The PLANSIM–Ttraffic simulation (nickname: LEGO)

Christian Gawron, Stefan Krauß, Peter Oertel, Peter Wagner
Center for Parallel Computing
University of Cologne
eMail: {gawron,krauss,p.wagner}@zpr.uni-koeln.de

April 15, 1998

Abstract

This text documents the "Lego" micro-simulation which is intended to simulate a road network. Lego's main feature is the ability to play with different traffic flow models in their natural environment, i.e. networks of streets.

The main features of Lego are:

- the ability to build any given highway topology from user supplied data files,
- easy modification of CA rules, and
- collection of statistics data (fundamental diagrams, time headways, lane usage distributions),
- build huge highway networks (via suitable converters) from databanks,
- model city traffic with all of its additional features (crossings, elaborate right of way rules),
- simulate traffic as fast as possible.

Warning! Unfortunately, not all of the documented features may be available, and others, though not documented, may exist. Please report any problems encountered to the PLANSIM–T-team.

Contents

Chapter 1

Overview

1.1 Starting the program

The simulation is started by the command `lego`, which has the following syntax:

```
lego [-f name] [-m] [-o] [-t steps] [-n] [-x] data
```

data is the datafile describing the topology. Options are: (A more current description of the command line options can be seen by typing `lego -?`)

`-f` *name*	specify outputfilename (default is `lego_xxx.out` where `xxxx` is the process id)
`-m`	enable the m4 preprocessor
`-o`	create statistics output
`-t` *steps*	number of time steps to be simulated
`-N` *threads*	run multithreaded (MP only).
`-S`	make screen dumps.
`-e` *name*	sets the file name for the emission table.
`-v`	verbose mode. (print lots of information concerning the program initialization)
`-x`	Enable X based graphics. (Default) All further options are passed to the XToolkit.

1.2 The main classes

Lego comprises mainly one object called SIMULATION. When the program is started it creates a table of all the network objects and puts them into one instance of this class, which then lets each of them do their job one after the other. Each round or timestep is cut into several stages called microsteps, so that all necessary actions can be arranged in the right order.

The network itself is built from several kinds of objects, which can communicate with each other. They fall into five categories:EDGES, CONNECTIONS, CROSSOVERS, INSTRUMENTS, and ROUTE-INFOS. The classes from which most features of lego derive are: EDGE, CONNECTION, and CARS. Derived from these are several subclasses. The different models (currently classical CA, continuous CA, and the so called SK-model) run on EDGES, in the form of SOURCES and SINKS they are also responsible for supplying the net with cars. CONNECTIONS hand the cars from the end of one EDGE to the beginning of another, while CROSSOVERS do something similar “horizontally”. They specify regions on one edge, where they take cars away and put them onto another one. They allow to simulate ramps and exits. Meanwhile INSTRUMENTS allow to direct the traffic according to specific events. TRAFFICLIGHTS and BUSSTOPS fall under this category.

Another base class is the CAR. As the name indicates CARS are the objects that are moved from site to site by the simulation. There are several subclasses to accomodate the different models. CONNECTIONS are responsible for customizing them if they change into a different model.

In addition, there are STATISTICSMODULES and GRAPHICSMODULES. They provide the data for the user.

1.3 The basic ideas

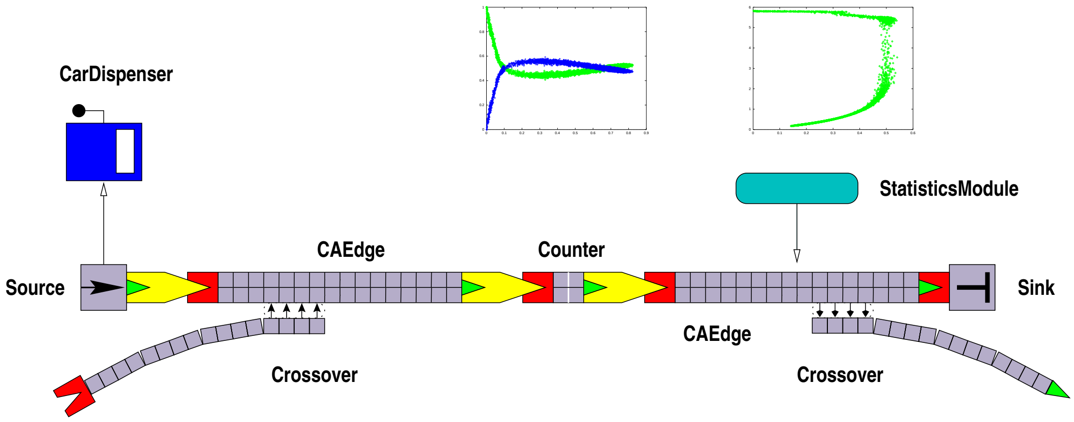

Figure 1.1: The main parts of a "Lego"-network and how they are put together

The structure of a "Lego" network is shown in fig. 1.1.

Basically, the lego system works like this: The cars drive on EDGES, which are connected to each other by CONNECTIONS. Cars that leave one edge are handed to the Connection that follows that edge and lane. The Connection knows which edge the car should go to and then hands it to that edge. In case that movement can be blocked by other cars or traffic lights, this is measured by INSTRUMENTS. They can tell the CONNECTIONS not to let cars pass through.

The following chapter describes all the different kinds of objects and how they can be put together to build a network. Then we give a description of the necessary data files. The appendix contains more detailed information about the car–following and lane–changing algorithms, and a number of examples for input files, that code simple networks.

Chapter 2

The Classes

All members of the simulation are derived from one class: the SIMULATIONOBJECT. This class has basic members which are common to all Objects actively taking part in the simulation. They have an *id*, which allows to identify them from all the other parts of the simulation. And they have an *operator (int)*. It is called by the simulation several times each timestep to allow the object to do its task.

Besides there is a pointer called *GraphicsModule*. If it is not zero the SIMULATIONOBJECT calls the related GRAPHICSMODULE every *timeStep* during the `MS_graphic` microStep.

This chapter is devoted to introducing all the different kinds of objects and their main features. The class hierarchy can be seen graphically in fig. 2.1.

2.1 The Edge Classes

In order to keep the program flexible, it is possible to use different kinds of edges. They all are derived from a class called EDGE.

The basic edge class EDGE has no CA! It is defined to have length zero and cars hitting an EDGE simply pass through it, without staying there. This class contains the methods and data structures which all kinds of edges have to possess. It has all the features, which are needed to connect it to other connections, but has no sites where cars can drive and no other functionality. Any edge class has to be derived from it, intended functionality is then added by overloading one or more methods of EDGE.

The derived classes CAEDGE HREDGE CONTEDGE SKEDGE are the heart of the simulation. They have an array called *cars* where a multilane road is stored and also have the methods to let the cars drive on the street.

2.1.1 The Most Important Methods of the Edge Classes

Length is the length in sites (not available in EDGE, default value is 0), *nLanes* the number of lanes of the edge (default 1). The lane with number 0 is always considered to be the rightmost lane. In the CAEDGE class and its descendants *cars* and *bcars* are the arrays, that contain the cars.

Each edge has structures called *toConnection* and *fromConnection* for each lane that tell it to which connections it is connected. They are set by the constructor of the respective CONNECTIONS.

lookForward and *lookBackward* give back how many sites in front of or behind a vehicle are free. The maximum horizon the car has is determined by its maximal velocity. If this implies that the car has to look beyond its current edge, the same methods exist on the CONNECTIONS which are pointed

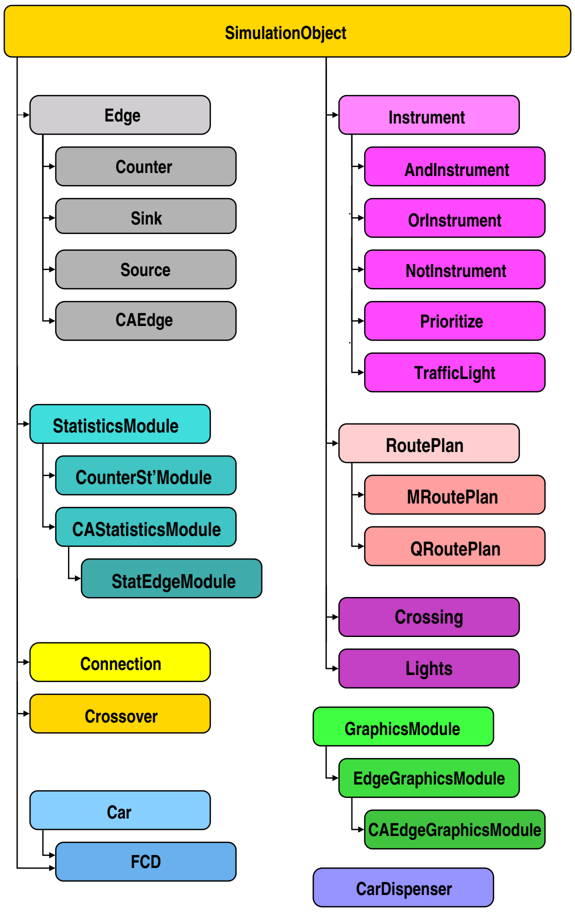

Figure 2.1: The classes of "lego" that are derived from SimulationObject

to by the *toConnection* and *fromConnection* structures. They then call the necessary routines on the following edge. This encapsulation is especially important when one puts different edges together.

If the edge is not connected to another one, *lookForward* will return zero and *lookBackward* will yield the "systemwide" maximum velocity (i. e. the maximum velocity the simulation allows).

Last but not least, there are two bool methods called *putNewVehicleOnEdge* and *giveVehicleToConnection*. When a car is going to leave an edge, the *giveVehicleToConnection* method is called, which in turn calls *receiveVehicleFromEdge* and then *giveVehicleToEdge* on the responsible CONNECTION. This method then hands the car to the following edge through *putNewVehicleOnEdge*. On CAEDGES this line of action is started by the *setVehicle* method.

2.1.2 The Cars

Each edge holds two arrays called *cars* and *bcars* of type pointer to class CAR.

Currently the following information is stored in the car class: *vmax* (the maximum velocity of the car), *pbrake* (the probability of the car to brake). *voffset*, *Prob_l2r*, and *Prob_r2l_ego* are parameters of the lane change rules. These parameters are for the classic CA. The continuous CA contain the same lane-change parameter, but has instead of *pbrake* two parameters: *bmax* and *sigma*, the maximum acceleration and the acceleration noise. The SK–Cars have another set of parameters: *vmax*, *bmax*, *dmax*, *sigma*, *voffset* (not used), *prob_l2r* and *length*. They refer to, respectively: maximum velocity, maximum acceleration, maximum deceleration, acceleration noise, and length of a vehicle. In addition there are parameters that tell the car which ROUTEINFO it should follow and allow it to keep track of its progress.

Beside this information the CAR class also has the following methods: *leaveEdge* and *enterEdge* which are called every time a car enters or leaves an edge, *ChangeToTheLeft* and *ChangeToTheRight* which are the lane changing rules and are called by the EDGES during the `MS_laneChange` microStep. (For further information on the rules see B.4.)

2.1.3 The Different Kinds of Edges

The following is a list of the different kinds of edges, which are available and already documented:

CAEdge is the oldest simulation model. It has a certain length and number of lanes. The vehicles on it are moved according to the STCA rules as developped by Kai Nagel and Michael Schreckenberg. The model is discrete in time, space, and velocity.

HREdge is an intermediate link of the evolution from CAEdge to ContEdge. It is no longer supported, and will probably go away soon. It allows to define more than *vmax*-velocity stages by refining the underlying grid. Use it on your own risk.

ContEdge is the extension of the original CA-rules to continuous space and velocity.

SKEdge is currently our "high–end" device. It implements a modified version of the Gipps–model, where the cars have finite braking abilities. Different from the Gipps' model, it is still very fast and more flexible: SKCars have no problems to drive together with normal CACars, making them a good vehicle for coupling different models together.

Sink is an end that swallows cars. Any car that drives beyond it vanishes.

Source is the contrary of the SINK class. It supplies the simulation with new cars. There are several options available which determine how many cars (and when) are sent onto the streets and which way they will go. (Both SOURCEs and SINKs are model-independent.)

Counter is intended as a statistical device, which measures local variables only and therefore mimics counting devices. It has length 0, which means, that cars are only going through it. Every time a car crosses the COUNTER it can be counted, and from the collected data statistics can be derived. The statistics are collected by a STATISTICSMODULE (*cf.* 2.5). This module is constructed automatically by the Counter and then added to the list of SIMULATIONOBJECTS. The COUNTER also parses the necessary options. The *id* of the COUNTERSTATISTICSMODULE is the same one as the one of the COUNTER with the Extension ".stat".

2.2 The Instruments

INSTRUMENTS are there to block streets due to "events" like cars that must pass first or traffic lights. The INSTRUMENTS are evaluated during the `MS_prepare` microstep. Their value is then stored in the bool variable *status*. It can then be seen from other objects (especially connections) by the method *value*. There are two kinds of INSTRUMENTS: basic ones and derived ones.

The value of the basic instruments depends only on certain conditions in the simulation like the actual timestep or a car on a certain site, while for derived ones their value is determined only by the value of other instruments.

2.2.1 The Basic Instruments

Currently there are two kinds of basic instruments:

Prioritize This instrument expects an edge (and lane) as a parameter. It looks into it from the end and switches to false (block), if there is a car within a certain number of sites.

Direction is a more sophisticated version of PRIORITIZE. It works just like PRIORITIZE, but instead of blocking whenever there is a car on the specified edge it blocks only if this car wants to go to a certain edge. This instrument is necessary for the famous „Linksabbieger-müssen-warten"-rule.

TrafficLight This instrument corresponds to a traffic light. It switches to false in a certain rhythm which is specified by the options given to its constructor.

BusStop recognizes if a Bus approaches the end of the edge it governs. If so, it blocks the CONNECTION at the end of the edge for a specified number of timesteps thus simulating a bus stop.

2.2.2 The Composite Instruments

The composite instruments are the following ones:

DeadLock is a special instrument. It was added to resolve traffic situations where no car would be allowed to drive on. It is supplied with a list of parameter instruments. If they all return false, then one of them is set to true, thus avoiding deadlock in the network.

Compound is a logical Instrument. It allows to use the other instruments as input to build more complicated Right of Way rules.

2.3 The Connections

CONNECTIONS turn the single edges into a network. Everytime a car wants to look or drive beyond its current edge and the respective Connection–Pointer of the edge is not null, it calls a connection. The connection then calls the edge it is connected to. In the case that there is a instrument associated with the connection to that edge, it respects the status of it. If the connection leads to more than one edge, it looks whether the car has specified a certain destination. If not it sets a default destination and also sets this destination in the car.

2.4 The Crossovers

CROSSOVERS have been supplied to simulate exits and ramps. They can be put "between" two edges, called *origin* and *destination* respectively. They then take cars from the *origin* and put them onto the *destination*. The user can specify several parameters: Whether they chose the cars randomly or according to a certain ROUTEINFO, and the exact location of the active region.

2.5 The StatisticsModules

In order to collect data from the simulation there is a special class hierarchy: the STATISTICSMODULES. They are usually called during the `MS-statistic` microStep. Then they collect data from the edges they are affiliated with, compute mean values and write them to the general output file of the simulation after a certain number of timeSteps.

There are currently three kinds of STATISTICSMODULES

CAEdgeStatisticsModule is a dummy class that has some basic features necessary for statistics modules.

StatEdgeModule writes the following line to the output file:

`Id timestep` $\overline{\rho}$ $\overline{v}$ q `[` $\overline{\rho}_i$ $\overline{v}_i$ q_i `%`$_i$ σ_{v_i} σ_{α_i} `]` [1]

Id is the id of the module.

timestep is the current timestep. The mean values are computed over the interval set by the options.

$\overline{\rho}$ is the mean density, first on the whole edge, then for each lane.

$\overline{v}$ is then mean velocity.

q is the mean flow.

%$_i$ is the percentage of cars using this lane.

σ_v is the standard deviation of the velocity.

σ_α is the standard deviation of the distances.

CounterStatisticsModule is handled in a special way, since it must always be connected to a COUNTER and vice versa. The COUNTERSTATISTICSMODULE is not mentioned explicitly in the datafile, but constructed by COUNTER automatically and then added to the list of simulation objects. This is why the corresponding options are set in the declaration of the Counter. COUNTER produces two output files: count.avg and count.dst. The count.dst contains the velocity distribution for any lane, measured during the time interval specified by the I option. The output-format of the count.dst-file is as follows:

`Id timestep [` v $p(v)$ `]`

where v is the velocity and $p(v)$ the associated probability. The are repeated for any value of v (currently COUNTER has $10 * vmax$ velocity-classes). The data written into count.avg contain the following information:

`Id timestep [` $\overline{v}_i$ q_i `%`$_i$ $q_i \cdot \overline{1/v}_i$ v_{mean} `]` $\overline{v}_i$ q_{sum}

Id is the id of the module.

timestep is the current timestep. The mean values are computed over the interval set by the options.

$\overline{v}_i$ is the mean velocity on lane i.

q_i is the flow on lane i.

%$_i$ is the percentage of cars using this lane.

$q_i \cdot \overline{1/v}$ is the product of flow q_i with the mean of $(1/v)_i$, and therefore an estimate for the density.

v_{mean} is the average velocitiy calculated from the distribution $p(v)$.

Finally, the total average velocities and flows (averaged over all lanes) are given.

[1] The braces do not appear in the output file, but the text within them appears as many times as there are lanes on the edge.

2.6 The RouteInfo System

To simulate traffic realistically, cars should start from a certain origin (a SOURCE) and go to a prespecified destination (a SINK). In order to do this there are special SIMULATIONOBJECTs called ROUTEINFOs. There should only be one ROUTEINFOobject specified in a given input file. The program has a pointer to one ROUTEINFOinstance which is used for all route choices etc.

Currently, there are two different kinds of ROUTEINFOs: QROUTEINFOs and MROUTEINFOs

2.6.1 General Ideas

The ROUTEINFO holds a trimmed down representation of the network in the form of a graph. The words node and edge refer to elements of this graph in this section. Nodes correspond to crossings and SOURCES, edges to roads and streets.

SOURCES that support the routeplans have a pointer to a ROUTEINFO. They use a method called *ReleaseRoutePlan* to ask the ROUTEINFO, whether it should issue a car. If so, a special object called ROUTEPLAN is created by the ROUTEINFO. This ROUTEPLAN has a method called *nextNode* that can tell the car the next node on its way to its destination.

Anytime the car reaches a point where it can decide between several possible routes, this method is called. This happens at two points: On CROSSOVERs and CONNECTIONs with more than one following edges. These objects have to be informed by the Data File, which nodes they lead to.

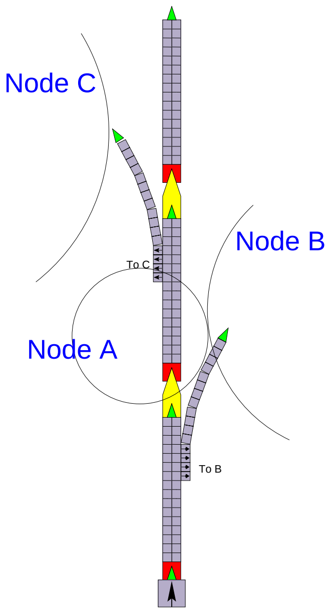

Figure 2.2: The graph of the routeplan system

Whenever a car enters the active region of a CROSSOVER, the CROSSOVER will find out whether the car has a routeplan. Then it looks up the next node on the car's shortest path from the current node to its destination node in the routeplan's shortest path matrix. According to this information, it puts the car on the new track if that is necessary.

To be able to collect statistics from the net, the EDGES must know (from the Data file) which edge of the graph they represent.

This system makes it possible to save the edge weights during the simulation, which is needed by the shortest path algorithm.

2.6.2 MRouteInfo

The ROUTEINFO has several tasks: First of all it holds two matrices, the OD-matrix and the shortest path matrix of the "supergraph".

The ROUTEINFO decides according to the OD-matrix whether it wants a given source to issue a car and where it should head to.

The graph, the ROUTEINFO stores, is a weighted graph, where the weights represent travel times on these edges. It computes new shortest paths (with Dijkstra's algorithm) in certain intervals, so changing road usage is considered in the paths for the cars.

Cars that reach a new node, tell the ROUTEINFO how long it took them to get from the last node to their current one. In this way the ROUTEINFO can keep a record of the current edge weights.

2.6.3 QRouteInfo

For this ROUTEINFO object, all cars, their starting times and their routes have to be supplied in an extra file with the extension `.rout`.

For each car its departure time (four bytes), estimated arrival time (four bytes), four empty bytes (for the actual arrival time) and a list of all the nodes a car visits on its route (except for those following a node with two adjacent nodes) (two bytes each) must be supplied in this file.

Chapter 3

The Data File

3.1 General Description

The data file consists simply of a list of objects. Each line corresponds to one object. Empty lines, lines that begin with a blank or `%` are ignored. If the description of an object is continued on the next line, you must end the line with a `/` (slash).

Each line must have the following syntax: First is the *id* of the edge (an arbitrary string), followed by the kind of object to be constructed.

Currently available are the following sorts of objects:

- CAEdge, HREdge, ContEgde, SKEdge, Sink, Counter,
- Source, CarDispenser,
- CACarDescriptor, ContCarDescriptor, SKCarDescriptor
- Connection, Crossover,
- StatEdgeModule, Counter,
- MRouteInfo, QRouteInfo.
- Crossings, and Lights.

The rest of the line is handed over to the constructor of the specified edge, which processes the rest of the string. The available options for the different objects are described in the next sections.

The program builds the objects in the order given by the data file. Therefore it will stop if one of the ids specified in an options string references an object which is declared later in the file.

The options have the following notation:

N *nLanes* [1]

which indicates that the value *N* is neccessary and stands for the numer of lanes, and that the default value, if this option is not set, is 1. All options, for which no default is given, have to be specified. The processor is case sensitive.

which indicates that the *N* is neccessary and that the default value, if this option is not set, is 1. All options, for which no default is given, have to be specified.

3.2 Edges

The following options are recognized by EDGE (and therefore by any class derived from it):

Edge *id* [] sets the (routeplan) id. On arrival on a new edge a car checks whether the edge's id is new. In that case it tells its ROUTEINFO how long it drove on the old edge. Note that this parameter refers to the edges of the routeplan graph.

Prerequisite: A routeplan has to be specified.

N *nLanes* [1] sets the number of lanes of the edge.

L *length* [0] sets the length of the edge in sites.

Polygon and

K *XStart YStart XEnde YEnde* are the coordinates of the edges. It is possible, to give more than one pair of coordinates, telling the graphics to connect them by a sequence of lines. (If these options are missing, there is no graphics output for this edge).

Shift *off* [0] is another graphics parameter.

Color *color* [-1] gives the color of the edge.

Speed *SpeedLimit* [systemwide maximum speed] allows cars to accelerate only if their velocity is less than the value set by *SpeedLimit*.

3.2.1 CAEdge

For all Edges there are the following additional options:

Dichte *density* [0.0] sets the initial car density on the edge.

Descriptor *CarDescriptorObject* determines that the Edge's Cars behave according to the CarDescriptor *CarDescriptorObject*.

Dispenser ?? determines the ratio with which the different car classes are distributed initially.

Only the CarDescriptor changes with the Edge, e.g. *SKEgde*s have *SKCarDescriptor*s etc.

3.2.2 Counter

The COUNTER class takes the following commands:

I *Intervall* [180] is the number of time steps for the averages of the COUNTERSTATISTICSMODULES.

3.2.3 Source

The SOURCE class takes the following options:

N *nLanes* [1] ? determines the source's number of lanes.

Route *routeplanId* sets the routeplan and

Node *id* [] sets the node of the routeplan graph the source belongs to.

Fourier expects a list of a_i b_i pairs. This option generates a flow $q(t)$ of the form

$$q(t) = q_0 + \sum_{k=0}^{N_{\text{Fourier}}} a_k \cos(k\omega t) + b_k \sin(k\omega t). \tag{3.1}$$

ω and q_0 are set with separate options (s.b.).

Rho q_0 [0] sets a constant flow (s. Fourier option).

Omega ω $[2 * M_{PI}/86400]$ sets the parameter ω in the Fourier equation above.

Tau τ [86400] sets ω to $2 * M_{PI}/\tau$.

Deterministic *distance* [0] puts a new car onto the street, whenever the last one is at least *distance* sites away.

Dispenser *CarDispenserObject* determines ratio with which different cars are emitted from SOURCE.

3.2.4 Sink

understands one parameter

I *Interval* [120] specifies the interval between statistic outputs

3.3 Connection

The Connections constructor expects directions as to which edges to connect.

Orig *fromId* [] tells the CONNECTION where its cars are coming from.

L *lane* [0] sets the lane on *fromId* that is to be connected.

Dest *toId lane Instrument Node* [] means the following: There is one connection to lane *lane* on CAEDGE*toId*. This connection is governed by INSTRUMENT *Instrument* and leads to Node *Node*. If you want to specify a different connection lane, just add the corresponding number.

Route *routeplanId* sets the Routeplan.

Node *id* [] tells the connection which Node it is assinged to.

$\star$ *lane1* $\star$ *dummy* or *lane1 InstrumentId Nodenumber* [] At the end of the string, this determines, which lane on this edge is connected to which lane at the other edge, and whether an instrument is associated with this connection. When using without instrument, one *must* use the syntax `Dest E2 1 * 0`, otherwise a core dump follows. This almost surely changes in later versions of PLANSIM–T.

As a default the connection assumes, that the rightmost lanes are connected.
Example:

```
C0a Connection    Orig SBahn2R   L 1   Dest Bahn2a  1 * 0
```

3.4 Crossover

understands the following options:

Route *routeplanId* sets the Routeplan.

Node *Id* [] is only necessary when routeplans are used. Tells the CROSSOVER which “super”node it is mapped to. (see also 2.6)

Out *probability* [0] gives the probability for cars on the origin edge to be stolen by the CROSSOVER.

Orig *Id* [] is the id of the edge the cars are to be taken from.

OOffset *sites* [0] is the number of sites from the beginning where the cars are to be taken from on the origin edge.

Dest *Id* is the id of the edge the cars are to be inserted on.

DOffset *sites* [0] is the number of sites from the beginning where the cars are to be inserted on the destination edge.

L *sites* [10] sets the length (in sites) of the active region of the Crossover.

Left Cars should be inserted from the left on the destination. (default)

Right Cars should be inserted from the right on the destination.

DNode *Id* [] is only necessary when routeplans are used. Tells the CROSSOVER which “super”nodes the destination EDGE is adjacent to. (see also 2.6)

Ego *probability* [1] sets the probability for cars to change EDGES even when they impede other cars.

Example:

```
Spurw Crossover   Orig Wechsel   Dest Bahn2b   L 20   Out 1   Right   Ego 0
```

3.5 Cars

3.5.1 CarDispenser

Determinies the probability with which different cars are emitted from a source or put on an edge initially. It accepts the values n, p_{type}

n is the number of car types

$\mathbf{p}_{type}$ specifies the probability for a car of the given type to emerge from the source.

The various CARDESCRIPTORS describe the parameters for the different car-classes distributed by the CARDISPENSER. Consequently, there is a different descriptor for each of the different edge-types.

3.5.2 CACarDescriptor

understands several parameters:
n, [v_{max}, p_{brake}, v_{offset}, p_{l2r}, $p_{r2l_{ego}}$]
Again, the terms in the brackets have to repeated as often as there are different car classes, as determined by n.

n is the number of car types

$\mathbf{v}_{max}$ is the maximum velocity of this type of car.

$\mathbf{p}_{brake}$ is the probability of an sudden decrease in speed of 1.

$\mathbf{v}_{offset}$ is the amount of space that has to be free on the destination lane to allow a car changing lanes.

$\mathbf{p}_{l2r}$ determines the probability of an change to the right, even when still following another car.

$\mathbf{p}_{r2l/ego}$ determines the egoistic factor, i.e. the probability of a car to change to the left even though forcing someone else to brake.

3.5.3 ContCarDescriptor

understands a somewhat different set of parameters:
n, [v_{max}, b_{max}, σ, v_{offset}, p_{l2r}, $p_{r2l_{ego}}$]
Again, the terms in the brackets have to repeated as often as there are different car classes, as determined by n.

n is the number of car types

$\mathbf{v}_{max}$ is the maximum velocity of this type of car.

$\mathbf{b}_{max}$ is the maximum acceleration of this type of car.

σ is the acceleration noise of this type of car.

$\mathbf{v}_{offset}$ is the amount of space that has to be free on the destination lane to allow a car changing lanes.

$\mathbf{p}_{l2r}$ determines the probability of an change to the right, even when still following another car.

$\mathbf{p}_{r2l/ego}$ determines the egoistic factor, i.e. the probability of a car to change to the left even though forcing someone else to brake.

3.5.4 SKCarDescriptor

The SKCARDESCRIPTOR understands:
n, [v_{max}, b_{max}, d_{max}, σ, v_{offset}, p_{l2r}, length]

n is the number of car types

$\mathbf{v}_{max}$ is the maximum velocity of this type of car.

$\mathbf{b}_{max}$ is the car's highest acceleration

$\mathbf{d}_{max}$ is the car's highest deceleration

σ sigma is the acceleration "noise" generated

v_{offset} is the amount of space that has to be free on the destination lane to allow a car changing lanes.

p_{l2r} determines the probability of an change to the right, even when still following another car,

length gives the effective length of that car, in CA-units, i.e. compared to $\lambda = 7.5m$. length=1.0 means that this car occupies 7.5m road when standing in a jam.

3.6 Instruments

3.6.1 Prioritize

The first parameter is the edge, the instrument has to look into. It may not be missing. Then there are three more options, which may follow:

L *lane* [0] is the lane number, the instrument should work on.

T *threshold* [systemwide maximum speed] is the number of sites Prioritize should check.

3.6.2 Direction

The first parameter is the edge, the instrument has to look into. It may not be missing. Then there are three more options, which may follow:

L *lane* [0] is the lane number, the instrument should work on.

T *threshold* [systemwide maximum speed] is the number of sites Prioritize should check.

D *destination* [] allows to specify a special direction. If a car is detected by the instrument, but its *ExitFlag* (a variable determining the next edge the car wants to go to) is equal to the *destination*, it will be ignored, so the INSTRUMENT is not blocked.

3.6.3 TrafficLight

expects a list of the length the different phases of the traffic light should have.

3.6.4 Probability

expects the probability, with which it should switch to green.

3.6.5 BusStop

Location *edge* (required parameter) is the *id* of the edge, where the BusStop is to be placed.

Halt *timesteps* [120] is the number of timesteps a bus should halt at the busstop.

3.6.6 Compound

This one allows to build a logical formula with variables, that are other instruments.

3.7 StatisticsModules

The first parameter is the *id* of the object, the STATISTICSMODULE has to evaluate. The options of COUNTERSTATISTICSMODULES are handled by the corresponding COUNTER.

3.7.1 StatEdgeModule

take two parameters:

I *steps* [60] is the number of timesteps over which the mean values are computed.

O [0] is the offset of the watched section on the specified lane.

L *sites* [edgelength] is the number of sites (from the beginning of the edge) on which the module works.

n *nstat* [∞] determines how many cars STATEDGEMODULE should sample in each time-step, therefore generating statistics with equal number of data-points.

3.8 RouteInfo

understands three parameters:

B *filename* [] must be stated and gives the base name of the OD-, routeplan, and graph files.

3.8.1 MRouteInfo

understands three parameters:

Beta *prob* [50] determines the prefactor for computing the probability that a car doesn't chose the best route, but a longer one. This probability is determined according to `exp(-Beta*(D-d)/d)`.This keeps the ROUTEINFO updated on the current travel times of all ROUTEINFO edges.

I *weight* [.5] states how strongly the latest edge weight is weight against the old ones.

F *timesteps* [100] is the number of time steps after which the updated edge weight list is written to the file.

3.8.2 QRouteInfo

has no parameters.

Appendix A

Example data files

The simplest possible network can be built in the form of a ring (shown here for twolane traffic only):

```
%
% twolane traffic simulation
%

% defining the CarDispenser
TheDisp            CarDispenser  Cars 2 0.8, 0.2

% syntax SKCars: nrclass {vmax amax dmax sigma voffset pl2r length},...
TheCars          SKCarDescriptor 2 6.0 0.2 0.5 0.3, 4.0 0.1 0.2 0.1

% the one and only edge
E1 SKEdge Dispenser TheDisp Descriptor TheCars N 2 L 10000 Dichte 0.1  K 0 0 100
 100

% two connections are needed for the two lanes
CC2a Connection Orig E1        Dest E1 0 * 0
CC2b Connection Orig E1 L 1    Dest E1 1 * 0
```

In this case, no output but the graphical output on the GUI is produced. A slightly more advanced example can be found in the following data-files, for CAEDGE and SKEDGE, simulating a highway with a ramp, where the inflow both in the ramp and the highway are periodical functions.

A.1 CAEdge

```
% Simulation of a ramp

DispL           CarDispenser  Cars 2 0.95, 0.05
DispR           CarDispenser  Cars 2 0.80, 0.20

% Anzahl, vmax, pbrake, voffset, prob_l2r, prob_r2l_ego
Cars            CACarDescriptor 2 4 0.2 0 0.05 0.2, 3 0.2 0 0.1 0.2

SBahn1  Source  Dispenser DispR  N 1  Rho 0.3 Fourier -0.3 0 Tau 400
```

```
SBahn2L Source  Dispenser DispL  N 1  Rho 0.25 Fourier -0.25 0 Tau 3000
SBahn2R Source  Dispenser DispR  N 1  Rho 0.25 Fourier -0.25 0 Tau 3000

Bahn2a  CAEdge Descriptor Cars N 2  L 100  K   0   0    0  100
Bahn2b  CAEdge Descriptor Cars N 2  L  20  K   0 100    0  120
Bahn2c  CAEdge Descriptor Cars N 2  L 100  K   0 120    0  220
Bahn1   CAEdge Descriptor Cars N 1  L 140  K 100   0   10  100
Wechsel CAEdge Descriptor Cars N 1  L  20  K  10 100   10  120
Fake    CAEdge Descriptor Cars N 1  L   0  K  10 120    5  130

SiBahn2 Sink I 100000

B2Stat1 StatEdgeModule  Bahn2a O  80  L 20  I 300
B2Stat2 StatEdgeModule  Bahn2c O  50  L 20  I 300
B1Stat  StatEdgeModule  Bahn1  O 100  L 40  I 300

Spurw   Crossover  Orig Wechsel   Dest Bahn2b L 20 Out 1 Right Ego 0

C0a     Connection Orig SBahn2R L 0 Dest Bahn2a   0 * 0
C1a     Connection Orig SBahn2L L 0 Dest Bahn2a   1 * 0
C0b     Connection Orig Bahn2a  L 0 Dest Bahn2b   0 * 0
C1b     Connection Orig Bahn2a  L 1 Dest Bahn2b   1 * 0
C0c     Connection Orig Bahn2b  L 0 Dest Bahn2c   0 * 0
C1c     Connection Orig Bahn2b  L 1 Dest Bahn2c   1 * 0

C2      Connection Orig SBahn1  L 0 Dest Bahn1   0 * 0
C3      Connection Orig Bahn1   L 0 Dest Wechsel 0 * 0

C4      Connection Orig Bahn2c  L 0 Dest SiBahn2 0 * 0
C5      Connection Orig Bahn2c  L 1 Dest SiBahn2 0 * 0
```

A.2 SKEdge

```
% Simulation of a ramp
DispL           CarDispenser  Cars 2 0.95, 0.05
DispR           CarDispenser  Cars 2 0.80, 0.20

% Anzahl, vmax, bmax, dmax, sigma, voffset, prob_l2r, length
SKCars          SKCarDescriptor 2 4 0.25 0.5 0.2 0 0.05 1, 3 0.2 0.4 0.15 0 0.1 2

SBahn1  Source  Dispenser DispR  N 1  Rho 0.3 Fourier -0.3 0 Tau 400
SBahn2L Source  Dispenser DispL  N 1  Rho 0.25  Fourier -0.25 0 Tau 3000
SBahn2R Source  Dispenser DispR  N 1  Rho 0.25  Fourier -0.25 0 Tau 3000

Bahn2a  SKEdge Descr SKCars N 2  L 100  K   0   0    0  100
Bahn2b  SKEdge Descr SKCars N 2  L  20  K   0 100    0  120
Bahn2c  SKEdge Descr SKCars N 2  L 100  K   0 120    0  220
Bahn1   SKEdge Descr SKCars N 1  L 140  K 100   0   10  100
Wechsel SKEdge Descr SKCars N 1  L  20  K  10 100   10  120
```

```
Fake    SKEdge Descr SKCars N 1  L   0  K  10 120    5  130

SiBahn2 Sink I 100000

B2Stat1 SKStatEdgeModule  Bahn2a O  80  L 20  I 300
B2Stat2 SKStatEdgeModule  Bahn2c O  50  L 20  I 300
B1Stat  SKStatEdgeModule  Bahn1  O 100  L 40  I 300

Spurw   Crossover  Orig Wechsel   Dest Bahn2b  L 20 Out 1 Right Ego 1.0

C0a     Connection Orig SBahn2R L 0 Dest Bahn2a   0 * 0
C1a     Connection Orig SBahn2L L 0 Dest Bahn2a   1 * 0
C0b     Connection Orig Bahn2a  L 0 Dest Bahn2b   0 * 0
C1b     Connection Orig Bahn2a  L 1 Dest Bahn2b   1 * 0
C0c     Connection Orig Bahn2b  L 0 Dest Bahn2c   0 * 0
C1c     Connection Orig Bahn2b  L 1 Dest Bahn2c   1 * 0

C2      Connection Orig SBahn1  L 0 Dest Bahn1   0 * 0
C3      Connection Orig Bahn1   L 0 Dest Wechsel 0 * 0

C4      Connection Orig Bahn2c  L 0 Dest SiBahn2 0 * 0
C5      Connection Orig Bahn2c  L 1 Dest SiBahn2 0 * 0
```

Appendix B

CA-rules implemented

B.1 Car–following rules

We use the STCA rules of Nagel and Schreckenberg with minor modifications due to german traffic regulations, i.e. overtaking on the right side is forbidden. `gap` is the number of free sites ahead.

```
if (v >= gap)
  v = gap;
else
  if(v < vmax && v < SpeedLimit)
    v++;

// no overtaking on the right side
if ((v>3) && (v>(gapLeftAhead + vLeftAhead)))
  v = g + vahead;
if (v>0 && random () < car->pbrake)
  v--;
```

B.2 Car–following (continuous, CONTEGDE)

We skip now the passing on the right side, it is implemented as in CAEDGE. The speed–limit is implemented by first forcing `vmax` to be be the minimum between `vmax` of the car and the current `SpeedLimit`. `gap` is now the free space between front–bumper and rear–bumper of the car ahead.

```
gap = gap < vmax ? gap : vmax;
v = v+bmax;
if (v > gap)
  v = gap;
v -= sigma * bmax * frandom ();
v = v > 0 ? v : 0;
```

B.3 Car–following (continuous, with finite braking, SKEGDE)

Anything concerned with gap and SpeedLimit is as in CONTEDGE, so we state only the car–following rules here (pred->v is the speed of the car ahead, which is needed in order to calculate vSafe):

```
if (v<vmax-bmax)
  v += bmax;
else
  v = vmax;

vSafe = pred->v + dmax * (gap-pred->v)/(0.5*(pred->v+v) + dmax);

if (vSafe > vmax)
  vSafe = vmax;

if (vSafe > gap)
  vSafe = gap;

if (v>vSafe)
  v = vSafe;

v -= sigma * bmax * frandom ();

if (v<0)
  v=0;
```

B.4 Lane–change rules

CAVEAT: The lane–change rules are still under construction and may be different in future releases. We give here the code for the CAEDGE and CONTEDGE only.

In the current release, lane changes are controlled by the three parameters *voffset*, *Prob_l2r* and *Prob_r2L_ego*. For those who want to know the very details, here is an excerpt from the code:

```
ChangeToTheRight (int Length, int Lane, int gap, int vmax)
{
  int vo, g;

  if (random()<Prob_l2r)
    {
    vo = site[Lane][Length];
    if (((int) site[Lane-1][Length] == EMPTY) &&
        (g = bestimmegap(Length, Lane-1, 1,vo+1), g>=vo) &&
        (g = bestimmegap(Length, Lane-1, -1, MAXVELO, &vo), g >= vo))
      {
        site [Lane-1][Length] = site[Lane][Length];
        site [Lane][Length] = (int) EMPTY;
        return true;
      }
```

```
   }
   else
     {
     if (((int) site[Lane-1][Length] == EMPTY) &&
        (gap>vmax) &&
        (g = bestimmegap(Length, Lane-1, 1,vmax+Voffset+1),
                         g>(vmax+Voffset)) &&
        (g = bestimmegap(Length, Lane-1, -1, MAXVELO, &vo), g >= vo))
      {
         site [Lane-1][Length] = site[Lane][Length];
         site [Lane][Length] = (int) EMPTY;
         return true;
       }
     }
  return false;
}

ChangeToTheLeft (int Length, int Lane, int gap, int vmax)
{
  int vo, g;

  if (random()<Prob_r2l_ego)
    {
      if (((int) site[Lane+1][Length]==EMPTY)&&(gap<vmax))
        {
          site [Lane+1][Length] = site[Lane][Length];
          site [Lane][Length] = (int) EMPTY;
          return true;
        }
    }
  else                                  // old rules apply
    if (( (int) site[Lane+1][Length] == EMPTY) && // can we do it?
        (gap<vmax) &&                 // should we do it?
        (g = bestimmegap(Length, Lane+1, -1, MAXVELO, &vo), g>=vo) &&
        (bestimmegap(Length, Lane+1, 1, site[Lane][Length]) > gap))
      {
        site [Lane+1][Length] = site[Lane][Length];
        site [Lane][Length] = (int) EMPTY;
        return true;
      }
    else
      return false;

}
```