GrooveNet Vehicle Network Simulator

Developer's Manual

Rahul Mangharam & Dan Weller
rahul@cmu.edu
May 2006

Notice

GrooveNet is a product of the hard-work of several Carnegie Mellon undergraduate and graduate students. You are requested NOT to reuse any of the images in this manual. Please credit the developers by citing:

1. R. Mangharam, D. S. Weller, R. Rajkumar. "GrooveNet: A Hybrid Simulator for Vehicle-to-Vehicle Networks" *Proceedings of Second International Workshop Vehicle-to-Vehicle Communications (V2VCOM)* Invited Paper. San Jose, USA. July 2006.

2. R. Mangharam, D. S. Weller, D. D. Stancil, R. Rajkumar, J. S. Parikh, "GrooveSim: A Topography-Accurate Simulator for Geographic Routing in Vehicular Networks" *Proceedings of Second ACM International Workshop on Vehicular Ad hoc Networks (Mobicom/VANET 2005).* Cologne, Germany. September 2005.

The source code, documentation and related material are distributed to with the understanding that all vehicular networking projects will be conducted in collaboration with Rahul Mangharam and Professor Raj Rajkumar of Carnegie Mellon University. We share this software to foster joint research projects and further the development of common tools for new research areas.

© 2006. Rahul Mangharam, Daniel S. Weller and Raj Rajkumar. Carnegie Mellon University, Pittsburgh, PA. USA

Table of Contents

1. Simulation Manager 1
  1.1. Creating New Simulations 1
  1.2. Loading Existing Simulations 1
    1.2.1. Simulation File Specification 1
    1.2.2. Simulation Loading Process 3
  1.3. Saving Simulations 4
  1.4. Unloading Simulations 4
  1.5. The Model Manager 4
  1.6. Running a Simulation 4
    1.6.1. Simulator Event 4
    1.6.2. Simulator Event Queue 5
2. Simulation Models and Visualizers 5
  2.1. Design and Implementation 5
    2.1.1. Model Lifecycle 5
    2.1.2. Model Hierarchy 8
    2.1.3. Creating a New Model Type 8
  2.2. Vehicle Models 9
    2.2.1. Communication Models 9
    2.2.2. Mobility Models 10
  2.3. Visualizers 11
  2.4. Other Models 11
3. User Interface Components 11
  3.1. Main Application Window 11
  3.2. Simulation Editor Dialog Box 12
  3.3. Auto-Generate Nodes Dialog Boxes 13
  3.4. Run Simulation Dialog Box 14
  3.5. Send Message Dialog Boxes 15
  3.6. Configure GrooveNet Dialog Box 16
4. Map Database 16
  4.1. GrooveNet Database Format 16
  4.2. TIGER/Line 2000+ Database Format 18
  4.3. Adding Other Formats 18
5. Network Interface 18
  5.1. Network Messages 18
  5.2. Network Abstraction Layer 19
  5.3. UDP Client/Server 19
  5.4. TCP Client/Server 19
  5.5. Adding Other Client/Servers 19
6. GPS Interface 19
  6.1. NMEA 0183 Processor 20
  6.2. Adding Other GPS Processors 20
7. Data Logger 20
  7.1. Logged Data Format 20
  7.2. Simulation Data Logging 20
  7.3. Debugger Logging 21
  7.4. Adding Other Data Formats 21

1. Simulation Manager

The simulation manager is in charge of creating, loading, saving, and running simulations. The important classes are Simulator, ModelMgr, SimEvent, and SimEventQueue. The Simulator class is the core class for managing simulations. It contains functions for creating new, loading existing, saving, and running simulations. It also contains a ModelMgr member, which maintains the list of simulation models and manages the model types. The simulation consists of the Simulator class pulling events from the event queue and processing them. These events are implemented in the SimEvent class, and the event queue is implemented in the SimEventQueue class.

1.1. Creating New Simulations

To create a new simulation, follow these steps:

1. Create all vehicles and infrastructure nodes using the Auto-Generate Nodes dialog accessible from the Simulation Editor dialog. Specify all parameters for these vehicles, in either or both dialog boxes. Each of these dialog boxes are described later in this manual.

2. Create additional associated models in the Other Models tab of the Simulation Editor dialog. In the very least, you will probably want to create visualizers (such as MapVisual or CarListVisual).

3. Once all models have been created, press the OK button to close the dialog and create the simulation.

4. The functions Simulator::New() and Simulator::internalAddNew() iterate through all the models and create the models. Details on model creation are provided in the section on loading an existing simulation. Models that depend on other models are not created until those other models are created successfully. This means that circular dependencies can cause simulation creation to fail.

5. If the function Simulator::New() fails (either a model failed to initialize successfully or the dependencies did not map properly), the software will return to the Simulation Editor dialog and provide a chance to correct the error. Otherwise, the simulation created successfully and is now loaded into the main application workspace. At this time, all models should be created and initialized, and the simulation can be run.

1.2. Loading Existing Simulations

The process of loading a simulation begins in the Simulator::Load() function with loading a Simulation file, with (optional) extension ".sim", an ASCII text file that has the format specified in the below section. These files can be either created in the GrooveNet software or edited in your favorite text editor (e.g. Vim or Xemacs).

1.2.1. Simulation File Specification

Each line is parsed in succession. Blank lines and lines beginning with the comment character ('%') are ignored. The simulation file is a list of simulation models, specified by their name (parameter="MODEL"), type (parameter="TYPE"), dependencies (parameter="DEPENDS"), and other parameters. Each non-comment line consists of parameter-value pairs separated by whitespace:

```
Parameter1=Value1 Parameter2=Value2 Parameter3=Value3...
```

Parameters are case-sensitive, except for the special parameters "MODEL", "TYPE", and "DEPENDS", and can contain whitespace. Values are not case-specific (may or may not be case-sensitive), and can

only contain spaces if contained in double-quotes. Values can also contain the following escape sequences:

- Single quote mark: \’
- Tab mark: \t
- Linefeed: \n
- Carriage return: \r
- Backslash: \\
- Hexadecimal character code: \x[0-F][0-F]
- Octal character code: \x[0-1][0-7][0-7]

Numeric values can also contain the following suffixes:

- Giga- (10^{9}): G
- Mega- (10^{6}): M
- Kilo- (10^{3}): K
- Milli- (10^{-3}): m
- Micro- (10^{-6}): u
- Nano- (10^{-9}): n
- Pico- (10^{-12}): p
- Femto- (10^{-15}): f
- Percent: %

These codes are specified in StringHelp.h.

Parameter-value pairs between model name (parameter=“MODEL”) parameters are considered to belong to the same model. Each model must have a unique name and a valid type associated with it; dependencies and other parameters are generally optional (although certain models may require specific parameters to function properly). All parameters are placed in a map, where duplicate parameters overwrite earlier instances of that parameter for the same model; this map is then sent to the model for use during initialization. The following in an example file:

```
1    % Created by GrooveNet Hybrid Simulator on 5/10/2006
2
3    MODEL="SightseeingModel0" TYPE="SightseeingModel"
4    MAXDISTANCE="1000" START="157 2nd Ave, New York, New York"
5
6    MODEL="UniformSpeedModel0" TYPE="UniformSpeedModel" DEPENDS="SightseeingModel0"
7    HIGH="-25%" LOW="25%" MULTILANE="YES" TRIP="SightseeingModel0"
8
9    MODEL="CarFollowingModel0" TYPE="CarFollowingModel"
     DEPENDS="SightseeingModel0;UniformSpeedModel0"
```

```
10    LEADER="UniformSpeedModel0" MULTILANE="YES" TRIP="SightseeingModel0"
11
12    MODEL="SightseeingModel1" TYPE="SightseeingModel"
13    MAXDISTANCE="1000" START="121 Wooster St, New York, New York"
14
15    MODEL="UniformSpeedModel1" TYPE="UniformSpeedModel" DEPENDS="SightseeingModel1"
16    HIGH="-25%" LOW="25%" MULTILANE="YES" TRIP="SightseeingModel1"
17
18    MODEL="CarFollowingModel1" TYPE="CarFollowingModel"
      DEPENDS="SightseeingModel1;UniformSpeedModel1"
19    LEADER="UniformSpeedModel1" MULTILANE="YES" TRIP="SightseeingModel1"
20
21    MODEL="CarListVisual0" TYPE="CarListVisual"
22    CAPTION="GrooveNet Widget" DELAY="0.2" HEIGHT="497" WIDTH="504"
23
24    MODEL="MapVisual0" TYPE="MapVisual"
25    CAPTION="GrooveNet Widget" DELAY="0.2" FOLLOW="" HEIGHT="497" START_POS="0, 0" WIDTH="504"
      ZOOM="4"
26
27    MODEL="SimplePhysModel0" TYPE="SimplePhysModel"
28    MAXDISTANCE="200" MULTICHANNEL="YES"
29
30    MODEL="SimpleLinkModel0" TYPE="SimpleLinkModel"
31
32    MODEL="SimpleCommModel0" TYPE="SimpleCommModel"
33    GATEWAY="NO" RBXJITTER="YES" REBROADCAST="YES" REBROADCASTINTERVAL="1"
34
35    MODEL="SimModel0" TYPE="SimModel"
      DEPENDS="SimplePhysModel0;SightseeingModel0;SimpleLinkModel0;CarFollowingModel0;SimpleCommModel0"
36    COMM="SimpleCommModel0" DELAY="0.2" DOLOG="YES" ID="127.0.0.1" LINK="SimpleLinkModel0"
      MOBILITY="CarFollowingModel0" PHYS="SimplePhysModel0" START="0" TRACKSPEED="NO"
      TRIP="SightseeingModel0"
37
38    MODEL="SimplePhysModel1" TYPE="SimplePhysModel"
39    MAXDISTANCE="200" MULTICHANNEL="YES"
40
41    MODEL="SimpleLinkModel1" TYPE="SimpleLinkModel"
42
43    MODEL="SimpleCommModel1" TYPE="SimpleCommModel"
44    GATEWAY="NO" RBXJITTER="YES" REBROADCAST="YES" REBROADCASTINTERVAL="1"
45
46    MODEL="SimModel1" TYPE="SimModel"
      DEPENDS="SimplePhysModel1;SimpleLinkModel1;SightseeingModel1;CarFollowingModel1;SimpleCommModel1"
47    COMM="SimpleCommModel1" DELAY="0.2" DOLOG="YES" ID="127.0.0.2" LINK="SimpleLinkModel1"
      MOBILITY="CarFollowingModel1" PHYS="SimplePhysModel1" START="0" TRACKSPEED="NO"
      TRIP="SightseeingModel1"
48
49    MODEL="TrafficLightModel0" TYPE="TrafficLightModel"
50    GREENLIGHTTIME="30"
```

1.2.2. Simulation Loading Process

Once the simulation file has been loaded into memory, models are added to the model manager using the ModelMgr::AddModel() function without specifying dependencies. This function creates a model of the specified name and type and initializes it by calling that model's initialization function, passing the provided parameters for that model. Then, the loader saves the model's dependency information for later.

Once all models have been created, the loader attempts to resolve dependencies by calling ModelMgr::BuildModelTree(). This function constructs a dependency graph and reduces that graph to a tree. If that graph contains any cycles (resulting from circular dependencies), or dependencies do not exist, this procedure will fail, and the simulation will fail to load.

If the simulator succeeds in loading, the simulation will appear ready to run. Otherwise, the simulation will fail to load, and the main application workspace will remain empty.

1.3. Saving Simulations

Saving a simulation creates a new or replaces an existing simulation file of the format described in the previous section. The Simulator::Save() function writes each existing model's parameters to the file using the model's Model::Save() function. The models are written in dependency order through recursive calls of Simulator::internalSave(), so that models are written after their dependencies in the file. This simulator creates relatively verbose simulation files since all model parameters, whether default values or not, are written to the file.

1.4. Unloading Simulations

Unloading simulations involves destroying all the models in dependency order, by first calling each model's Model::Cleanup() function, and freeing that model from memory and the model manager's list of models. Before a simulation can be unloaded, it must be stopped if it is running using the wait() function. Upon exiting the program, any currently loaded simulation will be unloaded before the software exits completely.

1.5. The Model Manager

The model manager, implemented in the ModelMgr class, maintains a registry of all loaded models and model types on behalf of the Simulator class. The model manager thus contains several important data structures, including a map of model creation functions for each model type, a list of pointers to each existing model instance mapped by model name, and an array of nodes in the model dependency tree. The array specifying the model dependency tree contains the model instance name, pointer, list of pointers to dependency nodes, and status bits (dirty, error, etc.) for each model instance. This tree is created by the ModelMgr::BuildModelTree() function which reduces the model dependency graph to a tree by sorting nodes by minimum degree and removing redundant edges. Use the ModelMgr::AddModel() function specifying dependencies to add a model during simulation runtime. The model manager uses a recursive-locking mutex to make accesses and modifications to the model list atomic and thread-safe.

1.6. Running a Simulation

To run a simulation, the main application window displays the Run Simuation dialog and calls the Simulator::start() function with the result. This function then stops the currently running simulation if a simulation is running, creates the log files for this run, and starts the main simulation thread using QThread::start() with the specified priority.

The thread's entry point is the Simulator::run() function, which continues execution until all iterations of the simulation are completed or the simulation is stopped (cancelled). For each iteration (trial) of the simulation, the simulator sets the current time mode (real-time or simulated), constructs a new event queue and fills the queue with the messages specified in the Run Simulation dialog, and initializes all models in dependency order by calling the models' Model::PreRun() function. Once this is done, the simulation thread will repeatedly check the event queue for events until this iteration of the simulation is complete or the simulation is stopped. As the simulation runs, events are dequeued and sent to the destination model's event processor (the Model::ProcessEvent() function).

1.6.1. Simulator Event

Simulation events, implemented by the SimEvent class, contain useful information. Each simulation event contains the time that event occurs at, the originating and destination model name, the event type and priority, and a pointer to any data associated with that event. For the developer's convenience, one can specify a destruction function to call when the event queue is cleared to free the associated data.

1.6.2. Simulator Event Queue

Events are stored in a first-in-first-out (FIFO) queue that organizes contained events based on timestamp and priority, where zero represents the highest priority. The event queue supports periodic events by allowing a model to reinsert an event with a new timestamp. The event queue uses a mutex to ensure that accesses and modifications are thread-safe. This is merely a precaution, however, since at present, events are only handled by the simulation thread.

2. Simulation Models and Visualizers

The simulator uses model and visualizer classes to represent particular behaviors in a modular fashion. Many of these models have more abstract forms that define capabilities and more specific implementations that realize those capabilities.

2.1. Design and Implementation

This section describes the design and implementation of simulation models and visualizers from a global perspective.

2.1.1. Model Lifecycle

When a model is first created, that model's constructor, as well as constructors for all parents are called. Then, the model is initialized using that model's Model::Init() function. This function provides a list of parameters specified in either the Simulation Creation dialog or the simulation file. An example of this function is provided below:

```
 1  int SimplePhysModel::Init(const std::map<QString, QString> & mapParams)
 2  {
 3          QString strValue;
 4
 5          if (CarPhysModel::Init(mapParams))
 6                  return 1;
 7
 8          strValue = GetParam(mapParams, SIMPLEPHYSMODEL_PARAM_DISTTHRESH,
    SIMPLEPHYSMODEL_PARAM_DISTTHRESH_DEFAULT);
 9          m_fDistanceThreshold = ValidateNumber(StringToNumber(strValue), 0., HUGE_VAL);
10
11          ...
12
13          return 0;
14  }
```

Figure 1: Model::Init() Code Example

In this example, a physical layer model is initialized, first by calling the parent's initialization function, then extracting parameters using the GetParam() function, and finally, by returning the status code (0 = success).

Once a model has been created and initialized, it can be used in as many tests as desired. Each test begins by performing pre-run initialization for all the models, using the Model::PreRun() function. An example of this function is provided below:

```
1  int SimplePhysModel::PreRun()
2  {
3          if (CarPhysModel::PreRun())
4                  return 1;
5
6          m_iMessages = 0;
```

```
7           return 0;
8   }
```

Figure 2: Model::PreRun() Initialization Code Example

In this example, the model's pre-run initialization begins by calling the parent's pre-run initialization function. Then, state variables are reset as necessary, and the function returns the status code (0 = successful).

Once the simulation has gotten started, events are dispatched by the Simulator class to the destination models; the models are expected to use the Model::ProcessEvent() function to process each event and/or pass that event back to the parent model.

```
1   int SimpleCommModel::ProcessEvent(SimEvent & event)
2   {
3           if (CarCommModel::ProcessEvent(event))
4                   return 1;
5
6           switch (event.GetEventID())
7           {
8           case EVENT_CARCOMMMODEL_REBROADCAST:
9           {
10                  RebroadcastMessage * pRBXMsg = (RebroadcastMessage *)event.GetEventData();
11                  if (pRBXMsg != NULL)
12                  {
13                          ...
14
15                          if (pRBXMsg->msg.m_tTime + pRBXMsg->msg.m_tLifetime <= tNext)
16                                  delete pRBXMsg;
17                          else
18                          {
19                                  event.SetTimestamp(tNext);
20                                  g_pSimulator->m_EventQueue.AddEvent(event);
21                          }
22                  }
23                  break;
24          }
25          default:
26                  break;
27          }
28
29          return 0;
30  }
```

Figure 3: Model::ProcessEvent() Code Example

As shown in the above example, the event is first sent to the parent model's event handler; then, if the event is of the specific type EVENT_CARCOMMMODEL_REBROADCAST, that event is then processed. This particular type of event has data associated with it; freeing this data is the programmer's responsibility in the case it is no longer needed. In the event that the event needs to occur again at a later time, that event's timestamp is modified, and the resulting event is added to the simulator's event queue, as shown in lines 19-20. Once the processing of the event is complete, the function returns the status code (0 = success).

Once the simulation has ended, the simulator performs post-run cleanup, calling each model's Model::PostRun() function. This function, shown below, essentially performs whatever additional steps need to be done before the current run of the simulation comes to an end.

```
1   int CarListVisual::PostRun()
2   {
3           if (TableVisualizer::PostRun())
4                   return 1;
5
6           UpdateTable();
```

```
7
8           return 0;
9     }
```

Figure 4: Model::PostRun() Code Example

In this example, the post-run cleanup is performed in essentially the same manner as the other functions discussed. This function is called each time a simulation ends, unlike the Model::Cleanup() function, which performs final cleanup before a model can be safely destroyed. An example of such a function is provided below.

```
1     int CarListVisual::Cleanup()
2     {
3           QDraggingTable * pTable = m_pWidget == NULL ? NULL : ((QTableVisualizer *)m_pWidget)-
      >m_pTable;
4           if (pTable != NULL)
5                 pTable->setNumRows(0);
6
7           if (TableVisualizer::Cleanup())
8                 return 1;
9
10          m_mapObjectsToRows.clear();
11
12          return 0;
13    }
```

Figure 5: Model::Cleanup() Code Example

Once this function completes successfully, the model is removed from memory, and the model's "life" is over. However, during the model's lifecycle, two other functions are commonly used: Model::GetParams() and Model::Save().

The Model::GetParams() function must be implemented in every model class and should call the base class' implementation as well as specify the parameters that this model accepts. This function is used when the user is creating or editing a model to populate the list of parameters for that model. For each parameter, the parameter's name, default value, description, and auxiliary data (optional) are specified. An example of this function is shown below.

```
1     void SimplePhysModel::GetParams(std::map<QString, ModelParameter> & mapParams)
2     {
3           CarPhysModel::GetParams(mapParams);
4
5           mapParams[SIMPLEPHYSMODEL_PARAM_DISTTHRESH].strValue =
      SIMPLEPHYSMODEL_PARAM_DISTTHRESH_DEFAULT;
6           mapParams[SIMPLEPHYSMODEL_PARAM_DISTTHRESH].strDesc =
      SIMPLEPHYSMODEL_PARAM_DISTTHRESH_DESC;
7           mapParams[SIMPLEPHYSMODEL_PARAM_DISTTHRESH].eType = ModelParameterTypeFloat;
8           mapParams[SIMPLEPHYSMODEL_PARAM_DISTTHRESH].strAuxData = "0.:";
9
10          mapParams[SIMPLEPHYSMODEL_MULTICHANNEL_PARAM].strValue =
      SIMPLEPHYSMODEL_MULTICHANNEL_PARAM_DEFAULT;
11          mapParams[SIMPLEPHYSMODEL_MULTICHANNEL_PARAM].strDesc =
      SIMPLEPHYSMODEL_MULTICHANNEL_PARAM_DESC;
12          mapParams[SIMPLEPHYSMODEL_MULTICHANNEL_PARAM].eType =
      (ModelParameterType)(ModelParameterTypeYesNo | ModelParameterFixed);
13    }
```

Figure 6: Model::GetParams() Code Example

The Model::Save() function is used to construct a list of parameter-value pairs from the model's settings to be written to a file or for editing. The result, if passed to the Model::Init() function, should create an equivalent model. An example of this function is provided below.

```
1     int SimpleCommModel::Save(std::map<QString, QString> & mapParams)
2     {
3             if (CarCommModel::Save(mapParams))
4                     return 1;
5
6             mapParams[SIMPLECOMMMODEL_REBROADCASTINTERVAL_PARAM] =
      QString("%1").arg(ToDouble(m_tRebroadcastInterval));
7             mapParams[SIMPLECOMMMODEL_DOREBROADCAST_PARAM] = BooleanToString(m_bRebroadcast);
8             mapParams[SIMPLECOMMMODEL_MOBILEGATEWAY_PARAM] = BooleanToString(m_bGateway);
9             mapParams[SIMPLECOMMMODEL_RBXJITTER_PARAM] = BooleanToString(m_bJitter);
10            return 0;
11    }
```

Figure 7: Model::Save() Code Example

This function follows the same structure as the Model::Init() function.

2.1.2. Model Hierarchy

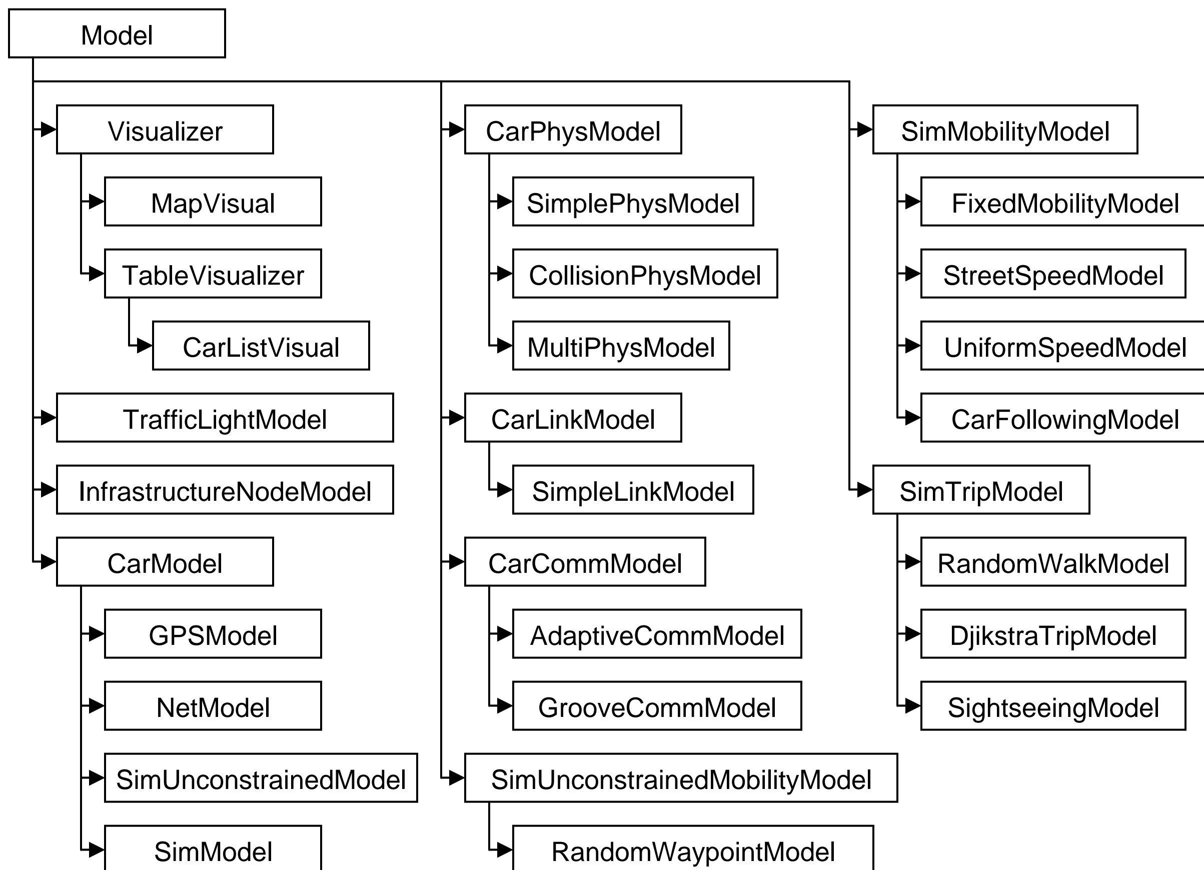

Figure 8: GrooveNet Model Class Inheritance Hierarchy

In GrooveNet, all simulation models are descended from the Model class. In addition, all visualizers are descended from the Visualizer class. All of these models are described in detail in the following sections.

2.1.3. Creating a New Model Type

In order to create a new type of model, follow this procedure:

1. Create a new class and derive it from the proper base class. All models should at least be derived from the Model class, and visual elements should be at least derived from the Visualizer class. Populate this class with member functions. Ensure that all functions that may be overridden by a derived class are declared virtual. Also, check to make sure that functions overriding base class functions also are declared virtual. Finally, check to make sure that the destructor is virtual. This allows C++ polymorphism to work properly with your model.

2. Override the important functions from the base class. These functions may include the lifecycle functions described earlier, such as Model::Init() or Model::ProcessEvent(). Make sure to call the base class equivalents where necessary.

3. Implement a model creation function, like the following example from the SimplePhysModel class. Substitute the name of your model where appropriate.

```
inline Model * SimplePhysModelCreator(const QString & strModelName)
{
	return new SimplePhysModel(strModelName);
}
```

4. In the ModelMgr::ModelMgr() function, add the following line. You will also probably have to #include the model's header file in ModelMgr.cpp. This step adds the model type to the model manager and associates the model's creation function with the model type name.

```
RegisterModel(<Model creation function name>, <Model type string>);
```

5. In the QSimCreateDialog::QSimCreateDialog() function, add the following lines. You will also probably have to #include the model's header file in QSimCreateDialog.cpp. This step adds the model type to the Simulation Creation dialog, associating a model type name and parameters with the particular model class.

```
<Model class>::GetParams(m_mapModelParams[<model type string>]);
m_vecModelTypes.push_back(<model type string>);
```

6. Recompile the software (adding source and header files to the makefile if necessary). The model should now be incorporated into the simulator.

2.2. Vehicle Models

Several types of vehicles are used in GrooveNet: the infrastructure node, a GPS-powered vehicle, a network-powered vehicle, a simulated vehicle, and a simulated vehicle not linked to the map database (unconstrained). All except the first are derived from the CarModel base class. The infrastructure node model is derived directly from the Model class. All these models use helper models for mobility and communication.

2.2.1. Communication Models

There are three basic layers used by all vehicles for communication. The physical layer, which is modeled by the CarPhysModel class and derived classes, represents the physical hardware and the communication medium itself (e.g. wireless). The link layer, which is modeled by the CarLinkModel class and derived classes, represents the filter protocol used to discriminate between relevant and irrelevant network traffic; the link layer passes relevant traffic up to the application. The communication layer, which is modeled by the CarCommModel class and derived classes, represents the rebroadcast policy used by a vehicle to forward network traffic. Each of the derived classes is discussed below:

- CarPhysModel

- SimplePhysModel – performs simple check to enforce communication range; supports multiple communication channels on the same receiver
- CollisionPhysModel – same as SimplePhysModel, but includes collision modeling
- MultiPhysModel – models multiple receivers and uses different physical layer models for messages from vehicles and messages from infrastructure nodes; especially useful for mobile gateways

- CarLinkModel
  - SimpleLinkModel – verifies that the packet has not been previously received and that the receiver lies within that packet's geographic region of relevance
- CarCommModel
  - SimpleCommModel – rebroadcasts packets at a specific rate; supports jittering
  - AdaptiveCommModel – same as above, but implements several advanced features: first-fast rebroadcasting, distance-based rate throttling, load-based rate throttling, and location-based broadcast suppression; each of these features can be enabled or disabled
  - GrooveCommModel – (not yet implemented at time of release) a placeholder for the GrooveNet Adaptive Broadcast Model

Additional classes can be added using the method discussed previously.

2.2.2. Mobility Models

There are several types of mobility models. First, the infrastructure node uses no mobility model, assuming that it is stationary. The infrastructure node's position can be specified in the InfrastructureNodeModel class itself. The GPSModel and NetModel classes derive their mobility from external sources (a Global Positioning System and a network, respectively). The SimUnconstrainedModel class uses special unconstrained mobility models. These models, such as that implemented in the RandomWaypointModel class, specify a starting location and a method for wandering in an unconstrained two-dimensional space. Finally, the SimModel class is the most sophisticated, using two types of models to simulate motion: a mobility model for executing motion and a trip model for planning routes. The derived classes for each of these is described below:

- SimUnconstrainedMobilityModel
  - RandomWaypointModel – wander at a random speed and direction continuously
- SimMobilityModel
  - FixedMobilityModel – remain at a fixed position/do not move
  - StreetSpeedModel – vehicle always moves at the speed limit
  - UniformSpeedModel – vehicle moves at some speed uniformly distributed about the speed limit; the user specifies the lower and upper bounds relative to the speed limit in either absolute or percentage-based values

- CarFollowingModel – vehicle will not exceed the speed of the vehicle in front, vehicle will use a different mobility model (user-specified) when vehicle is the leader

- SimTripModel

  - RandomWalkModel – vehicle will randomly choose where to go whenever vehicle approaches an intersection

  - DjikstraTripModel – vehicle plans the shortest route from the source to the destination location (possibly visiting waypoints if they are specified); if no destination is specified, vehicle will randomly walk after visiting waypoints

  - SightseeingModel – vehicle randomly walks until it is a certain distance from the starting point; the vehicle then takes the shortest path back to the starting point and starts again along a different path

Additional classes can be added using the method discussed previously.

2.3. Visualizers

Simulations are much more exciting if the user can see what's happening as it happens. For this purpose, several visual displays that can exist in the main application workspace are defined. These displays are implemented using the Qt user interface specification, but the details are hidden away by the Visualizer, TableVisualizer, and other classes. Specifically, the Visualizer class manages the display widget for the programmer, and the TableVisualizer class does the same for a QTable-based widget. To design visual displays based on other types of widgets, such as text editors, the recommended procedure is to create a new generic type of visualizer that has as a member variable the desired type of widget, binds all useful signals and slots of that widget to class member functions, and handles creation and destruction of the widget gracefully. The Visualizer and TableVisualizer classes are two good examples of such generic visualization classes. Then, one simply has to derive a specific class, such as a MapVisual or a CarListVisual, from the appropriate generic class and implement the display's particular behavior.

2.4. Other Models

At this point, only one other model, the TrafficLightModel class, exists. The purpose of the TrafficLightModel class is to manipulate virtual traffic signals on the map, so that vehicles do not "crash" in the intersection. At this point, the TrafficLightModel is very simple, implementing only red and green lights, and no turn signals. Of course, more specialized models can be added using the procedure discussed before.

3. User Interface Components

The GrooveNet user interface is designed to be both rich in features and easy to use. This section discusses the major features of the user interface.

3.1. Main Application Window

The main application window, implemented in MainWindow.cpp, consists of several major components. First, the workspace area is the large central region of the window that contains the different visual displays, including in this case, a vehicle status display and a map display. Below the workspace is the network manager, which allows the user to connect to instances of the GrooveNet software running on other machines using a client/server network connection, that is implemented in the QNetworkManager

class. The menu bar above allows the user to create, edit, load, save, or run simulations; configure the program options; pause or stop the running simulation; disconnect from or reconnect to the network; start or stop the server; and cascade, tile, or close the visual display windows. The status bar at the bottom of the screen displays status messages during program execution. Not shown is the terminal; the application will periodically print messages to the application's associated console window.

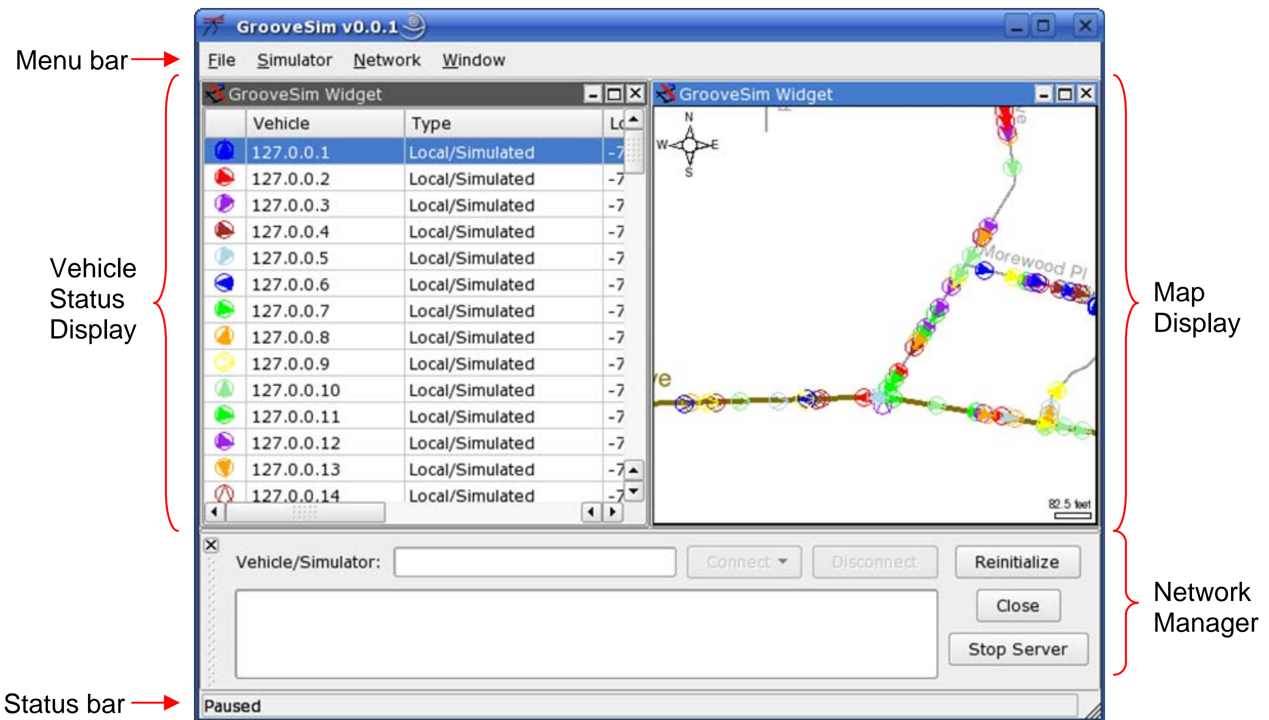

Figure 9: Main Application Window

3.2. Simulation Editor Dialog Box

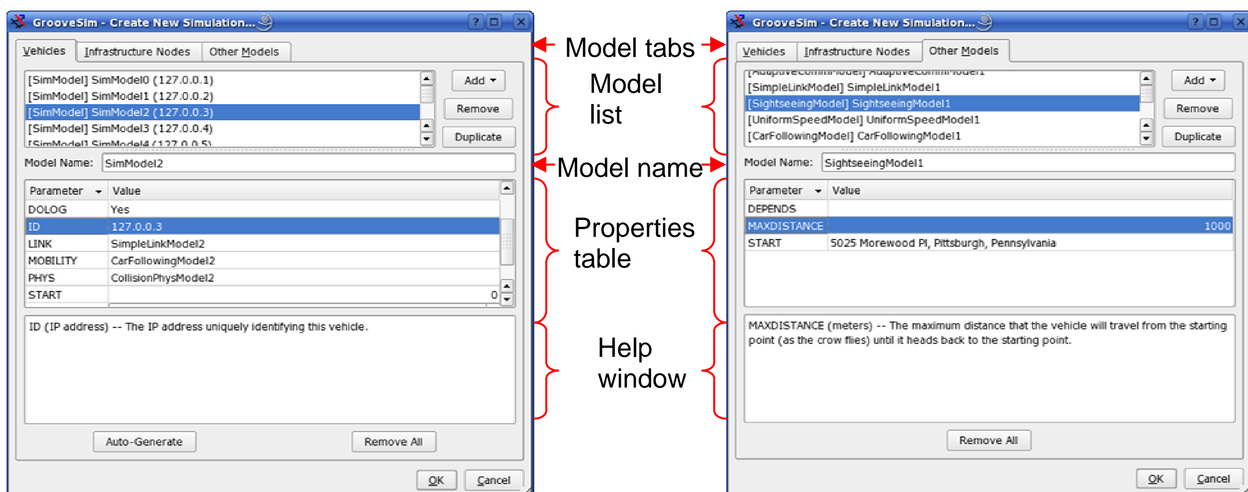

Figure 10: Simulation Editor Dialog

The Simulation Editor dialog contains multiple tabs, one for each model category (vehicles, infrastructure nodes, and other models). Each tab is essentially the same in layout: a list of models with buttons to add, remove, or duplicate them; a table of properties for the selected model below a text editor for the model name; and a help window that displays a caption for the selected property. The first two tabs have Auto-Generate buttons, which when pressed will display an Auto-Generate Nodes dialog allowing the user to create vehicles or infrastructure nodes in bulk; all tabs have Remove All buttons that remove all contained models. When the Remove All button is pressed for vehicles or infrastructure nodes, their associated models are removed as well from the third tab. This dialog box is used for both creating new and editing existing simulations.

3.3. Auto-Generate Nodes Dialog Boxes

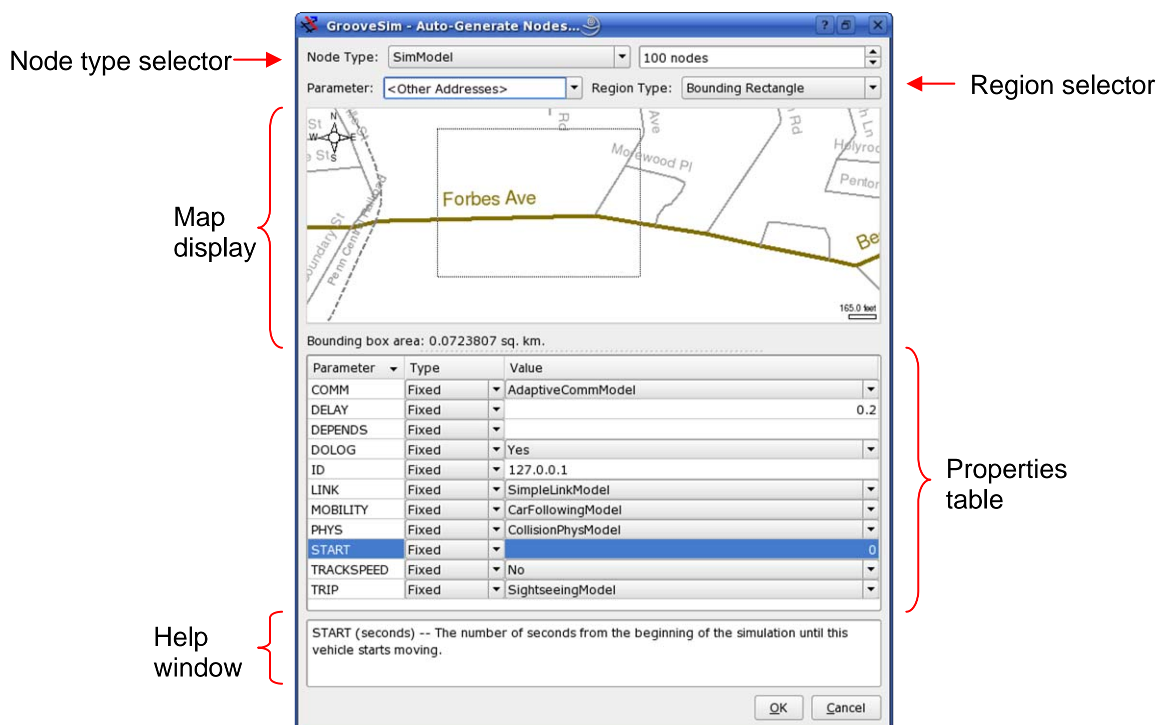

Figure 11: Auto-Generate Nodes Dialog

The first dialog box pops up when the user desires to automatically generate either vehicles or infrastructure nodes. This dialog box allows the user to create a large number of these nodes that share common properties. At the top, the user specifies the particular type of node and how many to create. Just below, the user can specify regions on the map display for use when generating random street addresses or coordinates. The software supports different regions for each parameter name; a parameter name can be typed into the dropdown box that says “<Other Addresses>”. The type of region can be set using the region type dropdown box to the right. The user can click and drag on the map display to select an area; to select an area not shown on the map, right-click on the map and select “Find address…” to recenter the map at a specific location. The area of the selected region is shown below the map. Below the map display is the table of properties for that node/vehicle type. These properties can be fixed to the same value for all the models; some properties can be randomized to different values or read from a text file listing values one to a line. Some of these properties ask the user to choose a model type; another dialog shown below will popup once the user presses the OK button, prompting the user to enter parameters for those models separately. The help window at the bottom portrays a helpful caption for the selected parameter.

Once the user presses the OK button, a new dialog box will appear for each associated model to be created (except for the NullModel class, which is a null-model placeholder for any model type). This dialog box, the Specify Model Parameters dialog, is shown below. This dialog works in the same way as the previous one, with the exception of the missing map display. The user can return to the Auto-Generate Nodes dialog by pressing the Cancel button.

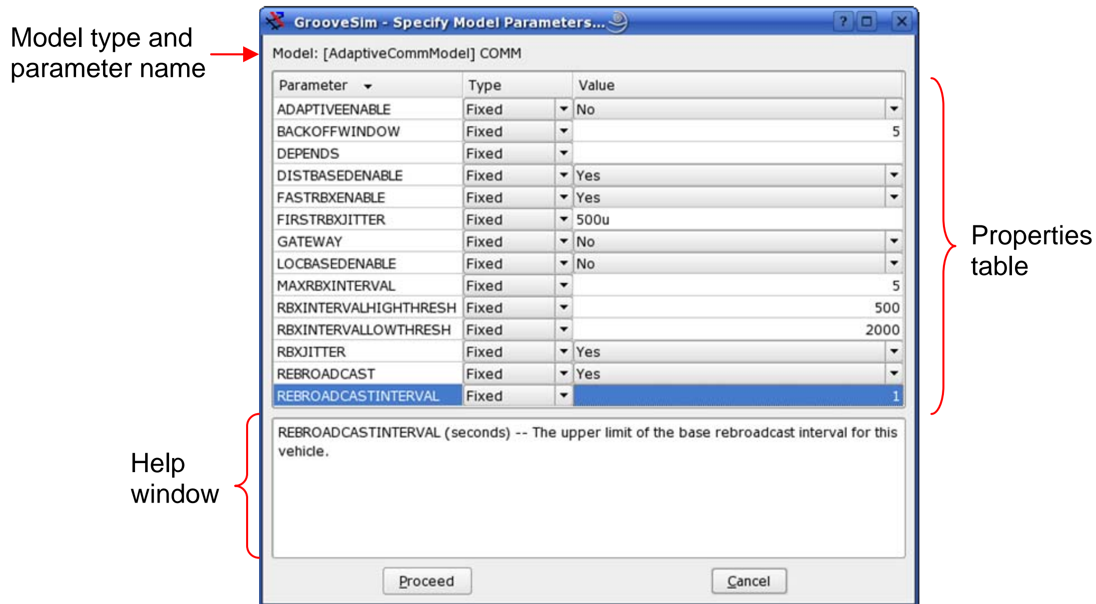

Figure 12: Specify Model Parameters Dialog

3.4. Run Simulation Dialog Box

This dialog allows the user to set up a single simulation run or a Monte Carlo series of simulation runs, with an optional maximum duration, a list of messages to automatically send during each run, and a list of paths for each type of log file. By pressing the Add… or Edit… buttons, the Send Message dialog appears, allowing the user to specify a message; these messages are automatically triggered by the simulator at the proper time. The simulation can run either in real time or in simulated fixed time increments.

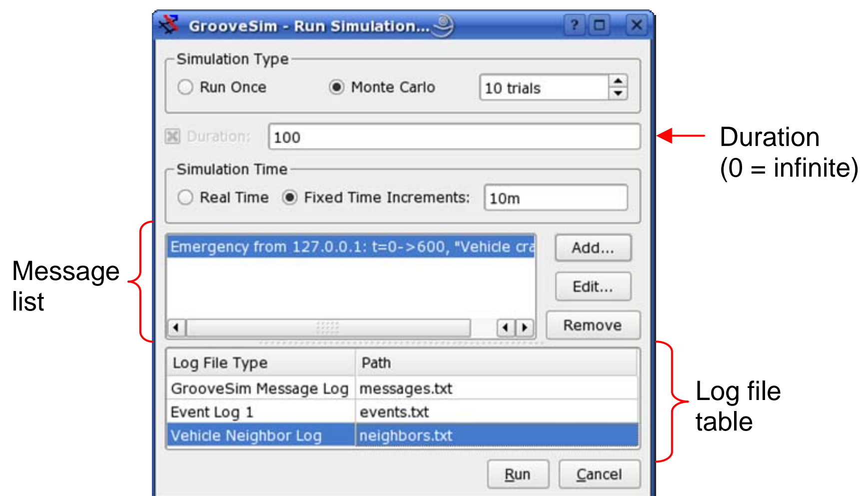

Figure 13: Run Simulation Dialog

3.5. Send Message Dialog Boxes

When the user either adds a message to the simulation to be triggered at runtime or attempts to send a message manually, the Send Message dialog appears, allowing the user to choose the message source, type, text, bounding region, optional destination street address, transmit time, and lifetime. When sending a message manually, the user right-clicks on a vehicle in the vehicle status display, selects "Send Message…" and this dialog appears with the message source automatically selected. The bounding region is optional, and if a bounding region is desired, the particular region can be configured by selecting the bounding region type or pressing the Configure… button. Either way, another dialog box appears, and the user can select the bounding region on a map and specify any parameters in the text box below. The message destination must be a valid street address or intersection; if the message reaches that location before expiring, a message will be output to the terminal informing the user of the time it took to reach that location.

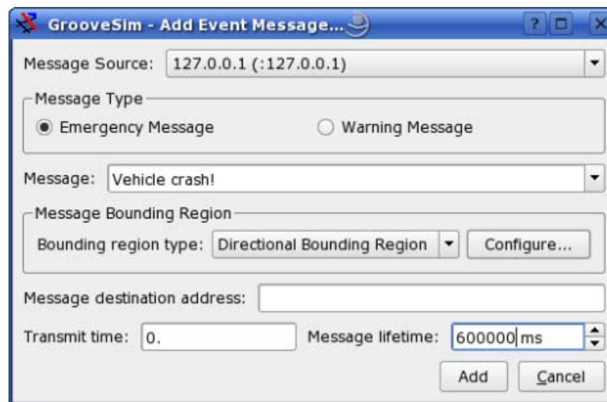

Figure 14: Send Message Dialog

By pressing the Add or Send button, the message is created by calling the function referred to by the m_pfnAcceptCallback member variable. For adding a message to the simulation, no function is actually called (m_pfnAcceptCallback is NULL), but the parameters are captured directly in the function QSimRunDialog::slotAddMessage() or QSimRunDialog::slotEditMessage().

3.6. Configure GrooveNet Dialog Box

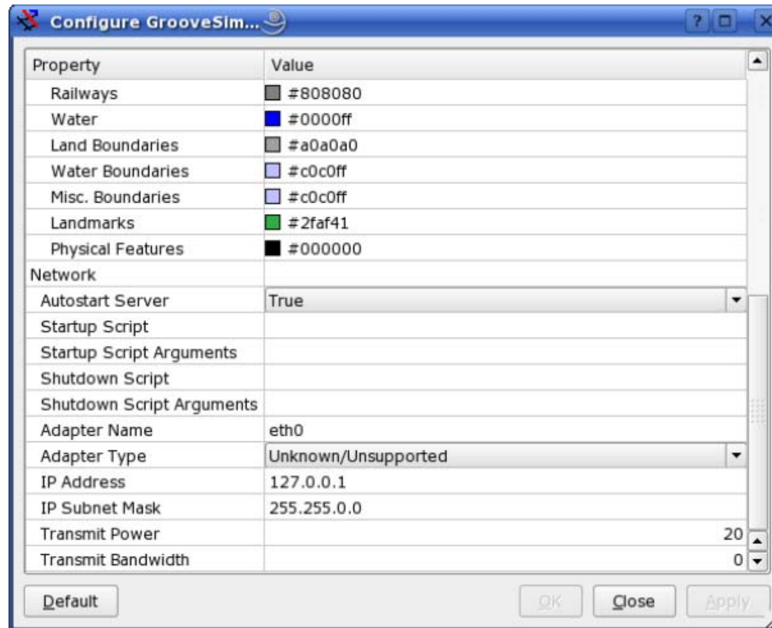

Figure 15: Configure GrooveNet Dialog

This dialog allows the user to customize the GrooveNet software by changing parameters controlling the program's appearance, network properties, and behavior. These parameters are read from the Settings class, which uses a QSettings class to store the settings in a file or in the system registry (platform dependent). Note that the colors are specified in hexadecimal code.

4. Map Database

The road database is actually a non-negative weighted edge graph where each road, rail, or water segment represents an edge, and every intersection between two edges is a vertex. The programmer can use the road database to draw maps, find the shortest path between two street addresses, locate street addresses on a map, and more.

4.1. GrooveNet Database Format

The GrooveNet map database is stored as a binary-encoded file with the following format (there are no linefeeds in the file). This map database is based on the US Census Bureau's TIGER/Line 2000+ database format, and many of the concepts used in the GrooveNet database format are analogous to their use in the TIGER/Line 2000+ database format. For more information, please consult the US Census Bureau's TIGER/Line 2000+ format specification, found online at http://www.census.gov/geo/www/tiger/.

The map database format is used by the MapDB class. It includes a list of strings for all the feature names in the database, where the features reference the strings by index rather than by value to conserve space. During the loading process (see the MapDB::LoadMap() function for details), two tasks are performed: the file's lists of strings, vertices (intersections), records, and water polygons are merged with those already present in the database; and the county is segmented into a 10x10 grid, and lists of records are generated for each section of the grid.

```
<county code><bounding rectangle>
<number of strings>
```

```
<string 1>
<string 2>
...
<string n>
<number of vertices>
<vertex 1 coordinates><vertex 1 number of neighbors>
<neighbor vertex 1><neighbor record 1>
<neighbor vertex 2><neighbor record 2>
...
<neighbor vertex k><neighbor record k>
<vertex 1 number of roads>
<road record 1>
<road record 2>
...
<road record k>
...
<vertex n coordinates><vertex n number of neighbors>
<neighbor vertex 1><neighbor record 1>
<neighbor vertex 2><neighbor record 2>
...
<neighbor vertex k><neighbor record k>
<vertex n number of roads>
<road record 1>
<road record 2>
...
<road record k>
<number of records>
<record 1 number of feature names>
<feature name string index 1><feature type string index 1>
<feature name string index 2><feature type string index 2>
...
<feature name string index k><feature type string index k>
<record 1 type, water on the right, water on the left>
<record 1 cost>
<record 1 coordinates of water on the left>
<record 1 coordinates of water on the right>
<record 1 number of street address ranges>
<address range 1 start><address range 1 end><address range 1 zip code, side of record>
<address range 2 start><address range 2 end><address range 2 zip code, side of record>
...
<address range k start><address range k end><address range k zip code, side of record>
<record 1 bounding rectangle>
<record 1 number of shape points>
<shape point 1 coordinates>
<shape point 2 coordinates>
...
<shape point k coordinates>
<record 1 number of vertices>
<vertex 1 index>
<vertex 2 index>
...
<vertex k index>
...
<record n number of feature names>
<feature name string index 1><feature type string index 1>
<feature name string index 2><feature type string index 2>
...
<feature name string index k><feature type string index k>
<record n type, water on the right, water on the left>
<record n cost>
<record n coordinates of water on the left>
<record n coordinates of water on the right>
<record n number of street address ranges>
<address range 1 start><address range 1 end><address range 1 zip code, side of record>
<address range 2 start><address range 2 end><address range 2 zip code, side of record>
...
<address range k start><address range k end><address range k zip code, side of record>
<record n bounding rectangle>
<record n number of shape points>
<shape point 1 coordinates>
<shape point 2 coordinates>
```

```
...
<shape point k coordinates>
<record n number of vertices>
<vertex 1 index>
<vertex 2 index>
...
<vertex k index>
<number of water polygons>
<water polygon 1 bounding rectangle>
<water polygon 1 number of shape points>
<shape point 1 coordinates>
<shape point 2 coordinates>
...
<shape point k coordinates>
...
<water polygon n bounding rectangle>
<water polygon n number of shape points>
<shape point 1 coordinates>
<shape point 2 coordinates>
...
<shape point k coordinates>
```

This data format changes between versions; the user may want to recreate the map database files when loading a new version onto a machine.

4.2. TIGER/Line 2000+ Database Format

The TIGER/Line 2000+ database format, described completely on the US Census Bureau's website at http://www.census.gov/geo/www/tiger/, is a free street map database for every county in the United States. The entire database consists of a .zip file for each county, where each .zip file contains several cross-referenced databases comprising information about every known street, waterway, and shape polygon in that county. The TIGERProcessor class is responsible for converting this data into a GrooveNet map database.

4.3. Adding Other Formats

In order to add other formats, the developer needs to implement a conversion program that converts that database into a complete GrooveNet-compatible map database. Then, the MapDB class can be programmed to automatically convert the database formats by adding code to the MapDB::DownloadCounties() function that selects the type of converter to use based on some criteria.

5. Network Interface

The network interface facilitates communication between vehicles running on the local machine and vehicles running remotely. When a vehicle sends a packet, that packet is sent across the network to any servers specified in the network manager in the main application window.

5.1. Network Messages

Several network messages are implemented in the file Message.cpp. The primary type of network message is the Packet class; this structure contains the minimum information that should be sent from vehicle to vehicle. Each packet upon transmit can be identified uniquely by the source vehicle's IP address and sequence number; each packet upon receipt can be identified uniquely by these two fields, combined with an additional sequence number assigned by the receiver.

The Message class implements all the features of the Packet class, with additional information suitable for a GPS or alert broadcast. This class contains vehicle information for both the message origin and the last transmitter, since such packets can take multiple hops until they reach their destination. The

Message data structure also contains bounding region information that specifies the geographic area in which the message is relevant. Another type of message not currently used is the SquelchMsg class; this message may be used in the GrooveNet Adaptive Broadcast Model to alert vehicles closer to the origin of the message as to how far the message has propagated, so the vehicles can adjust their transmission rate accordingly. The SquelchMsg contains only vehicle information for the originator of the message.

The other types of messages contained in Message.h are used for logging data, not network communication, and will be discussed later.

5.2. Network Abstraction Layer

The Network Abstraction Layer (NAL) is an interface that allows whatever uses it to treat the network as an abstract entity, whether the network exists only on one machine or thousands. The NAL begins in the TransmitPacket() and ReceivePacket() functions found in the CarModel and InfrastructureNodeModel classes. The TransmitPacket() function communicates to both local and remote vehicles using the same packet format; the ReceivePacket() function is called equivalently during both local and remote communication.

The NAL goes deeper, however. Message transmissions and receptions over the external network are handled by generic Client and Server classes that contain all the important functions that the application layer needs for network communication. However, the actual network communication might actually be happening over TCP, UDP, or even using the Denso kits' special WAVE functions. By abstracting the communication layer, the application can act the same no matter what network is underneath it.

One limitation of the current implementation is that only one server can run at a time; this needs to be fixed.

5.3. UDP Client/Server

The UDP client and server, implemented in the classes UDPClient and UDPServer, use connectionless sockets to send individual datagrams over the network. One packet is transmitted at a time, and one packet is received at a time. UDP is good for frequently rebroadcast or rapidly obsolete information such as GPS or alert messages because it is a fast and simple protocol.

5.4. TCP Client/Server

The UDP client and server, implemented in the classes TCPClient and TCPServer (with the help of TCPListener, a connection listener thread), use connection-based sockets to send bytes over the network. The server receives byte streams from multiple clients on different sockets simultaneously, and multiple packets may arrive at once, complicating the decoding process. TCP is good because it is generally highly reliable and robust to packet loss.

5.5. Adding Other Client/Servers

Other clients and servers can be added by deriving new classes from the Client and Server base classes. Using the TCP or UDP implementations as a guide, adding new communication types is a breeze. Finally, just add the new client and server to the InitNetworking() function in Network.cpp.

6. GPS Interface

The Global Positioning System (GPS) interface allows the GrooveNet software to gather real-world location and velocity information in real time. The GPSModel vehicle type actively gathers GPS data from a user-specified GPS processor. Currently, the only implemented GPS processor supports the NMEA

0183 protocol, a widely established standard, received via a serial port terminal interface; this interface is also presently limited to Linux. However, adding other GPS processors is easy.

6.1. NMEA 0183 Processor

The NMEA 0183 standard is maintained by the National Marine Electronics Association (NMEA) and is used on boats, aircraft, and a wide variety of other applications. This standard is implemented as a stream of messages periodically transmitted to the computer's serial port. The GPS processor for NMEA messages, implemented in the NMEAProcessor class, supports many common message types, including the $GPGGA, $GPGLL, $GPGSA, $GPGST, $GPGSV, $GPRMC, $GPRRE, $GPVTG, and $GPZDA messages. Please refer to the NMEA 0183 standard for more information (can be ordered from the NMEA). Together, these messages provide accurate information about a vehicle's position and velocity anywhere on the earth.

6.2. Adding Other GPS Processors

The NMEA 0183 standard is not the only method out there; others include Garmin, Magellan, and raw binary messages. Therefore, to communicate with devices that use these other standards, other GPS processors must be added. To do so, create a new class derived from the GPSProcessor class (in GPSModel.cpp), call a method for populating that processor's parameters in the GPSModel::GetParams() function, add the protocol name to the list in the strAuxData field of the structure corresponding to the GPSMODEL_PROTOCOL_PARAM parameter in the GPSModel::GetParams() function, and add the processor to the processor creation code in GPSModel::Init().

7. Data Logger

The Logger class is responsible for logging test data to files and writing information to the console. To keep with the modular design methodology, the logger possesses generic support for writing data to a file; implementations are contained in the Logger::WriteHeader() and Logger::WriteMessage() functions to write specific data to the files. At the same time, the Logger writes debug information to a file (debug*.txt) and echoes that information to the terminal.

7.1. Logged Data Format

Several file types exist for logging data; each file type has its own particular format. However, these formats have several characteristics in common. Each format consists of a header followed by a series of messages, one to a line. Lines beginning with a comment character (%) represent either the header or actual comments and should be ignored when parsing the contained messages. Finally, all these files are ASCII formatted.

7.2. Simulation Data Logging

During a simulation, several types of data are logged.

In the GrooveNet Message Log, each alert message as it is originally transmitted and each initial receipt are written to the file; in particular, for each entry in the file, the sequence number, time, originating vehicle IP address, vehicle position and velocity, the transmitter and receiver IP addresses, the transmit and receive times, the RSSI (Received Signal Strength Indicator) and SNR (Signal-to-Noise Ratio), the number of hops, and the contained data are all printed on a line.

In the Event Log 1, for each alert message, the maximum distance the message has traveled, the maximum distance the originator of the message has traveled, and the total number of vehicles reached is printed on a line for each time interval, up to that time in the simulation.

In the Vehicle Neighbor Log, for each time interval, each vehicle's number of neighbors, transmitted messages, collisions, and current lane is printed on a line.

7.3. Debugger Logging

The Logger class also handles the printing and logging of debugging messages. These messages can be turned off by compiling in release mode or changing the value of CURRENT_WARNING_LEVEL to some value other than WARNING_LEVEL_DEBUG (see Logger.h). To log debugging information to the terminal and a file, use the function Logger::LogInfo(), with a suitable warning level (the warning level must not be greater than the value of CURRENT_WARNING_LEVEL to be printed). To log warnings and give the user a chance to respond via a popup message box, use either the Logger::PromptLogWarning() function or the Logger::LogWarning() function. To log errors and again give the user a chance to respond, use either the Logger::PromptLogError() function or the Logger::LogError() function.

7.4. Adding Other Data Formats

To add other data formats, simply increment LOGFILES (in Logger.h), add a description to the g_strLogFileNames global variable (declared in Logger.cpp), add new handlers in the Logger::WriteHeader() and Logger::WriteMessage() functions, and call the Logger::WriteMessage() function where appropriate.