParallel`. This triggers a lot of technical stuff. Among the more interesting things: the standard configuration file `USER/application/ParallelDemo/Config` is evaluated.

2) Register the local `TParallelDemo` instance as recipient of events. We use eight events to control communication between the CPNs. As soon as an event is received by `TCommunication`, it is passed to the `HandleEvent` method.

3) *Master only*: Try to allocate as many CPNs as possible on the given host set. Output some information about hosts and CPNs.

4) Start parallel machine.

5) Retrieve information about the physical topology of the underlying communication network. It defaults to a simple bus topology but can be switched to a two-dimensional grid topology using configuration settings.

6) Output more information about our location in the physical topology and about who our neighbors are.

7) *Master only*: Call control method `RunMaster`, which executes the three parts of the demo.

8) Shut down the parallel machine.

3.39.4 TParallelDemo::HandleAnswer

`bool` **`TParallelDemo::HandleAnswer`** `(TBaseMessage*Message, int &Handled)`

Count acknowledgments from slaves. 'HandleAnswer' is only called for the master. Each time it is called, it decreases the number of missing acknowledgments by one.

3.39.5 TParallelDemo::HandleAnswerInCols

`bool` **`TParallelDemo::HandleAnswerInCols`** `(TBaseMessage *, int &Handled)`

Count each acknowledgment from slaves. 'HandleAnswer' is only called for the master. Each time it is called, it decreases the number of missing acknowledgments by one.

3.40 TProfileEvent

`enum` **`TProfileEvent`**

Predefined event types available.

3.41 TProfile

`class` **`TProfile`**

Class providing the interface to the MPI profiling library.

3.42 TRomioBufferInfo

`class` **`TRomioBufferInfo`**

Class providing the interface to the MPI ROMIO I/O subsystem.

3.43 TOPOLOGY_MY_RANK

`#define` **`TOPOLOGY_MY_RANK`** `(-0xFFFF)`

Predefined rank that evaluates to the rank of the CPN where it is used.

3.44 TTopology

`class` **`TTopology`** `: public TBaseObject`

Base class for all topology classes. It defines a trivial rank of 0 to the number of CPNs minus 1 by sorting the CPN ID. The master process usually has rank 0. A wrap-around is assumed for the ranks. So, -1 is equivalent to MaxRank+1 and vice versa.

3.44.1 GetNext

```
const TCPNInfo* GetNext (const TCPNInfo *aCPNInfo = 0) const
```

Return the info for CPN, which has a rank by one higher than `aCPNInfo`. If 0 is passed, the local CPN is assumed as reference.

3.44.2 GetPrev

```
const TCPNInfo* GetPrev (const TCPNInfo *aCPNInfo = 0) const
```

Return the info for CPN which has a rank by one lower than `aCPNInfo`. If 0 is passed, the local CPN is assumed as reference.

3.44.3 GetRank

```
virtual int GetRank (const TCPNInfo *aCPNInfo = 0) const
```

Get rank of CPN. If 0 is passed, the local CPN is assumed as reference.

3.44.4 GetType

```
virtual TTopologyType GetType (void) const
```

Return the type of the topology. There are several child classes for topology. Use this type to cast the topology pointer to the correct class.

3.44.5 TUnixFile

```
class TUnixFile : public TParallelFile
```

Provides the interface to a standard POSIX unix file.