Cadyts – calibration of dynamic traffic simulations
Version 1.1.0 manual

Gunnar Flötteröd*

November 4, 2010

1 Introduction

Iterated microsimulations have become prominent solution procedures for the dynamic traffic assignment (DTA) problem. Their arguably most frequently mentioned advantages are (i) structural consistency with real transportation systems, and (ii) intuitive means to understand the iterative simulation logic in terms of a "learning process" of individual travelers (Nagel and Flötteröd, 2009).

However, these advantages come with drawbacks: (i) transport microsimulations also require behavioral modeling at the individual level, which is a data hungry and methodologically challenging problem, and (ii) the intuition of "learning" alone is too weak to analyze the results of an iterated simulation.

These problems have limited the development of mathematically consistent calibration techniques for DTA microsimulations. The Cadyts software aims to overcome these difficulties. It adopts a mathematically well-defined stochastic process view on the simulation (Cascetta, 1989; Cascetta and Cantarella, 1991; Nagel et al., 1998) and calibrates its behavioral parameters in a Bayesian setting from traffic counts and/or vehicle re-identification data (Flötteröd, 2008, 2009; Flötteröd et al., 2010).

This text describes how to use Cadyts, but it does not provide a comprehensive description of its theoretical underpinnings. However, some basic concepts are

*gunnar.floetteroed@epfl.ch

provided in Section 2. Sections 3 and 4 provide step-by-step explanations of how to use the software. If you plan serious work with Cadyts, you are more than welcome to contact the author for support.

2 How Cadyts works

Cadyts is designed to interact with an stochastic and iterative DTA microsimulation. Stochastic means that at least the agent (simulated traveler) behavior is non-deterministic. The simulation's traffic flow model may either be deterministic or stochastic. Iterative means that the simulation runs according to the following logic:

1. Initialization.
2. Iterations. Repeat the following until stationary conditions are reached.
   (a) Demand simulation. All agents select new plans (e.g., routes) based on what they have learned in the previous iterations.
   (b) Supply simulation. The plans of all agents are simultaneously executed on the network (traffic flow simulation, network loading).

Note that this logic is equally applicable to simulate an equilibrium-based planning model and a telematics model where drivers are spontaneous and imperfectly informed. From a simulation point of view, the only difference between these two models is that an equilibrium demand simulator typically utilizes all information from the most recent supply simulation, whereas a telematics demand simulator generates every elementary decision within an agent's travel plan only based on such information that could have actually been gathered up to the according point in simulated time (Bottom, 2000).

Cadyts is able to calibrate the demand simulation, i.e., the individual-level behavior of all simulated travelers, from traffic counts and/or vehicle re-identification data. (A simultaneous calibration of the supply simulation is desirable not only from a practical but also from a mathematical point of view. This feature is likely to be added in a later version.)

In order to apply Cadyts to a concrete DTA microsimulation, the following is needed.

1. Before the simulation is started, Cadyts needs to be initialized. It also needs to be provided with the measurement data based on which the calibration is to be conducted. This is subsequently called the **initialization step**.

2. Cadyts must have a chance to affect the agents' plan choice and/or their behavioral parameters. There are various ways to realize this, depending on the DTA simulation at hand. This is subsequently called the **choice step**.

3. Cadyts must observe the simulated network conditions in order to compare them to the sensor data. This is subsequently called the **update step**.

How these three steps are realized depends on the simulation system at hand. Section 3 describes this at the software level, i.e., it is assumed that the calibration interacts with the simulation through direct function calls. Section 4 then demonstrates how a file-based interaction can be realized.

3 How to use the calibration – function calls

This section assumes that (an interface to) the simulation system is implemented in Java. It first specifies in Subsection 3.1 some data structures that are referred to throughout the text. Subsections 3.2 through 3.5 then describe different ways of applying Cadyts to particular calibration problems for particular simulations.

A few notes about software design. The Cadyts code tries to follows the guidelines given in Gamma et al. (1994). The eventual need for refactorings (Fowler, 1999) is hidden from the user as much as possible by separating the software into two parts: One set of internal packages that are likely to be frequently refactored, and one set of user packages that link the internal functionality to the simulation systems. The user packages serve as facades that shield the simulation from the internal calibration code and its potential changes.

If you plan to systematically use Cadyts, feel free to contact the author to agree on an user interface package that serves your needs.

3.1 Data structures

Unless stated otherwise, `L` represents the generic network link type in the simulation code. Time is generally represented in integer seconds after midnight.

3.1.1 Demand data structures

Travel demand is represented in terms of travel plans of individual agents. These plans are represented by instances of the `Plan<L>` interface. The functions of

`Plan<L>` are used only internally by the calibration. An instance of `Plan<L>` is build through a `PlanBuilder<L>` that provides the following relevant functions.

```
public void reset()
```

prepares the builder for the generation of a new `Plan<L>` instance. This instance is then built by calls to the functions

```
public void addEntry(L entryLink, int time_s)
public void addTurn(L toLink, int time_s)
public void addExit(int time_s)
```

that define the network entries, turning moves at intersections, and network exits in the considered travel plan, including all associated timing information. The result of this building process can be accessed via

```
public Plan<L> getResult().
```

3.1.2 Supply data structures

Supply is specified through the interface `SimResults<L>` that defines the single function

```
public double getSimValue(L link,
int startTime_s, int endTime_s,
SingleLinkMeasurement.TYPE type).
```

An implementation must provide the simulated traffic conditions of the specified `type` on every network link in every time interval from `startTime_s` (inclusive) to `endTime_s` (exclusive). Possible `SingleLinkMeasurement.TYPE` instances are `FLOW_VEH_H` (average flow rate in the considered time interval) and `COUNT_VEH` (total vehicle count in the considered time interval). The class `BasicSimResults<L>` provides a default implementation of the `SimResults<L>` interface.

3.1.3 Measurement data structures

There are two type of measurement data structures: single-link measurements and multi-link measurements.

Single-link measurements represent an observation that is made on (the upstream end of) a single link during a given time interval. Single-link measurements are represented by the class `SingleLinkMeasurement<L>`. In order to generate an instance of this class, use its constructor

```
public SingleLinkMeasurement(L link,
double measValue, double measVariance, int
startTime_s, int endTime_s, TYPE type)
```

or resort to one of the convenience functions of the `Calibrator<L>` class described further below.

Multi-link measurements represent multiple observations of the same vehicles at (the upstream ends of) several links in the network. Multi-link measurements are represented by the class `MultiLinkMeasurement<L>`. In order to generate an empty instance of this class, use its constructor

```
public MultiLinkMeasurement(int value_veh,
double detectionRate)
```

where `value_veh` represents the number of vehicles that are observed at *all* sensor locations and `detectionRate` (strictly positive and at most one) is the probability that a real vehicle that crosses all sensor locations is also detected by *all* these sensors.

The observations per sensor location are then sequentially added using the function

```
public void addObservation(L link, int
start_s, int end_s)
```

where `link` is the link on the upstream end of which the observation is made and `start_s` and `end_s` define the time span of this observation.

The variance of a multi-link measurement is internally inferred from its `detectionRate`, the `varianceScale` parameter, and the `minStddev` parameter for `COUNT_VEH` data.

3.2 The basic `Calibrator`

Here as well as in the following subsections, the presentation is in terms of the three steps **initialization**, **choice**, and **update**, which need to be implemented in order to link Cadyts to a concrete simulation, cf. Section 2.

The basic `Calibrator` can be used to adjust the simulated travel behavior in the agent population through additive utility corrections, which requires that the demand simulator implements a utility-based plan choice model.

3.2.1 Step 1: initialization

A `Calibrator` is constructed using

```
public Calibrator(String logFile, Long
randomSeed, int timeBinSize_s),
```

using the following parameters.

`logFile` A `String` that indicates a file where log messages are to be written. Logging to file is disabled by setting this to `null`.

`randomSeed` A `Long` random seed. Using the same random seed in several experiments enables reproducible calibration behavior. If a `null` value is provided, the default random seed zero of zero is used.

`timeBinSize_s` The size (in seconds) of the time bins in which the calibration internally collects simulation-related data. The length of a day (86400 s) must be an integer multiple of this value. Could be, e.g., 3600 s for the calibration of a planning simulation with slow within-day dynamics.

The following parameters can be set in the `Calibrator` class. (The according setters and getters are consistent with the variable names used here. For example, for parameter `foo` there is a `getFoo()` and a `setFoo(..)` function.)

`regressionInertia` A real number larger than zero and at most one. The calibration internally relies on regression models that approximate the simulation's workings. This parameter defines the inertia of this tracking. Too low of an inertia is likely to lead to oscillations and unstable results, whereas too large of an inertia leads to slow convergence. The default value is 0.95, which should work fine for most applications.

freezeIteration An integer number that is larger or equal to 0. It specifies after how many iteration the adaptivity of the calibration is turned of. When the calibrated simulation is started, there usually is a lot of change going on in the network conditions and the agent behavior, which the calibration needs to track in some way. However, when the situation stabilizes, the adaptivity of the calibration is useless and should be turned off. This happens after `freezeIteration` iterations. This parameter should be identified experimentally: First, run a calibration with `freezeIteration` set to its default value `Integer.MAX_VALUE` and observe after how many iterations the situation stabilizes. The `freezeIteration` parameter should then be chosen somewhat larger than this number of iterations.

varianceScale A strictly positive real number. If measurements that go without an explicitly specified variance are registered with the `Calibrator`, then it is assumed that their variance is `varianceScale` times the measured value. (In order to avoid numerical problems, the variance is bounded from below; see the next item.) The default value of `varianceScale` is 1.0, which is consistent with the assumption of Poisson distributed measurements.

minStddev A strictly positive real number that defines the smallest allowed standard deviation for every measurement type separately (Possible types are `SingleLinkMeasurement.TYPE.COUNT_VEH` and `SingleLinkMeasurement.TYPE.FLOW_VEH_H`.) The default value is 25 for all measurement types. The getter and setter for `minStddev` take a `SingleLinkMeasurement.TYPE` as an additional parameter.

preparatoryIterations A positive integer number. Defines the number of iterations during which the calibration merely observes the simulation without taking any effect. This might in some situations increase the stability of the calibration. The default value is one.

centerRegression A Boolean variable that specifies if the recursive regressions through which the calibration tracks the dynamics of the simulation should work with adaptive working points. Experimental feature with default value `false`.

statisticsFile A `String` variable that contains the name of a file where various statistics of the calibration are continuously written that helps to monitor convergence and identify problems. The default value is `calibration-stats.txt`. Table 1 describes the contents of this file.

Table 1: Columns of the calibration statistics file

#	column	explanation
1	`count-ll`	log-likelihood value (only for single-link measurements)
2	`count-ll-pred-err`	prediction error of `count-ll`; a measure of how well Cadyts estimates the network loading's reaction to changes in the plan choice distributions
3	`p2p-ll`	log-likelihood value (only for multi-link measurements)
4	`total-ll`	total log-likelihood value (sum of `count-ll` and `p2p-ll`)
5	`link-lambda-avg`	average link utility offset (only for single-link measurements)
6	`link-lambda-stddev`	standard deviation of link utility offsets (only for single-link measurements)
7	`link-lambda-min`	minimum link utility offset (only for single-link measurements)
8	`link-lambda-max`	maximum link utility offset (only for single-link measurements)
9	`plan-lambda-avg`	average plan utility offset (for all measurements)
10	`plan-lambda-stddev`	standard deviation of plan utility offset (for all measurements)
11	`plan-lambda-min`	minimum plan utility offset (for all measurements)
12	`plan-lambda-max`	maximum plan utility offset (for all measurements)
13	`replan-count`	number of replanning agents

Every row in the file corresponds to one iteration. It consists of 13 tab-separated columns. Columns 1 through 4 give log-likelihood values that allow to monitor the convergence of the calibration; in particular, column 4 captures the overall measurement fit. Columns 9 through 12 report statistics of (possibly hypothetical) plan utility corrections ("lambda-values") that can be interpreted as follows: If the behavioral model was logit with scale parameter one then the effect of the calibration would be equivalent to changing the utility of an alternative by a particular utility offset. For single-link measurements, the plan utility offsets are additively composed of utility offsets computed at every sensor location; the respective statistics are given in columns 5 through 8.

`proportionalAssignment` A Boolean variable that indicates if the calibration is to internally linearize the supply simulator based on a proportional assignment assumption. Such a setting is appropriate only in uncongested conditions, and the parameter is available mainly for experimental purposes. If it is set to `false`, Cadyts deploys a more sophisticated regression-based linearization that can also cope with congested network conditions (Flötteröd and Bierlaire, 2009). The default value is `false`.

`debugMode` A Boolean variable that indicates if fine-grained debug messages are to be logged. The default value is `false`.

`flowAnalysisFile` A `String` variable that contains the name of a file where detailed flow analysis information is written for all single-link measurement locations. The default value is `null`.

Once these parameters are set, the measurements can be registered with the calibration. (Since some parameters affect the way the measurements are processed, measurements should not be registered before all parameters are specified.)

If the measurements are already represented as instances of `SingleLinkMeasurement` or `MultiLinkMeasurement`, cf. Subsection 3.1.3, then the functions

```
public void addMeasurement(SingleLinkMeasurement<L>
meas)
public void addMeasurement(MultiLinkMeasurement<L>
meas)
```

should be used to feed them into the calibration.

For the registration of single-link measurements, `Calibrator` provides the following convenience functions:

```
public void addMeasurement(L link, int
start_s, int end_s, double value, double
stddev, SingleLinkMeasurement.TYPE type)
public void addMeasurement(L link,
int start_s, int end_s, double value,
SingleLinkMeasurement.TYPE type)
```

Both register the measurement `value` of the given `type` on the given `link` between `start_s` and `end_s`, both in seconds from midnight. The first function

specifies the standard deviation of the measurement explicitly in `stddev`. In the second version, the measurement variance is inferred from value according to

$$\text{stddev} = \min\left\{\text{minStddev}, \sqrt{\text{varScale} \cdot \text{value}}\right\}.$$

The abstract class `BasicMeasurementLoader<L>` provides most functionality that is needed feed measurements that are stored in an XML file directly into the `Calibrator`. A concrete subclass of `BasicMeasurementLoader` only needs to implement the function

```
protected abstract L label2link(String label)
```

that defines how the string label of a link in the XML file is mapped on a link instance of type `L`. After having created a subclass of `BasicMeasurementLoader` using its constructor

```
public BasicMeasurementLoader(Calibrator<L>
calibrator),
```

the measurement file(s) can be loaded with a call to

```
load(final String...  measurementFileNames).
```

An example of the expected XML file format is given below:

```
<measurements>

    <!-- a single-link measurement between
    second 25 200 and 28 800 after midnight
    -->
    <singlelink link="2" start="25200"
    end="28800" value="1200" stddev="15"
    type="COUNT_VEH"/>
    <!-- a multi-link observation of
    100 vehicles with perfect detection
    technology -->
    <multilink value="100"
    detectionrate="1.0">
```

```
        <observation link="one"
        start="14:00:00"
        end="14:30:00"/>
        <observation link="two"
        start="14:15:00"
        end="14:55:00"/>
    </multilink>

</measurements>
```

3.2.2 Step 2: choice

The basic `Calibrator` class calibrates the plan choices in a utility-based demand simulation through utility adjustments. The following description is given under the assumption that the demand simulation is a multinomial logit model with scale parameter μ. (That is, it is assumed that the choice probability of an alternative i with original utility V_i is, apart from normalization, proportional to $e^{\mu V_i}$). For this, the following three steps are necessary whenever the plan choice of an agent is to be simulated:

1. The utility modification for every plan in the agent's choice set is computed by using the function

   ```
   public double calcLinearPlanEffect(Plan<L>
   plan).
   ```

2. The agent selects a plan based on modified utilities, where the modified utility of a particular plan is the "linear plan effect" computed in Step 1 divided by the scale parameter μ.

3. The calibration is informed of the chosen plan through a call to

   ```
   public void addToDemand(Plan<L> plan).
   ```

3.2.3 Step 3: update

In order to inform the Calibrator of the network conditions that resulted from the most recent replanning stage, the network loading needs to be followed by a call to

```
public void afterNetworkLoading(SimResults<L>
simResults)
```

where `simResults` represents the most recently simulated network conditions, cf. Subsection 3.1.2.

3.3 The `SamplingCalibrator`

The `SamplingCalibrator` is a subclass of `Calibrator` that adjusts the simulated plan choice distributions of all agents without any further assumptions about the workings of the demand simulator: Whenever an agent selects a plan, it proposes this plan to the calibration. The calibration either accepts or rejects the plan. If the plan is rejected, the agent must select again. Eventually, the calibration accepts a plan, which is the plan the agent keeps.

3.3.1 Step 1: initialization

The `SamplingCalibrator` inherits all parameters of the basic `Calibrator`. It has the following additional parameters, which should be set together with the other `Calibrator` parameters.

`maxDraws` An integer number greater than or equal to 2. It specifies the largest number of draws until an accept occurs in a single choice situation. This parameter allows for a trade-off between computational speed and calibration precision. The larger `maxDraws` the slower and the more precise is the calibration. Vice versa, small values yield faster yet increasingly imprecise estimates. The default value is 20.

`choiceSamplerFactory` An implementation of the `ChoiceSamplerFactory<L>` interface that creates instances of the `ChoiceSampler<L>` interface described further below. By default, this parameter is configured with an instance of `RecursiveSamplerFactory<L>` that creates `RecursiveSampler<L>` instances.

3.3.2 Step 2: choice

Before an agent selects a plan, a `ChoiceSampler<L>` instance needs to be obtained from the `SamplingCalibrator` by calling

`public ChoiceSampler<L> getSampler(Object agent)`.

The `ChoiceSampler` interface defines the relevant function C

`public boolean isAccepted(Plan<L> plan)`,

which implements the accept/reject procedure. The agent must randomly and independently select new plans until `isAccepted(..)` returns true for the first time. This will happen after at most `maxDraws` draws. The accepted plan then needs to be treated as if it was originally chosen by the agent.

3.3.3 Step 3: update

Like in `Calibrator`.

3.4 The `AnalyticalCalibrator`

The `AnalyticalCalibrator` is applicable to simulations where the choice protocol is such that first a choice set is enumerated and then a choice is made according to explicitly specified choice probabilities. It works as follows: Whenever an agent will select a plan, the simulation provides the `AnalyticalCalibrator` with this agent's choice set, including all choice probabilities. The `AnalyticalCalibrator` then selects one plan from the choice set in consideration of both the (a priori) choice probabilities and the measurements. This plan is then executed by the simulation as if it was the original choice of the agent.

3.4.1 Step 1: Initialization

The `AnalyticalCalibrator` inherits all parameters of the basic `Calibrator`. It has the following additional parameter, which should be set together with the other `Calibrator` parameters.

`bruteForce` A Boolean variable that indicates if the calibration shall make a maximum ("brute force") effort to reproduce the measurements. From a model calibration point of view, this is not a meaningful option. However, it may be useful to identify inconsistent measurements that cannot be reproduced even when this parameter set to `true`. The default value is `false`.

3.4.2 Step 2: choice

The following is necessary to calibrate the choice distribution of a replanning agent with the `AnalyticalCalibrator`.

1. The complete choice set is put into a `List<? extends Plan<L>>`.
2. The a priori choice probabilities specified by the simulation are put into an instance of `Vector`. Their order needs to correspond to the order of the plans.
3. A choice is triggered by calling

```
public int selectPlanIndex(List<? extends
Plan<L>> plans, Vector choiceProbs),
```

which returns the index of the plan selected by the calibration. The `AnalyticalCalibrator` also has some convenience getters that provide later access to the most recent selection results:

```
public double getLastChoiceProb(int index)
public int getLastChoiceIndex()
```

3.4.3 Step 3: update

Like in `Calibrator`.

3.5 The `ChoiceParameterCalibrator`

The `ChoiceParameterCalibrator` is a subclass of the `AnalyticalCalibrator`. It affects the replanning behavior both directly (like the `AnalyticalCalibrator`) and through an adjustment of the demand model's parameters. Internally, it does so by implementing either a stochastic gradient ascent algorithm that maximizes the a posteriori probability of the behavioral model's parameters.

3.5.1 Step 1: Initialization

The constructor of the `ChoiceParameterCalibrator` takes one additional parameter.

`parameterDimension` A strictly positive integer that indicates the number of choice model parameters to be calibrated.

The `ChoiceParameterCalibrator` inherits all parameters of the `AnalyticalCalibrator` (and hence also of the basic `Calibrator`). It has the following additional parameters, which should be set together with the other `(Analytical)Calibrator` parameters.

`initialStepSize` A strictly positive real number that defines the initial step size of the parameter adjustment. (Also see the next item.) The default value is 1.0.

`msaExponent` A non-negative step size control parameter. Because only a stochastic evaluation of the log-likelihood function is possible due to the simulated environment, the step size of the parameter adjustment is decreased according to

$$\text{initialStepSize}/\text{iteration}^{\text{msaExponent}}.$$

If `msaExponent` is set to zero, the step size remains constant. For a decaying step size, recommended values are in the range (0.5,1.0]. The default value is 1.0, which corresponds to the method of successive averages (MSA; hence the name of the parameter).

`useApproximateNewton` A Boolean variable that indicates if an approximate Newton algorithm is to be used for the maximization of the a posteriori parameter probabilities. If it is set to false, a simple stochastic gradient ascent is deployed. The default value is `true`.

`initialParameters` A `Vector` of initial parameter values. It serves two purposes. First, it defines the starting point of the parameter calibration. Second, it represents the a priori estimate of the parameters. (That is, the calibration always starts from the a priori values.) This parameter may be `null`, in which case the starting point of the calibration is an all-zero vector and no a priori knowledge is accounted for in the calibration. The default value is `null`.

`initialParameterCovariance` A symmetric positive definite `Matrix` that captures the uncertainty associated with the a priori parameter estimates. It may be `null`, in which case no a priori knowledge is accounted for in the calibration. The default value is `null`. There also is a convenience setter `setInitialParameterVariances(Vector)` that defines a diagonal covariance matrix.

3.5.2 Step 2: choice

The replanning step is implemented differently from the `AnalyticalCalibrator` in that the original `selectPlanIndex(List<? extends Plan<L>> plans, Vector choiceProbs)` function is no more available (it throws an `UnsupportedOperationException`).

Instead, the following function available:

```
public int selectPlan(List<? extends
Plan<L>> plans, Vector choiceProbs, Matrix
dChoiceProb_dParam, List<? extends Matrix>
d2ChoiceProb_dParam2)
```

where the parameters have the following meaning:

- `plans` contains the choice set.
- `choiceProbs` contains the respective choice probabilities.
- `dChoiceProb_dParam` is a `Matrix` that contains the derivatives of all choice probabilities with respect to all choice model parameters in that the matrix element in row *i* and column *j* contains the derivative of the choice probability of the i*th* alternative with respect to the *j*th model parameter.
- `d2ChoiceProb_dParam2` is a list of symmetric matrices that contain the Hessians of the choice probabilities with respect to the parameters: The *i*th element of this list contains a `Matrix` where the element in row *j* and column *l* is the derivative of the choice probability of alternative *i* with respect to the parameters *j* and *l*.

If a multinomial logit plan choice model is used in the simulation, a convenience implementation `MultinomialLogit` is available that can be parametrized through self-explaining setters and provides getters for the choice probability vector, the derivative matrix, and the list of Hessians in the format required by the `selectPlan(..)` function.

3.5.3 Step 3: update

First, the `ChoiceParameterCalibrator` needs to be informed of the most recent network conditions as it is described for the basic `Calibrator` class.

After this, an updated parameter vector is available that can be obtained via the function

```
public Vector getParameters(),
```

which contains the parameters in an order that is consistent with the ordering of the columns of the `dChoiceProbs_dParam Matrix` in the `selectPlanIndex(...)` function described above. The current parameter estimates should be retrieved from the calibration and applied to the choice model immediately after their retrieval. (The `MultinomialLogit` class can be directly parametrized with the `Vector` returned by `getParameters()`.)

Finally, a rough estimate of the current parameter covariance matrix is available via the function

```
public Matrix getParameterCovariance().
```

4 How to use the calibration – file transfers

A file-based interaction between calibration and simulation requires to enable Cadyts to read all required data from file and to write its results to file. It is unlikely that the accept/reject procedure of the `SamplingCalibrator` can be efficiently implemented via files. The `AnalyticalCalibrator`, however, communicates in relatively large data chunks with the calibration in that it reads, for every agent, its complete choice set before generating a single choice.

The basic functionality for interfacing through files is implemented in the abstract `FileBasedController` that provides a command-line controlled interface to a (subclass of) `AnalyticalCalibrator`.

Examples for a file-based interaction can be found in the package `cadyts.interfaces.sumo` that binds the SUMO microsimulation (SUMO, accessed 2010) to Cadyts and in the package `cadyts.interfaces.dracula` that binds the DRACULA microsimulation (Liu, 2005; DRACULA, accessed 2010) to Cadyts.

4.1 Additional data structures

Since the agents and their plans are now read from files, additional data structures for their representation within the java code of Cadyts are necessary. For this, the class

```
Agent<P extends Plan<String>, D extends
PlanChoiceModel<P>>
```

serves as a container for a single agent, its choice set, and its `PlanChoiceModel<P extends Plan<?>>`, which is an interface that defines a single function

```
public Vector getChoiceProbabilities(final
List<?  extends P> plans)
```

that maps a list of plans on a respectively ordered vector of plan choice probabilities. A default implementation `PlanChoiceDistribution<P extends Plan<?>>` is available that can be configured with fixed plan choice probabilities; this is likely to be sufficient when the choice probabilities are explicitly written to file by an external demand simulator.

4.2 Necessary implementations to enable file-based interactions

The abstract class

```
FileBasedController<C extends
AnalyticalCalibrator<L>, A extends Agent<P,
?  extends PlanChoiceModel<P>>, P extends
Plan<L>, L>
```

needs to be subclassed in order to coordinate the interactions between the command line calls to the calibration and the function calls to the (subclass of) `AnalyticalCalibrator` that implements the desired calibration logic. For this, an implementation of `FileBasedController` needs to implement the factory method

```
protected abstract C newCalibrator(CommandLineParser
clp)
```

that returns the `AnalyticalCalibrator` (subclass) that is to be used for the calibration. The `CommandLineParser` provides access to the command line arguments that are passed to the `FileBasedController` (see below).

The function

```
abstract protected void loadMeasurements(C
calibrator, String measFile)
```

loads the measurements from file and registers them with the calibrator. For the reading and measurement data in XML format, a `BasicMeasurementLoader<L>` is available.

The factory method

```
protected abstract PopulationFileReader<A>
newPopulationReader(C calibrator,
CommandLineParser clp)
```

provides access to the agent population, including their current choice sets, by returning an instance of the interface `PopulationFileReader<A extends Agent<?, ?>>` that defines the single function

```
public Iterable<A> getPopulationSource(String
populationFile).
```

through which the `FileBasedController` is enabled to iterate over the agent population. (A is an instance of A that provides access to the agents, their choice sets, and their choice models, cf. Subsection 4.1.)

The factory method

```
protected abstract ChoiceFileWriter<A, P>
newChoiceFileWriter(C calibrator)
```

returns a `ChoiceFileWriter<A extends Agent<P, ?>, P extends Plan<?>>` interface that defines the three functions public void `open(String choiceFile)`, `public void write(A agent, P plan)`, and `public void close()`. An implementation of the `write(..)` function needs to write the chosen plan of the considered agent to file.

Finally, an implementation of

```
abstract protected void update(C calibrator,
CommandLineParser clp)
```

is responsible for loading the network conditions of the most recent network loading from file and communicating them to the calibrator through a call to its `afterNetworkLoading(..)` function, cf. Subsection 3.2.

4.3 Program flow control with the `FileBasedController`

Once a subclass of `FileBasedController` is implemented, the calibration can be run. `FileBasedController` provides a `run(String[] args)` function that can be called directly with the Java command line parameters received in any `main`-function. It identifies from the first entry in this array what to do. Possible entries are "INIT", "CHOICE", and "UPDATE", which trigger the initialization step, the choice step, and the update step, respectively.

All calibration parameters can be set from the command line. Tables 2 through 4 show the available parameters and their corresponding Cadyts variables. (Calling the `run` function with a single argument "INIT", "CHOICE", or "UPDATE" displays these parameters as well.)

In order to define additional command line parameters in a concrete subclass of `FileBasedController`, the following is necessary. First, any of the functions

```
protected void prepareCommandLineParserINIT(
CommandLineParser clp)
protected void prepareCommandLineParserCHOICE(
CommandLineParser clp)
protected void prepareCommandLineParserUPDATE(
CommandLineParser clp)
```

need to be overridden and the additional parameters need to be registered with the `CommandLineParser`. (The first call in the overriding function should be to the respective superclass function in order to maintain the parameters defined there). Second, the functions

```
protected void prepareCalibratorINIT(C
calibrator, CommandLineParser clp)
protected void prepareCalibratorCHOICE(C
calibrator, CommandLineParser clp)
```

Table 2: INIT parameters of the FileBasedController

keyword	explanation (corresponding Cadyts variable)
-BINSIZE	required; numerical bin size [s]; no default value (timeBinSize_s in Calibrator)
-BRUTEFORCE	optional; enforces best effort in measurement reproduction; default = false (bruteForce in AnalyticalCalibrator)
-CENTERREGR	optional; centering of internal regressions; default = false (centerRegression in Calibrator)
-DEBUG	optional; if fine-grained debug messages are to be generated; default = false (debugMode in Calibrator)
-FREEZEIT	optional; number of iterations until system freezes; default = 2147483647 (freezeIteration in Calibrator)
-LOGFILE	optional; logfile; no default value (logFile in Calibrator)
-MEASFILE	required; comma-separated list of files that contain the measurements; no default value (no Cadyts variable)
-MINCOUNTSTDDEV	optional; minimum count standard deviation [veh]; default = 25.0 (minStddev for SingleLinkMeasurement.TYPE.COUNT_VEH in Calibrator)
-MINFLOWSTDDEV	optional; minimum flow standard deviation [veh/h]; default = 25.0 (minStddev for SingleLinkMeasurement.TYPE.FLOW_VEH_H in Calibrator)
-PREPITS	optional; number of preparatory iterations; default = 1 (preparatoryIterations in Calibrator)
-PROPASSIGN	optional; if the calibration is to use a proportional assignment; default = false (proportionalAssignment in Calibrator)
-REGRINERTIA	optional; regression inertia; default = 0.95 (regressionInertia in Calibrator)
-RNDSEED	optional; random seed; default = 0 (randomSeed in Calibrator)
-STATSFILE	optional; name of file where statistics are written; default = calibration-stats.txt (statisticsFile in Calibrator)
-VARSCALE	optional; scales measurement variance; default = 1.0 (varianceScale in Calibrator)

Table 3: `CHOICE` parameters of the `FileBasedController`

keyword	explanation (corresponding Cadyts variable)
`-CHOICEFILE`	required; fine where the choices are to be written; no default value (no Cadyts variable)
`-CHOICESETFILE`	required; comma-separated list of files that contain the choice sets; no default value (no Cadyts variable)

Table 4: `UPDATE` parameters of the `FileBasedController`

keyword	explanation (corresponding Cadyts variable)
`-FLOWFILE`	optional; file in which to write comparisons of measured and simulated flows; no default value (`flowAnalysisFile` in `Calibrator`)
`-NETFILE`	required; file that contains the network conditions; no default value (no Cadyts variable)

```
protected void prepareCalibratorUPDATE(C
calibrator, CommandLineParser clp)
```

need to be overridden in order to communicate the new parameters contained in the `CommandLineParser` to the calibrator.

A Version history

Version 1.1.0

- calibration of continuous-valued demand parameters from traffic counts
- calibration from vehicle re-identification data
- default measurement XML loader
- default population XML loader
- file-based interface to DRACULA microsimulation
- enhanced logging and reporting facilities

Version 1.0.1

- writing of statistics logfile for debugging and convergence checking
- test cases

Version 1.0.0

- first release

References

J.A. Bottom. *Consistent Anticipatory Route Guidance*. PhD thesis, Massachusetts Institute of Technology, 2000.

E. Cascetta. A stochastic process approach to the analysis of temporal dynamics in transportation networks. *Transportation Research Part B*, 23(1):1–17, 1989.

E. Cascetta and G.E. Cantarella. A day-to-day and within-day dynamic stochastic assignment model. *Transportation Research Part A*, 25(5):277–291, 1991.

DRACULA. DRACULA web site. http://www.its.leeds.ac.uk/software/dracula/index.html, accessed 2010.

G. Flötteröd. Cadyts – a free calibration tool for dynamic traffic simulations. In *Proceedings of the 9th Swiss Transport Research Conference*, Monte Verita/Ascona, Switzerland, September 2009.

G. Flötteröd. *Traffic State Estimation with Multi-Agent Simulations*. PhD thesis, Berlin Institute of Technology, Berlin, Germany, 2008.

G. Flötteröd and M. Bierlaire. Improved estimation of travel demand from traffic counts by a new linearization of the network loading map. In *Proceedings of the European Transport Conference*, Noordwijkerhout, The Netherlands, October 2009.

G. Flötteröd, M. Bierlaire, and K. Nagel. Bayesian demand calibration for dynamic traffic simulations. Technical Report TRANSP-OR 100105, Ecole Polytechnique Fédérale de Lausanne & Berlin Institute of Technology, 2010.

M. Fowler. *Refactoring. Improving the Design of Existing Code*. The Addison-Wesley Object Technology Series. Addison-Wesley, 1999.

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison–Wesley Professional Computing Series. Addison–Wesley, 1994.

R. Liu. The DRACULA dynamic traffic network microsimulation model. In R. Kitamura and M. Kuwahara, editors, *Simulation Approaches in Transportation Analysis: Recent Advances and Challenges*, pages 23–56. Springer, 2005.

K. Nagel and G. Flötteröd. Agent-based traffic assignment: going from trips to behavioral travelers. In *Proceedings of 12th International Conference on Travel Behaviour Research*, Jaipur, India, December 2009. Invited resource paper.

K. Nagel, M. Rickert, P.M. Simon, and M. Pieck. The dynamics of iterated transportation simulations. In *Proceedings of the 3rd Triennial Symposium on Transportation Analysis*, San Juan, Puerto Rico, 1998.

SUMO. SUMO web site. http://sumo.sourceforge.net, accessed 2010.