A First Draft on how to Integrate High Fidelity and Cellular Automata Approaches to Microsimulation in TRANSIMS on a Distributed Computer Network Version 1.0

Marcus Rickert*†

February 11, 1995

*mr@zpr.uni-koeln.de, ZPR Zentrum für Paralleles Rechnen, Universität zu Köln, Germany
†rickert@tsasa.lan.gov, TSA-DO/SA, LANL, Los Alamos NM, USA

Contents

1 Introduction

1.1 General remarks

First the reader should know that is *only* a working paper which tries to summarize all ideas that might lead to a functioning application. The author strongly encourages critical comments on any part of this paper. To designate the author's 'conviction' concerning his ideas there are question marks added to the text with the following meaning: one ? states that the solution is considered good, but doubt remains. Two ?? mean that there are several options and the one chosen is (most?) suitable for the context. And three ??? suggest that the solution chosen was a first (and probably not the best) guess. Nevertheless those parts not containing any marks should be checked thoroughly, especially if they refer to source code that has already been implemented in either the demo or the current working version of the TRANSIMS simulation.

Also this summary is inhomogeneous as far as the level of detail is concerned. Some aspects of the implementation are still very unclear so that in places only handwaving arguments are given. Hopingly this will improve in future versions of this summary as implementation proceeds.

1.2 What's new?

High fidelity traffic simulations have been created by several scientific and engineering groups all around the world over the last decades. Due to its scale TRANSIMS will run into the same problems as any of its predecessors: computational speed provided by today's conventional[1] computers is still not sufficient to cope with the extremely large number of individual objects that have to be simulated in real time for a realistic network (e.g. of a major city like Albuquerque).

In this paper two methods will be described to increase the performance of the simulation and thus to make a large scale computation feasable:

- **Parallel computers** will be used to distribute the network onto several computational nodes. The toolkit PVM will provide a programming environment that allows the porting on a large variety of modern supercomputers like the CM-5, the Paragon, or the Parsytec. Also the implementation should comprise a means to perform a dynamic load balancing in case a high performance workstation cluster (e.g. SUN Sparcs or IBM Riscs) is chosen as a platform.
- **Cellular automata** will serve as an alternative model for simulating the motion of the vehicles. In contrast to the high fidelity model based upon intelligent objects CAs can be regarded as a low fidelity solution which still captures the main characteristics of traffic flow while boosting computational speed by a factor of 10 to 100. There are limitations to the concept of CAs which will probably emerge as soon as first test runs are performed. Certain problems will deliver unsatisfactory (or even qualitively wrong) results when solved in an CA approach. One of the main objectives of a program which integrates both models will thus be
  - to compare results quantitively and qualitively, and

[1] that is *normal* single node computers

– to find parameters by which to decide whether to use slow high fidelity or the fast low fidelity approach.

1.3 Technical terms

Here are some of the technical terms used in this draft just to make sure writer and readers have the same understanding of the matter.

- **Node:** Node of the traffic network to which segments are incedent. These are mainly junctions and intersections.
- **Segment:** Part of the traffic network representing a street, road or highway segment between two nodes.
- **Network:** Entity of all nodes and segments usually provided as ASCII input file.
- **Computational Node (CPN)** One unit in a parallel computer system.
- **Topology:** The way CPNs are arranged and accessed in a parallel computer system.
- **Tile** Part of the geometric area covered by the traffic network that will be assigned to a CPN.
- **Boundary** The interface part between two neighbouring CPNs usually containing data about objects close to the edges of the CPNs.

2 Objects

Objects will be the main means of storing and handling data in the simulation. As far as the high fidelity (HF) model is concerned object oriented source code has been implemented in the current version of TRANSIMS. As for CAs, however, the idea of high level contructs and dynamic memory allocation which usually goes along with objects contradicts the guiding line to keep structures easy and linear and thus fast. So there have to be some compromises between these two (normally) opposing approaches: In the hierarchical structure of the program with simulation control at the top and the individual vehicle at the bottom the border between strict object oriented approach should be set as low as neccessary to allow for the integration of the HF model but as high as possible to achieve good results.

2.1 Abstract data structures

In the implementation, five abstract data structures will be used to handle sets of objects 'in an orderly fashion'. They will either be taken from a library (if available) or be written especially fitted for this simulation. All should be provided as templates to allow for maximum programming comfort and to enhance readability of the source code.

Non intrusive singly linked list

This is just a regular singly linked list that handles type cast (through templates) pointers of objects.

Non intrusive doubly linked list

Equivalent to the singly linked list.

Binary tree

The binary tree will be used to organize message passing between the CPNs[2] especially for *broadcasting* and gathering statistical data by *reducing*.

Heap

The heap will be used to find the maximum or minimum of a characteristic of a dynamically changing set of objects. An example is the scheduler which sorts the received events by the time stamp of each scheduled event. The event with the minimum in time will be executed first.

AB-tree

An AB-tree is defined as a tree in which all non leaves have at least A sons and at most B sons. It can be shown that AB-trees have always a depth complexity of $O(\log n)$ and therefore all dynamic insertions and deletions can be done in $O(\log n)$ time complexity. Usually an infix order is applied to the AB-tree to allow for quick managing of sorted sets.

In this simulation the ID-tree which links IDs with pointers will be organized in an AB-tree.

2.2 Object inheritence tree

See figure 1. Arrows point from parent to child.

2.3 Object dependency tree

See figure 2. Arrows represent relationship *uses a*. Labels like $1 : n$ denote the number of objects used by another object. n may vary in each case.

2.4 Base objects

Three base objects are declared which all other objects will be derived from. They are abstract objects; that is, there will not be any instances of the base objects but only instances of derived objects although none of the object methods will be formally declared abstract to allow for incomplete virtual redefinition.

Object `TID`

The base object `TID` defines the ID by which each object will be identified during the simulation. A new ID will be assigned upon creation of an object which will be kept until the final destruction of the object. An ID once assigned will never be reused. In case an object is moved from one CPN (source) to another CPN (destination) it will be destroyed on its source CPN and recreated on its destination CPN with the same ID it had before.

[2] computational nodes

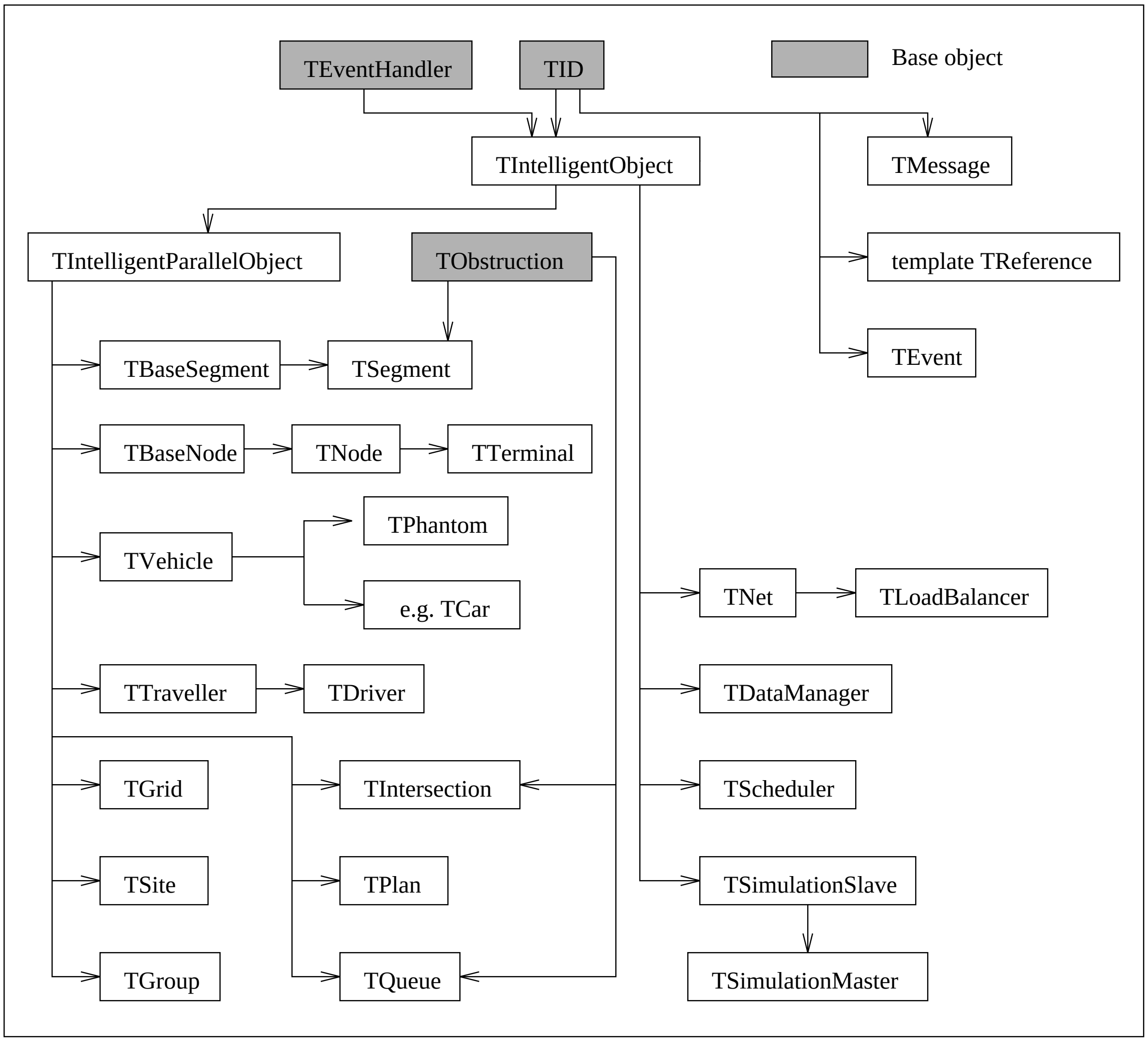

Figure 1: *Object inheritence tree*

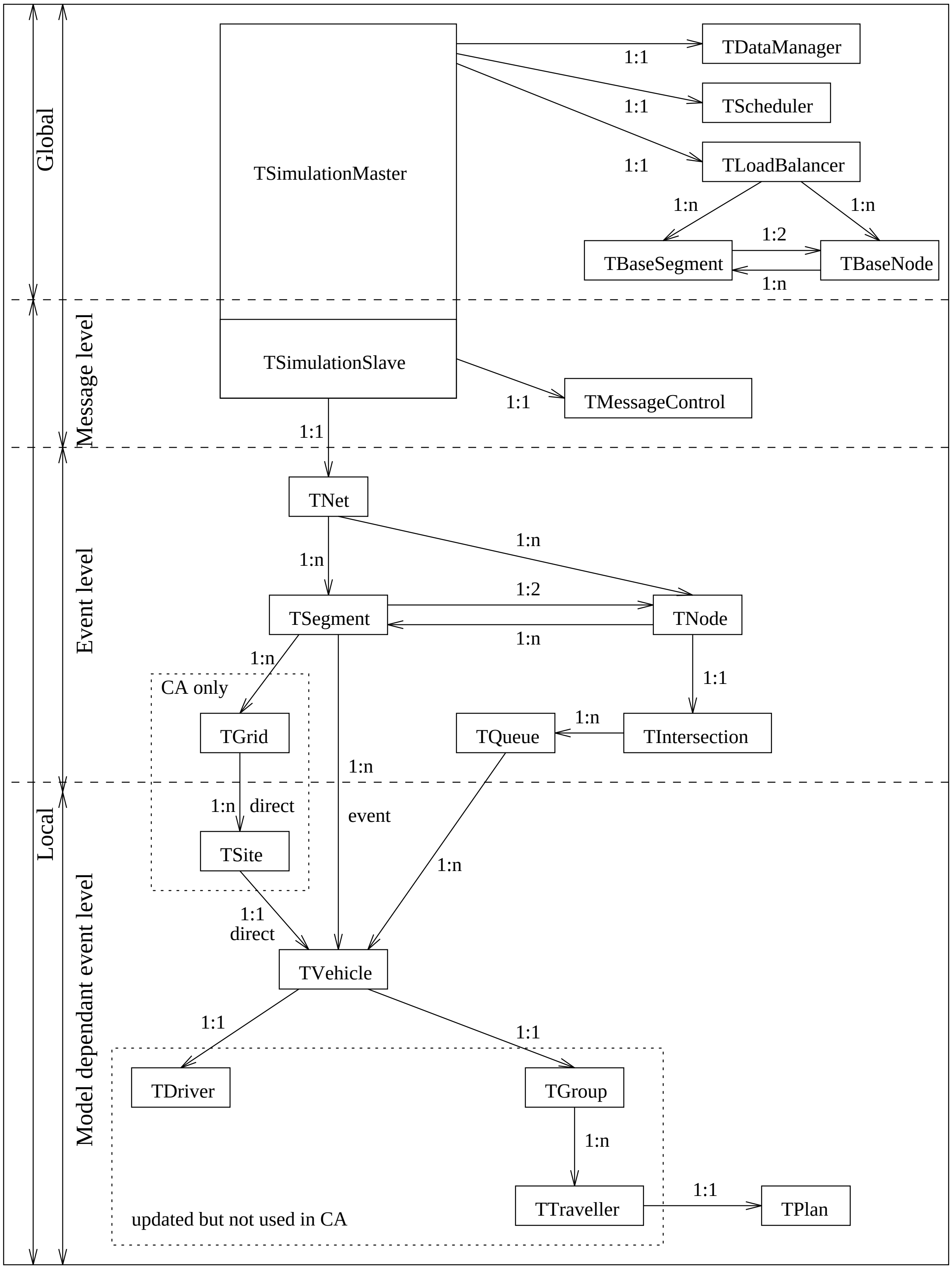

Figure 2: *Object dependency tree*

That way all references to its ID will remain valid although its physical location in the cluster may change.

In addition to the ID assigned by the system the object `TID` will also contain the following pieces of information:

- the original object number of the input file for future references, and
- a type ID that identifies the object as one of the object types used in the simulation.

Base object `TEventHandler`

The base object `TEventHandler` will be the parent of all objects required to react to events which are passed down to them in the event control structure.

Base object `TObstruction`

The base object `TObstruction` will be used as parent of all classes that can be regarded as temporary or permanent obstacle for traffic flow (??). For example, queues for the direction 'straight ahead' at an intersection which can be empty or occupied represent an obstacle for vehicles in the left turn queue of the opposite direction.

2.5 Context objects

Context objects are used to pass a huge sets of parameters to a function or an object method. Their main purpose is to avoid modifications in the declaration of object methods in case the number and/or type of parameters has changed. As an exception to the rule they are not derived from base objects and do not have any children either. Normally they do not have methods although it might be appropriate in some occasions to supply some for extended data conversion and consistency checking.

Context object `TObstructionContext`

The object `TObstructionContext` contains all data neccessary for an object of type `TObstruction` to decide whether it represents an obstruction or not.

Context object `TStateContext`

The context object `TStateContext` comprises all data neccessary for a driver to decide its next action when triggered to so by an appropriate event. It is generated at the highest hierarchy level that handles events and is gradually filled in all intermediate levels until it reaches its final destination object `TDriver`.

2.6 Intermediate objects

Intermediate objects combine characteristics of base objects but no instances will ever be created from them. They simply serve as parents for all simulation objects and control objects.

Intermediate object `TIntelligentObject`

The object `TIntelligentObject` combines the characteristics of base objects `TID` and `TEventHandler`. It is called intelligent because it can react to events representing e.g. real life incidents. In this object the two main virtual methods `DecideNextAction` and `DoNextAction` are located. They will be replaced by appropriate methods of their children if neccessary. Also the method `Parse` is rooted here which creates an instance of the object from an ASCII input file.

Intermediate object `TIntelligentParallelObject`

The object `TIntelligentParallelObject` is based upon `TIntelligentObject`. The main methods are `AddToMessage` which prompts the object to add itself to an object of type `TMessage` and `ExtractFromMessage` which will extract all data neccessary from the message and initialize a new instance of the object. They will be replaced by methods of their children.

2.7 Simulation objects

Simulation objects are central to the design approach described here. They represent those entities of the simulation that can be transferred to another CPN in the parallel computer topology. Most of them take an active part in the decision *DecideNextAction* either by actually making it (`TDriver`) or by contributing part of their current state which the decision will be based upon (e.g. `TVehicle`, `TSegment`).

Simulation object `TBaseSegment`

This object will not be described in detail. It should be regarded as a simple parent of `TSegment` which only contains data neccessary to do the load balancing so that memory requirements can be kept as low as possible on the master CPN.

Simulation object `TSegment`

`TSegment` contains all information about one street segment from one junction/intersection to another. Parameters such as number of lanes or speed limit need to be unique for the whole segment so that realistic segments with multiple values for the same parameter may have to be split.

In the simulation hierarchy the segment will be the topmost level that is shared by CA and HF approaches. At lower levels in the hierarchy it the two approaches will differ:

- In the current **HF** version all vehicles are sorted according to their positions on the segment before each `DecideNextAction` event. In future versions this sorting should be avoided by handling vehicles dynamically in a sorted container class. One approach is to supply each segment with an AB-tree (???) with all vehicles that currently reside on the segment. Events arriving at the segment will be passed to each of the associated vehicles. Statistics are collected from each vehicle and then evaluated at the segment until aggregated results are passed to the upper layers.

  Besides the AB-tree (called *AB*) there are two other structures to handle the vehicles on a segment: the first one simply replaces the AB-tree with a doubly linked list (called

DLL) and does not require any further explanation. The second one (called *CA*) uses the object `TGrid` of the CA approach to keep track of the relative positions of the vehicles. Every vehicle would have a grid position (which is simply an integer number identifying the site that the vehicles is closest to) and a relative continuous offset (or deplacement) within that site. In the following table advantages and disadvantages are summarized. e denotes the number of vehicles currently residing on the segment, l the length of the segment in arbitrary units, and v the vision range in arbitrary units.

aspect	AB	DLL	CA
memory requirements	$O(e)$	$O(e)$	$O(l)$
implementation	easy	easy	tricky
move vehicle past n successors	$O(\log e)$	$O(n)$	$O(1)$
insert vehicle at random position	$O(\log e)$	$O(e)$	$O(1)$
check predecessors and successors	$O(v)$	$O(v)$	$O(v)$
independence of CA	yes	yes	no
modifications of original CA data structures	no	no	yes
implicit transformation CA to HF	no	no	yes

Table 1: *Comparison of vehicle handling methods*

- In the **CA** version each segment will have an array of grids (type `TGrid`), one grid for each lane and direction. All events are passed to grids. All statistics are gathered from the grids.

Boundary segments connecting two CPNs will exist on both CPNs with identical IDs. Depending on whether the first or the second node is local, the object will handle the vehicles on the first or second half of the segment and take of the boundaries. It may be convenient to define a child object of `TSegment` called `TBoundarySegment` (??).

Simulation object `TGrid` (CA only)

A `TGrid` represents one lane (for one direction) on a segment. It consists of an array of `TSites`. In the CA model this level is the lowest that still evaluates events. At lower levels of the hierarchy only *direct* evaluation of rules, execution of commands, and gathering of statistics take place in order to maintain the high performance.

Simulation object `TSite` (CA only)

A `TSite` represents a box of approx. 7.5 meters length on a lane that can be either occupied by a vehicle (in which case in contains a pointer to the vehicle of type `TVehicle`) or empty (in which case the pointer contains a `NULL`).

Simulation object `TVehicle`

The object `TVehicle` represents the main object of the simulation that is updated and moved through the network as the simulation proceeds. It has local methods defining its current state and driving characteristics which are passed in `TStateContext` to its associated driver for the decision `DecideNextAction`. The methods will be overwritten by its children representing a certain vehicle class such as `TCar` or `TTruck`.

Each vehicle contains a group (`TGroup`) of travellers (`TTraveller`) that currently reside in the vehicle. Of those one is designated to be a driver (`TDriver`).

subsubsection*Simulation object `TTraveller` The object `TTraveller` represents a person having a travel plan. This person need not neccesarily be able to drive so that the method `DecideNextAction` is not overwritten. Every traveller will be *assigned* to a vehicle, as long as he is transported in that vehicle. He might be assigned to a queue or pool of vehicles for the time he is independent from any means of transportation.

Simulation object `TDriver`

The object `TDriver` is the only object which justifies the name intelligent object since this is the one who actually makes the decision in the method `DecideNextAction`[3]. All other intelligent objects simply contribute their state to make the decision possible. In the first version of the simulation `TDriver` will only be tought how to drive a passenger car but in future versions this should be extended to allow for a skill (or behaviour) pattern for a set of vehicles.

As for the CA model the object `TDriver` will be moved along with vehicle it belongs as well as groups and vehicle type but it will never be accessed directly during motion since the CA rules do not require it.

Simulation object `TGroup`

The object `TGroup` administers a set of travellers that reside in the same vehicle. It will supply the common partial route plan; that is, shared by all travellers in the group. Also it will organize adding and removing travellers from a vehicle.

Simulation object `TRoute`

The object `TRoute` will manage a set of pairs of the form (segment, node) that define the travellers route plan in the network. Travellers on the same vehicle have at least one subrange of their plans in common.

It may be suitable in future implementations to enhance the (..., node) part of a travel route step by (..., queue, delay) to model travellers that are detached from the vehicle network because they are spending their time at an office, school, shopping center or the like.

Simulation object `TBaseNode`

Like `TBaseSegment` the object `TBaseNode` is a simple parent of `TNode` restricted for the use with `TLoadBalancer`.

Simulation object `TNode`

The object `TNode` represents locations where the end of at least one segment resides. There will be two major types of nodes: most of them will represent logical elements found in

[3] of course, this applies only the simulation, since we know that in reality the fact of being permitted to drive has nothing to with intelligence

real traffic like terminals[4], ramps (2 outgoing segments), junctions (3 outgoing segments), and intersections (4 outgoing segments). Some of them will only have the auxiliary task of providing a means to split a realistic segment into two or more smaller ones whenever the characterictics of that segment change.

Simulation object `TTerminal`

The object `TTerminal` is a child of `TNode` that has special methods to handle the insertion of vehicles into the network and the absorption of vehicles from the network at time dependant rates given in vehicles per time unit (?).

Simulation object `TQueue`

The object `TQueue` will hold vehicles that are in progress of passing through an intersection. Each queue corresponds to a pair of (outgoing lane, incoming direction). In contrast to the high fidelity representation of vehicles on the segments (in the HF model) the intersections will be modelled in a medium resolution approach in which they have no physical extent and vehicles passing through them have no influence on each other except blocking.

Since queues only hold pointers to the vehicles and these vehicles are not updated they are independent of the model (CA or HF). Therefore they represent a way for integrating the two models by using them as an interface. The only problem may be that vehicles should still have a velocity while going through intersections, especially for the lanes going straight. Otherwise there would be a clear discontinuity in the HF model.

Simulation object `TIntersection`

The object `TIntersection` has mainly three functions:

- It will serve as a event dispatcher for a two dimensional matrix containing pointer to queues, one entry for each queue in the intersection.
- For each queue there will be a list of objects of parent type `TObstruction` which determine whether a queue can be *emptied*; that is, whether a vehicle can leave the queue (??).
- At signalized intersections there will be a time dependant matrix (probably bit coded, ??) that determines when vehicles may *enter* the queue.

2.8 Control objects

In contrast to simulation objects control objects either remain on the same CPN once they have been created or if they move they serve only as temporary data storage (`TMessage`). The main tasks handled by control objects are:

- **Data retrieval** from storage devices like hard disks,
- **Data processing** from ASCII format to internal object format,

[4] *terminals* (1 outgoing segment) are used in this context for locations where vehicles enter or leave the edge of the system, such as the city border or state border

- **Organizing network topology** by creating a graph from input data and distributing it onto the CPNs,
- **Message passing** from one CPN to another or to a set of other CPNs,
- **Event handling** for communication within a CPN,
- **Gathering of statistics** by aggregating single vehicle characteristics.

Control object `TReference`

The object `TReference` will be used to access another object in the simulation. It will be supplied as template to force type checking.

Control object `TSimulationSlave`

`TSimulationSlave` is the topmost object in the simulation hierarchy. There is one instance on each CPN that has pointers to one instance each of `TNet` and `TMessageControl`. After startup it will take care of initialization of local (CPN related) data structures. Then it will enter the main loop which checks `TMessageControl` whether new messages have arrived and if so will pass them to `TNet` for distribution. It will only leave this loop at the end of the simulation.

Control object `TSimulationMaster`

The object `TSimulationMaster` is a child of `TSimulationSlave`. There is only CPN in the parallel topology that has a `TSimulationMaster` instead of a `TSimulationSlave`. In addition to the latter it has pointers to one instance each of the objects `TScheduler`, `TDataManager`, and `TLoadBalancer`.

After startup it will use `TDataManager` to read data from the ASCII input files and setup the network with `TLoadBalancer`. Later it will start to distribute the network via messages to each CPN (including itself). During the simulation it will switch between slave mode in which it behaves like any other CPN and master mode in which it checks the scheduler which events should be sent to the network.

Control object `TNet`

The object `TNet` handles the lists of those segments and nodes that reside a CPN. Its main task is to distribute events it receives from `TSimulationSlave` and to gather statistical data. Moreover it should be able to provide information about what percentage of time the CPN is idle. This will be sent to `TSimulationMaster` to organize load balancing.

Control object `TLoadBalancer`

The `TLoadBalancer` object keeps track of segments and nodes as the `TNet` object does, but it will do it for the *whole* network represented by `TBaseSegment` and `TBaseNode` objects. During the run of the simulation it will always be informed about which subsets of segments and nodes reside on which CPN. According to the *idle time* information it receives from each CPN it will try to restructure the topology by sending out messages to move network elements from one CPN to another.

Control object `TScheduler`

The `TScheduler` object will provide the master clock of the simulation. In the first version it will (as in the demo version) emulate a time driven simulation by regularly sending out `DecideNextAction` and `DoNextAction` events which will be dispatched by `TSimulationMaster` to all CPNs. Each of the events will have a time stamp on it that can be used by intelligent objects dependant on behavioural patterns repeated in regular intervals such as `TIntersection`.

Control object `TMessageControl`

In its initialization phase `TMessageControl` will setup all data structures neccessary for inter CPN communication. It will serve as an interface to the PVM library and the PVM group library. During the simulation it will receive and store (if neccessary) all messages from other CPN. It will also be used to send out messages.

Control object `TDataManager`

The `TDataManagerObject` will read formatted ASCII data and transform it into a form that will be used by `TLoadBalancer` to build a whole network. For this object most of the source code of the demo version or later versions should be used.

Control object `TEvent`

A `TEvent` is a temporary object used to transmit commands and data within a CPN or if the event is encoded into a message even to other CPNs. Each event has an ID, a sender, an addressee, and a time stamp. Children derived from `TEvent` will add local variables according to their needs.

Control object `TMessage`

A `TMessage` is also a temporary object. It will be used to transmit events from one CPN to another. Each message has an ID, a sender CPN, and an addressee which may be `ALL` if the message is to be broadcast. In contrast to `TEvent` there will probably not be any children from `TMessage` to allow for different message types. Yet they will have different lengths due to the variety of events encoded.

3 Simulation

3.1 Cellular automata (CA) approach

3.1.1 Basic characteristics

The CA model that will be used for this implementation was developed by Kai Nagel and Michael Schreckenberg [3]. The author has extended it to allow for freeway traffic on two lanes and interactions at junctions and intersections [4]. The main idea is that vehicles are forced to move in a grid with boxes of approx. length 7.5 [m] called *sites* and that they can only have the velocities $0, 1, \ldots 5 =: v_{max}$ which are measured in boxes per time step. During validation it turned out that the timestep is roughly a second and maximum speed lies at approx. 112 [km/h].

The update of the vehicles on each lane is defined by three simple rules that represent:

- **acceleration** until maximum velocity is reached,
- **decelaration** due to obstacles (predecessors),
- additional **randomized decelaration** modeling driver behaviour.

In case of multilane streets or road three more rules are added to allow for lane changing:

- check whether the vehicle is going at the driver's desired velocity
- if no, check whether a higher velocity could be reached in the left or right lane
- if yes, check whether lane change is permitted; that is, check minimum gaps to predecessors and successors.

In this basic model the following parameters are available: The column R/V/S denotes

Symbol	Values	R/V/S	Meaning
a	1 $[site/s^2]$	R	acceleration
d	1,2,3,4,5 $[site/s^2]$	R	deceleration
v_d	1,2,3,4,5 $[site/s]$	V	desired speed
v_{max}	1,2,3,4,5 $[site/s]$	S	speed limit
p_{proh}	yes or no	S	passing prohibition

Table 2: *Parameters of the CA model*

whether the parameter is dependant on the rules, the vehicle, or the site that the vehicle is on.

3.1.2 Optional characteristics

In future versions it may be appropriate to slightly enhance the CA rules to make them more realistic. Among other things the following aspects may be taken into consideration:

- **Asymmetry** So far the CA model treats all lane equally which does not correspond to traffic regulations. In real traffic the left lane is more often used as a passing lane the the right one. The CA model should be able to simulate this behaviour by breaking the symmetry.

- Both **acceleration and decelaration** yield values that are too high compared to real life vehicle characteristics. In avarage it takes a vehicle 10 seconds to reach its maximum speed of 112 [km/h] which is probably twice as high as the realistic average. On the other hand a vehicle that is going maximum speed can come to a full stop within 5*7.5=37.5 meters which is also very unlikely. Therefore the basic rules of CA may need some tuning to garantee that vehicle behaviour is consistent with reality.

- Both **maximum speed and desired speed** are currently restricted to integer values which allows only 3 reasonable speeds: 3,4,5. Through a minor change in the decelaration rule it may be possible to set the average maximum speed or desired speed respectively to a continous value.

3.1.3 Implementation

The major objective of a CA implementation is simplicity in data structures and data access which results in superior computational performance. To achieve this one has to follow mainly four guidelines:

- Use arrays as means of storage.
- Avoid pointers.
- Update array elements linearly in a single simple loop.
- Use as few update rules as possible.

Guidelines three and four are easy to accomplish since the update itself only affects the CA part of the simulation. The first to guidelines, however, directly interact with the HF model since data structures are involved which should be common to both approaches.

In the following the data structure of the CA model will be described. As mentioned earlier the object `TSegment` is the highest object in the simulation hierarchy that is still common to both models. Each segment has as many pointers to grids as lanes are defined for that segment. Each grid (object `TGrid`) consists of array of sites each representing a site of length 7.5 [m] as defined by the CA model. Each site contains a *pointer* which can either be `NULL` or point to a vehicle of type `TVehicle`. So in contrast to the original model the sites do not contain the vehicle itself but only a reference to it. Moving a vehicle means moving a *pointer* in memory. In terms of speed this, of course, slightly slows down the computation since one level of indirection is added. The advantage is, however, that the *same instances* of vehicles can used for both approaches. Switching between the models is then reduced to updating a few local variables of the object `TVehicle`.

3.2 High fidelity (HF) approach

The features of the HF model are mainly defined by the TRANSIMS demo version plus the enhancements added inbetween by John Prior and Doug Roberts [1]. It is important to mention that the first step of this project merely consists of implementing all components neccessary for the CA part. Nevertheless all data structures should be such that the HF model can be added in a second step in *a natural way* without modifications of neither object inheritence nor object dependence.

3.3 Integration of CA and HF approaches

During simulation it should be possible to

- simulate the network partly according to the CA model and partly according to the HF model
- change the underlying model for any given segment.

The second feature is the easier one of the two because one only has to define routines that convert a configuration of vehicles given in CA parameters into a configuration in HF parameters and vice versa (with a loss of accuracy).

The first feature requires that there has to be a mechanism to transfer vehicles from a CA segment to a HF segment which is consistent with both models and yet fast enough in order not to slow down the CA model if running without the HF part.

An easy approach would be to use the queues that exist on each node as an interface. These queues will be provided with methods to delete vehicles from segments and insert vehicles into segments respectively. It has to be mentioned that as far as performance is concerned the implementation of the queues will play an important role that should not be underestimated especially in networks with a short average segment lengths and thus a high node to segment length ratio.

3.4 Queue management

Queues will be modelled as a site oriented CA that should be as similar to the CA used on the segments as possible (see 3). The number of sites will be determined by the length of the queues (left and right turns) or the extent of the intersection (through lanes). Yet all queues will have at least a length of v_{max} sites so that a vehicle traversing an intersection will have to spend *at least one timestep* on the intersection. This will simplify implementation considerably.

As far as the update is concerned the following rules apply:

- All vehicles (HF and CA) will be updated according to the CA rules although HF vehicles will maintain their velocity while going through straight ahead (provided that there is no congestion).
- For turning lanes there will be a speed limit which will be imposed the moment the vehicle enters the queue.
- The intersection provides the feeding segment with boundaries (CA) or predecessors (HF).
- The feeding segment actively insert its vehicles into the queue.
- The destination segment provides the intersection with boundaries (CA) or predecessors (HF).
- The intersection actively inserts its vehicles into the destination segment.
- Blocking will take place either
  - at the beginning of the queue according to the state of the traffic lights, or
  - at certain places within the queue according to obstructions preventing vehicles to leave the queue.

3.5 Memory management

For all objects that are likely to change CPNs (such as `TDriver`, `TVehicle` or `TRoute`) optimized allocation (`getmem`) and deallocation methods (`free`) will be supplied to keep administrive overhead as small as possible. Free memory for instances of these objects will be handled in arrays with lists linking the free entries. Dynamic memory allocation will only be neccessary if the number of objects of a given type temporarily exceeds the estimated maximum number of array entries.

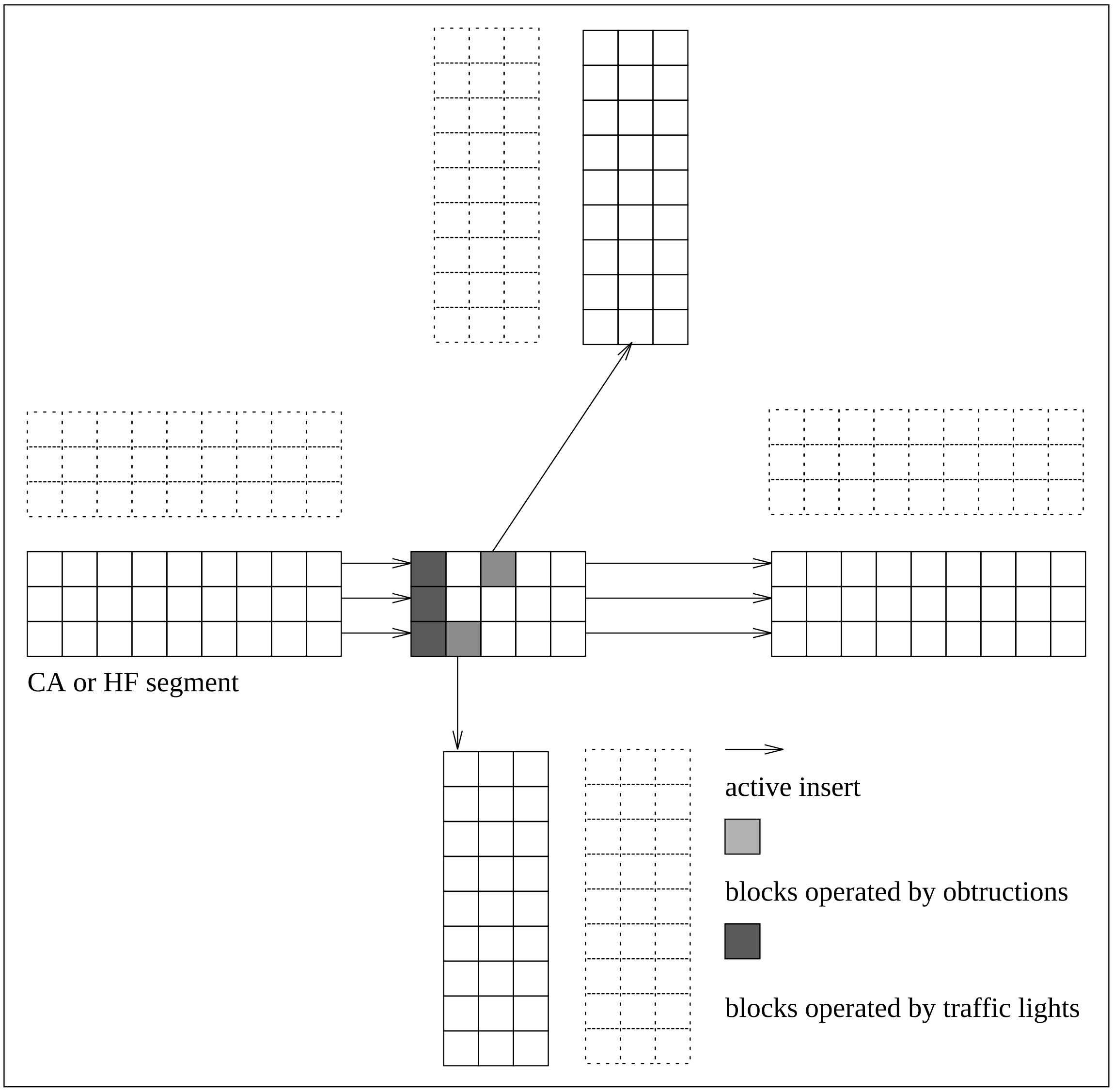

Figure 3: *Queue management*

3.6 Definition of driver behaviour

So far the definition of driver behaviour is accomplished by actually writing C++-Code which results in several disadvantages:

- The code produced is very likely to have errors, since behavioural rules usually transform into 'endless' IF–THEN–ELSE structures which are difficult to overview once they have been written.
- The code is programmer dependant since there might be some matters left to individual interpretation.
- The code is system dependant since the transformation of behavioural rules into checking of parameters depends on the combinaton of parameters available in a specific platform.

Therefore the option of implementing a cross compiler for behavioural rules should be taken into consideration as has been done in the project PARAMICS [2]. Maybe it is even possible to use the same compiler by simply adapting it to the TRANSIMS environment.

4 Parallelization

In microsimulations computational speed is one of the main objectives. Any simulation should run as fast or faster than the problem it tries to model takes in real time. For a large scale simulation of a traffic network this is only possible by distributing the network onto several computational nodes.

In the first step of the implementation the CA version should be able to

- distribute the network onto several computational nodes,
- perform a *dynamic* load balancing during the progress of the simulation in a straight forward easy approach,
- remove or add computational nodes (except the master control node) during the progress of the simulation after an appropriate waiting period.

In the second step efforts can be taken to

- optimize the load balancing so that the frequency and/or amount of communication between nodes is minimized,
- enhance the system so that *any* node of the parallel system can take over the position of the master control node.

4.1 Platform

Roughly spoken at the moment there two major types of parallel super computer hardware:

- The first type can be called the *high end* version because it usually includes especially designed hardware and/or software for both on node computation as well communication between the nodes such as the CM-5 (Thinking Machines) and the Paragon (Intel). On these systems programs are usually assigned to *dedicated* node; that is,

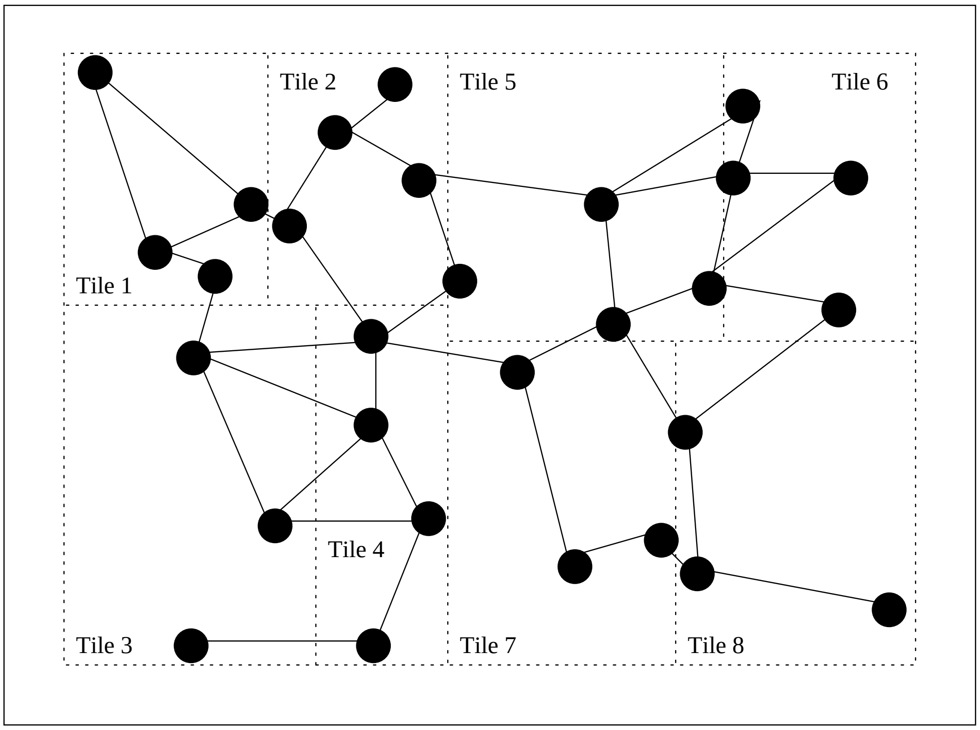

Figure 4: *Initial distribution of nodes onto CPNS*

they run in a multiple task single user environment. Dynamic load balancing is only neccessary if the load that is imposed by the simulation itself changes during the run of the simulation. In case of a microsimulation this very likely to happen since the computation is more or less proportional to the number of vehicles residing on a CPN and as the vehicles travel through the network the load might move accordingly.

- The second type of parallel systems is simply a cluster of workstations connected by a LAN such as Ethernet or better FDDI. Compared to their high end counterparts they are far less expensive but often suffer from poor performance as far as communication is concerned. Moreover being multi user multiple task systems they are more demanding in respect to dynamic load balancing since nodes are very likely to be blocked by co-users or even have to be rebootet once in while.

To keep the implementation as portable as possible the library PVM was chosen (?) to take care of the communication between the computational nodes. As of the latest version of README files PVM exists in *native* ports for both the CM-5 and the Paragon so that a relatively high speed can be expected on these platforms. As for the workstation clusters PVM allows to combine different operating systems and/or hardware in a LAN or even globally via Internet.

4.2 Distribution

As for distribution one has to destinguish between the *initial* distribution onto a given number of computational nodes and the *dynamic* distribution that takes place to keep the load balanced. Of course, if the *dynamic* load balancing is sophisticated enough it should

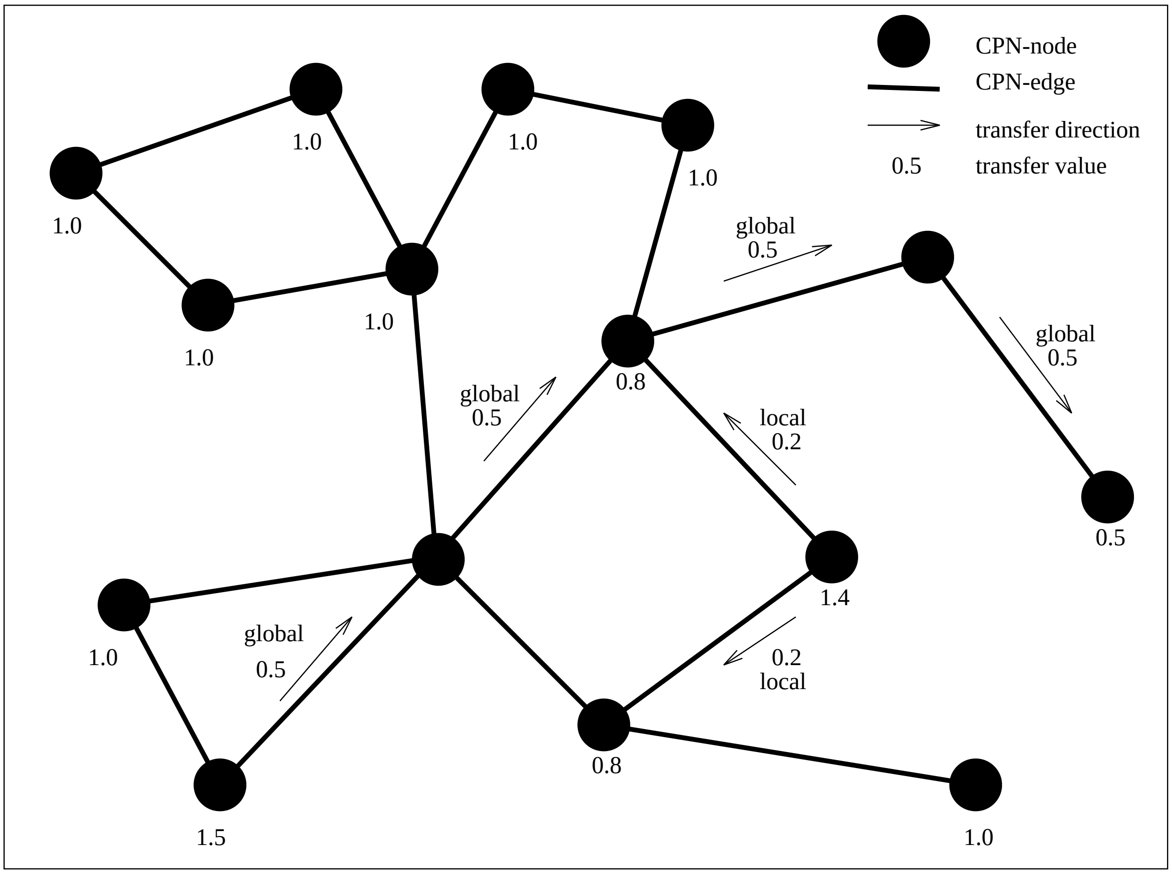

Figure 5: *Local and global dynamic load balancing*

be sufficient to start with single node and simply add the others one after another hoping that the system will adapt accordingly. But since in the first step of the implementation only a straight forward algorithm will be realized this is probably to naive an approach.

Therefore the first distribution will be *geometric* (?); that is, the area covered by the network is split into as many parts as there are CPNs available in the topology. Since the density of nodes and segments is not homogeneous measures will be taken to allow for differently sized *tiles*. Figure 4 contains a network that is equally distributed onto 8 CPNs each taking care of the nodes residing on its associated tile. In this example the number of nodes was used to determine load. The future version may consider the number of edges, the lengths of the edges and the number of nodes or combination of thereof.

As communication is more or less linear to the number of segments crossing the edges of the tiles which in turn is linear to the circumference it will be the objective of the *dynamic* load balacing to keep the tiles convex and as square as possible.

4.3 Dynamic load balancing

In order to handle dynamic load balancing the object `TLoadBalancer` will maintain a CPN graph of the following structure:

- Each CPN will represent a CPN–node. This node contains
  - a list of all CPN–edges (see below) incedent to the node, and

  - the load value (which is computed from the idle time value) of the CPN

- Two CPN–nodes will be connected by a CPN–edge if they have *at least one* common boundary segment.
- Each CPN–edge will have
  - a list of all its associated boundarie segments, and
  - a transfer value that will determine how much load will be transferred over this edge during the next distribution.

During the simulation `TLoadBalancer` will regularly try to reconfigure the load of the CPNs by transferring parts of the network between the CPNs. The reconfiguration depends of a *local* and a *global* balancing (refer to figure 5). The following steps will be executed:

1. Receive idle time infos from each CPN and convert them into load values.
2. The *global* balancing will pick out the CPN with the highest load and as many CPNs with insufficient load as are neccessary to compensate for it. The transfer values of all the CPN-edges leading from the insufficiently loaded CPNs to the overloaded CPN will be set accordingly.
3. During the *local* balancing each CPN will check the load deficiency and load surplus on its neighbours and try to balance it. It will change the transfer values on the CPN-edges accordingly (add them, take mean or take maximum (???)).
4. For each CPN–edge having a transfer value greater zero `TLoadBalancer` will try to move nodes from one associated CPN to the other. If there is more than one choice it will take the node the transfer of which least increases (or even decreases) the number of boundary segments belonging to that CPN-edge. Each node chosen for transfer will be marked. An edge is marked if it changes its CPN association.
5. Synchronize.
6. Send out events triggering the transfer of the marked nodes.
7. Send out events triggering the transfer of the edges. Depending on whether the edge used to be boundary edge or not the event will contain the full edge and dependant information or 'half' of it.
8. Send out events triggering the deletion of the edges on the source CPNs.
9. Send out events triggering the deletion of the nodes on the source CPNs.
10. Update the CPN–graph.
11. Synchronize (??).

Terms

In the next two sections the following terms will be used:

term	meaning
n	number of CPN–nodes
l_i	current load on CPN–nodes i
v_i	valence of CPN–node i
$l_{opt} = 1$	optimal load of a CPN
A_i	set of neighbours of CPN i

Table 3: *Terms used in descriptions of dynamic load balancing*

4.3.1 Global load balancing

`TLoadBalancer` will look for the CPN j having greatest load l_j. Then it will look for k CPNs $n_1 \dots n_k$ having least loads so that just

$$\sum_{i=1}^{k}(1 - l_{n_i}) > l_j - 1$$

A simple Dijkstra will be run to determine the shortest paths (where the weight of each edge will be assumed as one) $p_1 = e_{11} \dots e_{1l_1}$ through $p_k = e_{k1} \dots e_{kl_k}$. The edges on paths p_m for $m = 1, \dots, k-1$ will be assigned the transfer values

$$\bigwedge_{i=1}^{l_m} transfer(e_{mi}) := 1 - l_{n_m}.$$

On the last path the edges will be assigned

$$\bigwedge_{i=1}^{l_k} transfer(e_{ki}) := l_j - 1 - \sum_{p=1}^{k-1}(1 - l_{n_p}).$$

4.3.2 Local load balancing

The local load balancing will regard the local load (of the CPN itself) and on all the neighbouring CPNs. CPN i will compute load surplus S_i on the neighbours according to

$$S_i = \sum_{n_j \in A_i, l_j > 1} \frac{l_j - 1}{v_j},$$

and a load deficiency D_i on the neighbours according to

$$D_i = \sum_{n_j \in A_i, l_j < 1} \frac{1 - l_j}{v_j}.$$

The net load transfered to all neighbours is

$$T_i = S_i + l_i - 1$$

which will be distributed on neighbouring CPN–node n_j and assigned to the associated CPN-edge as follows:

$$t_{ij} = T_i \frac{1 - l_j}{v_j D_i}$$

4.4 Boundaries

As in the approach by Kai Nagel the network will be cut in such a way that boundaries are in the middle of those segments that have nodes in neighbouring tiles. The reason for this is that the interactions that take place on an open stretch of a highway are regarded to be simpler than those at an intersection or junction.

4.4.1 Timing

Every CPN will send its boundaries to the neighbouring CPNs as soon as possible at the beginning of a timestep. After that it will start waiting for its neighbours' boundaries to arrive. As soon as all boundaries have arrived the motion part of the timestep is executed. As Nagel pointed out this system cannot deadlock.

4.4.2 CA model

In the CA model the width of the boundary will be restricted to the maximum number of sites n that a vehicle looks back or ahead which is currently 10. Each CPN will a send message of the first/last n sites to its neighbours which will be used to determine the behaviour of the vehicles leaving or entering the CPN respectively. Since vehicles residing on the boundaries have to behave identically no matter whether they are updated in the original CPN or in their future CPN one has to make sure that all data dertermining random decision will also be transferred by the boundary.

4.4.3 HF model

Since the HF model is vehicle oriented and not site oriented vehicles will be transferred in messages to the neighbours instead of sites. The number of vehicles depend on the behavioural rules implemented. So far only the immediate successor and predecessor of every lane is taken into consideration. Each CPN will only update the vehicles residing on it at the *beginning* of a time step. Vehicles that have left the part of a segment that is handled by its current CPN will be transferred to its new CPN at the *end* of the time step.

To easily find the vehicles next to a boundary (in the HF model) it is probably convenient to introduce *phantom cars* (object `TPhantom`) that are fixed to the boundary so that during update they will not change their positions (??). All methods of the object returning data about its current state will be rewritten so that it returns its predecessor's/successor's data instead.

4.5 Data access

All entities that are used in the simulation will be assigned a unique number through the inheritence from object `TID`. On each CPN an AB-tree (called *ID-tree*) will be maintained containing all pairs of (ID, pointer to object) currently residing on that CPN. Whenever an entity moves from one CPN to another it will keep its ID but its entry in the old ID-tree will be deleted and inserted in the tree on the new CPN.

All *references* to objects will be handled through objects of type `TReference`. The first time a `TReference` is accessed it will search the local ID-tree, retrieve the corresponding pointer to the object and return this pointer. On all subsequent calls it will simply return the stored pointer.

4.6 Event handling

In the simulation four different kind of events will be distiguished:

- **Control events** that are issued by
  - `TSimulationMaster` to control the general flow of the simulation such as start, stop and synchronize which affect all CPNs, or
  - `TLoadBalancer` to trigger the reconfiguration of the network which affects only a subset or a single CPN, or
  - `TScheduler` to trigger certain steps in a simulation update such as `DecideNextAction` and `DoNextAction`. They affect all CPNs in the system.
- **Structural events** are used to encode parts of the network and sent it to another CPN. A structural event has exactly one source CPN and exactly one destination CPN. A vehicle leaving a CPN will be encoded into an event triggering its recreation on the new CPN.
- **Boundary events** are similar to structural events except that the information they carry is not *moved* from one CPN to another but *copied* and has usually a restricted life time on the destination CPN.
- **Inquiry events** resemble control events. They are also sent by `TSimulationMaster` but ususally trigger actions that send data *back* to `TSimulationMaster` which is aggregated on its way.

Before an event leaves its CPN of origin it is encoded into a message and handled by `TMessageMaster`. On the destination CPN the message part is stripped with the event part remaining.

In case an complex object (e.g. `TSegment`) is transferred to another CPN, a single event only containing the object itself is not enough since it might contain both contain local objects and references to dependant objects that will have to be transferred as well. Consider an object like the one in figure 6. `TChild` is an object derived from object `TParent`. It has local objects such as `TObject` and references to dependants like `TDependant`. The following order will be used to encode the complex object recursively:

1. Encode the parent object `TParent`.
2. Encode all local variables that are not objects and have global scope and encode heap variables referenced by local variables.
3. Encode all local objects such as `TObject`.
4. Encode depandant objects like `TDependant`.

Note that every object is only a depandant of exactly *one* other object although it may be references by several others. Indepandant objects that are primarily handled by other objects are not encoded. Otherwise they would be multiply created on the destination CPN.

On the destination node the object is decoded in exactly the same order. In fact it is the `decoding` and not the `encoding` that determines the order. Part A of the object may have to encoded before part B because part A may contain information about how to decode part B.

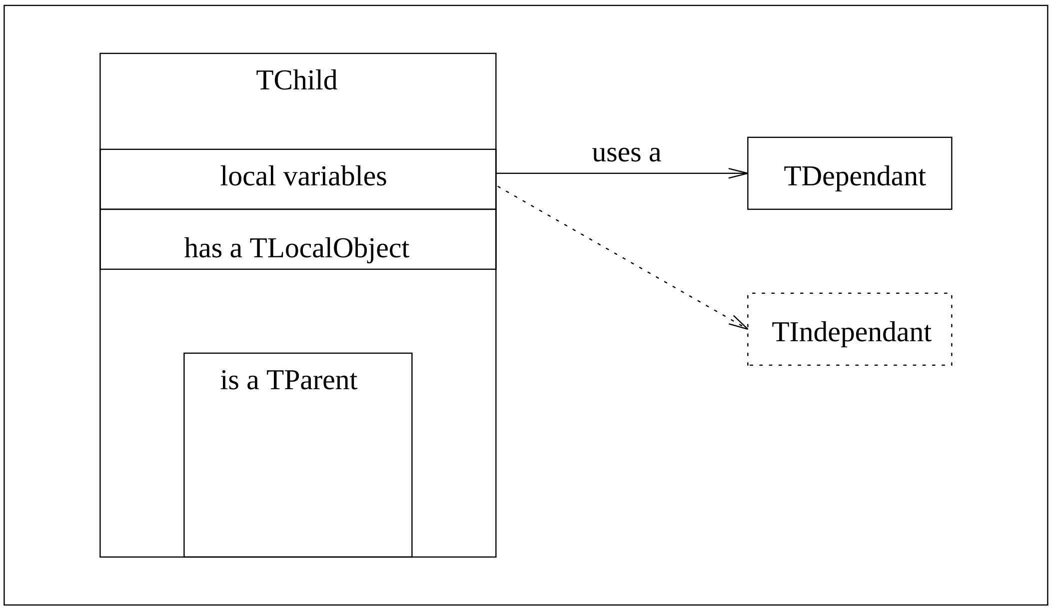

Figure 6: *Order of encoding*

4.7 A simple examplary run

1. The user initializes PVM by calling pvm and adding all CPNs manually or starts a script to do it automatically.
2. The user starts the program on the master CPN with a start configuration of slave CPNs.
3. `TSimulationMaster` starts all other instances of the program on the slave CPNs (`TSimulationSlave`). They will enter the main message loop and wait for messages.
4. `TSimulationMaster` reads the network structure from the input data and pass the information to `TLoadBalancer`.
5. `TLoadBalancer` distributes the network and sends out messages to the slave CPNs with encoded network elements.
6. `TSimulationMaster` sends an event to all CPNs to start the simulation.
7. `TScheduler` enters the main loop:
   (a) It prompts all CPNs to send their boundaries to the neighbouring CPNs.
   (b) It sends the events `DecideNextAction` and `DoNextAction` to all CPNs.
8. After a certain number of iterations there are several options:
   - `TSimulationMaster` prompts all CPNs to send their idle time data. This information is used by `TLoadBalancer` to redistribute the network.
   - `TSimulationMaster` prompts all CPNs to send local statistical data about their traffic load which is aggregated by `TSimulationMaster`.

- The X-Windows event queue is checked if there is need to update the graphics output.
- The interactive user interface is checked if
  - commands regarding the progress of the simulation were given,
  - CPNs have to be removed from the topology or new CPNs can be added to the topology.

9. `TSimulationMaster` stops the simulation.

5 Problems

5.1 Handling of queues

In the current approach of the implementation of queues there is a bijective projection between the queues in an intersection and the lanes leading up to the intersection. Unfortunately this does not cover the special case that the same lane contains two lanes blocking each other such as the left turn queue and the through lane queue on a single lane (per direction) street.

5.2 Granularity

Boundaries will be located in the middle of segments to avoid the complex behaviour and references close to nodes. Nevertheless the point where a segment is split is only *indepedent* of the associated nodes if the segment is twice as long as vision range. This reduces the potential number of segments that are suited for the placement of a boundary considerably and thus creates a coarser grid for load balancing.

5.3 Synchronization

Synchronization is usually done by emptying the message queues on all CPNs in order to continue the next part of the simulation either simultanously (which is difficult to achive and fortunately hardly ever neccessary) or consistenly; that is, referring to the same data and time configuration.

Since synchronization is very time expensive it should be restricted to cases in which it is absolutely neccessary. Whenever possible the simulation should be permitted to run asynchronously with different time steps being processed on the CPNs provided that the difference in time steps between *neighbouring* CPNs never exceeds one.

5.4 Scalability

Although the microsimulation part itself scales well with the number of CPNs provided[5] there are several aspects that scale only poorly:

- Processing of the input data is done by the master CPN. This also results in a severe disparity of memory requirements since the whole network has to be stored to execute the load balancing.

[5]This only applies to parallel computer hardware that provides simultaneous local communication scaling with the number of CPNs

- Graphics output is handled by a single CPN.
- Global actions such as gathering statistics or command dispatch triggers dense cascades of local and/or global communication. This will especially affect workstation clusters where communication between any pair of CPNs is always *sequential*.

References

[1] C. Barret and D.J. Roberts. The transims microsimulation status. Technical report, TSA-DO/SA, Los Alamos National Lab, New Mexico, USA, 1994.

[2] D. McArthur. The *paramics* model: Present and future directions. Technical report, SIAS Ltd., Edinburgh, 1994.

[3] K. Nagel and M. Schreckenberg. A cellular automaton model for freeway traffic. *J. Physique I*, 2:2221, 1992.

[4] M. Rickert. Simulation zweispurigen Verkehrsflusses auf der Basis zellularer Automaten. Master's thesis, Universität zu Köln, 1994.