

**TRANSPORTATION ANALYSIS SIMULATION SYSTEM
(TRANSIMS)**

VOLUME 1.0

Simulation Output Subsystem for IOC-1

**B. W. Bush
Energy and Environment Analysis Group
Los Alamos National Laboratory**

**K. P. Berkbigler
Computer Research and Applications Group
Los Alamos National Laboratory**

March 1998

[Tab 8]

Contents

1. INTRODUCTION.....	5
2. DESIGN	7
2.1 CONCEPTS.....	7
2.1.1 Types of Data Collection	7
2.1.2 Filtering	7
2.2 CLASSES	7
2.2.1 TOutDispatcher.....	10
2.2.2 TOutFactory	12
2.2.3 TOutGeneralSpecification	13
2.2.4 TOutGeneralSpecificationReader.....	14
2.2.5 TOutSpecificationReader.....	15
2.2.6 TOutProcessor.....	15
2.2.7 TOutEvolutionProcessor.....	16
2.2.8 TOutEventProcessor	17
2.2.9 TOutSummaryProcessor	18
2.2.10 TOutObserver.....	19
2.2.11 TOutVehicleObserver	19
2.2.12 TOutNodeEvolutionObserver.....	20
2.2.13 TOutLinkEvolutionObserver	20
2.2.14 TOutSignalCoordinatorEvolutionObserver	20
2.2.15 TOutSignalizedControlObserver.....	20
2.2.16 TOutIntersectionObserver.....	21
2.2.17 TOutLinkSpaceObserver	21
2.2.18 TOutLinkTimeObserver	22
2.2.19 TOutRetriever	22
2.2.20 TOutEvolutionRetriever	23
2.2.21 TOutEventRetriever.....	23
2.2.22 TOutSummaryRetriever.....	23
2.2.23 TOutWriter	24
2.2.24 TOutTextWriter	24
2.2.25 TOutStorage	24
2.2.26 TOutRecord.....	26
2.2.27 TOutException	28
3. IMPLEMENTATION.....	29
3.1 C++ LIBRARIES	29
3.2 FILE SYSTEM.....	29
3.3 INTEGRATION INTO THE MICROSIMULATION	31
4. USAGE.....	32
4.1 SPECIFICATION FORMATS	32
4.1.1 Output Specification	32
4.1.2 Output Node Specification.....	34
4.1.3 Output Link Specification.....	35
4.2 DATA RETRIEVAL.....	35

4.3 OUTPUT FORMATS	36
4.3.1 Snapshot Data	36
4.3.21. Event Data	37
4.3.3 Summary Data	38
4.4 EXAMPLE OF RETRIEVAL USING C++	39
4.5 NOTES.....	41
4.5.1 Database Setup	41
4.5.2 Empty Storage Files	41
5. FUTURE WORK.....	42
6. REFERENCES	43

1. INTRODUCTION

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 1 shows the position of the simulation output subsystem within the TRANSIMS software architecture. This subsystem depends only 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.

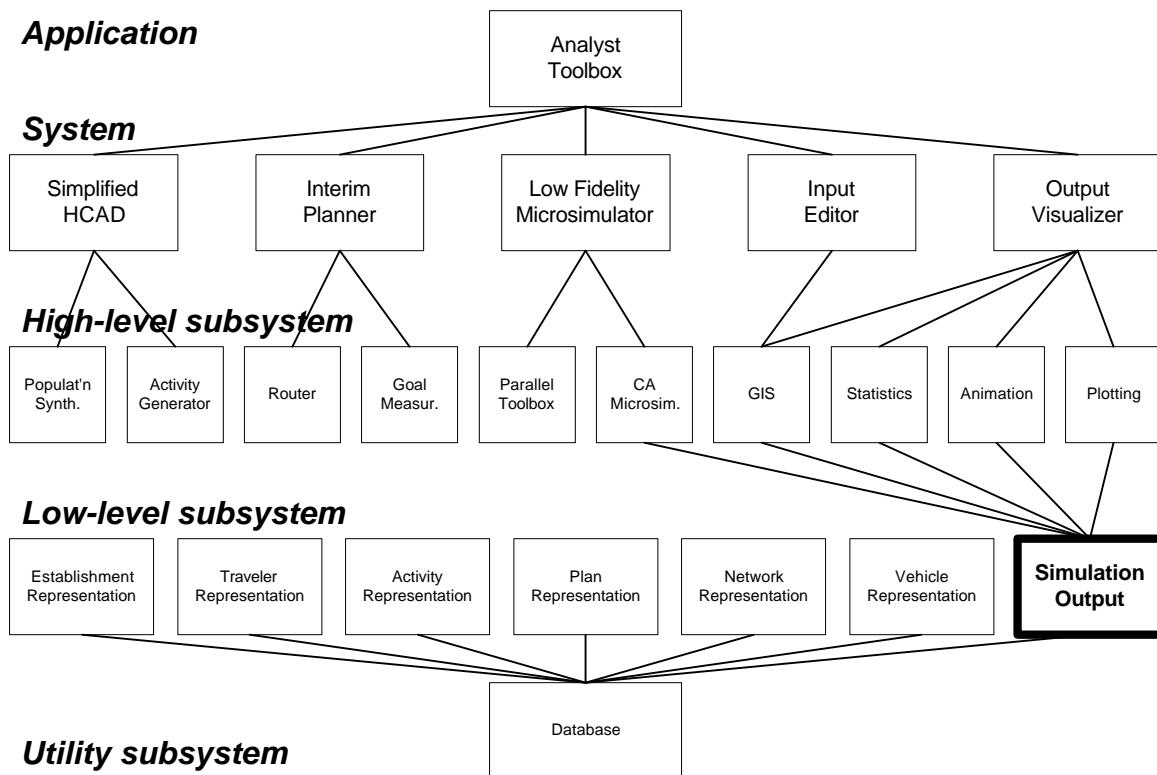


Figure 1: Location of the Simulation Output Subsystem in the TRANSIMS Software Architecture

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 2 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.

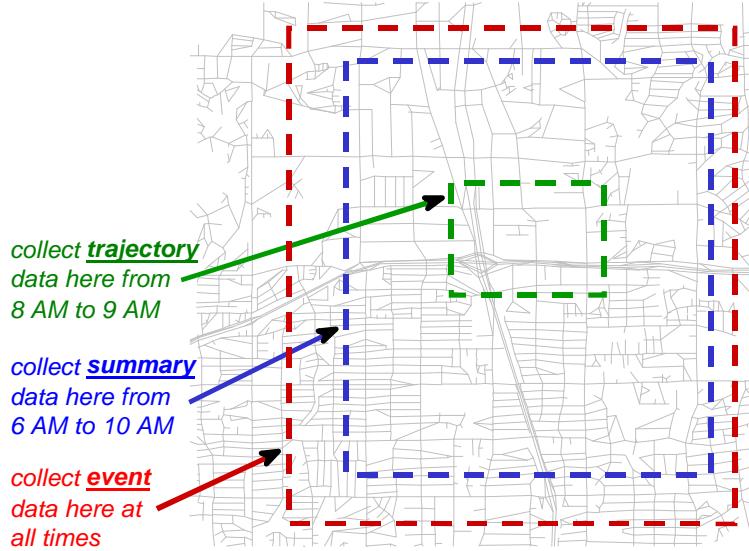


Figure 2: Data Collection Filtered by Space and Time

The subsystem has been put to a wide variety of uses in the first TRANSIMS case study. Snapshot 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 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 3 illustrates the dual uses of the subsystem.

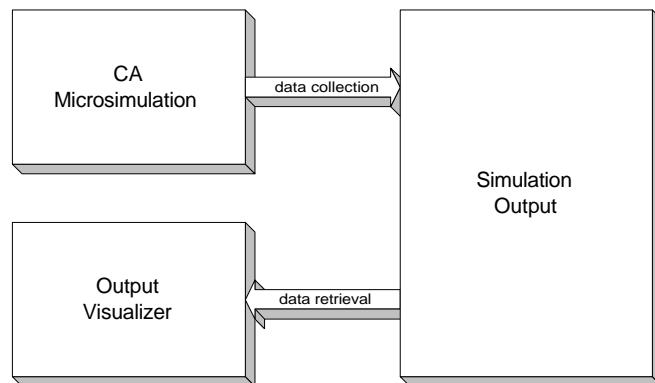


Figure 3: Dual Uses of the Simulation Output Subsystem

The body of this document outlines the design, implementation, and usage of the subsystem.

2. DESIGN

2.1 Concepts

2.1.1 Types of Data Collection

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

- 1) *Snapshot 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. Snapshot data may be collected for each time step; the data is not summarized (i.e., totaled or averaged) in any way.
- 2) *Event data* supplies information on exceptional conditions of vehicle status. Examples include when a vehicle becomes lost (unable to follow its plan), when the plan for a vehicle is invalid, and when the vehicle enters or exits the study area. Event data is collected only when an event occurs.
- 3) *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.

2.1.2 Filtering

The simulation output subsystem has the capability to collect data on any subset of nodes and links in the road network (Figure 2). It is also possible to set the starting and ending times within which data collection occurs (also Figure 2). A user can also specify the frequency of reporting for snapshot 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 Classes

The simulation output subsystem has classes containing domain knowledge and classes forming a domain-independent data management layer (Figure 4). Figure 5, Figure 6, and Figure 7 show the relationships between classes used for snapshot, event, and summary data collection, respectively. Figure 8 shows the relationships between classes used for data retrieval.

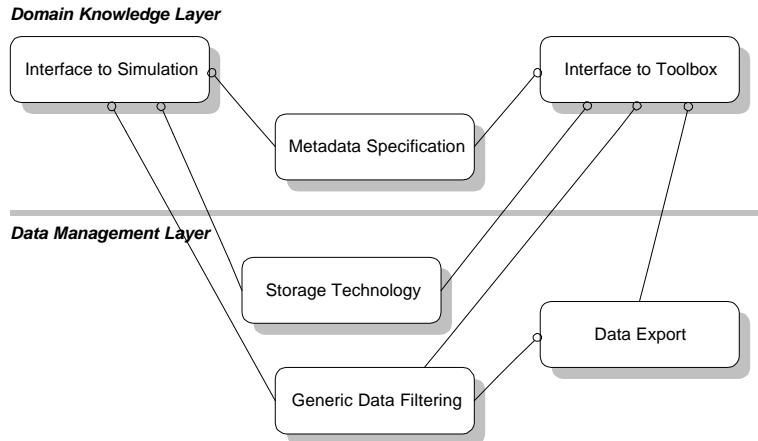


Figure 4: Categories of Classes

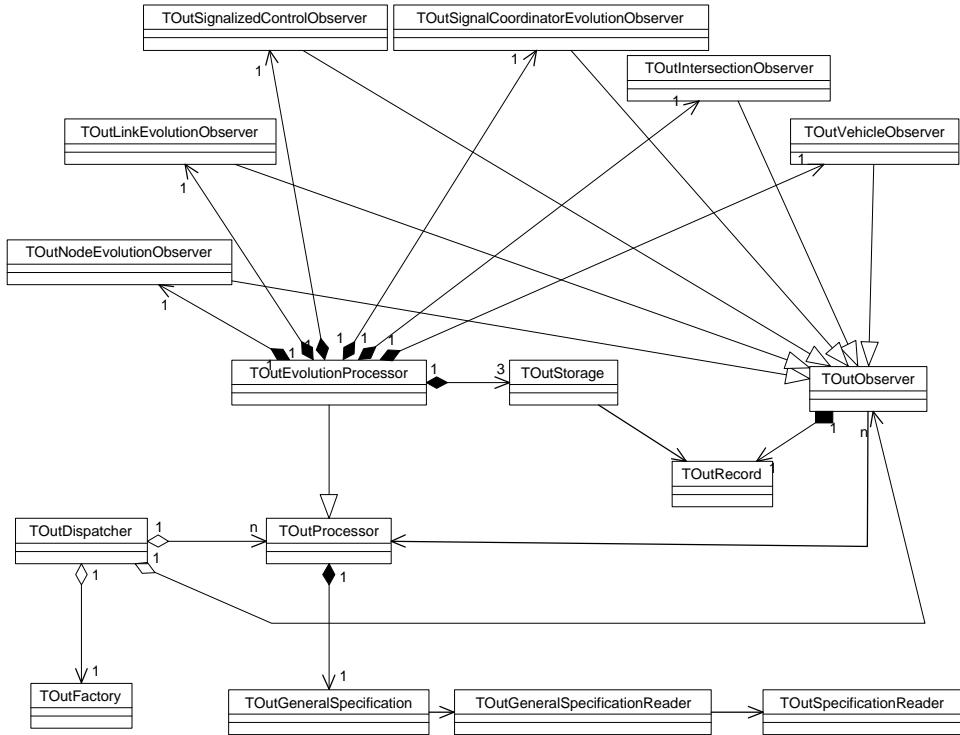


Figure 5: Class Diagram for Classes Involved in Snapshot Data Collection (unified notation)

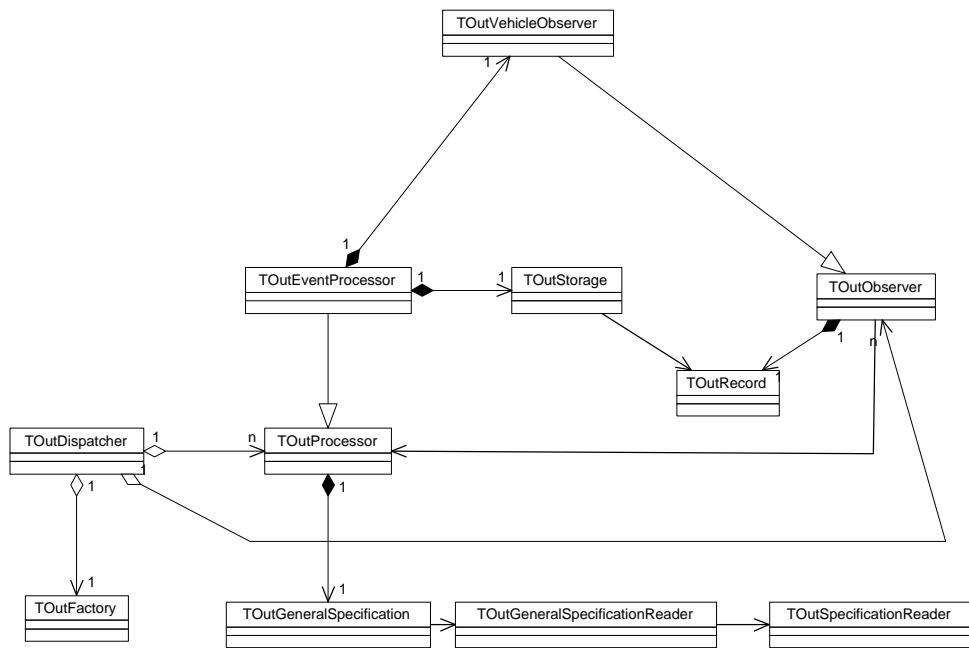


Figure 6: Class Diagram for Classes Involved in Event Data Collection (unified notation)

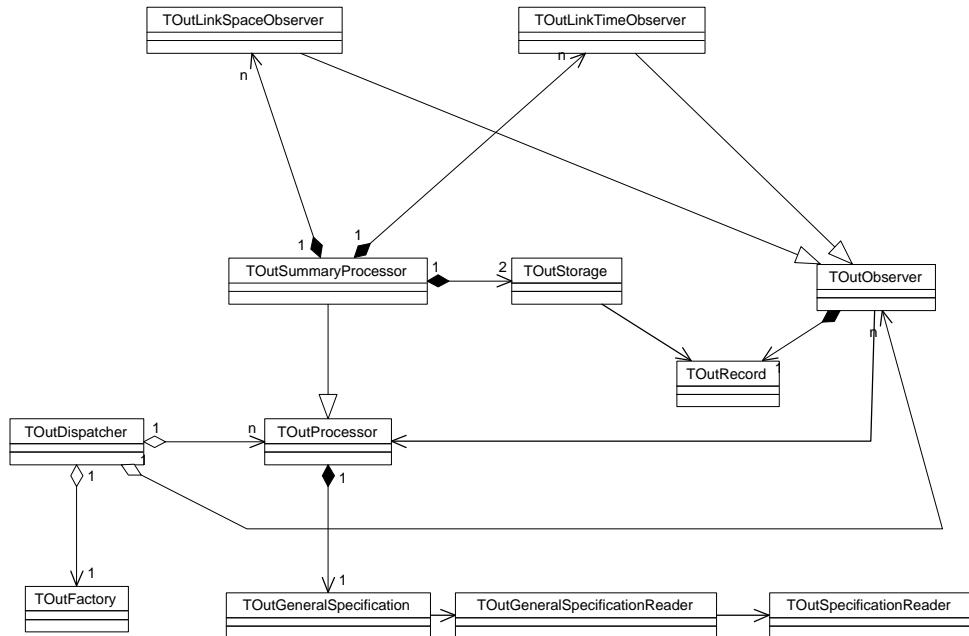


Figure 7: Class Diagram for Classes Involved in Summary Data Collection (unified notation)

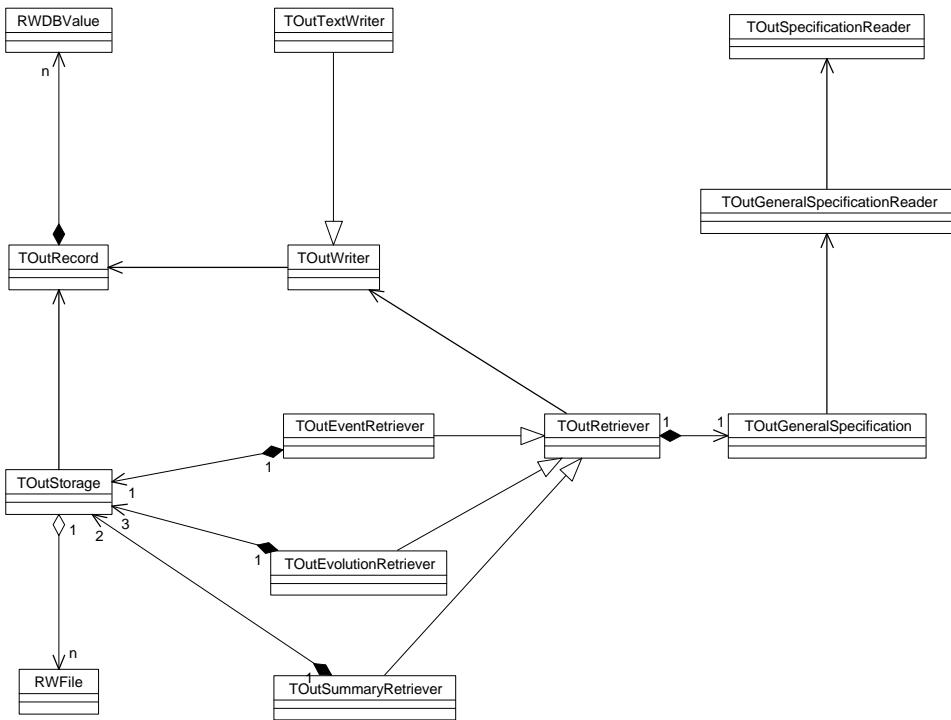


Figure 8: Class Diagram for Classes Involved in Data Retrieval (unified notation)

2.2.1 TOutDispatcher

An output dispatcher coordinates the construction of simulation output objects and supervises the transfer of data. Each dispatcher has processors and observers; it also remembers the factory it uses to construct objects.

```

enum EObserverType {kNodeEvolutionObserver,
                   kLinkEvolutionObserver, kVehicleObserver,
                   kIntersectionObserver,
                   kSignalCoordinatorEvolutionObserver,
                   kSignalizedControlObserver, kLinkSpaceObserver,
                   kLinkTimeObserver}
Observer types.

TOutDispatcher(TOutSpecificationReader& reader, TOutFactory&
               factory)
Construct 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)
Begin output recording for this time step.
  
```

```

ProcessorMap& GetProcessors()

    Return the processors.

const ObserverMap& GetObservers() const

EvolutionProcessorSet& GetEvolutionProcessors()

    Return the evolution processors.

const EventProcessorSet& GetEventProcessors() const

SummaryProcessorSet& GetSummaryProcessors()

    Return the summary processors.

Define the node observers.

Define the link observers.

Define the vehicle observers.

    observers)

void SetSignalCoordinatorObservers(TOutObserver::ObserverMap&

    Define the signal coordinator observers.

    observers)

void SetLinkSpaceObservers(TOutObserver::ObserverMap& observers)

void SetLinkTimeObservers(TOutObserver::ObserverMap& observers)

void ClearLinkSpaceObservers(TOutObserver::ObserverMap&

    Undefine the link space observers.

```

```
void ClearLinkTimeObservers(TOutObserver::ObserverMap&
                            observers)
    Undefine the link time observers.
```

2.2.2 TOutFactory

An output factory allocates and constructs new simulation output objects.

```
TOutFactory()
```

Construct a factory.

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

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

```
virtual TOutSummaryProcessor* NewSummaryProcessor(OutProcessorId
                                                 id, const TOutGeneralSpecification& specification)
    Return a new summary processor from the specification.
```

```
virtual TOutLinkEvolutionObserver*
    NewLinkEvolutionObserver(OutObserverId id)
    Return a new link evolution observer.
```

```
virtual TOutVehicleObserver* NewVehicleObserver(OutObserverId
                                               id)
    Return a new vehicle observer.
```

```
virtual TOutNodeEvolutionObserver*
    NewNodeEvolutionObserver(OutObserverId id)
    Return a new node evolution observer.
```

```
virtual TOutIntersectionObserver*
    NewIntersectionObserver(OutObserverId id)
    Return a new intersection observer.
```

```
virtual TOutSignalCoordinatorEvolutionObserver*
    NewSignalCoordinatorEvolutionObserver(OutObserverId
                                         id)
    Return a new signal coordinator evolution observer.
```

```
virtual TOutSignalizedControlObserver*
    NewSignalizedControlObserver(OutObserverId id)
    Return a new signalized control observer.
```

```

virtual TOutLinkSpaceObserver*
    NewLinkSpaceObserver(OutObserverId id)
    Return a new link space observer.

virtual TOutLinkTimeObserver* NewLinkTimeObserver(OutObserverId
    id);
    Return a new link time observer.

```

2.2.3 TOutGeneralSpecification

The general specification defines the frequency and extent of data to be collected or retrieved in both space and time. Each specification has a root, a name, a minimum time, a maximum time, a time step, a time sample, a box length, a collection region, a set of node ids, and a set of link ids.

```

static const REAL kMinusInfinity
static const REAL kPlusInfinity
    Time constants.

TOutGeneralSpecification(TOutGeneralSpecificationReader& reader)
    Construct a general specification from a reader.

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

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

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

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

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

REAL GetBoxLength() const
    Return the box length.

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

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

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

```

2.2.4 TOutGeneralSpecificationReader

A general specification reader reads specifications from the database. Each general specification reader has database table accessors for the specification and for node and link tables.

```
enum EProcessorType {kEvolutionProcessor, kEventProcessor,
                     kSummaryProcessor}
Processor types.

TOutGeneralSpecificationReader(TOutSpecificationReader& reader)
    Construct a general specification reader.

void Reset()
    Reset the iteration over the table.

void GetNextSpecification()
    Get the next specification in the table.

bool MoreSpecifications() const
    Return whether there are any more specifications in the table.

string GetRoot() const
    Return the root of the specification.

string GetName() const
    Return the name of the specification.

REAL GetMinimumTime() const
    Return the minimum time of the specification.

REAL GetMaximumTime() const
    Return the maximum time of the specification.

REAL GetTimeStep() const
    Return the time step of the specification.

REAL GetTimeSample() const
    Return the time sampling of the specification.

REAL GetBoxLength() const
    Return the box length of the specification.

TGeoRectangle GetRegion() const
    Return the geographic region of the specification.

NodeIDSet GetNodes() const
    Return the nodes in the specification.

LinkIdSet GetLinks() const
    Return the links in the specification.
```

Return the processor type in the specification. A `TOutInvalidProcessor` is thrown if the processor is not a valid type.

2. .5 `TOutSpecificationReader`

specification table, a node specification table, and a link specification table.

```
TOutSpecificationReader(TDbTable generalTable, TDbTable
```

Construct a reader for the specified tables.

```
TDbTable& GetGeneralTable()
```

```
TDbTable& GetNodeTable()
```

Return the node specification table.

Return the link specification table.

2. .6 `TOutProcessor`

representation that is filtered and summarized before storage. A processor has an id and a general output specification; the class keeps track of the next available processor id.

```
kSummaryProcessor}
```

Processor types.

```
specification)
```

Construct a processor.

Return the processor type.

```
virtual void RecordOutput(REAL time)
```

```
OutProcessorId GetId()
```

```
const OutProcessorId GetId() const
```

```
static OutProcessorId GetNextId()
```

Return the next unused processor id.

```
TOutGeneralSpecification& GetGeneralSpecification()
const TOutGeneralSpecification& GetGeneralSpecification() const
    Return the general specification.
```

2.2.7 TOutEvolutionProcessor

An output evolution processor deals with evolving data such as that needed for animation, space-time plots, etc. An evolution processor has a node observer, a link observer, a vehicle observer, an intersection observer, a signal coordinator observer, and a signalized control observer. It is connected to corresponding storage objects.

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

EProcessorType GetProcessorType() const
    Return the processor type.

TOutEvolutionSpecification& GetEvolutionSpecification()
const TOutEvolutionSpecification& GetEvolutionSpecification()
    const
    Return the evolution specification.

virtual void RecordOutput(REAL time)
    Begin recording output for this time step.

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.

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

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

```

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

    Define the link observer.

TOutObserver& GetVehicleObserver()

    Return the vehicle observer.

void SetVehicleObserver(TOutObserver& observer)

TOutObserver& GetIntersectionObserver()
const TOutObserver& GetIntersectionObserver() const

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

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

    Define the signal coordinator observer.

TOutObserver& GetSignalizedControlObserver()

    Return the signalized control observer.

void SetSignalizedControlObserver(TOutObserver& observer)

```

2.2 TOutEventProcessor

An output event processor deals with conditions occurring in the simulation such as vehicle entry, and is connected to a vehicle storage.

```

TOutEventProcessor(OutProcessorId id, const
                    vehicleMask)
Construct an event processor.

    Return the processor type.

```

```

virtual void RecordOutput(REAL time)
    Begin recording output for this time step.

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

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

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

UINT GetVehicleMask() const
    Return the vehicle status mask.

```

2.2.9 TOutSummaryProcessor

An output summary processor collects statistics on the simulation. A summary processor has space and time observers and is connected to corresponding storages.

```

TOutSummaryProcessor(OutProcessorId id, const
                     TOutGeneralSpecification& specification)
    Construct a summary processor.

EProcessorType GetProcessorType() const
    Return the processor type.

virtual void RecordOutput(REAL time)
    Begin recording output for this time step.

virtual void RecordSpace(const TOutObserver& observer)
    Finish recording output for link space data.

virtual void RecordTime(const TOutObserver& observer)
    Finish recording output for link time data.

ObserverSet& GetSpaceObservers()
const ObserverSet& GetSpaceObservers() const
    Return the space observers.

ObserverSet& GetTimeObservers()
const ObserverSet& GetTimeObservers() const
    Return the time observers.

void AddSpaceObserver(TOutObserver& observer)
    Define a space observer.

void AddTimeObserver(TOutObserver& observer)
    Define a time observer.

```

```
void RemoveSpaceObserver(TOutObserver& observer)

void RemoveTimeObserver(TOutObserver& observer)
    Undefine a time observer.
```

2.10

An observer converts data from the object to which it is attached into the generic form understood by the output subsystem. An observer has an id and an output record. The class keeps track of the

```
TOutObserver(OutObserverId)
    Construct an observer.
```

```
processor)
```

An observer has an observe function for noting the values of data members of interest in the view.

```
OutObserverId GetId()
```

Return the observer's id.

```
TOutRecord& GetRecord()
```

Return the associated record.

```
static OutObserverId GetNextId()
```

```
void SetTime(REAL time)
```

Define the record's time.

2.11

A vehicle observer observes data related to vehicles.

```
TOutVehicleObserver(OutObserverId id)
```

```
void SetId(UINT vehicle)
```

Define the vehicle's id.

Define the id of the link the vehicle is on.

```

void SetLane(NetLaneNumber lane)
    Define the lane number the vehicle is on.

void SetDistance(REAL distance)
    Define the vehicle's distance from the node.

void SetNode(NetNodeId node)
    Define the id of the node from which the distance is measured.

void SetVelocity(REAL velocity)
    Define the vehicle's velocity.

void SetStatus(BYTE status)
    Define the vehicle's status.

```

2.2.12 TOutNodeEvolutionObserver

A node evolution observer observes evolving data related to nodes.

```
TOutNodeEvolutionObserver(OutObserverId id)
    Construct a node evolution observer.
```

2.2.13 TOutLinkEvolutionObserver

A link evolution observer observes evolving data related to links.

```
TOutLinkEvolutionObserver(OutObserverId id)
    Construct a link evolution observer.
```

2.2.14 TOutSignalCoordinatorEvolutionObserver

A signal coordinator evolution observer observes evolving data related to signal coordinators.

```
TOutSignalCoordinatorEvolutionObserver(OutObserverId id)
    Construct a signal coordinator evolution observer.
```

2.2.15 TOutSignalizedControlObserver

A signalized control observer observes data related to signals.

```
TOutSignalizedControlObserver(OutObserverId id)
    Construct a signalized control observer.
```

```
void SetLink(NetLinkId link)
    Define the id of the link entering the intersection.
```

```
void SetLane(NetLaneNumber lane)
    Define the lane number of the lane entering the intersection.
```

```
void SetNode(NetNodeId node)

void SetSignal(TNetTrafficControl::ETrafficControl state)
    Define the current signal state.
```

2.16

An intersection observer observes data related to intersections.

```
TOutIntersectionObserver(OutObserverId id)
```

```
void SetId(UINT vehicle)
    Define the vehicle's id.
```

Define the id of the link from which the vehicle entered.

```
void SetLane(NetLaneNumber lane)
```

```
void SetNode(NetNodeId node)
    Define the id of the node with which the intersection is associated.
```

Define the index of the vehicle's position in the queue.

2. .17 TOutLinkSpaceObserver

box length, and box data.

```
TOutLinkSpaceObserver(OutObserverId id)
```

```
bool IsInitialized() const
    Return whether the observer has been initialized.
```

Clear the accumulated spatial summary data for the observer.

```
void SetLink(NetLinkId id)
```

```
void SetNode(NetNodeId id)
    Set the departure node id.
```

```

void SetLengths(REAL linkLength, REAL boxLength, REAL
                cellLength)
    Set the link, box, and cell lengths.

void AddVehicle(REAL distance, REAL velocity)
    Add a vehicle to the summary.

void ReportObservations(TOutProcessor& processor)
    Report the observations to the specified processor.

```

2.2.18 TOutLinkTimeObserver

A link time observer records vehicle travel times on a link. Each link time observer has a vehicle count, a total of travel times, and a total of squared travel times.

```

TOutLinkTimeObserver(OutObserverId id)
    Construct a link time observer.

void AddVehicle(REAL time)
    Add a vehicle to the summary.

void ClearData()
    Clear the data for the observer.

void SetLink(NetLinkId id)
    Set the link id.

void SetNode(NetNodeId id)
    Set the departure node id.

void ReportObservations(TOutProcessor& processor)
    Report the observations to the specified processor.

```

2.2.19 TOutRetriever

An output retriever acts as an interface to coordinate the retrieval of data stored in a simulation. Each retriever has a general specification and may refer to a network.

```

virtual void Retrieve()
    Perform the retrieval.

TOutRetriever(const TOutGeneralSpecification& specification,
              const TNetNetwork* network = NULL)
    Construct a retriever based on the given specification and network.

TOutGeneralSpecification& GetGeneralSpecification()
const TOutGeneralSpecification& GetGeneralSpecification() const
    Return the general specification.

```

Return the network, if any.

```
void BasicRetrieve(TOutStorage& storage, TOutWriter& writer,
```

Retrieve data from the specified storage and put it in the specified writer, sorting it if indicated, and performing time and space filtering if indicated.

2.20

An evolution retriever gets specific trajectory data from storage and coordinates its conversion and filtering. The retriever is connected to a vehicle storage, an intersection storage, and a signal

```
TOutEvolutionRetriever(const TOutStorage::HostSet& hosts, const  
TOutGeneralSpecification& specification, const
```

Construct a reader for the specified hosts, given specification, and network.

```
void Retrieve(TOutWriter& vehicleWriter, TOutWriter&  
sort = TRUE)
```

Perform the retrieval on the specified writers.

2.21

An event retriever gets specific event data from storage and coordinates its conversion and filtering. The retriever is connected to a vehicle storage.

```
TOutGeneralSpecification& specification, const  
TNetNetwork* network = NULL)
```

```
void Retrieve(TOutWriter& vehicleWriter, bool sort = TRUE)
```

Perform the retrieval on the specified writers.

2.22

An event retriever gets specific summary data from storage and coordinates its conversion and filtering. The retriever is connected to a space storage and a time storage.

```
TOutGeneralSpecification& specification, const  
TNetNetwork* network = NULL)
```

```
void Retrieve(TOutWriter& spaceWriter, TOutWriter& timeWriter,  
bool sort = TRUE)
```

2.2.23 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.

```
TOutWriter()
    Construct a writer.

virtual void Write(const TOutRecord& record)
virtual void Write(const TOutRecord& record, const
                  FieldCollection& fields)
    Write the specified record.
```

2.2.24 TOutTextWriter

A text writer puts simulation output data into a formatted text file. The exception `TOutWriterFailure` is thrown if an operation fails. Each text writer is connected to a file and has a delimiter. A text writer may have to include header information in its output.

```
TOutTextWriter(const string& file, const string& delimiter =
               "\t", bool includeHeader = FALSE)
    Open the specified file for writing, including the record header.

void Write(const TOutRecord& record)
void Write(const TOutRecord& record, const FieldCollection&
           fields)
    Write the specified record with fields in the given order.
```

2.2.25 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. The class has a local host name. Each instance has a file suffix, a root location, a basic name, and a file on each host.

```
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 HostSet& hosts, const string& root, const
            string& name, Mode mode = kRead)
    Connect the storage with the given root and basic name to the specified hosts.
```

```

const string& GetRoot() const
    Return the root name.

    Return the basic name.

HostSet GetHosts() const

HostHandle GetHostHandle(const string& host) const
    Return the host handle.

long GetOffset(HostHandle host) const
long GetOffset(const string& host) const

bool AtEnd() const
bool AtEnd(HostHandle host) const

    Return whether an end-of-file has occurred for the specified host file.

void Seek(long position = kBegin)

void Seek(const string& host, long position = kBegin)
    Position the specified host file to the given location.

void Write(HostHandle host, const TOutRecord& record)
void Write(const string& host, const TOutRecord& record)

bool Read(TOutRecord& record)
bool Read(const string& host, TOutRecord& record)

    Read the given record on the specified host. Return whether a record was available for
    reading.

void WriteHeader(const string& host, const TOutRecord& record)
void WriteHeader(HostHandle host, const TOutRecord& record)

bool ReadHeader(TOutRecord& record)
bool ReadHeader(const string& host, TOutRecord& record)

    Read the given record header on the specified host. Return whether a record was available
    for reading.

    Flush any pending operations.

```

2.2.26 TOutRecord

This class is used for storing the values of a collection of fields. Each record has a field map holding the current values of the fields.

```
enum Type {kNoType, kChar, kUnsignedChar, kShort,
           kUnsignedShort, kInt, kUnsignedInt, kLong,
           kUnsignedLong, kFloat, kDouble, kString}
```

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.

```
void SetField(const string& field, char value)
void SetField(const string& field, unsigned char value)
void SetField(const string& field, short value)
void SetField(const string& field, unsigned short value)
void SetField(const string& field, int value)
void SetField(const string& field, unsigned int value)
void SetField(const string& field, long value)
void SetField(const string& field, unsigned long value)
void SetField(const string& field, float value)
void SetField(const string& field, double value)
void SetField(const string& field, const string& value)
void SetField(const string& field)
```

Set the value of the specified field.

```
void GetField(const string& field, char& value) const
void GetField(const string& field, unsigned char& value) const
void GetField(const string& field, short& value) const
void GetField(const string& field, unsigned short& value) const
void GetField(const string& field, int& value) const
void GetField(const string& field, unsigned int& value) const
void GetField(const string& field, long& value) const
void GetField(const string& field, unsigned long& value) const
void GetField(const string& field, float& value) const
void GetField(const string& field, double& value) const
void GetField(const string& field, string& value) const
```

Get the value of the specified field.

```
Type GetType(const string& field) const
```

Return the type of the specified field.

```
FieldMap& GetMap()
```

```
const FieldMap& GetMap() const
```

Return the map for the record.

```
FieldMapIterator GetIterator() const
```

2. .27 TOutException

has a message. Figure 9 shows the hierarchy of exception classes.

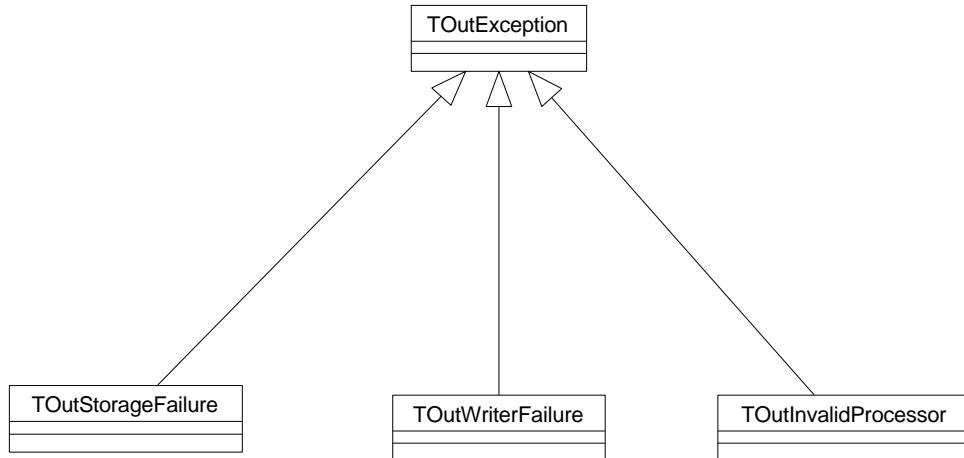


Figure 9: Exception Hierarchy (unified notation)

`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.

3. IMPLEMENTATION

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).

3.2 File System

Although the simulation output subsystem runs on multiple computational nodes (CPNs), it stores output data locally (Figure 10) 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 11).

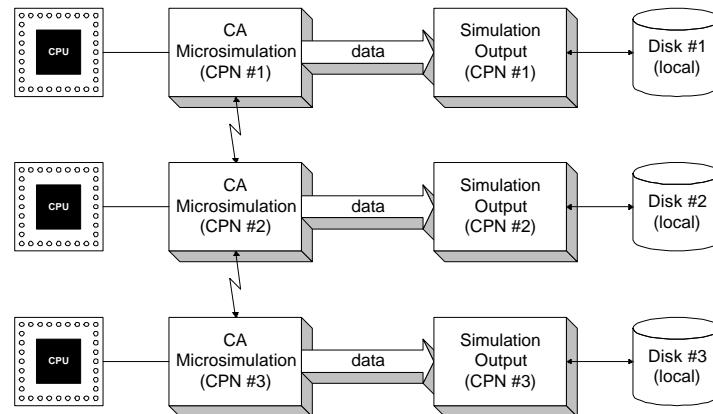


Figure 10: Configuration for Simultaneous Data Collection on Multiple CPNs

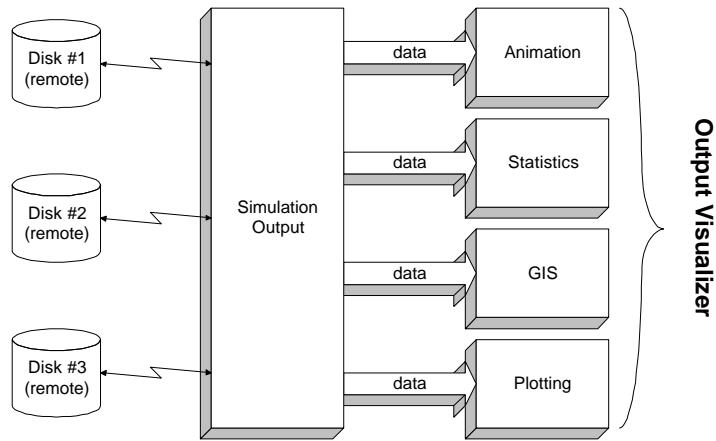


Figure 11: Configuration for Data Retrieval from Multiple Disks

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 name (typically the name of the CPN) so that it can be accessed remotely. Figure 12 illustrates this scheme. Several other workable arrangements are possible, however.

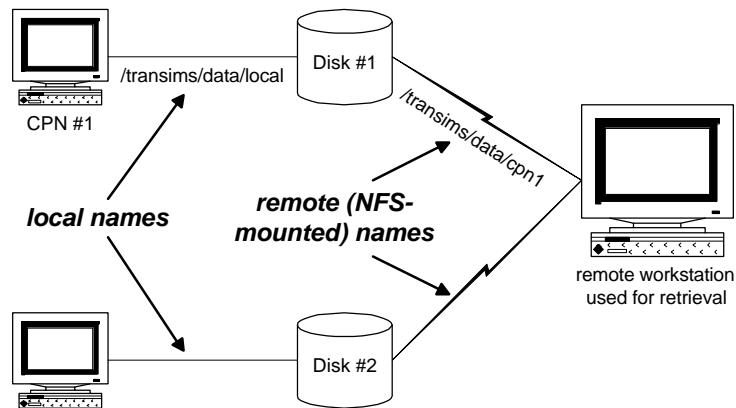


Figure 12: Typical Naming and NFS-Mounting Scheme for Simulation Output Directories

3.3 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 13 illustrates an example of how this subclassing can be implemented.

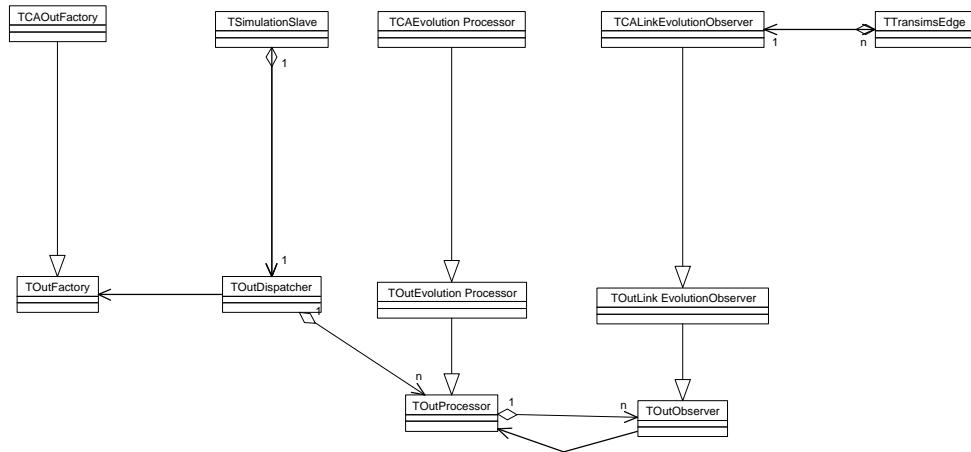


Figure 13: Subclassing Simulation Output Classes in a Simulation (unified notation)

4. USAGE

4.1 Specification Formats

This section describes the files that a user must currently prepare in order to use the simulation output subsystem in conjunction with the CA microsimulation.

The general specification output tables describe the general characteristics of the output that is to be collected during a microsimulation run. These specifications are stored in the Oracle database management system prior to running the microsimulation, retrieved by the microsimulation during execution, and used to guide the simulation output subsystem's collection and storage of data.

Tables may be created in Oracle by using the database functions of the Input Editor. Tables may also be created by using the database subsystem import utility, for which the usage is:

```
Import file
```

where `file` contains information and commands for the construction of one or more data tables. Import prints a brief help message if invoked without a file. When constructing new tables, it is recommended that, when possible, an existing table description file that is similar to the desired new table description be copied and edited. This reduces the chances for errors in the SQL statements in the file.

Three tables are currently used to provide the general specification for the simulation output subsystem. Preparation of the files for each of these tables is described below.

4.1.1 Output Specification

The output specification table provides information about the time frame for collecting data and where the data should be stored. Table 1 defines the format for this table. The combination of ROOT and NAME must be unique among multiple simultaneous users of the subsystem, or else the output files will be overwritten. For example, evolutionary output might be collected on each CPN for vehicle, intersection, and signal data. The files created would be:

<code>ROOT + "/local" + NAME + ".veh" + ".stg"</code>	for vehicle data
<code>ROOT + "/local" + NAME + ".int" + ".stg"</code>	for intersection data
<code>ROOT + "/local" + NAME + ".sig" + ".stg"</code>	for signal data

Typically, the directory `ROOT + "/local" + NAME` on each host is just an NFS link to the physical directory `ROOT + hostname + NAME`; this allows both local and global views of the output directories (see Figure 12). Note that the output files must be deleted manually when they are no longer needed.

Table 1: Format for the Output Specification Data Table (an “Output Specification” data source in the database subsystem)

Field	Interpretation
ROOT	The directory where the output should be written
NAME	The output file name
PROCESSOR	The type of processor: “Evolution”: an evolution processor “Event”: an event processor “Summary”: a summary processor
TIMEMIN	The first time (in seconds from simulation start) at which to collect data
TIMEMAX	The last time (in seconds from simulation start) at which to collect data.
TIMESTP	The frequency (in seconds) at which to report data (i.e., write it to disk)
TIMESMP	The frequency (in seconds) at which to accumulate sample data
BOXLEN	The length of the boxes used for summary data
ABSCISSAMN	The minimum abscissa for which to collect data (currently ignored)
ABSCISSAMX	The maximum abscissa for which to collect data (currently ignored)
ORDINATEMN	The minimum ordinate for which to collect data (currently ignored)
ORDINATEMX	The maximum ordinate for which to collect data (currently ignored)

An example output specification file is reproduced here:

```

Benchmark 1 (1 sq. mi.) Output Spec. II          1
This is a sample output specification table.      2
OUTSPECBENCH1III                                3
Output Specification                             4
CREATE TABLE OUTSPECBENCH1III (                  5
    PROCESSOR VARCHAR(25),                      6
    ROOT VARCHAR(50),                         7
    NAME VARCHAR(50),                         8
    TIMEMIN NUMBER(10),                        9
    TIMEMAX NUMBER(10),                        10
    TIMESTP NUMBER(10),                         11
    TIMESMP NUMBER(10),                         12
    BOXLEN NUMBER(10),                          13
    ABSCISSAMIN FLOAT,                         14
    ABSCISSAMAX FLOAT,                         15
    ORDINATEMIN FLOAT,                         16
    ORDINATEMAX FLOAT,                         17
    PRIMARY KEY (NAME)                         18
);
PROCESSOR, ROOT, NAME, TIMEMIN, TIMEMAX, TIMESTP, TIMESMP, BOXLEN,
ABSCISSAMIN, ABSCISSAMAX, ORDINATEMIN, ORDINATEMAX          20
'Evolution', '/transims/output4', 'bn1_evolution_1', 0, 36000,
300, 0, 0, 3000, 0, 3000                                     21
'Summary', '/transims/output4', 'bn1_summary_1', 0, 36000, 180,
10, 150, 0, 3000, 0, 3000                                    22
'Summary', '/transims/output4', 'bn1_summary_2', 0, 36000, 300,
60, 50, 0, 3000, 0, 3000                                     23
'Event', '/transims/output4', 'bn1_event_1', 0, 36000, 1, 1, 1, 0,
3000, 0, 3000                                              24

```

The first line of the file is the unique table name, with no more than 50 characters. The second line is a comment describing the table. Line 3 is a unique SQL table name. Line 4 is the data source name. Line 5 begins the SQL command for creating the table with the name specified in line 3. This command continues for fourteen additional lines and is terminated with a closing parenthesis and a semicolon. Line 20 names the columns whose values are specified in the same order and delimited by commas on lines 21 through 24. One type of evolution data, two types of summary data, and one type of event data are specified for collection in this example.

4.1.2 Output Node Specification

The output node specification table is used to specify the nodes at which data should be collected. Table 2 defines the format for this table. Until geographic filtering is supported this is the only way to indicate the nodes for which data collection occurs.

Table 2: Format for the Output Node Specification Data Table (an “Output Node Specification” data source in the database subsystem)

Field	Interpretation
NAME	The output file name
NODE	The node id

An example output node specification file is reproduced here:

```

Benchmark 1 (1 sq. mi.) Output Node Spec. II          1
This is a sample output node specification table.      2
OUTNODEBENCH11II                                     3
Output Node Specification                            4
CREATE TABLE OUTNODEBENCH11II (                      5
    NAME VARCHAR(50),                                6
    NODE NUMBER(10),                                 7
    PRIMARY KEY (NAME,NODE)                         8
);
NAME, NODE                                         9
'bn1_evolution_1',40006                           10
'bn1_evolution_1',1793                            11
'bn1_evolution_1',1808                            12
'bn1_evolution_1',2413                            13
'bn1_evolution_1',2423                            14
'bn1_summary_2',2423                            15
'bn1_summary_2',2424                            16

```

The first line of the file is the unique table name, with no more than 50 characters. The second line is a comment describing the table. Line 3 is a unique SQL table name. Line 4 is the data source name. Line 5 begins the SQL command for creating the table with the name specified in line 3. This command continues for four additional lines. Line 10 names the columns whose values are specified on lines 11 and following. The NAME column refers to the output filename and must correspond to the filename given in the Output Specification. Evolution data is collected on five nodes and summary data on two nodes.

4.1.3 Output Link Specification

The output link specification table is used to specify the links at which data should be collected. Table 3 defines the format for this table. Until geographic filtering is supported this is the only way to indicate the links for which data collection occurs.

Table 3: Format for the Output Link Specification Data Table (an “Output Link Specification” data source in the database subsystem)

Field	Interpretation
NAME	The output file name
LINK	The link id

An example output link specification file is reproduced here:

```
Benchmark 1 (1 sq. mi.) Output Link Spec. II          1
This is a sample output link specification table.      2
OUTLINKBENCH11II          3
Output Link Specification          4
CREATE TABLE OUTLINKBENCH11II (          5
    NAME VARCHAR(50),          6
    LINK NUMBER(10),          7
    PRIMARY KEY (NAME,LINK)          8
);
NAME, LINK          9
'bn1_summary_1',11150000          10
'bn1_summary_1',11140001          11
'bn1_summary_1',8830507          12
'bn1_summary_1',8830507          13
```

The first line of the file is the unique table name, with no more than 50 characters. The second line is a comment describing the table. Line 3 is a unique SQL table name. Line 4 is the data source name. Line 5 begins the SQL command for creating the table with the name specified in line 3. This command continues for four additional lines. Line 10 names the variables whose values are specified on lines 11 and following. Summary data is collected on three links.

4.2 Data Retrieval

The binary output stored in a distributed manner by the microsimulation is generally postprocessed to collect it into a single location. The DumpStorage utility may be used to postprocess the output into a text format for display or analysis. The usage is:

```
DumpStorage outname root name hosts ...
```

where `outname` is the destination file name, `root` is the root directory for the host subdirectories, `name` is the name of the storage file, and `hosts ...` are the host subdirectory names. For example,

```
DumpStorage bn1_summary_1.txt /transims/output4 bn1_summary_1.tim
bach faure sousa
```

collates the data from the files

```

/transims/output4/bach/bn1_summary_1.tim.stg
/transims/output4/faure/bn1_summary_1.tim.stg
/transims/output4/sousa/bn1_summary_1.tim.stg

```

into the file

```
./bn1_summary_1.txt
```

in a tab-delimited ASCII text format.

4.3 Output Formats

4.3.1 Snapshot Data

Vehicle snapshot data files have the storage file suffix `.veh.stg`. Table 4 lists the fields present in such files; each record in the file represents a single vehicle. The data reporting start time, finish time, and reporting frequency are given by the output specification. The output specification also determines on which links the data is collected.

Table 4: Data Format for Vehicle Snapshot Storage (`.veh.stg`) files

Field	Interpretation
VEHICLE	The vehicle id
NODE	The node from which the vehicle was traveling away
LINK	The link on which the vehicle was traveling
LANE	The lane on which the vehicle was traveling
TIME	The time the data was taken (in seconds from simulation start)
DISTANCE	The distance (in meters) the vehicle was away from the node setback
VELOCITY	The velocity (in meters per second) the vehicle was traveling
STATUS	The vehicle status bits: 1: The vehicle is lost. 2: The vehicle is at a dead end. 4: The vehicle has just entered the study area. 8: The vehicle has just exited the study area. 16: The vehicle is in the study area. 32: The vehicle has an invalid plan.

Intersection evolution data files have the storage file suffix `.int.stg`. Table 5 lists the fields present in such files; each record in the file represents a single vehicle. The data reporting start time, finish time, and reporting frequency are given by the output specification. The output specification also determines on which nodes the data is collected.

Table 5: Data Format for Intersection Evolution Storage (.int.stg) files

Field	Interpretation
VEHICLE	The vehicle id
NODE	The node where the vehicle is located
TIME	The time the data was taken (in seconds from simulation start)
LINK	The link from which the vehicle entered
LANE	The lane from which the vehicle entered
QINDEX	The vehicle position in the queue

Signal evolution data files have the storage files suffix `.sig.stg`. Table 6 lists the fields present in such files; each record in the file represents an incoming lane at an intersection. The data reporting start time, finish time, and reporting frequency are given by the output specification. The output specification also determines on which nodes the data is collected.

Table 6: Data Format for Signal Evolution Storage (.sig.stg) files

Field	Interpretation
NODE	The node where the signal is located
TIME	The time the data was taken (in seconds from simulation start)
LINK	The link entering the signal
LANE	The lane entering the signal
SIGNAL	The type of control present: 0: None. 1: Stop. 2: Yield. 3: Wait. 4: Caution. 5: Permitted. 6: Protected.

4.3.2 Event Data

Vehicle event data files have the storage file suffix `.veh.stg`. Table 7 lists the fields present in such files; each record in the file represents a single vehicle event. The data reporting start time, finish time, and reporting frequency are given by the output specification. This data is collected for all links—i.e., the output link specification is ignored.

Table 7. Data format for vehicle event storage (.veh.stg) files

Field	Interpretation
VEHICLE	The vehicle id
NODE	The node from which the vehicle was traveling away
LINK	The link on which the vehicle was traveling
LANE	The lane on which the vehicle was traveling
TIME	The time the data was taken (in seconds from simulation start)
DISTANCE	The distance (in meters) the vehicle was away from the node setback
VELOCITY	The velocity (in meters per second) the vehicle was traveling
STATUS	The vehicle status bits: 1: The vehicle is lost. 2: The vehicle is at a dead end. 4: The vehicle has just entered the study area. 8: The vehicle has just exited the study area. 16: The vehicle is in the study area. 32: The vehicle has an invalid plan.

4.3.3 Summary Data

Link space summary data files have the storage file suffix `.spa.stg`. Table 8 lists the fields present in such files; each record in the file represents the summary for a single box on a link. If there is a short box, it is at the beginning of the link. The beginning distance of the box, which is not written in the file, is the ending distance of the box minus the box size. The data reporting start time, finish time, the sampling frequency, the data reporting frequency, and the box size are given by the output specification. The output specification also determines on which links the data is collected. Note that no data is reported at the reporting start time. Also, there may be two entries for links that are split by a CPN boundary. When this happens, it is necessary to add the respective COUNT and SUM entries for the duplicate box records.

Table 8: Data Format for Link Space Summary Storage (.spa.stg) files

Field	Interpretation
LINK	The link being reported
NODE	The node from which vehicles were traveling
DISTANCE	The ending distance of the box (in meters) from the node setback
TIME	The time the data was taken (in seconds from simulation start)
COUNT	The number of vehicles in the box
SUM	The sum of the vehicle velocities (in meters per second) in the box

Link time summary data files have the storage file suffix `.tim.stg`. Table 9 lists the fields present in such files. Each record in the file represents the summary for a single direction of a link. The data reporting start time, finish time, and reporting frequency are given by the output specification. The output specification also determines on which links the data is collected. Note that no data is reported at the reporting start time. Also, there may be two entries for links that are split by a CPN boundary. When this happens, it is necessary to add the respective COUNT, SUM, and SUMSQUARES entries for the duplicate link records.

Table 9: Data Format for Link Time Summary Storage (.tim.stg) files

Field	Interpretation
LINK	The link being reported
NODE	The node from which vehicles were traveling
TIME	The time the data was taken (in seconds from simulation start)
COUNT	The number of vehicles leaving the link
SUM	The sum of the vehicle travel times (in seconds) for vehicles leaving the link. The time spent in the previous intersection is included in this value
SUMSQUARES	The sum of the squared vehicle travel times (in seconds squared) for vehicles leaving the link. The time spent in the previous intersection is included in this value.

4.4 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;
```

4.5 Notes

4.5.1 Database Setup

The CreateSources utility must be executed before the first use of the simulation output subsystem. This application registers the data sources in the database subsystem.

4.5.2 Empty Storage Files

The present implementation of the TOut...Retriever classes cannot handle input files of zero length. If no data is collected on a CPN, it is advisable to delete the empty storage file(s) created for that CPN.

5. 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
- eliminating the dependence on commercial products such as DBtools.h++ and Tools.h++

6. 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).