Robust Timetabling Documentation

Release 1.0

Anna Pessarrodona Marfà & Tim Seppelt

Jul 29, 2018

Contents

This package contains all modules which have been developed within the scientific computing pratical by Anna Pessarrodona Marfà <a.pessarrodonamarf@stud.uni-goettingen.de> and Tim Seppelt <t.seppelt@stud.uni-goettingen.de> in the summer semester 2018.

Please see the related report for further information on the background and the functionality.

The whole project is based on the LinTim framework which is developed at the Universität Göttingen.

The package is structured in the following way:

- *robtim* contains general functionality for example for input/output, e.g. `Dataset`
- *robtim.eval* contains all functionality related to the evaluation of robustness. This includes:
  - the `robtim.eval.RobustnessEvaluator` which manages the evaluation
  - several scenario generators which generate delay scenarios
  - several delay managers which take care of the delay management
  - severel statisticians which collect and analyze statistical data during the evaluation.
- *robtim.opt* contains all functionality related to the optimization of a timetable by robustness. This includes:
  - the `robtim.opt.RobustnessOptimizer` which manages the optimization
  - several EAN generators which generate event activitiy networks
  - several timetablers which compute timetables based on the EANs
  - several supervisors which provide a stop criterion and collect / analyze statistical data

CHAPTER 1

Input/Output

class `robtim.`**`Dataset`**(*name: str*, *path: str*, *env=None*, *silent: bool = False*)

Models a LinTim dataset and takes care of all input/output operations.

Example Dataset('toy','LinTim/datasets')

Parameters

- **`name`** – name of the dataset, i.e. name of the directory in which the dataset is stored
- **`path`** – path to the dataset
- **`env`** – system environment which should be used when communicating with LinTim
- **`silent`** – whether the entire LinTim output should be printed to stdout

`applyConfig`(*config: dict*, *unset: list = []*, *filename: str = 'basis/Private-Config.cnf'*)

Applies configuration to config files.

Parameters

- **`conf`** – a dict containing config keys and their values
- **`unset`** (`list`) – an optional list of config keys which should be removed from the file
- **`filename`** (`str`) – path to the config file, default 'basis/Private-Config.cnf'

`copy`(*origin: str*, *target: str*)

Copies a file in the dataset's directory

Parameters

- **`origin`** (`str`) – relative path of the file in the dataset's directory which should be copied
- **`target`** (`str`) – relative path of the target file in the dataset's directory

`delete`(*filename: str*)

Deletes the given file from the dataset's directory

Parameters **`filename`** (`str`) – relative path of the file in the dataset's directory

`make`(*target: str*)

Executes make on this dataset with the target specified. The custom system environment is used.

Outputs and errors are send to stdout. Verbose outputs are only written if the dataset was initialized with silent == False (default). If an error occurs an exception is raised.

Example make('tim-timetable')

Parameters **`target`** (`str`) – make target

Raises **`Exception`** – if the execution failed

`prepareFor` (*target: str*)

Prepares the dataset for executing the specified make target. This means that all targets in the following list are executed until the specified target is reached. The specified target will not be executed.

1. "lc-line-concept"
2. "ean"
3. "tim-timetable"
4. "ro-rollout"
5. "dm-disposition-timetable"

Example prepareFor('tim-timetable') will call make('lc-line-concept') and make('ean')

Parameters **`target`** (`str`) – target which the dataset should be prepared for

`readCSV` (*filename: str*, *columns: list = None*)

Reads a CSV file from the dataset's directory.

Parameters

- **`filename`** (`str`) – relative path
- **`columns`** (`list`) – list of indices of columns which should be stored. If columns == None all columns will be included

Returns list of rows. Every row is a list which contains the i-th column's value at the i-th position

`realPath` (*filename: str = ''*)

Returns the absolute path of a given file in the dataset's directory or of the dataset itself if filename == ""

Parameters **`filename`** (`str`) – relative path of the file

Returns absolute path of filename

`resetConfig` (*filename: str = 'basis/Private-Config.cnf'*)

Deletes the given config file

Parameters **`filename`** (`str`) – config file which should be deleted or "basis/Private-Config.cnf"

`statistics` (*filename: str = 'statistic.sta'*)

Reads a statistic file from LinTim. If no file name is specified 'statistics/statistics.sta' in the dataset directory will be read Returns are dict with strings, floats or booleans as values

Parameters **`filename`** (`str`) – filename of the statistical file or 'statistics/statistics.sta'

Returns dict of statistical values

class `robtim.`**`LinTimCSVWriter`** (*file*)

writes LinTim CSV files

Parameters **`file`** – file object, e.g. returned by open()

`escape` (*s: str*) → str

escapes values

Return type `str`

Parameters **s** (`str`) – value

write(*x*, *escape: bool = True*) → None

writes plain string or list of values which are then escaped according to CSV file definition

If x is of type str it will be written to the file directly. Otherwise x is assumed to be iterable and treated as a list of values in a row.

Set escape = False if you do not want all values to be escaped by *`escape()`*.

Parameters

- **x** – row or plain string
- **escape** (`bool`) – whether values should be escaped

CHAPTER 2

Evaluating Robustness

The *robtim.eval* package contains all functionality which is related to evaluating the robustness of a timetable. In order to provide a general framework which can be used for the large variety of robustness concepts and settings it has a modular design.

The main class is the `RobustnessEvaluator` which takes care of the general process and coordinates the work of the different components. These components are:

- scenario generators which generate a set of delay scenarios
- delay managers which compute disposition timetables based on the delays and the given timetable
- statisticians which collect and analyze statistical data about the timetables, e.g. in order to compute a robustness coefficient.

How these three components work must be specified by extending the classes `ScenarioGenerator`, `DelayManager` and `Statistician`.

Apart from the general classes this package provides several concrete implementations. Most interesting for the user is probably the `TreeOnTrackEvaluator` which combines the components used in the report.

2.1 The *RobustnessEvaluator*

class robtim.eval.**RobustnessEvaluator**(*scenarios: robtim.eval.scenario_generator.ScenarioGenerator = None*, *delayManager: robtim.eval.delay_manager.DelayManager = None*, *statistician: robtim.eval.statistician.Statistician = None*, *silent: bool = False*)

Evaluates the Robustness of the current timetable stored in the dataset.

This class assumes that ro-rollout (and all previous planning steps) was done.

Please call `evaluate()` do actually evaluate the timetable

Example RobustnessEvaluator().evaluate(dataset) will evaluate the robustness of the current timetable in the dataset which default scenario generator (ScAlbertU1), default delay manager (OrdinaryDelayManager) and default statistician (OrdinaryStatistician). Please see these classes for details.

Parameters

- **`scenario`** – instance of *`ScenarioGenerator`* which models the set of scenarios which should be taken into accunt
- **`delayManager`** – instance of *`DelayManager`* which models the delay management planning step
- **`statistician`** – instance of *`Statistician`* which takes care of the statistical evaluation of the collected data
- **`silent`** – whether there should be no additional messages about the progress

`evaluate`(*dataset: robtim.io.Dataset*)

Evaluates the robustness of the timetable in the dataset.

The following steps are done:

1. ScenarioGenerator, DelayManager, Statistican are initialized, e.g. the respective functions of the objects called
2. Statistical information about the nominal timetable is collected and handed over to statistician.collectNominal
3. **For all scenarios provided by *`ScenarioGenerator.scenarios()`* the following is done:**
   - delay management
   - statistical data is collected
4. After exhausting the scenario set the statistican is asked for its results which are then returned

Parameters **`dataset`** (`Dataset`) – dataset which the robustness of the timetable is evaluated on

Returns result of *`Statistician.interpret()`*

2.1.1 Implementations

Note: All classes in this section inherit from *`RobustnessEvaluator`* and have therefore the same functions. This documentation lists only the constructors and their parameters.

`class` `robtim.eval.`**`TreeOnTrackEvaluator`**(*p*, *beta*, *n*)

RobustnessEvaluator which uses the TreeOnTrackScenarioGenerator together with the CoefficientStatistician. It returns the average robustness coefficient of all delays in the sample.

Parameters

- **`p`** – TreeOnTrack's p parameter
- **`beta`** – TreeOnTrack's beta parameter
- **`n`** – number of iterations which should be used

2.2 The *ScenarioGenerator*

`class` `robtim.eval.`**`ScenarioGenerator`**

Abstract super class of all scenario generators. A scenario generator models a set of delay scenarios.

A scenario generator uses a python generator, implemented in *`scenarios()`* to yield delay scenarios. Be aware that a scenario generator may yield infinetly many delay scenarios. When using *`RobustnessEvaluator`* you have to take care of the termination yourself. Use for example *`ScenarioScheduler`*.

initialize (*dataset: robtim.io.Dataset*)

Initializes the generator. This is where configuration changes for LinTim components should be applied.

Parameters **dataset** (`Dataset`) – dataset which delays should be generated on

reset (*dataset: robtim.io.Dataset*)

Deletes all generated delays from the dataset. This affects the following files in the dataset's directory:

- delay-management/Delays-Activities.giv
- delay-management/Delays-Events.giv

Parameters **dataset** (`Dataset`) – dataset which delays should be generated on

scenarios (*dataset: robtim.io.Dataset*)

This function has to be a python generator which generates a new delay scenario and yields a dict of its properties as long as more delays are available.

The dict of properties should correspond to the parameters of LinTim component dm-delays.

Parameters **dataset** (`Dataset`) – dataset which delays should be generated on

2.2.1 Helper classes

Note: All classes in this section inherit from *`ScenarioGenerator`* and have therefore the same functions. This documentation lists only the constructors and their parameters.

class robtim.eval.**ScenarioScheduler** (**args*, *iterations: list = []*)

Schedules different scenario generators for successive execution. In this way it ensures that the evaluation terminates.

Example ScenarioScheduler(ScAlbertU1(4), iterations=[5]) models the scenario set constisting of 5 scenarios generated by ScAlbertU1 with parameter 4.

Parameters

- **args** – one or several scenario generators
- **iterations** – list of integers. The i-th value states how many scenarios from the i-th scenario generator in args should be taken

class robtim.eval.**ConfigurableScenarioGenerator** (*config*)

Generates delay scenarios using LinTim's dm-delays component. The given configuration is used.

Be aware that this generator will generate infinetly many delays. Use *`ScenarioScheduler`* to guarantee that your evaluation terminates.

Parameters **config** – configuration for LinTim's dm-delays

class robtim.eval.**DistributionScenarioGenerator** (*randomizer*, *count: int = 10*, *count_abs: bool = True*, *events: bool = False*, *activities: bool = True*, *info: dict = None*)

Generates delay scenarios with a custom distribution

In addition to the regular fields described in *`ScenarioGenerator`* this generator yields for every scenario the following values:

- *delays_total* sum of all delays on activities and events in seconds
- *delays_total_weighted* sum of all delays weighted by the amount of passengers on the activities / events in seconds

- *delays_total_passengers* total amount of passengers affected by the delays, i.e. number of passengers on activities / events with delays
- *delays_average = delays_total / delays_total_passengers*
- *delays_average_weighted = delays_total_weighted / delays_total_passengers*

Parameters

- **`randomizer`** – function int -> list[int] which returns a given amount of random delays
- **`count`** – number/percantage of events/activites which shall be delayed
- **`count_abs`** – whether count is absolute (True) or relative (False)
- **`events`** – whether events shall be delayed
- **`activities`** – whether activities shall be delays
- **`info`** – info about the distribution for statistical evaluation

2.2.2 Implementation

Note: All classes in this section inherit from `ScenarioGenerator` or from classes in the prior section and have therefore the same functions. This documentation lists only the constructors and their parameters.

Warning: Note that all scenario generators in this section yield infinetly many delay scenarios. Use `ScenarioScheduler` to ensure that the evaluation terminates.

class robtim.eval.**TreeOnTrackScenarioGenerator**(*binomial_p: float, exp_beta: float, events: bool = False, activities: bool = True*)

Generates delay scenarios with tree-on-track distribution. The amount of trees, e.g. the amount of activities/events is binomial distributed with parameters lambda and stretch. The delay each tree causes is exponentially distributed with parameter beta.

Parameters

- **`binomial_p`** – binomial p parameter in [0,1]
- **`exp_beta`** – exponential beta parameter
- **`events`** – whether events shall be delayed
- **`activities`** – whether activities shall be delays

class robtim.eval.**NormalDistScenarioGenerator**(*mean: float, deviation: float, count: int = 10, count_abs: bool = True, events: bool = False, activities: bool = True*)

Generates normal distributed delay scenarios

Formula: numpy.random.normal(mean, deviation)

Parameters

- **`mean`** – mean
- **`deviation`** – standard deviation
- **`count`** – number/percantage of events/activites which shall be delayed
- **`count_abs`** – whether count is absolute (True) or relative (False)
- **`events`** – whether events shall be delayed

- **activities** – whether activities shall be delays

class robtim.eval.**PoissonDistScenarioGenerator**(*lam: float*, *stretch: float = 1*, *count: int = 10*, *count_abs: bool = True*, *events: bool = False*, *activities: bool = True*)

Generates poisson distributed delay scenarios

Formula: stretch * numpy.random.poisson(lam)

Parameters

- **lam** – poisson distribution's lambda parameter
- **stretch** – stretch parameter
- **count** – number/percantage of events/activites which shall be delayed
- **count_abs** – whether count is absolute (True) or relative (False)
- **events** – whether events shall be delayed
- **activities** – whether activities shall be delays

class robtim.eval.**ScAlbertU1**(*s*, *smin=0*)

Scenario generator for the scenario set U_1 defined in Bachelor thesis by Albert

Parameters

- **s** – max source delay
- **smin** – (optional) min source delays

class robtim.eval.**ScAlbertU2**(*s*, *k*, *smin=0*)

Scenario generator for the scenario set U_1 defined in Bachelor thesis by Albert

Parameters

- **s** – max source delay
- **k** – max amount of activities delayed
- **smin** – (optional) min source delays

2.3 The *DelayManager*

class robtim.eval.**DelayManager**

Abstract super class of all delay managers. A delay manager takes care of the planning step dm-delay-management and provides functionality similar to the functionality of the related LinTim component.

dispositionTimetable(*dataset: robtim.io.Dataset*)

Function which is called by the evaluator when a disposition timetable is required.

Parameters **dataset** (Dataset) – Dataset which the delay management is done on

initialize(*dataset: robtim.io.Dataset*)

Initializes the delay manager. This is where the configuration of the used LinTim components should be done.

Parameters **dataset** (Dataset) – Dataset which the delay management is done on

2.3.1 Implementations

Note: All classes in this section inherit from *DelayManager* and have therefore the same functions. This documentation lists only the constructors and their parameters.

class `robtim.eval.`**`ConfigurableDelayManager`**(*config: dict*)

Simple implementation of a *`DelayManager`*. It calls the LinTim make target dm-disposition-timtable with the given configuration parameters

Parameters **`config`** – dict of configuration paramters for LinTim

class `robtim.eval.`**`OrdinaryDelayManager`**(*method: str = 'propagate'*)

Simplifies the *`ConfigurableDelayManager`* by allowing only certain configuration parameters.

Parameters **`method`** – value for LinTim's DM_method paramter

2.4 The *Statistician*

class `robtim.eval.`**`Statistician`**

Abstract super class of all statisticians. A statistician collects, stores and interprets statistical data about delays and their effects.

`collectNominal`(*dataset: robtim.io.Dataset*)

Collects data about the nominal timetable, i.e. without delays.

Parameters **`dataset`** (`Dataset`) – dataset which statistics should be done on

`collectStats`(*dataset: robtim.io.Dataset*, *scenario: dict*)

Collects data about a disposition timetable, i.e. with delays.

Parameters

- **`dataset`** (`Dataset`) – dataset which statistics should be done on
- **`scenario`** (`dict`) – dict with information about the delay scenario provided by the scenario generator, i.e. an instance of *`ScenarioGenerator`*.

`initialize`(*dataset: robtim.io.Dataset*)

Initializes the statistician. This is where configuration changes for LinTim components should be applied.

Parameters **`dataset`** (`Dataset`) – dataset which statistics should be done on

`interpret`(*dataset: robtim.io.Dataset*)

Interprets the data and returns the final result of the analysis.

Parameters **`dataset`** (`Dataset`) – dataset which statistics should be done on

2.4.1 Implementations

Note: All classes in this section inherit from *`Statistician`* and have therefore the same functions. This documentation lists only the constructors and their parameters.

class `robtim.eval.`**`OrdinaryStatistician`**(*extended: bool = False*)

Statistician which stupidly stores everything what it gets.

The result is a tuple (nominal, delayed) where

- nominal is a dict of information about the nominal timetable
- delayed is a list of tuples where the first entry is the dict of scenario info and the second entry is the dict of evaluation info

Parameters **`extended`** – whether extended info should be included (DM_eval_extended)

class robtim.eval.**MatrixStatistician**(*keysScenario: list*, *keysEvaluation: list*, *extended: bool = False*)

Statistician which filters the data and strores it in a matrix.

The result is a tuple (nominal, delayed) where

- nominal is a list of the selected info about the nominal timetable
- delayed is a list of lists of the selected info about the delayed timetables

Parameters

- **keysScenario** – properties of the scenarios which should be included
- **keysEvaluation** – properties of the evaluations which should be included
- **extended** – whether extended info should be included (DM_eval_extended)

class robtim.eval.**CoefficientStatistician**(*keysEvaluation: list*, *extended: bool = False*)

Statistician which computes the averages of a evaluation keys and devides them by the corresponding values for the nominal timetable.

Parameters

- **keysEvaluation** – list of evaluation keys which should be used
- **extended** – use extended evaluation

CHAPTER 3

Optimizing Robustness

The *robtim.opt* package contains all functionality which is related to the optimization of a timetable by robustness. In order to provide a general framework which can be used for the large variety of robustness concepts and settings it has a modular design.

The main class is the `RobustnessOptimizer` which takes care of the general process and coordinates the work of the different components. These components are:

- EAN generators compute new, e.g. more robust, event activity networks
- timetablers compute periodic and aperiodic timetables based on the EAN
- supervisors provide a stop criterion and collect and analyze statistical data about the optimization.

How these three components work must be specified by extending the classes `EANGenerator`, `Timetabler` and `Supervisor`.

3.1 The *RobustnessOptimizer*

class robtim.opt.**RobustnessOptimizer**(*ean_generator: robtim.opt.ean_generator.EANGenerator, supervisor: robtim.opt.supervisor.Supervisor, timetabler: robtim.opt.timetabler.Timetabler = <robtim.opt.timetabler.DefaultTimetabler object>, silent: bool = False*)

Optimizes the robustness of a timetable of a dataset.

It is assumed that lc-line-concept and all previous planning steps have been done.

To run the optimization please call `optimize()`.

Parameters

- **ean_generator** – EANGenerator which generates new (more robust) EANs
- **supervisor** – Supervisor which takes care of the interruption of the (possibly infinite) optimization and stores statistical data.
- **silent** – whether there should be additional output or not

optimize(*dataset: robtim.io.Dataset*)

Optimizes the EAN in dataset for robustness.

The following steps are done:

1. EANGenerator, Supervisor and Timetabler are initialized, i.e. the the respective functions on the objects are called.
2. For all EANs provided by *`EANGenerator.eans()`* the following steps are done:

- timetabling, i.e. *`Timetabler.timetable()`*, which computes a periodic and aperiodic timetable for the current EAN.
- statistical data is collected by the supervisor.
- the supervisor is asked whether the optimization should be interrupted, i.e. if the objective is reached. See *`Supervisor.interrupt()`*.

3. Information about the final EAN and the report of the supervisor are returned to the callee.

Parameters **`dataset`** (`Dataset`) – dataset which the optimization should be done on

Returns tuple (ean, report) where ean is a dictionary with information about the final EAN and report the result of *`Supervisor.report()`*.

3.2 The *EANGenerator*

`class` `robtim.opt.`**`EANGenerator`**

Abstract super class of all EANGenerators. An EANGenerator generates EANs with the objective of optimizing their robustness.

`eans`(*dataset: robtim.io.Dataset*)

Python generator for EANs. This function must for every iteration write an EAN into the respective files in the dataset and yield a dict with the following information:

- *activitiesModified*: the number of modified activities
- *activitiesModifiedWeighted*: the number of modified activities weighted by the amount passengers on the activity
- *slackAdded*: the number of minutes added to the lower bounds of the activities
- *slackAddedWeighted*: the number of minutes added to the lower bounds of the activities weighted by the amount of passengers on the activity

Implementations of this function must ensure this exact order. All values are relative to the dataset's EAN at the point this function is called.

Note that in contrary to the *`robtim.eval.ScenarioGenerator`* an EANGenerator must not take care of the termination of the optimization. This function can or cannot yield infinitely many EANs. The interruption of the process is done by the *`Supervisor`*.

Parameters **`dataset`** (`Dataset`) – Dataset which the EANs are generated for

`initialize`(*dataset: robtim.io.Dataset*) → None

Initialize the EANGenerator. This is where the configuration of LinTim components should take place.

Parameters **`dataset`** (`Dataset`) – Dataset which the EANs are generated for

3.2.1 Helper classes

Note: All classes in this section inherit from *`EANGenerator`* and have therefore the same functions. This documentation lists only the constructors and their parameters.

class robtim.opt.**IncrementalSlackEANGenerator**

EANGenerator which increased the slack time incrementally by calling its function distribute in every loop for every activity.

The behaviour of the generator depends on the implementation of `distribute()` which must be overwritten by inheriting classes. In every iteration and for every activity this function is called to determine the new lower and upper bounds.

distribute(*dataset: robtim.io.Dataset*, *activity_index: int*, *activity_type: str*, *from_event: int*, *to_event: int*, *lower_bound: int*, *upper_bound: int*, *passengers: int*)

Decides how much slack an activity receives. This function is called in every iteration and for every activitiy.

Parameters

- **dataset** (Dataset) – dataset which the optimization is done on
- **activity_index** (int) – index of the activity in the respective file
- **activity_type** (str) – type of the activity, e.g. *drive*, *wait* etc.
- **from_event** (int) – id of the starting event
- **to_event** (int) – id of the end event
- **lower_bound** (int) – current lower bound
- **upper_bound** (int) – current upper bound
- **passengers** (int) – amount of passengers on the activitiy

Returns tuple (lower_bound, upper_bound) with the new bounds

3.2.2 Implementations

Note: All classes in this section inherit from *EANGenerator* and have therefore the same functions. This documentation lists only the constructors and their parameters.

class robtim.opt.**IncreasingSlackEANGenerator**(*step: int*, *increaseUpperBounds: bool = False*, *activitiyTypes: list = None*)

EANGenerator which increases the slack of all activities by a given number of minutes (step). If increaseUpperBounds == False, the generator will stop making changes after the upper bounds are reached. Otherwise it will increase lower and upper bounds simultaneously.

Parameters

- **step** – minutes of slack which should be added to the lower bounds in every loop.
- **increaseUpperBounds** – whether upper bound should be increased as well.
- **activitiyTypes** – list of activities, e.g. "drive", "wait", or None if all types of activities should be affected

class robtim.opt.**GiveTheRichEANGenerator**(*totalSlack: int, step: int = 1, stepAbsolute: bool = True, activitiyTypes: list = ['drive', 'wait']*)

EANGenerator which instead of increasing the slack re-distributes it in every iteration. The total amount of slack which is added to the EAN ist kept constant. The following procedure is used:

- sort all activities by their number of passengers in reverse order
- **iterate n = number of non-zero activities to 0 with given step:**
  - add slack time to the first n most important activities
  - the amount of slack which an activity receives equals its ratio of the number of passengers

In addition to the EAN properties which are defined by *EANGenerator* this generator yields the following information

- *ratioRich*: ratio of the non-zero activities which are considered rich, i.e. which received slack, value in [0,1]

  Parameters

  - **totalSlack** – total amount of slack which is then redistributed (in min)
  - **step** – step which the amount of rich activities is reduced by in every iteration
  - **stepAbsolute** – whether the step is an absolute number, if not the absolute step is computed by (number of non-zero acitivities)*step//100
  - **activityTypes** – list of acitivity types, e.g. *drive*, *wait* etc. which may receive slack, default: *drive* and *wait*.

class robtim.opt.**GiveTheRichWeightedEANGenerator**(*initalSlackPerPassenger: float, step: int = 1, stepAbsolute: bool = True, activitiyTypes: list = ['drive', 'wait']*)

EANGenerator which instead of increasing the slack re-distributes it in every iteration. The *weighted* total amount of slack which is added to the EAN ist kept constant. The following procedure is used:

- sort all activities by their number of passengers in reverse order
- **iterate n = number of non-zero activities to 0 with given step:**
  - add slack time to the first n most important activities
  - the amount of slack which an activity receives equals its ratio of the number of passengers

In addition to the EAN properties which are defined by *EANGenerator* this generator yields the following information

- *ratioRich*: ratio of the non-zero activities which are considered rich, i.e. which received slack, value in [0,1]
- *slackPerPassenger*: slack which an activitiy got for one passenger

  Parameters

  - **totalSlack** – total amount of slack which is then redistributed (in min)
  - **step** – step which the amount of rich activities is reduced by in every iteration
  - **stepAbsolute** – whether the step is an absolute number, if not the absolute step is computed by (number of non-zero acitivities)*step//100
  - **activityTypes** – list of acitivity types, e.g. *drive*, *wait* etc. which may receive slack, default: *drive* and *wait*.

3.3 The *Timetabler*

class robtim.opt.**Timetabler**

Abstract super class of all timetablers. A timetabler should take care of the planning steps tim-timetable and ro-rollout and provide functionality similar to the functionality of the related LinTim make target.

initialize(*dataset: robtim.io.Dataset*)

Initializes the timetabler. This is where the configuration of the used LinTim components should be done.

Parameters dataset (Dataset) – Dataset which the timetabling is done on

timetable (*dataset: robtim.io.Dataset*)
Function which is called by the optimizer when a timetable is required.

Parameters **dataset** (`Dataset`) – Dataset which the timetabling is done on

3.3.1 Implementations

Note: All classes in this section inherit from *`Timetabler`* and have therefore the same functions. This documentation lists only the constructors and their parameters.

class `robtim.opt.`**ConfigurableTimetabler** (*config: dict*)
Simple implementation of a *`Timetabler`*. It calls the LinTim make target tim-timetable and ro-rollout with the given configuration parameters

Parameters **config** – dict of configuration paramters for LinTim

class `robtim.opt.`**DefaultTimetabler** (*model: str = 'MATCH'*)
Simplifies the *`ConfigurableTimetabler`* by allowing only certain configuration parameters.

Parameters **model** – value for LinTim's tim_model paramter

3.4 The *Supervisor*

class `robtim.opt.`**Supervisor**
Abstract super class of all supervisors.

A supervisor takes care of interruption an optimization if the objective is reached and of collecting statistical information.

initialize (*dataset: robtim.io.Dataset*)
Initializes the supervisor. This is when LinTim components should be configured.

Parameters **dataset** (`Dataset`) – dataset which the optimization is done on

interrupt (*dataset: robtim.io.Dataset, iteration: int, ean: dict*) → bool
This function is called in every loop of the optimization and should do two things:

- store statistical information about the optimization
- interrupt the optimization when the objective is reached

Return type `bool`

Parameters

- **dataset** (`Dataset`) – dataset which the optimization is done on
- **iteration** (`int`) – number of the current iteration counting from 1
- **ean** (`dict`) – dict with information about the current EAN, provided by *`EANGenerator.eans()`*

Returns True, if the optimization should be interrupted, False otherwise

report (*dataset: robtim.io.Dataset, iteration: int, ean: dict*)
Returns the collected statistical information at the end of the optimization.

Parameters

- **dataset** (`Dataset`) – dataset which the optimization is done on
- **iteration** (`int`) – number of the current iteration counting from 1

- **ean** (dict) – dict with information about the current ean, provided by *EANGenerator.eans()*

Returns info about the optimization

3.4.1 Implementations

Note: All classes in this section inherit from *Supervisor* and have therefore the same functions. This documentation lists only the constructors and their parameters.

class robtim.opt.**DefaultSupervisor**(*n: int*)

Supervisor which only looks at the number of iterations and stops at a certain point

Parameters n – number of iterations which should be done

class robtim.opt.**MatrixRobustnessSupervisor**(*rob: robtim.eval.evaluator.RobustnessEvaluator*, *condition=False*, *n: int = -1*)

Supervisor which uses robustness evaluation for interrupting the optimization. It supports the following stop criteria:

- a function which checks if the current result is good enough
- a fixed maximal number of iterations.

Data is stored in a matrix where every row stands for an iteration of the RobustnessEvaluator. In every row the following data is stored in this order:

1. info about the EAN, provided by *EANGenerator.eans()*
2. info about the nominal timetable, provided by *robtim.eval.RobustnessEvaluator*
3. info about the delayed timetable, provided by *robtim.eval.RobustnessEvaluator*

This function expects the robustness evaluator to return a tuple (nominal, delayed) where delayed is a matrix where every row represents one iteration of the evaluator. See *robtim.eval.MatrixStatistician* for a statistian which works in this way.

If you use *robtim.eval.MatrixStatistician* the order of columns in the returned matrix is as follows:

1. info about the EAN
2. info about the nominal timetable
3. info about the delay scenario
4. info about the delayed timetable

Example

```
evaluator = RobustnessEvaluator(
scenarios = ScenarioScheduler(TreeOnTrackScenarioGenerator(0.8, 3000),␣
↪iterations=[evalIterations]),
statistician = MatrixStatistician([], ["dm_time_average"])
)

opt = RobustnessOptimizer(
        IncreasingSlackEANGenerator(5, True, ["change"]),
        MatrixRobustnessSupervisor(evaluator, n = 15)
    )

ean, report = opt.optimize(d)
```

Parameters

- **rob** (*robtim.eval.RobustnessEvaluator*) – RobustnessEvaluator
- **condition** – function which receives the tuple (nominal, delayed), i.e. the result of the RobustnessEvaluator and returns True if the optimization should be interrupted, default is False, which means that this criterion will not be used
- **n** – maximal amount of iterations or -1 if this criterion should not be used

class robtim.opt.**RobustnessSupervisor**(*rob: robtim.eval.evaluator.RobustnessEvaluator*, *condition=False*, *n: int = -1*)

Supervisor which works basically like *MatrixRobustnessSupervisor* but does not make any assumptions about the format of the evaluation result.

The result is matrix where every row represents one iteration of the optimization. Every row contains firstly information about the EAN and secondly the result of the robustness evaluation.

Parameters

- **rob** (*robtim.eval.RobustnessEvaluator*) – RobustnessEvaluator
- **condition** – function which receives the tuple (nominal, delayed), i.e. the result of the RobustnessEvaluator and returns True if the optimization should be interrupted, default is False, which means that this criterion will not be used
- **n** – maximal amount of iterations or -1 if this criterion should not be used

CHAPTER 4

Examples

4.1 Setting up a dataset

First one needs to set up a `robtim.Dataset` object. It takes care of the communication with LinTim. In this example we have moved most configuration parameters to a special file *config.py*. The following things are defined there:

Execution environment

In order to run *make* properly one has to specify the execution environment of the shell. This is especially necessary if you want to use third-party solvers like Gurobi or Xpress.

```
import sys, os, getpass

user = getpass.getuser()

# Execution environment for make
env = os.environ.copy()
env["GUROBI_HOME"] = "/home/"+ user +"/gurobi800/linux64/"
env["PATH"] = env["PATH"] + ":" + env["GUROBI_HOME"]
env["LD_LIBRARY_PATH"] = env["GUROBI_HOME"] + "lib/"
env["GRB_LICENSE_FILE"] = "/home/"+ user +"/gurobi.lic"
env["CLASSPATH"] = env["GUROBI_HOME"] + "/lib/gurobi.jar"
```

Configuration for LinTim

LinTim comes with many configuration keys. A default configuration may look like this:

```
# Config for LinTim components
config = {
    "lc_solver": "GUROBI",
    "DM_solver": "Gurobi",
    "DM_method" : "propagate",
    "tim_solver": "gurobi",
    }
```

Path to the datasets

The last thing which needs to be specified is the path to the datasets, e.g.

```
# Path to LinTim's dataset directory
path = "/home/"+ user +"/LinTim/datasets"
```

Initialize dataset

Now a dataset object can be created in the main file. To reset the configuration (`robtim.Dataset.resetConfig()`) may not be necessary in most cases.

```
import config
from robtim import Dataset as Dataset

d = Dataset("bahn-01", config.path, config.env, silent=True)
d.resetConfig()
d.applyConfig(config.config)
```

4.2 Evaluating Robustness

For evaluating the robustness of a timetable one needs an instance of an `robtim.eval.RobustnessEvaluator` object. It comes with several parameters which are described in the documentation of this class.

The following code sets up an RobustnessEvaluator with the TreeOnTrack model, i.e. `robtim.eval.TreeOnTrackScenarioGenerator`, and a `robtim.eval.MatrixStatistician` which collects the field *dm_time_average* from the LinTim component *dm-disposition-timetable-evaluate* in every iteration. `robtim.eval.ScenarioScheduler` takes care of the amount of iterations. The scenario generator would otherwise yield infinitely many delay scenarios.

```
evalIterations = 5

evaluator = RobustnessEvaluator(
        scenarios = ScenarioScheduler(
                TreeOnTrackScenarioGenerator(0.8, 3000),
                iterations=[evalIterations]),
        statistician = MatrixStatistician([], ["dm_time_average"])
```

Now the robustness can be evaluated using

```
nominal, delayed = evaluator.evaluate(d)
```

nominal contains the value of *dm_time_average* for the nominal timetable. delayed is a matrix with rows for every delay scenario. Each row contains the value of *dm_time_average* for the disposition timetable

4.3 Optimizing Robustness

For optimizing the timetable with respect to the robustness one neeeds an instance of a `robtim.opt.RobustnessOptimizer` object. It comes with several parameters which are described in the documentation of this class.

The following code sets up an RobustnessOptimizer with `robtim.opt.IncreasingSlackEANGenerator`. This EANGenerator increases in this example the slack of all activities by 5 min in every iteration (first parameter), increases the lower bounds as well as the upper bound (second paramter: True) and modifies only the *change* activities (third parameter). A `robtim.opt.MatrixRobustnessSupervisor` is used to provide a stop criterion and collect statistical data. This supervisor will stop after 15 iterations and uses the RobustnessEvaluator object from the last section to compute the robustness of the timetables.

```
opt = RobustnessOptimizer(
        IncreasingSlackEANGenerator(5, True, ["change"]),
        MatrixRobustnessSupervisor(evaluator, n = 15)
    )

ean, report = opt.optimize(d)
```

After executing these lines, ean contains a dict with information about the final ean, see *robtim.opt.EANGenerator.eans()*, and report contains a matrix provided by the supervisor with the robustness information from every iteration, see *robtim.opt.Supervisor.report()*.

Python Module Index

Index