rmf_traffic

Release 1.0.0

Open Source Robotics Corporation

Mar 20, 2022

CONTENTS:

A package for managing traffic in OpenRMF.

CHAPTER

ONE

RMF_TRAFFIC API

1.1 File Hierarchy

1.2 Full API

1.2.1 Namespaces

Namespace rmf_traffic

Contents

Namespaces

Classes

Functions

Typedefs

Namespace rmf_traffic::@19

Namespace rmf_traffic::agv

Contents

Classes

- *Class Graph::Waypoint*
- *Class Interpolate*
- *Class Interpolate::Options*
- *Class invalid_traits_error*
- *Class LaneClosure*
- *Class NegotiatingRouteValidator*
- *Class NegotiatingRouteValidator::Generator*
- *Class Plan*
- *Class Plan::Waypoint*
- *Class Planner*
- *Class Planner::Configuration*
- *Class Planner::Debug*
- *Class Debug::Progress*
- *Class Planner::Goal*
- *Class Planner::Options*
- *Class Planner::Result*
- *Class Planner::Start*
- *Class Rollout*
- *Class RouteValidator*
- *Class ScheduleRouteValidator*
- *Class SimpleNegotiator*
- *Class SimpleNegotiator::Debug*
- *Class SimpleNegotiator::Options*
- *Class VehicleTraits*
- *Class VehicleTraits::Differential*
- *Class VehicleTraits::Holonomic*
- *Class VehicleTraits::Limits*

Functions

- *Function rmf_traffic::agv::compute_plan_starts*
- *Function rmf_traffic::agv::interpolate_time_along_quadratic_straight_line*

Namespace rmf_traffic::blockade

Contents

Classes

Functions

Typedefs

Namespace rmf_traffic::debug

Contents

Classes

Namespace rmf_traffic::detail

Contents

Classes

Namespace rmf_traffic::geometry

Contents

Classes

Functions

- *Template Function rmf_traffic::geometry::make_final(Args&&...)*
- *Template Function rmf_traffic::geometry::make_final(const T&)*
- *Template Function rmf_traffic::geometry::make_final_convex(Args&&...)*
- *Template Function rmf_traffic::geometry::make_final_convex(const T&)*
- *Function rmf_traffic::geometry::operator!=(const Circle&, const Circle&)*
- *Function rmf_traffic::geometry::operator!=(const Space&, const Space&)*
- *Function rmf_traffic::geometry::operator==(const Circle&, const Circle&)*
- *Function rmf_traffic::geometry::operator==(const Space&, const Space&)*

Typedefs

- *Typedef rmf_traffic::geometry::ConstConvexShapePtr*
- *Typedef rmf_traffic::geometry::ConstFinalConvexShapePtr*
- *Typedef rmf_traffic::geometry::ConstFinalShapePtr*
- *Typedef rmf_traffic::geometry::ConstShapePtr*
- *Typedef rmf_traffic::geometry::ConvexShapePtr*
- *Typedef rmf_traffic::geometry::FinalConvexShapePtr*
- *Typedef rmf_traffic::geometry::FinalShapePtr*
- *Typedef rmf_traffic::geometry::ShapePtr*

Namespace rmf_traffic::internal

Namespace rmf_traffic::schedule

Contents

Classes

- *Struct Add::Item*
- *Struct Inconsistencies::Element*
- *Struct Ranges::Range*
- *Template Struct Negotiation::SearchResult*
- *Struct Negotiation::Submission*
- *Struct Negotiation::VersionedKey*
- *Struct Rectifier::Range*
- *Struct View::Element*
- *Class Change*
- *Class Change::Add*
- *Class Change::Cull*
- *Class Change::Delay*
- *Class Change::Erase*
- *Class Change::Progress*
- *Class Change::RegisterParticipant*
- *Class Change::UnregisterParticipant*
- *Class Change::UpdateParticipantInfo*
- *Class Database*
- *Class DatabaseRectificationRequesterFactory*
- *Class Inconsistencies*
- *Class Inconsistencies::Ranges*
- *Class ItineraryViewer*
- *Class ItineraryViewer::DependencySubscription*
- *Class Mirror*
- *Class Negotiation*
- *Class Negotiation::Evaluator*
- *Class Negotiation::Table*
- *Class Table::Viewer*
- *Class Viewer::Endpoint*
- *Class Negotiator*
- *Class Negotiator::Responder*
- *Class ParticipantDescription*
- *Class Patch*
- *Class Patch::Participant*
- *Class Query*

- *Class Query::Participants*
- *Class Participants::All*
- *Class Participants::Exclude*
- *Class Participants::Include*
- *Class Query::Spacetime*
- *Class Spacetime::All*
- *Class Spacetime::Regions*
- *Class Spacetime::Timespan*
- *Class QuickestFinishEvaluator*
- *Class RectificationRequester*
- *Class RectificationRequesterFactory*
- *Class Rectifier*
- *Class SimpleResponder*
- *Class Snappable*
- *Class Snapshot*
- *Class StubbornNegotiator*
- *Class Viewer*
- *Class Viewer::View*
- *Class Writer*
- *Class Writer::Registration*

Functions

- *Function rmf_traffic::schedule::make_query(std::vector<Region>)*
- *Function rmf_traffic::schedule::make_query(std::vector<std::string>, const Time *, const Time *)*
- *Function rmf_traffic::schedule::operator!=*
- *Function rmf_traffic::schedule::operator==*
- *Function rmf_traffic::schedule::query_all*

Typedefs

- *Typedef rmf_traffic::schedule::Itinerary*
- *Typedef rmf_traffic::schedule::ItineraryVersion*
- *Typedef rmf_traffic::schedule::ItineraryView*
- *Typedef rmf_traffic::schedule::ParticipantDescriptionsMap*
- *Typedef rmf_traffic::schedule::ParticipantId*
- *Typedef rmf_traffic::schedule::ProgressVersion*

- *Typedef rmf_traffic::schedule::StorageId*
- *Typedef rmf_traffic::schedule::Version*

Namespace rmf_traffic::time

Contents

- *Functions*

Functions

- *Function rmf_traffic::time::apply_offset*
- *Function rmf_traffic::time::from_seconds*
- *Function rmf_traffic::time::to_seconds*

Namespace std

Contents

- *Classes*

Classes

- *Template Struct hash< rmf_traffic::agv::LaneClosure >*

1.2.2 Classes and Structs

Struct Plan::Checkpoint

- Defined in file_latest_rmf_traffic_include_rmf_traffic_agv_Planner.hpp

Nested Relationships

This struct is a nested type of *Class Plan*.

Struct Documentation

struct rmf_traffic::agv::*Plan*::**Checkpoint**

Public Members

RouteId **route_id**

CheckpointId **checkpoint_id**

Struct Plan::Progress

- Defined in file_latest_rmf_traffic_include_rmf_traffic_agv_Planner.hpp

Nested Relationships

This struct is a nested type of *Class Plan*.

Struct Documentation

struct rmf_traffic::agv::*Plan*::**Progress**

Public Members

std::size_t **graph_index**

Checkpoints **checkpoints**

rmf_traffic::*Time* **time**

Struct Debug::Node

- Defined in file_latest_rmf_traffic_include_rmf_traffic_agv_debug_debug_Planner.hpp

Nested Relationships

This struct is a nested type of *Class Planner::Debug*.

Nested Types

- *Struct Node::Compare*

Struct Documentation

struct rmf_traffic::agv::*Planner*::*Debug*::**Node**

A *Node* in the planning search. A final Planning solution will be a chain of these Nodes, aggregated into a *Plan* data structure.

Public Types

using SearchQueue = std::priority_queue<*ConstNodePtr*, std::vector<*ConstNodePtr*>, *Compare*>

using Vector = std::vector<*ConstNodePtr*>

Public Members

ConstNodePtr **parent**

The parent of this *Node*. If this is a nullptr, then this was a starting node.

std::vector<*Route*> **route_from_parent**

The route that goes from the parent *Node* to this *Node*.

double **remaining_cost_estimate**

An estimate of the remaining cost, based on the heuristic.

double **current_cost**

The actual cost that has accumulated on the way to this *Node*.

rmf_utils::optional<std::size_t> **waypoint**

The waypoint that this *Node* stops on.

double **orientation**

The orientation that this *Node* ends with.

agv::*Graph*::*Lane*::*EventPtr* **event**

A pointer to an event that occured on the way to this *Node*.

rmf_utils::optional<std::size_t> **start_set_index**

If this is a starting node, then this will be the index.

std::size_t **id**

A unique ID that sticks with this node for its entire lifetime. This will also (roughly) reflect the order of node creation.

struct Compare

Public Functions

inline bool **operator()**(**const** *ConstNodePtr* &*a*, **const** *ConstNodePtr* &*b*)

Struct Node::Compare

- Defined in file_latest_rmf_traffic_include_rmf_traffic_agv_debug_debug_Planner.hpp

Nested Relationships

This struct is a nested type of *Struct Debug::Node*.

Struct Documentation

struct rmf_traffic::agv::*Planner*::*Debug*::*Node*::**Compare**

Public Functions

inline bool **operator()**(**const** *ConstNodePtr* &*a*, **const** *ConstNodePtr* &*b*)

Struct RouteValidator::Conflict

- Defined in file_latest_rmf_traffic_include_rmf_traffic_agv_RouteValidator.hpp

Nested Relationships

This struct is a nested type of *Class RouteValidator*.

Struct Documentation

struct rmf_traffic::agv::*RouteValidator*::**Conflict**

Public Members

Dependency **dependency**

Time **time**

std::shared_ptr<**const** rmf_traffic::*Route*> **route**

Struct TimeVelocity

- Defined in file_latest_rmf_traffic_include_rmf_traffic_agv_Interpolate.hpp

Struct Documentation

struct rmf_traffic::agv::**TimeVelocity**

Public Members

Time **time**

Eigen::Vector2d **velocity**

Struct ReservedRange

- Defined in file_latest_rmf_traffic_include_rmf_traffic_blockade_Status.hpp

Struct Documentation

struct rmf_traffic::blockade::**ReservedRange**

Public Functions

inline bool **operator==**(**const** *ReservedRange* &*other*) **const**

Public Members

std::size_t **begin**

std::size_t **end**

Struct Status

- Defined in file_latest_rmf_traffic_include_rmf_traffic_blockade_Status.hpp

Struct Documentation

struct rmf_traffic::blockade::**Status**

Public Members

ReservationId **reservation**

std::optional<*CheckpointId*> **last_ready**

CheckpointId **last_reached**

bool **critical_error**

Struct Writer::Checkpoint

- Defined in file_latest_rmf_traffic_include_rmf_traffic_blockade_Writer.hpp

Nested Relationships

This struct is a nested type of *Class Writer*.

Struct Documentation

struct rmf_traffic::blockade::*Writer*::**Checkpoint**

Public Members

Eigen::Vector2d **position**

std::string **map_name**

bool **can_hold**

Struct Writer::Reservation

- Defined in file_latest_rmf_traffic_include_rmf_traffic_blockade_Writer.hpp

Nested Relationships

This struct is a nested type of *Class Writer*.

Struct Documentation

struct rmf_traffic::blockade::*Writer*::**Reservation**

Public Members

std::vector<*Checkpoint*> **path**

double **radius**

Struct Dependency

- Defined in file_latest_rmf_traffic_include_rmf_traffic_Route.hpp

Struct Documentation

struct rmf_traffic::**Dependency**

Bundle of integers representing a dependency on a checkpoint within a specific participant's plan.

Public Functions

bool **operator==**(**const** *Dependency* &*other*) **const**

Equality operator.

Public Members

uint64_t **on_participant**

uint64_t **on_plan**

uint64_t **on_route**

uint64_t **on_checkpoint**

Struct DependsOnPlan::Dependency

- Defined in file_latest_rmf_traffic_include_rmf_traffic_Route.hpp

Nested Relationships

This struct is a nested type of *Class DependsOnPlan*.

Struct Documentation

struct rmf_traffic::*DependsOnPlan*::**Dependency**

Public Members

RouteId **on_route**

CheckpointId **on_checkpoint**

Struct DetectConflict::Conflict

- Defined in file_latest_rmf_traffic_include_rmf_traffic_DetectConflict.hpp

Nested Relationships

This struct is a nested type of *Class DetectConflict*.

Struct Documentation

struct rmf_traffic::*DetectConflict*::**Conflict**

Public Members

Trajectory::*const_iterator* **a_it**

Trajectory::*const_iterator* **b_it**

Time **time**

Struct Add::Item

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_Change.hpp

Nested Relationships

This struct is a nested type of *Class Change::Add*.

Struct Documentation

struct rmf_traffic::schedule::*Change*::*Add*::**Item**
A description of an addition.

Public Members

RouteId **route_id**
The ID of the route being added, relative to the plan it belongs to.

StorageId **storage_id**
The storage ID of the route.

ConstRoutePtr **route**
The information for the route being added.

Struct Inconsistencies::Element

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_Inconsistencies.hpp

Nested Relationships

This struct is a nested type of *Class Inconsistencies*.

Struct Documentation

struct rmf_traffic::schedule::*Inconsistencies*::**Element**

An element of the *Inconsistencies* container. This tells the ranges of inconsistencies that are present for the specified Participant.

Public Members

ParticipantId **participant**

Ranges **ranges**

Struct Ranges::Range

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_Inconsistencies.hpp

Nested Relationships

This struct is a nested type of *Class Inconsistencies::Ranges*.

Struct Documentation

struct rmf_traffic::schedule::*Inconsistencies*::*Ranges*::**Range**

A single range of inconsistencies within a participant.

Every version between (and including) the lower and upper versions have not been received by the *Database*.

Public Members

ItineraryVersion **lower**

ItineraryVersion **upper**

Template Struct Negotiation::SearchResult

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_Negotiation.hpp

Nested Relationships

This struct is a nested type of *Class Negotiation*.

Struct Documentation

template<typename **Ptr**>
struct rmf_traffic::schedule::*Negotiation*::**SearchResult**

Public Functions

inline bool **deprecated**() **const**

inline bool **absent**() **const**

inline bool **found**() **const**

inline **operator** **bool**() **const**

Public Members

SearchStatus **status**
The status of the search.

Ptr **table**
The *Table* that was searched for (or nullptr if status is Deprecated or Absent)

Struct Negotiation::Submission

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_Negotiation.hpp

Nested Relationships

This struct is a nested type of *Class Negotiation*.

Struct Documentation

struct rmf_traffic::schedule::*Negotiation*::**Submission**

Public Members

ParticipantId **participant**

PlanId **plan**

Itinerary **itinerary**

Struct Negotiation::VersionedKey

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_Negotiation.hpp

Nested Relationships

This struct is a nested type of *Class Negotiation*.

Struct Documentation

struct rmf_traffic::schedule::*Negotiation*::**VersionedKey**

This struct is used to select a child table, demaning a specific version.

Public Functions

inline bool **operator==**(**const** *VersionedKey* &*other*) **const**

inline bool **operator!=**(**const** *VersionedKey* &*other*) **const**

Public Members

ParticipantId **participant**

Version **version**

Struct Rectifier::Range

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_Rectifier.hpp

Nested Relationships

This struct is a nested type of *Class Rectifier*.

Struct Documentation

struct rmf_traffic::schedule::*Rectifier*::**Range**

A range of itinerary change IDs that is currently missing from a database. All IDs from lower to upper are missing, including lower and upper themselves.

It is undefined behavior if the value given to upper is less than the value given to upper.

Public Members

ItineraryVersion **lower**
The ID of the first itinerary change in this range that is missing.

ItineraryVersion **upper**
The ID of the last itinerary change in this range that is missing.

Struct View::Element

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_Viewer.hpp

Nested Relationships

This struct is a nested type of *Class Viewer::View*.

Struct Documentation

struct rmf_traffic::schedule::*Viewer*::*View*::**Element**

Public Members

const *ParticipantId* **participant**

const *PlanId* **plan_id**

const *RouteId* **route_id**

const std::shared_ptr<**const** *Route*> **route**

const *ParticipantDescription* &**description**

Struct Trajectory::InsertionResult

- Defined in file_latest_rmf_traffic_include_rmf_traffic_Trajectory.hpp

Nested Relationships

This struct is a nested type of *Class Trajectory*.

Struct Documentation

struct rmf_traffic::*Trajectory*::**InsertionResult**

Public Members

iterator **it**

bool **inserted**

Template Struct hash< rmf_traffic::agv::LaneClosure >

- Defined in file_latest_rmf_traffic_include_rmf_traffic_agv_LaneClosure.hpp

Struct Documentation

template<>
struct std::**hash**<rmf_traffic::agv::*LaneClosure*>

Public Functions

inline std::size_t **operator()**(**const** rmf_traffic::agv::*LaneClosure* &*closure*) **const** **noexcept**

Class CentralizedNegotiation

- Defined in file_latest_rmf_traffic_include_rmf_traffic_agv_CentralizedNegotiation.hpp

Nested Relationships

Nested Types

- *Class CentralizedNegotiation::Agent*
- *Class CentralizedNegotiation::Result*

Class Documentation

class rmf_traffic::agv::**CentralizedNegotiation**

Public Types

using **Proposal** = std::unordered_map<schedule::*ParticipantId*, *Plan*>
When a proposal is found, it will provide a plan for each agent.

Public Functions

CentralizedNegotiation(std::shared_ptr<**const** schedule::*Viewer*> *viewer*)
Constructor

Parameters

- `[in] viewer`: A viewer for the traffic schedule. You may provide a std::shared_ptr<const schedule::Database> for this. The negotiation will avoid creating any new conflicts with schedule participants that are not part of the negotiation.

const std::shared_ptr<**const** schedule::*Viewer*> &**viewer**() **const**
Get the schedule viewer.

CentralizedNegotiation &**viewer**(std::shared_ptr<**const** schedule::*Viewer*> *v*)
Set the schedule viewer.

CentralizedNegotiation &**optimal**(bool *on* = true)
Require the negotiation to consider all combinations so that it finds the (near-)optimal solution. Off by default.

CentralizedNegotiation &**log**(bool *on* = true)
Toggle on/off whether to log the progress of the negotiation and save it in the *Result*. Off by default.

CentralizedNegotiation &**print**(bool *on* = true)
Toggle on/off whether to print the progress of the negotiation while it is running. Off by default.

Result **solve**(**const** std::vector<*Agent*> &*agents*) **const**
Solve a centralized negotiation for the given agents.

class Agent

Public Functions

Agent(schedule::*ParticipantId* *id*, *Plan*::*Start* *start*, *Plan*::*Goal* *goal*, std::shared_ptr<**const** *Planner*> *planner*, std::optional<*SimpleNegotiator*::*Options*> *options* = std::nullopt)
Constructor

Parameters

- `[in] id`: This agent's ID within the schedule database. If multiple agents are given the same ID in a negotiation, then a runtime exception will be thrown.
- `[in] starts`: The starting condition for this agent.
- `[in] goal`: The goal for this agent.
- `[in] planner`: The single-agent planner used for this agent. Each agent can have its own planner or they can share planners. If this is set to nullptr when the negotiation begins, then a runtime exception will be thrown.
- `[in] options`: Options to use for the negotiator of this agent. If nullopt is provided, then the default options of the *SimpleNegotiator* will be used.

Agent(schedule::*ParticipantId* *id*, std::vector<*Plan*::*Start*> *starts*, *Plan*::*Goal* *goal*, std::shared_ptr<**const** *Planner*> *planner*, std::optional<*SimpleNegotiator*::*Options*> *options* = std::nullopt)
Constructor

The planner will use whichever starting condition provides the optimal plan.

Parameters

- `[in] id`: This agent's ID within the schedule database. If multiple agents are given the same ID in a negotiation, then a runtime exception will be thrown.
- `[in] starts`: One or more starting conditions for this agent. If no starting conditions are provided before the negotiation begins, then a runtime exception will be thrown.

Parameters

- `[in] goal`: The goal for this agent.
- `[in] planner`: The single-agent planner used for this agent. Each agent can have its own planner or they can share planners. If this is set to nullptr when the negotiation begins, then a runtime exception will be thrown.
- `[in] options`: Options to use for the negotiator of this agent. If nullopt is provided, then the default options of the *SimpleNegotiator* will be used.

schedule::*ParticipantId* **id**() **const**
Get the ID for this agent.

Agent &**id**(schedule::*ParticipantId* *value*)
Set the ID for this agent.

const std::vector<*Plan*::*Start*> &**starts**() **const**
Get the starts for this agent.

Agent &**starts**(std::vector<*Plan*::*Start*> *values*)
Set the starts for this agent.

const *Plan*::*Goal* &**goal**() **const**
Get the goal for this agent.

Agent &**goal**(*Plan*::*Goal* *value*)
Set the goal for this agent.

const std::shared_ptr<**const** *Planner*> &**planner**() **const**
Get the planner for this agent.

Agent &**planner**(std::shared_ptr<**const** *Planner*> *value*)
Set the planner for this agent.

const std::optional<*SimpleNegotiator*::*Options*> &**options**() **const**
Get the options for this agent.

Agent &**options**(std::optional<*SimpleNegotiator*::*Options*> *value*)
Set the options for this agent.

class Result

Public Functions

const std::optional<*Proposal*> &**proposal**() **const**
If a solution was found, it will be provided by this proposal.

const std::unordered_set<schedule::*ParticipantId*> &**blockers**() **const**
This is a list of schedule Participants that were not part of the negotiation who blocked the planning effort. Blockers do not necessarily prevent a solution from being found, but they do prevent the optimal solution from being available.

const std::vector<std::string> &**log**() **const**
A log of messages related to the negotiation. This will be empty unless the *log()* function of the *CentralizedNegotiation* is toggled on before solving.

Class CentralizedNegotiation::Agent

- Defined in file_latest_rmf_traffic_include_rmf_traffic_agv_CentralizedNegotiation.hpp

Nested Relationships

This class is a nested type of *Class CentralizedNegotiation*.

Class Documentation

class rmf_traffic::agv::*CentralizedNegotiation*::**Agent**

Public Functions

Agent(schedule::*ParticipantId* *id*, *Plan*::*Start* *start*, *Plan*::*Goal* *goal*, std::shared_ptr<**const** *Planner*> *planner*, std::optional<*SimpleNegotiator*::*Options*> *options* = std::nullopt)
Constructor

Parameters

- `[in] id`: This agent's ID within the schedule database. If multiple agents are given the same ID in a negotiation, then a runtime exception will be thrown.
- `[in] starts`: The starting condition for this agent.
- `[in] goal`: The goal for this agent.
- `[in] planner`: The single-agent planner used for this agent. Each agent can have its own planner or they can share planners. If this is set to nullptr when the negotiation begins, then a runtime exception will be thrown.
- `[in] options`: Options to use for the negotiator of this agent. If nullopt is provided, then the default options of the *SimpleNegotiator* will be used.

Agent(schedule::*ParticipantId* *id*, std::vector<*Plan*::*Start*> *starts*, *Plan*::*Goal* *goal*, std::shared_ptr<**const** *Planner*> *planner*, std::optional<*SimpleNegotiator*::*Options*> *options* = std::nullopt)
Constructor

The planner will use whichever starting condition provides the optimal plan.

Parameters

- `[in] id`: This agent's ID within the schedule database. If multiple agents are given the same ID in a negotiation, then a runtime exception will be thrown.
- `[in] starts`: One or more starting conditions for this agent. If no starting conditions are provided before the negotiation begins, then a runtime exception will be thrown.

Parameters

- `[in] goal`: The goal for this agent.

- `[in] planner`: The single-agent planner used for this agent. Each agent can have its own planner or they can share planners. If this is set to nullptr when the negotiation begins, then a runtime exception will be thrown.
- `[in] options`: Options to use for the negotiator of this agent. If nullopt is provided, then the default options of the *SimpleNegotiator* will be used.

schedule::*ParticipantId* **id**() **const**
Get the ID for this agent.

Agent &**id**(schedule::*ParticipantId* *value*)
Set the ID for this agent.

const std::vector<*Plan*::*Start*> &**starts**() **const**
Get the starts for this agent.

Agent &**starts**(std::vector<*Plan*::*Start*> *values*)
Set the starts for this agent.

const *Plan*::*Goal* &**goal**() **const**
Get the goal for this agent.

Agent &**goal**(*Plan*::*Goal* *value*)
Set the goal for this agent.

const std::shared_ptr<**const** *Planner*> &**planner**() **const**
Get the planner for this agent.

Agent &**planner**(std::shared_ptr<**const** *Planner*> *value*)
Set the planner for this agent.

const std::optional<*SimpleNegotiator*::*Options*> &**options**() **const**
Get the options for this agent.

Agent &**options**(std::optional<*SimpleNegotiator*::*Options*> *value*)
Set the options for this agent.

Class CentralizedNegotiation::Result

- Defined in file_latest_rmf_traffic_include_rmf_traffic_agv_CentralizedNegotiation.hpp

Nested Relationships

This class is a nested type of *Class CentralizedNegotiation*.

Class Documentation

class rmf_traffic::agv::*CentralizedNegotiation*::**Result**

Public Functions

const std::optional<*Proposal*> &**proposal**() **const**
If a solution was found, it will be provided by this proposal.

const std::unordered_set<schedule::*ParticipantId*> &**blockers**() **const**
This is a list of schedule Participants that were not part of the negotiation who blocked the planning effort. Blockers do not necessarily prevent a solution from being found, but they do prevent the optimal solution from being available.

const std::vector<std::string> &**log**() **const**
A log of messages related to the negotiation. This will be empty unless the *log()* function of the *CentralizedNegotiation* is toggled on before solving.

Class Graph

- Defined in file_latest_rmf_traffic_include_rmf_traffic_agv_Graph.hpp

Nested Relationships

Nested Types

- *Class Graph::Lane*
- *Class Lane::Dock*
- *Class Lane::Door*
- *Class Lane::DoorClose*
- *Class Lane::DoorOpen*
- *Class Lane::Event*
- *Class Lane::Executor*
- *Class Lane::LiftDoorOpen*
- *Class Lane::LiftMove*
- *Class Lane::LiftSession*
- *Class Lane::LiftSessionBegin*
- *Class Lane::LiftSessionEnd*
- *Class Lane::Node*
- *Class Lane::Properties*
- *Class Lane::Wait*
- *Class Graph::OrientationConstraint*
- *Class Graph::Waypoint*

Class Documentation

class rmf_traffic::agv::**Graph**

Public Functions

Graph()
Default constructor.

Waypoint &**add_waypoint**(std::string *map_name*, Eigen::Vector2d *location*)
Make a new waypoint for this graph. It will not be connected to any other waypoints until you use make_lane() to connect it.

Note Waypoints cannot be erased from a *Graph* after they are created.

Waypoint &**get_waypoint**(std::size_t *index*)
Get a waypoint based on its index.

const *Waypoint* &**get_waypoint**(std::size_t *index*) **const**
const-qualified *get_waypoint()*

Waypoint ***find_waypoint**(**const** std::string &*key*)
Find a waypoint given a key name. If the graph does not have a matching key name, then a nullptr will be returned.

const *Waypoint* ***find_waypoint**(**const** std::string &*key*) **const**
const-qualified *find_waypoint()*

bool **add_key**(**const** std::string &*key*, std::size_t *wp_index*)
Add a new waypoint key name to the graph. If a new key name is given, then this function will return true. If the given key name was already in use, then this will return false and nothing will be changed in the graph.

bool **remove_key**(**const** std::string &*key*)
Remove the waypoint key with the given name, if it exists in this *Graph*. If the key was removed, this will return true. If the key did not exist, this will return false.

bool **set_key**(**const** std::string &*key*, std::size_t *wp_index*)
Set a waypoint key. If this key is already in the *Graph*, it will be changed to the new association.

This function will return false if wp_index is outside the range of the waypoints in this *Graph*.

const std::unordered_map<std::string, std::size_t> &**keys**() **const**
Get the map of all keys in this *Graph*.

std::size_t **num_waypoints**() **const**
Get the number of waypoints in this *Graph*.

Lane &**add_lane**(**const** *Lane*::*Node* &*entry*, **const** *Lane*::*Node* &*exit*, *Lane*::*Properties* *properties* = *Lane*::*Properties*())
Make a lane for this graph. Lanes connect waypoints together, allowing the graph to know how the robot is allowed to traverse between waypoints.

Lane &**get_lane**(std::size_t *index*)
Get the lane at the specified index.

const *Lane* &**get_lane**(std::size_t *index*) **const**
const-qualified *get_lane()*

std::size_t **num_lanes**() **const**
Get the number of Lanes in this *Graph*.

const std::vector<std::size_t> &**lanes_from**(std::size_t *wp_index*) **const**
Get the indices of lanes that come out of the given *Waypoint* index.

const std::vector<std::size_t> &**lanes_into**(std::size_t *wp_index*) **const**
Get the indices of lanes that arrive into the given *Waypoint* index.

Lane ***lane_from**(std::size_t *from_wp*, std::size_t *to_wp*)
Get a reference to the lane that goes from from_wp to to_wp if such a lane exists. If no such lane exists, this will return a nullptr. If multiple exist, this will return the one that was added most recently.

const *Lane* ***lane_from**(std::size_t *from_wp*, std::size_t *to_wp*) **const**
const-qualified *lane_from()*

class Lane
Add a lane to connect two waypoints.

Public Types

using EventPtr = rmf_utils::clone_ptr<*Event*>

Public Functions

Node &**entry**()
Get the entry node of this *Lane*. The lane represents an edge in the graph that goes away from this node.

const *Node* &**entry**() **const**
const-qualified *entry()*

Node &**exit**()
Get the exit node of this *Lane*. The lane represents an edge in the graph that goes into this node.

const *Node* &**exit**() **const**
const-qualified *exit()*

Properties &**properties**()
Get the properties of this *Lane*.

const *Properties* &**properties**() **const**
const-qualified *properties()*

std::size_t **index**() **const**
Get the index of this *Lane* within the *Graph*.

class Dock

Public Functions

Dock(std::string *dock_name*, *Duration* *duration*)
Constructor

Parameters
- `[in] Name`: of the dock that will be approached
- `[in] How`: long the robot will take to dock

const std::string &**dock_name**() **const**
Get the name of the dock.

Dock &**dock_name**(std::string *name*)
Set the name of the dock.

Duration **duration**() **const**
Get an estimate for how long the docking will take.

Dock &**duration**(*Duration* *d*)
Set an estimate for how long the docking will take.

class Door
A door in the graph which needs to be opened before a robot can enter a certain lane or closed before the robot can exit the lane.

Subclassed by *rmf_traffic::agv::Graph::Lane::DoorClose*, *rmf_traffic::agv::Graph::Lane::DoorOpen*

Public Functions

Door(std::string *name*, *Duration* *duration*)
Constructor

Parameters
- `[in] name`: Unique name of the door.
- `[in] duration`: How long the door takes to open or close.

const std::string &**name**() **const**
Get the unique name (ID) of this *Door*.

Door &**name**(std::string *name*)
Set the unique name (ID) of this *Door*.

Duration **duration**() **const**
Get the duration incurred by waiting for this door to open or close.

Door &**duration**(*Duration* *duration*)
Set the duration incurred by waiting for this door to open or close.

class DoorClose : **public** rmf_traffic::agv::*Graph*::*Lane*::*Door*

class DoorOpen : **public** rmf_traffic::agv::*Graph*::*Lane*::*Door*

class Event
An abstraction for the different kinds of *Lane* events.

Public Functions

virtual *Duration* **duration**() **const** = 0
An estimate of how long the event will take.

template<typename **DerivedExecutor**>
inline *DerivedExecutor* &**execute**(*DerivedExecutor* &*executor*) **const**

virtual *Executor* &**execute**(*Executor* &*executor*) **const** = 0
Execute this event.

virtual *EventPtr* **clone**() **const** = 0
Clone this event.

virtual ~Event() = default

Public Static Functions

static *EventPtr* **make**(*DoorOpen* *open*)

static *EventPtr* **make**(*DoorClose* *close*)

static *EventPtr* **make**(*LiftSessionBegin* *open*)

static *EventPtr* **make**(*LiftSessionEnd* *close*)

static *EventPtr* **make**(*LiftMove* *move*)

static *EventPtr* **make**(*LiftDoorOpen* *open*)

static *EventPtr* **make**(*Dock* *dock*)

static *EventPtr* **make**(*Wait* *wait*)

class Executor
A customizable *Executor* that can carry out actions based on which *Event* type is present.

Public Types

using DoorOpen = *Lane*::*DoorOpen*

using DoorClose = *Lane*::*DoorClose*

using LiftSessionBegin = *Lane*::*LiftSessionBegin*

using LiftDoorOpen = *Lane*::*LiftDoorOpen*

using LiftSessionEnd = *Lane*::*LiftSessionEnd*

using LiftMove = *Lane*::*LiftMove*

using Dock = *Lane*::*Dock*

using Wait = *Lane*::*Wait*

Public Functions

virtual void **execute**(**const** *DoorOpen* &*open*) = 0

virtual void **execute**(**const** *DoorClose* &*close*) = 0

virtual void **execute**(**const** *LiftSessionBegin* &*begin*) = 0

virtual void **execute**(**const** *LiftDoorOpen* &*open*) = 0

virtual void **execute**(**const** *LiftSessionEnd* &*end*) = 0

virtual void **execute**(**const** *LiftMove* &*move*) = 0

virtual void **execute**(**const** *Dock* &*dock*) = 0

virtual void **execute**(**const** *Wait* &*wait*) = 0

virtual ~Executor() = default

class LiftDoorOpen : **public** rmf_traffic::agv::*Graph*::*Lane*::*LiftSession*

class LiftMove : **public** rmf_traffic::agv::*Graph*::*Lane*::*LiftSession*

class LiftSession

A lift door in the graph which needs to be opened before a robot can enter a certain lane or closed before the robot can exit the lane.

Subclassed by *rmf_traffic::agv::Graph::Lane::LiftDoorOpen*, *rmf_traffic::agv::Graph::Lane::LiftMove*, *rmf_traffic::agv::Graph::Lane::LiftSessionBegin*, *rmf_traffic::agv::Graph::Lane::LiftSessionEnd*

Public Functions

LiftSession(std::string *lift_name*, std::string *floor_name*, *Duration* *duration*)

Constructor

Parameters

- `[in] lift_name`: Name of the lift that this door belongs to.
- `[in] floor_name`: Name of the floor that this door belongs to.
- `[in] duration`: How long the door takes to open or close.

const std::string &**lift_name**() **const**

Get the name of the lift that the door belongs to.

LiftSession &**lift_name**(std::string *name*)

Set the name of the lift that the door belongs to.

const std::string &**floor_name**() **const**

Get the name of the floor that this door is on.

LiftSession &**floor_name**(std::string *name*)

Set the name of the floor that this door is on.

Duration **duration**() **const**

Get an estimate of how long it will take the door to open or close.

LiftSession &**duration**(*Duration* *duration*)

Set an estimate of how long it will take the door to open or close.

class LiftSessionBegin : **public** rmf_traffic::agv::*Graph*::*Lane*::*LiftSession*

class LiftSessionEnd : **public** rmf_traffic::agv::*Graph*::*Lane*::*LiftSession*

class Node

A *Lane Node* wraps up a *Waypoint* with constraints. The constraints stipulate the conditions for entering or exiting the lane to reach this waypoint.

Public Functions

Node(std::size_t *waypoint_index*, rmf_utils::clone_ptr<*Event*> *event* = nullptr, rmf_utils::clone_ptr<*OrientationConstraint*> *orientation* = nullptr)

Constructor

Parameters

- `waypoint_index`: The index of the waypoint for this *Node*
- `event`: An event that must happen before/after this *Node* is approached (before if it's an entry *Node* or after if it's an exit *Node*).
- `orientation`: Any orientation constraints for moving to/from this *Node* (depending on whether it's an entry *Node* or an exit *Node*).

Node(std::size_t *waypoint_index*, rmf_utils::clone_ptr<*OrientationConstraint*> *orientation*)

Constructor. The event parameter will be nullptr.

Parameters

- `waypoint_index`: The index of the waypoint for this *Node*
- `orientation`: Any orientation constraints for moving to/from this *Node* (depending on whether it's an entry *Node* or an exit *Node*).

std::size_t **waypoint_index**() **const**

Get the index of the waypoint that this *Node* is wrapped around.

const *Event* ***event**() **const**

Get a reference to an event that must occur before or after this *Node* is visited.

Note Before if this is an entry node or after if this is an exit node

Node &**event**(rmf_utils::clone_ptr<*Event*> *new_event*)

Set the event that must occur before or after this *Node* is visited.

const *OrientationConstraint* ***orientation_constraint**() **const**

Get the constraint on orientation that is tied to this *Node*.

class Properties

The *Lane Properties* class contains properties that apply across the full extent of the lane.

Public Functions

Properties()

Construct a default set of properties

- speed_limit: nullopt

std::optional<double> **speed_limit**() **const**

Get the speed limit along this lane. If a std::nullopt is returned, then there is no specified speed limit for the lane.

Properties &**speed_limit**(std::optional<double> *value*)

Set the speed limit along this lane. Providing a std::nullopt indicates that there is no speed limit for the lane.

class Wait

Public Functions

Wait(*Duration value*)
Constructor

Parameters
- `[in] duration`: How long the wait will be.

Duration **duration**() **const**
Get how long the wait will be.

Wait &**duration**(*Duration value*)
Set how long the wait will be.

class OrientationConstraint
A class that implicitly specifies a constraint on the robot's orientation.

Public Types

enum Direction
Values:

enumerator Forward

enumerator Backward

Public Functions

virtual bool **apply**(Eigen::Vector3d &*position*, **const** Eigen::Vector2d &*course_vector*) **const** = 0
Apply the constraint to the given homogeneous position.

Return True if the constraint is satisfied with the new value of position. False if the constraint could not be satisfied.

Parameters
- `[inout] position`: The position which needs to be constrained. The function should modify this position such that it satisfies the constraint, if possible.
- `[in] course_vector`: The direction that the robot is travelling in. Given for informational purposes.

virtual rmf_utils::clone_ptr<*OrientationConstraint*> **clone**() **const** = 0
Clone this *OrientationConstraint*.

virtual ~OrientationConstraint() = default

Public Static Functions

static rmf_utils::clone_ptr<*OrientationConstraint*> **make**(std::vector<double> *acceptable_orientations*)
Make an orientation constraint that requires a specific value for the orientation.

static rmf_utils::clone_ptr<*OrientationConstraint*> **make**(*Direction* *direction*, **const** Eigen::Vector2d &*forward_vector*)
Make an orientation constraint that requires the vehicle to face forward or backward.

class Waypoint

Public Functions

const std::string &**get_map_name**() **const**
Get the name of the map that this *Waypoint* exists on.

Waypoint &**set_map_name**(std::string *map*)
Set the name of the map that this *Waypoint* exists on.

const Eigen::Vector2d &**get_location**() **const**
Get the position of this *Waypoint*.

Waypoint &**set_location**(Eigen::Vector2d *location*)
Set the position of this *Waypoint*.

bool **is_holding_point**() **const**
Returns true if this *Waypoint* can be used as a holding point for the vehicle, otherwise returns false.

Waypoint &**set_holding_point**(bool *_is_holding_point*)
Set whether this waypoint can be used as a holding point for the vehicle.

bool **is_passthrough_point**() **const**
Returns true if this *Waypoint* is a passthrough point, meaning a planner should not have a robot wait at this point, even just briefly to allow another robot to pass. Setting passthrough points reduces the branching factor of a planner, allowing it to run faster, at the cost of losing possible solutions to conflicts.

Waypoint &**set_passthrough_point**(bool *_is_passthrough*)
Set this *Waypoint* to be a passthrough point.

bool **is_parking_spot**() **const**
Returns true if this *Waypoint* is a parking spot. Parking spots are used when an emergency alarm goes off, and the robot is required to park itself.

Waypoint &**set_parking_spot**(bool *_is_parking_spot*)
Set this *Waypoint* to be a parking spot.

bool **is_charger**() **const**
Returns true if this *Waypoint* is a charger spot. Robots are routed to these spots when their batteries charge levels drop below the threshold value.

Waypoint &**set_charger**(bool *_is_charger*)
Set this *Waypoint* to be a parking spot.

std::size_t **index**() **const**
The index of this waypoint within the *Graph*. This cannot be changed after the waypoint is created.

const std::string *__name__() **const**
If this waypoint has a name, return a reference to it. If this waypoint does not have a name, return a nullptr.

The name of a waypoint can only be set using *add_key()* or *set_key()*.

std::string **name_or_index**(**const** std::string &*name_format* = "%s", **const** std::string &*index_format* = "#%d") **const**

If this waypoint has a name, the name will be returned. Otherwise it will return the waypoint index, formatted into a string based on the index_format argument.

Parameters

- `[in] name_format`: If this waypoint has an assigned name, the first instance of "%s" within name_format will be replaced with the name of the waypoint. If there is no s in the name_format string, then this function will simply return the name_format string as-is when the waypoint has a name.
- `[in] index_format`: If this waypoint does not have an assigned name, the first instance of "%d" within the index_format string will be replaced with the stringified decimal index value of the waypoint. If there is no "%d" in the index_format string, then this function will simply return the index_format string as-is when the waypoint does not have a name.

Class Graph::Lane

- Defined in file_latest_rmf_traffic_include_rmf_traffic_agv_Graph.hpp

Nested Relationships

This class is a nested type of *Class Graph*.

Nested Types

- *Class Lane::Dock*
- *Class Lane::Door*
- *Class Lane::DoorClose*
- *Class Lane::DoorOpen*
- *Class Lane::Event*
- *Class Lane::Executor*
- *Class Lane::LiftDoorOpen*
- *Class Lane::LiftMove*
- *Class Lane::LiftSession*
- *Class Lane::LiftSessionBegin*
- *Class Lane::LiftSessionEnd*
- *Class Lane::Node*
- *Class Lane::Properties*
- *Class Lane::Wait*

Class Documentation

class rmf_traffic::agv::*Graph*::**Lane**

Add a lane to connect two waypoints.

Public Types

using **EventPtr** = rmf_utils::clone_ptr<*Event*>

Public Functions

Node &**entry**()

Get the entry node of this *Lane*. The lane represents an edge in the graph that goes away from this node.

const *Node* &**entry**() **const**

const-qualified *entry()*

Node &**exit**()

Get the exit node of this *Lane*. The lane represents an edge in the graph that goes into this node.

const *Node* &**exit**() **const**

const-qualified *exit()*

Properties &**properties**()

Get the properties of this *Lane*.

const *Properties* &**properties**() **const**

const-qualified *properties()*

std::size_t **index**() **const**

Get the index of this *Lane* within the *Graph*.

class **Dock**

Public Functions

Dock(std::string *dock_name*, *Duration duration*)

Constructor

Parameters

- [in] Name: of the dock that will be approached
- [in] How: long the robot will take to dock

const std::string &**dock_name**() **const**

Get the name of the dock.

Dock &**dock_name**(std::string *name*)

Set the name of the dock.

Duration **duration**() **const**

Get an estimate for how long the docking will take.

Dock &**duration**(*Duration d*)

Set an estimate for how long the docking will take.

class Door

A door in the graph which needs to be opened before a robot can enter a certain lane or closed before the robot can exit the lane.

Subclassed by *rmf_traffic::agv::Graph::Lane::DoorClose*, *rmf_traffic::agv::Graph::Lane::DoorOpen*

Public Functions

Door(std::string *name*, *Duration* *duration*)

Constructor

Parameters

- `[in] name`: Unique name of the door.
- `[in] duration`: How long the door takes to open or close.

const std::string &**name**() **const**

Get the unique name (ID) of this *Door*.

Door &**name**(std::string *name*)

Set the unique name (ID) of this *Door*.

Duration **duration**() **const**

Get the duration incurred by waiting for this door to open or close.

Door &**duration**(*Duration* *duration*)

Set the duration incurred by waiting for this door to open or close.

class DoorClose : **public** rmf_traffic::agv::*Graph*::*Lane*::*Door*

class DoorOpen : **public** rmf_traffic::agv::*Graph*::*Lane*::*Door*

class Event

An abstraction for the different kinds of *Lane* events.

Public Functions

virtual *Duration* **duration**() **const** = 0

An estimate of how long the event will take.

template<typename **DerivedExecutor**>
inline *DerivedExecutor* &**execute**(*DerivedExecutor* &*executor*) **const**

virtual *Executor* &**execute**(*Executor* &*executor*) **const** = 0

Execute this event.

virtual *EventPtr* **clone**() **const** = 0

Clone this event.

virtual ~Event() = default

Public Static Functions

static *EventPtr* **make**(*DoorOpen open*)

static *EventPtr* **make**(*DoorClose close*)

static *EventPtr* **make**(*LiftSessionBegin open*)

static *EventPtr* **make**(*LiftSessionEnd close*)

static *EventPtr* **make**(*LiftMove move*)

static *EventPtr* **make**(*LiftDoorOpen open*)

static *EventPtr* **make**(*Dock dock*)

static *EventPtr* **make**(*Wait wait*)

class Executor

A customizable *Executor* that can carry out actions based on which *Event* type is present.

Public Types

using DoorOpen = *Lane*::*DoorOpen*

using DoorClose = *Lane*::*DoorClose*

using LiftSessionBegin = *Lane*::*LiftSessionBegin*

using LiftDoorOpen = *Lane*::*LiftDoorOpen*

using LiftSessionEnd = *Lane*::*LiftSessionEnd*

using LiftMove = *Lane*::*LiftMove*

using Dock = *Lane*::*Dock*

using Wait = *Lane*::*Wait*

Public Functions

virtual void **execute**(**const** *DoorOpen* &*open*) = 0

virtual void **execute**(**const** *DoorClose* &*close*) = 0

virtual void **execute**(**const** *LiftSessionBegin* &*begin*) = 0

virtual void **execute**(**const** *LiftDoorOpen* &*open*) = 0

virtual void **execute**(**const** *LiftSessionEnd* &*end*) = 0

virtual void **execute**(**const** *LiftMove* &*move*) = 0

virtual void **execute**(**const** *Dock* &*dock*) = 0

virtual ~Executor() = default

class LiftDoorOpen : **public** rmf_traffic::agv::*Graph*::*Lane*::*LiftSession*

class LiftMove : **public** rmf_traffic::agv::*Graph*::*Lane*::*LiftSession*

class LiftSession

A lift door in the graph which needs to be opened before a robot can enter a certain lane or closed before the robot can exit the lane.

Subclassed by *rmf_traffic::agv::Graph::Lane::LiftDoorOpen*, *rmf_traffic::agv::Graph::Lane::LiftMove*, *rmf_traffic::agv::Graph::Lane::LiftSessionBegin*, *rmf_traffic::agv::Graph::Lane::LiftSessionEnd*

Public Functions

LiftSession(std::string *lift_name*, std::string *floor_name*, *Duration* *duration*)

Constructor

Parameters

- `[in] lift_name`: Name of the lift that this door belongs to.
- `[in] floor_name`: Name of the floor that this door belongs to.
- `[in] duration`: How long the door takes to open or close.

const std::string &**lift_name**() **const**

Get the name of the lift that the door belongs to.

LiftSession &**lift_name**(std::string *name*)

Set the name of the lift that the door belongs to.

const std::string &**floor_name**() **const**

Get the name of the floor that this door is on.

LiftSession &**floor_name**(std::string *name*)

Set the name of the floor that this door is on.

Duration **duration**() **const**

Get an estimate of how long it will take the door to open or close.

LiftSession &**duration**(*Duration* *duration*)

Set an estimate of how long it will take the door to open or close.

class LiftSessionBegin : **public** rmf_traffic::agv::*Graph*::*Lane*::*LiftSession*

class LiftSessionEnd : **public** rmf_traffic::agv::*Graph*::*Lane*::*LiftSession*

class Node

A *Lane Node* wraps up a *Waypoint* with constraints. The constraints stipulate the conditions for entering or exiting the lane to reach this waypoint.

Public Functions

Node(std::size_t *waypoint_index*, rmf_utils::clone_ptr<*Event*> *event* = nullptr, rmf_utils::clone_ptr<*OrientationConstraint*> *orientation* = nullptr)

Constructor

Parameters

- `waypoint_index`: The index of the waypoint for this *Node*
- `event`: An event that must happen before/after this *Node* is approached (before if it's an entry *Node* or after if it's an exit *Node*).
- `orientation`: Any orientation constraints for moving to/from this *Node* (depending on whether it's an entry *Node* or an exit *Node*).

Node(std::size_t *waypoint_index*, rmf_utils::clone_ptr<*OrientationConstraint*> *orientation*)
Constructor. The event parameter will be nullptr.

Parameters
- `waypoint_index`: The index of the waypoint for this *Node*
- `orientation`: Any orientation constraints for moving to/from this *Node* (depending on whether it's an entry *Node* or an exit *Node*).

std::size_t **waypoint_index**() **const**
Get the index of the waypoint that this *Node* is wrapped around.

const *Event* ***event**() **const**
Get a reference to an event that must occur before or after this *Node* is visited.

Note Before if this is an entry node or after if this is an exit node

Node &**event**(rmf_utils::clone_ptr<*Event*> *new_event*)
Set the event that must occur before or after this *Node* is visited.

const *OrientationConstraint* ***orientation_constraint**() **const**
Get the constraint on orientation that is tied to this *Node*.

class Properties
The *Lane Properties* class contains properties that apply across the full extent of the lane.

Public Functions

Properties()
Construct a default set of properties
- speed_limit: nullopt

std::optional<double> **speed_limit**() **const**
Get the speed limit along this lane. If a std::nullopt is returned, then there is no specified speed limit for the lane.

Properties &**speed_limit**(std::optional<double> *value*)
Set the speed limit along this lane. Providing a std::nullopt indicates that there is no speed limit for the lane.

class Wait

Public Functions

Wait(*Duration value*)
Constructor

Parameters
- `[in] duration`: How long the wait will be.

Duration **duration**() **const**
Get how long the wait will be.

Wait &**duration**(*Duration value*)
Set how long the wait will be.

Class Lane::Dock

- Defined in file_latest_rmf_traffic_include_rmf_traffic_agv_Graph.hpp

Nested Relationships

This class is a nested type of *Class Graph::Lane*.

Class Documentation

class rmf_traffic::agv::*Graph*::*Lane*::**Dock**

Public Functions

Dock(std::string *dock_name*, *Duration* *duration*)
Constructor

Parameters

- `[in] Name`: of the dock that will be approached
- `[in] How`: long the robot will take to dock

const std::string &**dock_name**() **const**
Get the name of the dock.

Dock &**dock_name**(std::string *name*)
Set the name of the dock.

Duration **duration**() **const**
Get an estimate for how long the docking will take.

Dock &**duration**(*Duration* *d*)
Set an estimate for how long the docking will take.

Class Lane::Door

- Defined in file_latest_rmf_traffic_include_rmf_traffic_agv_Graph.hpp

Nested Relationships

This class is a nested type of *Class Graph::Lane*.

Inheritance Relationships

Derived Types

- `public rmf_traffic::agv::Graph::Lane::DoorClose` (*Class Lane::DoorClose*)
- `public rmf_traffic::agv::Graph::Lane::DoorOpen` (*Class Lane::DoorOpen*)

Class Documentation

class `rmf_traffic::agv::Graph::Lane::Door`

A door in the graph which needs to be opened before a robot can enter a certain lane or closed before the robot can exit the lane.

Subclassed by *rmf_traffic::agv::Graph::Lane::DoorClose*, *rmf_traffic::agv::Graph::Lane::DoorOpen*

Public Functions

Door(std::string *name*, *Duration* *duration*)

Constructor

Parameters

- `[in] name`: Unique name of the door.
- `[in] duration`: How long the door takes to open or close.

const std::string &**name**() **const**

Get the unique name (ID) of this *Door*.

Door &**name**(std::string *name*)

Set the unique name (ID) of this *Door*.

Duration **duration**() **const**

Get the duration incurred by waiting for this door to open or close.

Door &**duration**(*Duration* *duration*)

Set the duration incurred by waiting for this door to open or close.

Class Lane::DoorClose

- Defined in file_latest_rmf_traffic_include_rmf_traffic_agv_Graph.hpp

Nested Relationships

This class is a nested type of *Class Graph::Lane*.

Inheritance Relationships

Base Type

- `public rmf_traffic::agv::Graph::Lane::Door` (*Class Lane::Door*)

Class Documentation

class DoorClose : **public** rmf_traffic::agv::*Graph*::*Lane*::*Door*

Class Lane::DoorOpen

- Defined in file_latest_rmf_traffic_include_rmf_traffic_agv_Graph.hpp

Nested Relationships

This class is a nested type of *Class Graph::Lane*.

Inheritance Relationships

Base Type

- `public rmf_traffic::agv::Graph::Lane::Door` (*Class Lane::Door*)

Class Documentation

class DoorOpen : **public** rmf_traffic::agv::*Graph*::*Lane*::*Door*

Class Lane::Event

- Defined in file_latest_rmf_traffic_include_rmf_traffic_agv_Graph.hpp

Nested Relationships

This class is a nested type of *Class Graph::Lane*.

Class Documentation

class `rmf_traffic::agv::Graph::Lane::`**Event**
An abstraction for the different kinds of *Lane* events.

Public Functions

virtual *Duration* **duration**() **const** = 0
An estimate of how long the event will take.

template<typename **DerivedExecutor**>
inline *DerivedExecutor* &**execute**(*DerivedExecutor* &*executor*) **const**

virtual *Executor* &**execute**(*Executor* &*executor*) **const** = 0
Execute this event.

virtual *EventPtr* **clone**() **const** = 0
Clone this event.

virtual ~Event() = default

Public Static Functions

static *EventPtr* **make**(*DoorOpen* *open*)

static *EventPtr* **make**(*DoorClose* *close*)

static *EventPtr* **make**(*LiftSessionBegin* *open*)

static *EventPtr* **make**(*LiftSessionEnd* *close*)

static *EventPtr* **make**(*LiftMove* *move*)

static *EventPtr* **make**(*LiftDoorOpen* *open*)

static *EventPtr* **make**(*Dock* *dock*)

static *EventPtr* **make**(*Wait* *wait*)

Class Lane::Executor

- Defined in file_latest_rmf_traffic_include_rmf_traffic_agv_Graph.hpp

Nested Relationships

This class is a nested type of *Class Graph::Lane*.

Class Documentation

class rmf_traffic::agv::*Graph*::*Lane*::**Executor**
A customizable *Executor* that can carry out actions based on which *Event* type is present.

Public Types

using DoorOpen = *Lane*::*DoorOpen*

using DoorClose = *Lane*::*DoorClose*

using LiftSessionBegin = *Lane*::*LiftSessionBegin*

using LiftDoorOpen = *Lane*::*LiftDoorOpen*

using LiftSessionEnd = *Lane*::*LiftSessionEnd*

using LiftMove = *Lane*::*LiftMove*

using Dock = *Lane*::*Dock*

using Wait = *Lane*::*Wait*

Public Functions

virtual void **execute**(**const** *DoorOpen* &*open*) = 0

virtual void **execute**(**const** *DoorClose* &*close*) = 0

virtual void **execute**(**const** *LiftSessionBegin* &*begin*) = 0

virtual void **execute**(**const** *LiftDoorOpen* &*open*) = 0

virtual void **execute**(**const** *LiftSessionEnd* &*end*) = 0

virtual void **execute**(**const** *LiftMove* &*move*) = 0

virtual void **execute**(**const** *Dock* &*dock*) = 0

virtual void **execute**(**const** *Wait* &*wait*) = 0

virtual ~Executor() = default

Class Lane::LiftDoorOpen

- Defined in file_latest_rmf_traffic_include_rmf_traffic_agv_Graph.hpp

Nested Relationships

This class is a nested type of *Class Graph::Lane*.

Inheritance Relationships

Base Type

- `public rmf_traffic::agv::Graph::Lane::LiftSession` (*Class Lane::LiftSession*)

Class Documentation

class LiftDoorOpen : **public** rmf_traffic::agv::*Graph*::*Lane*::*LiftSession*

Class Lane::LiftMove

- Defined in file_latest_rmf_traffic_include_rmf_traffic_agv_Graph.hpp

Nested Relationships

This class is a nested type of *Class Graph::Lane*.

Inheritance Relationships

Base Type

- `public rmf_traffic::agv::Graph::Lane::LiftSession` (*Class Lane::LiftSession*)

Class Documentation

class LiftMove : **public** rmf_traffic::agv::*Graph*::*Lane*::*LiftSession*

Class Lane::LiftSession

- Defined in file_latest_rmf_traffic_include_rmf_traffic_agv_Graph.hpp

Nested Relationships

This class is a nested type of *Class Graph::Lane*.

Inheritance Relationships

Derived Types

- `public rmf_traffic::agv::Graph::Lane::LiftDoorOpen` (*Class Lane::LiftDoorOpen*)
- `public rmf_traffic::agv::Graph::Lane::LiftMove` (*Class Lane::LiftMove*)
- `public rmf_traffic::agv::Graph::Lane::LiftSessionBegin` (*Class Lane::LiftSessionBegin*)
- `public rmf_traffic::agv::Graph::Lane::LiftSessionEnd` (*Class Lane::LiftSessionEnd*)

Class Documentation

class rmf_traffic::agv::*Graph*::*Lane*::**LiftSession**

A lift door in the graph which needs to be opened before a robot can enter a certain lane or closed before the robot can exit the lane.

Subclassed by *rmf_traffic::agv::Graph::Lane::LiftDoorOpen*, *rmf_traffic::agv::Graph::Lane::LiftMove*, *rmf_traffic::agv::Graph::Lane::LiftSessionBegin*, *rmf_traffic::agv::Graph::Lane::LiftSessionEnd*

Public Functions

LiftSession(std::string *lift_name*, std::string *floor_name*, *Duration* *duration*)

Constructor

Parameters

- `[in] lift_name`: Name of the lift that this door belongs to.
- `[in] floor_name`: Name of the floor that this door belongs to.
- `[in] duration`: How long the door takes to open or close.

const std::string &**lift_name**() **const**

Get the name of the lift that the door belongs to.

LiftSession &**lift_name**(std::string *name*)

Set the name of the lift that the door belongs to.

const std::string &**floor_name**() **const**

Get the name of the floor that this door is on.

LiftSession &**floor_name**(std::string *name*)

Set the name of the floor that this door is on.

Duration **duration**() **const**

Get an estimate of how long it will take the door to open or close.

LiftSession &**duration**(*Duration* *duration*)

Set an estimate of how long it will take the door to open or close.

Class Lane::LiftSessionBegin

- Defined in file_latest_rmf_traffic_include_rmf_traffic_agv_Graph.hpp

Nested Relationships

This class is a nested type of *Class Graph::Lane*.

Inheritance Relationships

Base Type

- `public rmf_traffic::agv::Graph::Lane::LiftSession` (*Class Lane::LiftSession*)

Class Documentation

class LiftSessionBegin : **public** rmf_traffic::agv::*Graph*::*Lane*::*LiftSession*

Class Lane::LiftSessionEnd

- Defined in file_latest_rmf_traffic_include_rmf_traffic_agv_Graph.hpp

Nested Relationships

This class is a nested type of *Class Graph::Lane*.

Inheritance Relationships

Base Type

- `public rmf_traffic::agv::Graph::Lane::LiftSession` (*Class Lane::LiftSession*)

Class Documentation

class LiftSessionEnd : **public** rmf_traffic::agv::*Graph*::*Lane*::*LiftSession*

Class Lane::Node

- Defined in file_latest_rmf_traffic_include_rmf_traffic_agv_Graph.hpp

Nested Relationships

This class is a nested type of *Class Graph::Lane*.

Class Documentation

class `rmf_traffic::agv::Graph::Lane::Node`

A *Lane Node* wraps up a *Waypoint* with constraints. The constraints stipulate the conditions for entering or exiting the lane to reach this waypoint.

Public Functions

Node(std::size_t *waypoint_index*, rmf_utils::clone_ptr<*Event*> *event* = nullptr, rmf_utils::clone_ptr<*OrientationConstraint*> *orientation* = nullptr)
Constructor

Parameters

- `waypoint_index`: The index of the waypoint for this *Node*
- `event`: An event that must happen before/after this *Node* is approached (before if it's an entry *Node* or after if it's an exit *Node*).
- `orientation`: Any orientation constraints for moving to/from this *Node* (depending on whether it's an entry *Node* or an exit *Node*).

Node(std::size_t *waypoint_index*, rmf_utils::clone_ptr<*OrientationConstraint*> *orientation*)
Constructor. The event parameter will be nullptr.

Parameters

- `waypoint_index`: The index of the waypoint for this *Node*
- `orientation`: Any orientation constraints for moving to/from this *Node* (depending on whether it's an entry *Node* or an exit *Node*).

std::size_t **waypoint_index**() **const**
Get the index of the waypoint that this *Node* is wrapped around.

const *Event* ***event**() **const**
Get a reference to an event that must occur before or after this *Node* is visited.

Note Before if this is an entry node or after if this is an exit node

Node &**event**(rmf_utils::clone_ptr<*Event*> *new_event*)
Set the event that must occur before or after this *Node* is visited.

const *OrientationConstraint* ***orientation_constraint**() **const**
Get the constraint on orientation that is tied to this *Node*.

Class Lane::Properties

- Defined in file_latest_rmf_traffic_include_rmf_traffic_agv_Graph.hpp

Nested Relationships

This class is a nested type of *Class Graph::Lane*.

Class Documentation

class rmf_traffic::agv::*Graph*::*Lane*::**Properties**

The *Lane Properties* class contains properties that apply across the full extent of the lane.

Public Functions

Properties()

Construct a default set of properties

- speed_limit: nullopt

std::optional<double> **speed_limit**() **const**

Get the speed limit along this lane. If a std::nullopt is returned, then there is no specified speed limit for the lane.

Properties &**speed_limit**(std::optional<double> *value*)

Set the speed limit along this lane. Providing a std::nullopt indicates that there is no speed limit for the lane.

Class Lane::Wait

- Defined in file_latest_rmf_traffic_include_rmf_traffic_agv_Graph.hpp

Nested Relationships

This class is a nested type of *Class Graph::Lane*.

Class Documentation

class rmf_traffic::agv::*Graph*::*Lane*::**Wait**

Public Functions

Wait(*Duration value*)

Constructor

Parameters

- `[in] duration`: How long the wait will be.

Duration **duration**() **const**

Get how long the wait will be.

Wait &**duration**(*Duration value*)

Set how long the wait will be.

Class Graph::OrientationConstraint

- Defined in file_latest_rmf_traffic_include_rmf_traffic_agv_Graph.hpp

Nested Relationships

This class is a nested type of *Class Graph*.

Class Documentation

class `rmf_traffic::agv::Graph::OrientationConstraint`
A class that implicitly specifies a constraint on the robot's orientation.

Public Types

enum Direction
Values:

enumerator Forward

enumerator Backward

Public Functions

virtual bool **apply**(Eigen::Vector3d &*position*, **const** Eigen::Vector2d &*course_vector*) **const** = 0
Apply the constraint to the given homogeneous position.

Return True if the constraint is satisfied with the new value of position. False if the constraint could not be satisfied.

Parameters

- `[inout] position`: The position which needs to be constrained. The function should modify this position such that it satisfies the constraint, if possible.
- `[in] course_vector`: The direction that the robot is travelling in. Given for informational purposes.

virtual rmf_utils::clone_ptr<*OrientationConstraint*> **clone**() **const** = 0
Clone this *OrientationConstraint*.

virtual ~OrientationConstraint() = default

Public Static Functions

static rmf_utils::clone_ptr<*OrientationConstraint*> **make**(std::vector<double> *acceptable_orientations*)
Make an orientation constraint that requires a specific value for the orientation.

static rmf_utils::clone_ptr<*OrientationConstraint*> **make**(*Direction* *direction*, **const** Eigen::Vector2d &*forward_vector*)
Make an orientation constraint that requires the vehicle to face forward or backward.

Class Graph::Waypoint

- Defined in file_latest_rmf_traffic_include_rmf_traffic_agv_Graph.hpp

Nested Relationships

This class is a nested type of *Class Graph*.

Class Documentation

class rmf_traffic::agv::*Graph*::**Waypoint**

Public Functions

const std::string &**get_map_name**() **const**
Get the name of the map that this *Waypoint* exists on.

Waypoint &**set_map_name**(std::string *map*)
Set the name of the map that this *Waypoint* exists on.

const Eigen::Vector2d &**get_location**() **const**
Get the position of this *Waypoint*.

Waypoint &**set_location**(Eigen::Vector2d *location*)
Set the position of this *Waypoint*.

bool **is_holding_point**() **const**
Returns true if this *Waypoint* can be used as a holding point for the vehicle, otherwise returns false.

Waypoint &**set_holding_point**(bool *_is_holding_point*)
Set whether this waypoint can be used as a holding point for the vehicle.

bool **is_passthrough_point**() **const**
Returns true if this *Waypoint* is a passthrough point, meaning a planner should not have a robot wait at this point, even just briefly to allow another robot to pass. Setting passthrough points reduces the branching factor of a planner, allowing it to run faster, at the cost of losing possible solutions to conflicts.

Waypoint &**set_passthrough_point**(bool *_is_passthrough*)
Set this *Waypoint* to be a passthrough point.

bool **is_parking_spot**() **const**
Returns true if this *Waypoint* is a parking spot. Parking spots are used when an emergency alarm goes off, and the robot is required to park itself.

Waypoint &**set_parking_spot**(bool *_is_parking_spot*)
Set this *Waypoint* to be a parking spot.

bool **is_charger**() **const**
Returns true if this *Waypoint* is a charger spot. Robots are routed to these spots when their batteries charge levels drop below the threshold value.

Waypoint &**set_charger**(bool *_is_charger*)
Set this *Waypoint* to be a parking spot.

std::size_t **index**() **const**
The index of this waypoint within the *Graph*. This cannot be changed after the waypoint is created.

const std::string ***name**() **const**
If this waypoint has a name, return a reference to it. If this waypoint does not have a name, return a nullptr.

The name of a waypoint can only be set using *add_key()* or *set_key()*.

std::string **name_or_index**(**const** std::string &*name_format* = "%s", **const** std::string &*index_format* = "#%d") **const**
If this waypoint has a name, the name will be returned. Otherwise it will return the waypoint index, formatted into a string based on the index_format argument.

Parameters

- `[in] name_format`: If this waypoint has an assigned name, the first instance of "%s" within name_format will be replaced with the name of the waypoint. If there is no s in the name_format string, then this function will simply return the name_format string as-is when the waypoint has a name.
- `[in] index_format`: If this waypoint does not have an assigned name, the first instance of "%d" within the index_format string will be replaced with the stringified decimal index value of the waypoint. If there is no "%d" in the index_format string, then this function will simply return the index_format string as-is when the waypoint does not have a name.

Class Interpolate

- Defined in file_latest_rmf_traffic_include_rmf_traffic_agv_Interpolate.hpp

Nested Relationships

Nested Types

- *Class Interpolate::Options*

Class Documentation

class rmf_traffic::agv::**Interpolate**

Public Static Functions

static *Trajectory* **positions**(**const** *VehicleTraits* &*traits*, *Time* *start_time*, **const** std::vector<Eigen::Vector3d> &*input_positions*, **const** *Options* &*options* = *Options*())

class Options

Public Functions

Options(bool *always_stop* = false, double *translation_thresh* = 1e-3, double *rotation_thresh* = 1.0 * M_PI / 180.0, double *corner_angle_thresh* = 1.0 * M_PI / 180.0)

Options &**set_always_stop**(bool *choice*)
The robot must always come to a complete stop at every position. When this is true, all other properties in the options will have no effect.

bool **always_stop**() **const**

Options &**set_translation_threshold**(double *dist*)
If a waypoint is closer than this distance to its prior or subsequent waypoint, then it is allowed to be skipped.

double **get_translation_threshold**() **const**
Get the translation threshold.

Options &**set_rotation_threshold**(double *angle*)
If a waypoint's orientation is closer than this angle to the prior or subsequent waypoint, then it is allowed to be skipped.

double **get_rotation_threshold**() **const**
Get the rotation threshold.

Options &**set_corner_angle_threshold**(double *angle*)
If two line segments make a corner that is greater than this angle, then the waypoint must not be ignored.

double **get_corner_angle_threshold**() **const**
Get the corner angle threshold.

Class Interpolate::Options

- Defined in file_latest_rmf_traffic_include_rmf_traffic_agv_Interpolate.hpp

Nested Relationships

This class is a nested type of *Class Interpolate*.

Class Documentation

class rmf_traffic::agv::*Interpolate*::**Options**

Public Functions

Options(bool *always_stop* = false, double *translation_thresh* = 1e-3, double *rotation_thresh* = 1.0 * M_PI / 180.0, double *corner_angle_thresh* = 1.0 * M_PI / 180.0)

Options &**set_always_stop**(bool *choice*)
The robot must always come to a complete stop at every position. When this is true, all other properties in the options will have no effect.

bool **always_stop**() **const**

Options &**set_translation_threshold**(double *dist*)
If a waypoint is closer than this distance to its prior or subsequent waypoint, then it is allowed to be skipped.

double **get_translation_threshold**() **const**
Get the translation threshold.

Options &**set_rotation_threshold**(double *angle*)
If a waypoint's orientation is closer than this angle to the prior or subsequent waypoint, then it is allowed to be skipped.

double **get_rotation_threshold**() **const**
Get the rotation threshold.

Options &**set_corner_angle_threshold**(double *angle*)
If two line segments make a corner that is greater than this angle, then the waypoint must not be ignored.

double **get_corner_angle_threshold**() **const**
Get the corner angle threshold.

Class invalid_traits_error

- Defined in file_latest_rmf_traffic_include_rmf_traffic_agv_Interpolate.hpp

Inheritance Relationships

Base Type

- public exception

Class Documentation

class rmf_traffic::agv::**invalid_traits_error** : **public** exception

This exception is thrown by *Interpolate* functions when the *VehicleTraits* that are provided cannot be interpolated as requested.

Public Functions

const char ***what**() **const noexcept override**

Class LaneClosure

- Defined in file_latest_rmf_traffic_include_rmf_traffic_agv_LaneClosure.hpp

Class Documentation

class rmf_traffic::agv::**LaneClosure**

This class describes the closure status of lanes in a *Graph*, i.e. whether a lane is open or closed. Open lanes can be used by the planner to reach a goal. The planner will not expand down a lane that is closed.

Public Functions

LaneClosure()

Default constructor.

By default, all lanes are open.

bool **is_open**(std::size_t *lane*) **const**

Check whether the lane corresponding to the given index is open.

Parameters

- [in] lane: The index for the lane of interest

bool **is_closed**(std::size_t *lane*) **const**

Check whether the lane corresponding to the given index is closed.

Parameters

- [in] lane: The index for the lane of interest

LaneClosure &**open**(std::size_t *lane*)

Set the lane corresponding to the given index to be open.

Parameters

- [in] lane: The index for the opening lane

LaneClosure &**close**(std::size_t *lane*)

Set the lane corresponding to the given index to be closed.

Parameters

- `[in] lane`: The index for the closing lane

std::size_t **hash**() **const**
Get an integer that uniquely describes the overall closure status of the graph lanes.

bool **operator==**(**const** *LaneClosure* &*other*) **const**
Equality comparison operator.

Class NegotiatingRouteValidator

- Defined in file_latest_rmf_traffic_include_rmf_traffic_agv_RouteValidator.hpp

Nested Relationships

Nested Types

- *Class NegotiatingRouteValidator::Generator*

Inheritance Relationships

Base Type

- `public rmf_traffic::agv::RouteValidator` (*Class RouteValidator*)

Class Documentation

class `rmf_traffic::agv::`**NegotiatingRouteValidator** : **public** rmf_traffic::agv::*RouteValidator*

Public Functions

NegotiatingRouteValidator &**mask**(schedule::*ParticipantId* *id*)
Mask the given Participant so that conflicts with it will be ignored. In the current implementation, only one participant can be masked at a time.

Parameters

- `[in] id`: The ID of a participant whose conflicts should be ignored when checking for collisions.

NegotiatingRouteValidator &**remove_mask**()
Remove any mask that has been applied using the *mask()* function.

NegotiatingRouteValidator **next**(schedule::*ParticipantId* *id*) **const**
Get a *NegotiatingRouteValidator* for the next rollout alternative offered by the given participant.

const schedule::*Negotiation*::*VersionedKeySequence* &**alternatives**() **const**
Get the set of child Table alternatives used by this *NegotiatingRouteValidator*.

operator bool() const
Implicitly cast this validator instance to true if it can be used as a validator. If it cannot be used as a validator, return false. This will have the opposite value of *end()*.

bool **end() const**
Return true if this validator object has gone past the end of its limits. Return false if it can still be used as a validator.

virtual rmf_utils::optional<Conflict> **find_conflict**(**const** *Route* &*route*) **const final**
If the specified route has a conflict with another participant, this will return the participant ID for the first conflict that gets identified. Otherwise it will return a nullopt.

Parameters

- `[in] route`: The route that is being checked.

virtual std::unique_ptr<*RouteValidator*> **clone**() **const final**
Create a clone of the underlying *RouteValidator* object.

class Generator
The *Generator* class begins the creation of *NegotiatingRouteValidator* instances. *NegotiatingRouteValidator* may be able to brach in multiple dimensions because of the rollout alternatives that are provided during a rejection.

Public Functions

Generator(schedule::*Negotiation*::*Table*::*ViewerPtr* *viewer*, rmf_traffic::*Profile* *profile*)
Constructor

This version is safe to use even if the participant being negotiated for is not in the schedule yet.

Parameters

- `[in] viewer`: A viewer for the Negotiation Table that the generated validators are concerned with
- `[in] profile`: The profile of the participant whose routes are being validated.

Generator(schedule::*Negotiation*::*Table*::*ViewerPtr* *viewer*)
Constructor

This version looks for the participant in the schedule to find its profile.

Parameters

- `[in] table`: A viewer for the Negotiation Table that the generated validators are concerned with

Generator &**ignore_unresponsive**(bool *val* = true)
Toggle whether to ignore "unresponsive" (also called "read-only") schedule participants when determining conflicts. By default, conflicts with unresponsive participants will be caught.

Generator &**ignore_bystanders**(bool *val* = true)
Toggle whether to ignore "bystanders" which means schedule participants that are not being involved in the negotiation. By default, conflicts with bystanders will be caught.

NegotiatingRouteValidator **begin**() **const**
Start with a *NegotiatingRouteValidator* that will use all the most preferred alternatives from every participant.

std::vector<rmf_utils::clone_ptr<*NegotiatingRouteValidator*>> **all**() **const**
Get all the Negotiating *Route* Validators that can be generated.

const std::vector<schedule::*ParticipantId*> &**alternative_sets**() **const**
Get the set of participants who have specified what their available rollouts are.

std::size_t **alternative_count**(schedule::*ParticipantId* *participant*) **const**
Get the number of alternative rollouts for the specified participant. This function will throw an excpetion if participant does not offer an alternative set.

Class NegotiatingRouteValidator::Generator

- Defined in file_latest_rmf_traffic_include_rmf_traffic_agv_RouteValidator.hpp

Nested Relationships

This class is a nested type of *Class NegotiatingRouteValidator*.

Class Documentation

class rmf_traffic::agv::*NegotiatingRouteValidator*::**Generator**
The *Generator* class begins the creation of *NegotiatingRouteValidator* instances. *NegotiatingRouteValidator* may be able to brach in multiple dimensions because of the rollout alternatives that are provided during a rejection.

Public Functions

Generator(schedule::*Negotiation*::*Table*::*ViewerPtr* *viewer*, rmf_traffic::*Profile* *profile*)
Constructor

This version is safe to use even if the participant being negotiated for is not in the schedule yet.

Parameters

- `[in] viewer`: A viewer for the Negotiation Table that the generated validators are concerned with
- `[in] profile`: The profile of the participant whose routes are being validated.

Generator(schedule::*Negotiation*::*Table*::*ViewerPtr* *viewer*)
Constructor

This version looks for the participant in the schedule to find its profile.

Parameters

- `[in] table`: A viewer for the Negotiation Table that the generated validators are concerned with

Generator &**ignore_unresponsive**(bool *val* = true)
Toggle whether to ignore "unresponsive" (also called "read-only") schedule participants when determining conflicts. By default, conflicts with unresponsive participants will be caught.

Generator &**ignore_bystanders**(bool *val* = true)
Toggle whether to ignore "bystanders" which means schedule participants that are not being involved in the negotiation. By default, conflicts with bystanders will be caught.

NegotiatingRouteValidator **begin**() **const**
Start with a *NegotiatingRouteValidator* that will use all the most preferred alternatives from every participant.

std::vector<rmf_utils::clone_ptr<*NegotiatingRouteValidator*>> **all**() **const**
Get all the Negotiating *Route* Validators that can be generated.

const std::vector<schedule::*ParticipantId*> &**alternative_sets**() **const**
Get the set of participants who have specified what their available rollouts are.

std::size_t **alternative_count**(schedule::*ParticipantId* *participant*) **const**
Get the number of alternative rollouts for the specified participant. This function will throw an excpetion if participant does not offer an alternative set.

Class Plan

- Defined in file_latest_rmf_traffic_include_rmf_traffic_agv_Planner.hpp

Nested Relationships

Nested Types

- *Struct Plan::Checkpoint*
- *Struct Plan::Progress*
- *Class Plan::Waypoint*

Class Documentation

class rmf_traffic::agv::**Plan**

Public Types

using Start = *Planner*::*Start*

using StartSet = *Planner*::*StartSet*

using Goal = *Planner*::*Goal*

using Options = *Planner*::*Options*

using Configuration = *Planner*::*Configuration*

using Result = *Planner*::*Result*

using Checkpoints = std::vector<*Checkpoint*>

Public Functions

const std::vector<*Route*> &**get_itinerary**() **const**
If this *Plan* is valid, this will return the trajectory of the successful plan. If the Start satisfies the Goal, then the itinerary will be empty.

const std::vector<*Waypoint*> &**get_waypoints**() **const**
If this plan is valid, this will return the waypoints of the successful plan.

const *Start* &**get_start**() **const**
Get the start condition that was used for this plan.

double **get_cost**() **const**
Get the final cost of this plan.

struct Checkpoint

Public Members

RouteId **route_id**

CheckpointId **checkpoint_id**

struct Progress

Public Members

std::size_t **graph_index**

Checkpoints **checkpoints**

rmf_traffic::*Time* **time**

class Waypoint
A *Waypoint* within a *Plan*.

This class helps to discretize a *Plan* based on the Waypoints belonging to the *agv::Graph*. Each *Graph::Waypoint* that the *Plan* stops or turns at will be accounted for by a *Plan::Waypoint*.

To indicate the intended orientation, each of these Waypoints provides an Eigen::Vector3d where the third element is the orientation.

The time that the position is meant to be arrived at is also given by the *Waypoint*.

Note Users are not allowed to make their own *Waypoint* instances, because it is too easy to accidentally get inconsistencies in the position and graph_index fields. Plan::Waypoints can only be created by *Plan* instances and can only be retrieved using *Plan::get_waypoints()*.

Public Functions

const Eigen::Vector3d &**position**() **const**
Get the position for this *Waypoint*.

rmf_traffic::*Time* **time**() **const**
Get the time for this *Waypoint*.

std::optional<std::size_t> **graph_index**() **const**
Get the graph index of this *Waypoint*.

const std::vector<std::size_t> &**approach_lanes**() **const**
Get the graph indices of the lanes that will be traversed on the way to this *Waypoint*. This will have multiple values if the robot is able to move straight through multiple lanes without stopping to reach this *Waypoint*. It will be empty if the robot does not need to traverse any lanes to reach this *Waypoint* (e.g. it is simply turning in place).

const std::vector<*Progress*> &**progress_checkpoints**() **const**
Points on the graph that will be passed along the way to this waypoint.

const *Checkpoints* &**arrival_checkpoints**() **const**
Points in the itinerary that have been reached when the robot arrives at this waypoint.

std::size_t **itinerary_index**() **const**

std::size_t **trajectory_index**() **const**

const *Graph*::*Lane*::*Event* ***event**() **const**
An event that should occur when this waypoint is reached.

const *Dependencies* &**dependencies**() **const**
The dependencies on other traffic participants that must be satisfied before leaving this waypoint.

Class Plan::Waypoint

- Defined in file_latest_rmf_traffic_include_rmf_traffic_agv_Planner.hpp

Nested Relationships

This class is a nested type of *Class Plan*.

Class Documentation

class rmf_traffic::agv::*Plan*::**Waypoint**
A *Waypoint* within a *Plan*.

This class helps to discretize a *Plan* based on the Waypoints belonging to the *agv::Graph*. Each *Graph::Waypoint* that the *Plan* stops or turns at will be accounted for by a *Plan::Waypoint*.

To indicate the intended orientation, each of these Waypoints provides an Eigen::Vector3d where the third element is the orientation.

The time that the position is meant to be arrived at is also given by the *Waypoint*.

Note Users are not allowed to make their own *Waypoint* instances, because it is too easy to accidentally get inconsistencies in the position and graph_index fields. Plan::Waypoints can only be created by *Plan* instances and can only be retrieved using *Plan::get_waypoints()*.

Public Functions

const Eigen::Vector3d &**position**() **const**
Get the position for this *Waypoint*.

rmf_traffic::*Time* **time**() **const**
Get the time for this *Waypoint*.

std::optional<std::size_t> **graph_index**() **const**
Get the graph index of this *Waypoint*.

const std::vector<std::size_t> &**approach_lanes**() **const**
Get the graph indices of the lanes that will be traversed on the way to this *Waypoint*. This will have multiple values if the robot is able to move straight through multiple lanes without stopping to reach this *Waypoint*. It will be empty if the robot does not need to traverse any lanes to reach this *Waypoint* (e.g. it is simply turning in place).

const std::vector<*Progress*> &**progress_checkpoints**() **const**
Points on the graph that will be passed along the way to this waypoint.

const *Checkpoints* &**arrival_checkpoints**() **const**
Points in the itinerary that have been reached when the robot arrives at this waypoint.

std::size_t **itinerary_index**() **const**

std::size_t **trajectory_index**() **const**

const *Graph*::*Lane*::*Event* ***event**() **const**
An event that should occur when this waypoint is reached.

const *Dependencies* &**dependencies**() **const**
The dependencies on other traffic participants that must be satisfied before leaving this waypoint.

Class Planner

- Defined in file_latest_rmf_traffic_include_rmf_traffic_agv_Planner.hpp

Nested Relationships

Nested Types

- *Class Planner::Configuration*
- *Class Planner::Debug*
- *Struct Debug::Node*
- *Struct Node::Compare*
- *Class Debug::Progress*
- *Class Planner::Goal*
- *Class Planner::Options*

- *Class Planner::Result*
- *Class Planner::Start*

Class Documentation

class rmf_traffic::agv::**Planner**

Public Types

using StartSet = std::vector<*Start*>

Public Functions

Planner(*Configuration config*, *Options default_options*)
Constructor

Parameters

- `[in] config`: This is the *Configuration* for the *Planner*. The *Planner* instance will maintain a cache while it performs planning requests. This cache will offer potential speed ups to subsequent planning requests, but the correctness of the cache depends on the fields in the *Configuration* to remain constant. Therefore you are not permitted to modify a *Planner*'s *Configuration* after the *Planner* is constructed. To change the planning *Configuration*, you will need to create a new *Planner* instance with the desired *Configuration*.
- `[in] default_options`: Unlike the *Configuration*, you are allowed to change a *Planner*'s *Options*. The parameter given here will be used as the default options, so you can set them here and then forget about them. These options can be overriden each time you request a plan.

const *Configuration* &**get_configuration**() **const**
Get a const reference to the configuration for this *Planner*. Note that the configuration of a planner cannot be changed once it is set.

Note The *Planner* maintains a cache that allows searches to become progressively faster. This cache depends on the fields in the *Planner*'s configuration, so those fields cannot be changed without invalidating that cache. To plan using a different configuration, you should create a new *Planner* instance with the desired configuration.

Planner &**set_default_options**(*Options default_options*)
Change the default planning options.

Options &**get_default_options**()
Get a mutable reference to the default planning options.

const *Options* &**get_default_options**() **const**
Get a const reference to the default planning options.

Result **plan**(**const** *Start* &*start*, *Goal goal*) **const**
Produce a plan for the given starting conditions and goal. The default *Options* of this *Planner* instance will be used.

Parameters

- `[in] start`: The starting conditions
- `[in] goal`: The goal conditions

Result **plan**(**const** *Start* &*start*, *Goal goal*, *Options options*) **const**
Product a plan for the given start and goal conditions. Override the default options.

Parameters

- `[in] start`: The starting conditions
- `[in] goal`: The goal conditions
- `[in] options`: The *Options* to use for this plan. This overrides the default *Options* of the *Planner* instance.

Result **plan**(**const** *StartSet* &*starts*, *Goal goal*) **const**
Produces a plan for the given set of starting conditions and goal. The default *Options* of this *Planner* instance will be used.

The planner will choose the start condition that allows for the shortest plan (not the one that finishes the soonest according to wall time).

At least one start must be specified or else this is guaranteed to return a nullopt.

Parameters

- `[in] starts`: The set of available starting conditions
- `[in] goal`: The goal conditions

Result **plan**(**const** *StartSet* &*starts*, *Goal goal*, *Options options*) **const**
Produces a plan for the given set of starting conditions and goal. Override the default options.

The planner will choose the start condition that allows for the shortest plan (not the one that finishes the soonest according to wall time).

At least one start must be specified or else this is guaranteed to return a nullopt.

Parameters

- `[in] starts`: The starting conditions
- `[in] goal`: The goal conditions
- `[in] options`: The options to use for this plan. This overrides the default *Options* of the *Planner* instance.

Result **setup**(**const** *Start* &*start*, *Goal goal*) **const**
Set up a planning job, but do not start iterating.

See plan(const Start&, Goal)

Result **setup**(**const** *Start* &*start*, *Goal goal*, *Options options*) **const**
Set up a planning job, but do not start iterating.

See plan(const Start&, Goal, Options)

Result **setup**(**const** *StartSet* &*starts*, *Goal* *goal*) **const**
Set up a planning job, but do not start iterating.

See plan(const StartSet&, Goal)

Result **setup**(**const** *StartSet* &*starts*, *Goal* *goal*, *Options* *options*) **const**
Set up a planning job, but do not start iterating.

See plan(const StartSet&, Goal, Options)

class Configuration
The *Configuration* class contains planning parameters that are immutable for each *Planner* instance.

These parameters generally describe the capabilities or behaviors of the AGV that is being planned for, so they shouldn't need to change in between plans anyway.

Public Functions

Configuration(*Graph* *graph*, *VehicleTraits* *traits*, *Interpolate*::*Options* *interpolation* = *Interpolate*::*Options*())
Constructor

Parameters
- `[in] vehicle_traits`: The traits of the vehicle that is being planned for
- `[in] graph`: The graph which is being planned over
- `[in] interpolation`: The options for how the planner will perform trajectory interpolation

Configuration &**graph**(*Graph* *graph*)
Set the graph to use for planning.

Graph &**graph**()
Get a mutable reference to the graph.

const *Graph* &**graph**() **const**
Get a const reference to the graph.

Configuration &**vehicle_traits**(*VehicleTraits* *traits*)
Set the vehicle traits to use for planning.

VehicleTraits &**vehicle_traits**()
Get a mutable reference to the vehicle traits.

const *VehicleTraits* &**vehicle_traits**() **const**
Get a const reference to the vehicle traits.

Configuration &**interpolation**(*Interpolate*::*Options* *interpolate*)
Set the interpolation options for the planner.

Interpolate::*Options* &**interpolation**()
Get a mutable reference to the interpolation options.

const *Interpolate*::*Options* &**interpolation**() **const**
Get a const reference to the interpolation options.

Configuration &**lane_closures**(*LaneClosure closures*)
Set the lane closures for the graph. The planner will not attempt to expand down any lanes that are closed.

LaneClosure &**lane_closures**()
Get a mutable reference to the *LaneClosure* setting.

const *LaneClosure* &**lane_closures**() **const**
Get a const reference to the *LaneClosure* setting.

Configuration &**traversal_cost_per_meter**(double *value*)
How much the cost should increase per meter travelled. Besides this, cost is measured by the number of seconds spent travelling.

double **traversal_cost_per_meter**() **const**
Get the traversal cost.

class Debug
This class exists only for debugging purposes. It is not to be used in live production, and its API is to be considered unstable at all times. Any minor version increment

Public Types

using ConstNodePtr = std::shared_ptr<**const** *Node*>

Public Functions

Debug(**const** *Planner* &*planner*)
Create a debugger for a planner.

Progress **begin**(**const** std::vector<*Start*> &*starts*, *Goal goal*, *Options options*) **const**
Begin debugging a plan. Call step() on the *Progress* object until it returns a plan or until the queue is empty (the *Progress* object can be treated as a boolean for this purpose).

Public Static Functions

static std::size_t **queue_size**(**const** *Planner*::*Result* &*result*)
Get the current size of the frontier queue of a *Planner Result*.

static std::size_t **expansion_count**(**const** *Planner*::*Result* &*result*)
Get the number of search nodes that have been expanded for a *Planner Result*

static std::size_t **node_count**(**const** *Planner*::*Result* &*result*)
Get the current number of nodes that have been created for this *Planner Result*. This is equal to queue_size(r) + expansion_count(r).

struct Node
A *Node* in the planning search. A final Planning solution will be a chain of these Nodes, aggregated into a *Plan* data structure.

Public Types

using SearchQueue = std::priority_queue<*ConstNodePtr*, std::vector<*ConstNodePtr*>, *Compare*>

using Vector = std::vector<*ConstNodePtr*>

Public Members

ConstNodePtr **parent**
The parent of this *Node*. If this is a nullptr, then this was a starting node.

std::vector<*Route*> **route_from_parent**
The route that goes from the parent *Node* to this *Node*.

double **remaining_cost_estimate**
An estimate of the remaining cost, based on the heuristic.

double **current_cost**
The actual cost that has accumulated on the way to this *Node*.

rmf_utils::optional<std::size_t> **waypoint**
The waypoint that this *Node* stops on.

double **orientation**
The orientation that this *Node* ends with.

agv::*Graph*::*Lane*::*EventPtr* **event**
A pointer to an event that occured on the way to this *Node*.

rmf_utils::optional<std::size_t> **start_set_index**
If this is a starting node, then this will be the index.

std::size_t **id**
A unique ID that sticks with this node for its entire lifetime. This will also (roughly) reflect the order of node creation.

struct Compare

Public Functions

inline bool **operator()**(**const** *ConstNodePtr* &*a*, **const** *ConstNodePtr* &*b*)

class Progress

Public Functions

rmf_utils::optional<*Plan*> **step**()
Step the planner forward one time. This will expand the current highest priority *Node* in the queue and move it to the back of expanded_nodes. The nodes that result from the expansion will all be added to the queue.

inline operator bool() **const**
Implicitly cast the *Progress* instance to a boolean. The value will be true if the plan can keep expanding, and it will be false if it cannot expand any further.

After finding a solution, it may be possible to continue expanding, but there is no point because the first solution returned is guaranteed to be the optimal one.

const *Node*::*SearchQueue* &**queue**() **const**
A priority queue of unexpanded Nodes. They are sorted based on g(n)+h(n) in ascending order (see *Node::Compare*).

const *Node*::*Vector* &**expanded_nodes**() **const**
The set of Nodes that have been expanded. They are sorted in the order that they were chosen for expansion.

const *Node*::*Vector* &**terminal_nodes**() **const**
The set of Nodes which terminated, meaning it was not possible to expand from them.

class Goal
Describe the goal conditions of a plan.

Public Functions

Goal(std::size_t *goal_waypoint*)
Constructor

Note With this constructor, any final orientation will be accepted.
Parameters
- `[in] goal_waypoint`: The waypoint that the AGV needs to reach.

Goal(std::size_t *goal_waypoint*, double *goal_orientation*)
Constructor

Parameters
- `[in] goal_waypoint`: The waypoint that the AGV needs to reach.
- `[in] goal_orientation`: The orientation that the AGV needs to end with.

Goal(std::size_t *goal_waypoint*, std::optional<rmf_traffic::*Time*> *minimum_time*, std::optional<double> *goal_orientation* = std::nullopt)
Constructor

Parameters
- `[in] goal_waypoint`: The waypoint that the AGV needs to reach.
- `[in] minimum_time`: The AGV must be on the goal waypoint at or after this time for the plan to be successful. This is useful if a robot needs to wait at a location, but you want it to give way for other robots.
- `[in] goal_orientation`: An optional goal orientation that the AGV needs to end with.

Goal &**waypoint**(std::size_t *goal_waypoint*)
Set the goal waypoint.

std::size_t **waypoint**() **const**
Get the goal waypoint.

Goal &**orientation**(double *goal_orientation*)
Set the goal orientation.

Goal &**any_orientation**()
Accept any orientation for the final goal.

const double ***orientation**() **const**
Get a reference to the goal orientation (or a nullptr if any orientation is acceptable).

Goal &**minimum_time**(std::optional<rmf_traffic::*Time*> *value*)
Set the minimum time for the goal. Pass in a nullopt to remove the minimum time.

std::optional<rmf_traffic::*Time*> **minimum_time**() **const**
Get the minimum time for the goal (or a nullopt is there is no minimum time).

class Options
The *Options* class contains planning parameters that can change between each planning attempt.

Public Functions

Options(rmf_utils::clone_ptr<*RouteValidator*> *validator*, *Duration* *min_hold_time* = *DefaultMinHoldingTime*, std::shared_ptr<**const** std::atomic_bool> *interrupt_flag* = nullptr, std::optional<double> *maximum_cost_estimate* = std::nullopt, std::optional<std::size_t> *saturation_limit* = std::nullopt)
Constructor

Parameters
- `[in] validator`: A validator to check the validity of the planner's branching options.
- `[in] min_hold_time`: The minimum amount of time that the planner should spend waiting at holding points. Smaller values will make the plan more aggressive about being time-optimal, but the plan may take longer to produce. Larger values will add some latency to the execution of the plan as the robot may wait at a holding point longer than necessary, but the plan will usually be generated more quickly.
- `[in] interrupt_flag`: A pointer to a flag that should be used to interrupt the planner if it has been running for too long. If the planner should run indefinitely, then pass in a nullptr.
- `[in] maximum_cost_estimate`: A cap on how high the best possible solution's cost can be. If the cost of the best possible solution ever exceeds this value, then the planner will interrupt itself, no matter what the state of the interrupt_flag is. Set this to nullopt to specify that there should not be a cap.
- `[in] saturation_limit`: A cap on how many search nodes the planner is allowed to produce.

Options(rmf_utils::clone_ptr<*RouteValidator*> *validator*, *Duration* *min_hold_time*, std::function<bool>
> *interrupter*std::optional<double> *maximum_cost_estimate* = std::nullopt, std::optional<std::size_t> *saturation_limit* = std::nulloptConstructor

Parameters
- `[in] validator`: A validator to check the validity of the planner's branching options.
- `[in] validator`: A validator to check the validity of the planner's branching options.
- `[in] min_hold_time`: The minimum amount of time that the planner should spend waiting at holding points. Smaller values will make the plan more aggressive about being time-optimal, but the plan may take longer to produce. Larger values will add some latency to the execution of the plan as the robot may wait at a holding point longer than necessary, but the plan will usually be generated more quickly.
- `[in] interrupter`: A function that can determine whether the planning should be interrupted. This is an alternative to using the interrupt_flag.
- `[in] maximum_cost_estimate`: A cap on how high the best possible solution's cost can be. If the cost of the best possible solution ever exceeds this value, then the planner will interrupt itself, no matter what the state of the interrupt_flag is. Set this to nullopt to specify that there should not be a cap.
- `[in] saturation_limit`: A cap on how many search nodes the planner is allowed to produce.

Options &**validator**(rmf_utils::clone_ptr<*RouteValidator*> *v*)
Set the route validator.

const rmf_utils::clone_ptr<*RouteValidator*> &**validator**() **const**
Get the route validator.

Options &**minimum_holding_time**(*Duration holding_time*)
Set the minimum amount of time to spend waiting at holding points.

Duration **minimum_holding_time**() **const**
Get the minimum amount of time to spend waiting at holding points.

Options &**interrupter**(std::function<bool)
> *cb*Set an interrupter callback that can indicate to the planner if it should stop trying to plan.

Warning Using this function will replace anything that was given to interrupt_flag, and it will nullify the *interrupt_flag()* field.

const std::function<bool()> &**interrupter**
constGet the interrupter that will be used in these *Options*.

Options &**interrupt_flag**(std::shared_ptr<**const** std::atomic_bool> *flag*)
Set an interrupt flag to stop this planner if it has run for too long.

Warning Using this function will replace anything that was given to interrupter.

const std::shared_ptr<**const** std::atomic_bool> &**interrupt_flag**() **const**
Get the interrupt flag that will stop this planner if it has run for too long.

Options &**maximum_cost_estimate**(std::optional<double> *value*)
Set the maximum cost estimate that the planner should allow. If the cost estimate of the best possible plan that the planner could produce ever exceeds this value, the planner will pause itself (but this will not be considered an interruption).

std::optional<double> **maximum_cost_estimate**() **const**
Get the maximum cost estimate that the planner will allow.

Options &**saturation_limit**(std::optional<std::size_t> *value*)
Set the saturation limit for the planner. If the planner produces more search nodes than this limit, then the planning will stop.

std::optional<std::size_t> **saturation_limit**() **const**
Get the saturation limit.

Options &**dependency_window**(std::optional<*Duration*> *value*)
Set the dependency window for generated plans. Any potential conflicts with the generated plan that happen within this window will be added as dependencies to the plan waypoints. If set to a nullopt, the plan will not have any dependencies.

std::optional<*Duration*> **dependency_window**() **const**
Dependency window for the planner.

Options &**dependency_resoution**(*Duration value*)
Set the dependency resolution for generated plans. To check for dependencies, the planner will step the generated routes back in time by this value and check for conflicts. Detected conflicts get added to the list of dependencies. This backstepping happens until dependency_window is reached. If dependency_window is nullopt, this value will not be used.

Duration **dependency_resolution**() **const**
Get the dependency resolution for generated plans.

Public Static Attributes

static constexpr *Duration* **DefaultMinHoldingTime** = std::chrono::seconds(1)

class Result

Public Functions

bool **success**() **const**
True if a plan was found and this *Result* can be dereferenced to obtain a plan.

bool **disconnected**() **const**
True if there is no feasible path that connects the start to the goal. In this case, a plan will never be found.

operator bool() **const**
Implicitly cast the result to a boolean. It will return true if a plan was found, otherwise it will return false.

const *Plan* ***operator->**() **const**
If the *Result* was successful, drill into the plan.

const *Plan* &**operator***() **const** &
If the *Result* was successful, get a reference to the plan.

Plan &&**operator***() &&
If the *Result* was successful, move the plan.

const *Plan* &&**operator***() **const** &&
If the *Result* was successful, get a reference to the plan.

Result **replan**(**const** *Start* &*new_start*) **const**
Replan to the same goal from a new start location using the same options as before.

Parameters
- `[in] new_start`: The starting conditions that should be used for replanning.

Result **replan**(**const** *Start* &*new_start*, *Options* *new_options*) **const**
Replan to the same goal from a new start location using a new set of options.

Parameters
- `[in] new_start`: The starting conditions that should be used for replanning.
- `[in] new_options`: The options that should be used for replanning.

Result **replan**(**const** *StartSet* &*new_starts*) **const**
Replan to the same goal from a new set of start locations using the same options.

Parameters
- `[in] new_starts`: The set of starting conditions that should be used for replanning.

Result **replan**(**const** *StartSet* &*new_starts*, *Options* *new_options*) **const**
Replan to the same goal from a new set of start locations using a new set of options.

Parameters
- `[in] new_starts`: The set of starting conditions that should be used for replanning.
- `[in] new_options`: The options that should be used for replanning.

Result **setup**(**const** *Start* &*new_start*) **const**
Set up a new planning job to the same goal, but do not start iterating.

See replan(const Start&)

Result **setup**(**const** *Start* &*new_start*, *Options* *new_options*) **const**
Set up a new planning job to the same goal, but do not start iterating.

See replan(const Start&, Options)

Result **setup**(**const** *StartSet* &*new_starts*) **const**
Set up a new planning job to the same goal, but do not start iterating.

See replan(const StartSet&)

Result **setup**(**const** *StartSet* &*new_starts*, *Options* *new_options*) **const**
Set up a new planning job to the same goal, but do not start iterating.

See replan(const StartSet&, Options)

bool **resume**()
Resume planning if the planner was paused.

Return true if a plan has been found, false otherwise.

bool **resume**(std::shared_ptr<**const** std::atomic_bool> *interrupt_flag*)
Resume planning if the planner was paused.

Return true if a plan has been found, false otherwise.
Parameters
- `[in] interrupt_flag`: A new interrupt flag to listen to while planning.

Options &**options**()
Get a mutable reference to the options that will be used by this planning task.

const *Options* &**options**() **const**
Get the options that will be used by this planning task.

Result &**options**(*Options* *new_options*)
Change the options to be used by this planning task.

std::optional<double> **cost_estimate**() **const**
Get the best cost estimate of the current state of this planner result. This is the value of the lowest f(n)=g(n)+h(n) in the planner's queue. If the node queue of this planner result is empty, this will return a nullopt.

double **initial_cost_estimate**() **const**
Get the cost estimate that was initially computed for this plan. If no valid starts were provided, then this will return infinity.

std::optional<double> **ideal_cost**() **const**
Get the cost that this plan would have if there is no traffic. If the plan is impossible (e.g. the starts are disconnected from the goal) this will return a nullopt.

const std::vector<*Start*> &**get_starts**() **const**
Get the start conditions that were given for this planning task.

const *Goal* &**get_goal**() **const**
Get the goal for this planning task.

const *Configuration* &**get_configuration**() **const**
If this *Plan* is valid, this will return the *Planner::Configuration* that was used to produce it.

If *replan()* is called, this *Planner::Configuration* will be used to produce the new *Plan*.

bool **interrupted**() **const**
This will return true if the planning failed because it was interrupted. Otherwise it will return false.

bool **saturated**() **const**
This will return true if the planner has reached its saturation limit.

std::vector<schedule::*ParticipantId*> **blockers**() **const**
This is a list of schedule Participants who blocked the planning effort. Blockers do not necessarily prevent a solution from being found, but they do prevent the optimal solution from being available.

class Start
Describe the starting conditions of a plan.

Public Functions

Start(*Time* *initial_time*, std::size_t *initial_waypoint*, double *initial_orientation*, std::optional<Eigen::Vector2d> *location* = std::nullopt, std::optional<std::size_t> *initial_lane* = std::nullopt)
Constructor

Parameters
- `[in] inital_time`: The starting time of the plan.
- `[in] initial_waypoint`: The waypoint index that the plan will begin from.
- `[in] initial_orientation`: The orientation that the AGV will start with.
- `[in] initial_location`: Optional field to specify if the robot is not starting directly on the initial_waypoint location. When planning from this initial_location to the initial_waypoint the planner will assume it has an unconstrained lane.
- `[in] initial_lane`: Optional field to specify if the robot is starting in a certain lane. This will only be used if an initial_location is specified.

Start &**time**(*Time* *initial_time*)
Set the starting time of a plan.

Time **time**() **const**
Get the starting time.

Start &**waypoint**(std::size_t *initial_waypoint*)
Set the starting waypoint of a plan.

std::size_t **waypoint**() **const**
Get the starting waypoint.

Start &**orientation**(double *initial_orientation*)
Set the starting orientation of a plan.

double **orientation**() **const**
Get the starting orientation.

const std::optional<Eigen::Vector2d> &**location**() **const**
Get the starting location, if one was specified.

Start &**location**(std::optional<Eigen::Vector2d> *initial_location*)
Set the starting location, or remove it by using std::nullopt.

const std::optional<std::size_t> &**lane**() **const**
Get the starting lane, if one was specified.

Start &**lane**(std::optional<std::size_t> *initial_lane*)
Set the starting lane, or remove it by using std::nullopt.

Class Planner::Configuration

- Defined in file_latest_rmf_traffic_include_rmf_traffic_agv_Planner.hpp

Nested Relationships

This class is a nested type of *Class Planner*.

Class Documentation

class rmf_traffic::agv::*Planner*::**Configuration**
The *Configuration* class contains planning parameters that are immutable for each *Planner* instance.

These parameters generally describe the capabilities or behaviors of the AGV that is being planned for, so they shouldn't need to change in between plans anyway.

Public Functions

Configuration(*Graph graph*, *VehicleTraits traits*, *Interpolate*::*Options interpolation* = *Interpolate*::*Options*())
Constructor

Parameters

- [in] vehicle_traits: The traits of the vehicle that is being planned for
- [in] graph: The graph which is being planned over
- [in] interpolation: The options for how the planner will perform trajectory interpolation

Configuration &**graph**(*Graph graph*)
Set the graph to use for planning.

Graph &**graph**()
Get a mutable reference to the graph.

const *Graph* &**graph**() **const**
Get a const reference to the graph.

Configuration &**vehicle_traits**(*VehicleTraits traits*)
Set the vehicle traits to use for planning.

VehicleTraits &**vehicle_traits**()
Get a mutable reference to the vehicle traits.

const *VehicleTraits* &**vehicle_traits**() **const**
Get a const reference to the vehicle traits.

Configuration &**interpolation**(*Interpolate*::*Options interpolate*)
Set the interpolation options for the planner.

Interpolate::*Options* &**interpolation**()
Get a mutable reference to the interpolation options.

const *Interpolate*::*Options* &**interpolation**() **const**
Get a const reference to the interpolation options.

Configuration &**lane_closures**(*LaneClosure closures*)
Set the lane closures for the graph. The planner will not attempt to expand down any lanes that are closed.

LaneClosure &**lane_closures**()
Get a mutable reference to the *LaneClosure* setting.

const *LaneClosure* &**lane_closures**() **const**
Get a const reference to the *LaneClosure* setting.

Configuration &**traversal_cost_per_meter**(double *value*)
How much the cost should increase per meter travelled. Besides this, cost is measured by the number of seconds spent travelling.

double **traversal_cost_per_meter**() **const**
Get the traversal cost.

Class Planner::Debug

- Defined in file_latest_rmf_traffic_include_rmf_traffic_agv_debug_debug_Planner.hpp

Nested Relationships

This class is a nested type of *Class Planner*.

Nested Types

- *Struct Debug::Node*
- *Struct Node::Compare*
- *Class Debug::Progress*

Class Documentation

class rmf_traffic::agv::*Planner*::**Debug**
This class exists only for debugging purposes. It is not to be used in live production, and its API is to be considered unstable at all times. Any minor version increment

Public Types

using ConstNodePtr = std::shared_ptr<**const** *Node*>

Public Functions

Debug(**const** *Planner* &*planner*)
Create a debugger for a planner.

Progress **begin**(**const** std::vector<*Start*> &*starts*, *Goal goal*, *Options options*) **const**
Begin debugging a plan. Call step() on the *Progress* object until it returns a plan or until the queue is empty (the *Progress* object can be treated as a boolean for this purpose).

Public Static Functions

static std::size_t **queue_size**(**const** *Planner*::*Result* &*result*)
Get the current size of the frontier queue of a *Planner Result*.

static std::size_t **expansion_count**(**const** *Planner*::*Result* &*result*)
Get the number of search nodes that have been expanded for a *Planner Result*

static std::size_t **node_count**(**const** *Planner*::*Result* &*result*)
Get the current number of nodes that have been created for this *Planner Result*. This is equal to queue_size(r) + expansion_count(r).

struct Node
A *Node* in the planning search. A final Planning solution will be a chain of these Nodes, aggregated into a *Plan* data structure.

Public Types

using SearchQueue = std::priority_queue<*ConstNodePtr*, std::vector<*ConstNodePtr*>, *Compare*>

using Vector = std::vector<*ConstNodePtr*>

Public Members

ConstNodePtr **parent**
The parent of this *Node*. If this is a nullptr, then this was a starting node.

std::vector<*Route*> **route_from_parent**
The route that goes from the parent *Node* to this *Node*.

double **remaining_cost_estimate**
An estimate of the remaining cost, based on the heuristic.

double **current_cost**
The actual cost that has accumulated on the way to this *Node*.

rmf_utils::optional<std::size_t> **waypoint**
The waypoint that this *Node* stops on.

double **orientation**
The orientation that this *Node* ends with.

agv::*Graph*::*Lane*::*EventPtr* **event**
A pointer to an event that occured on the way to this *Node*.

rmf_utils::optional<std::size_t> **start_set_index**
If this is a starting node, then this will be the index.

std::size_t **id**
A unique ID that sticks with this node for its entire lifetime. This will also (roughly) reflect the order of node creation.

struct Compare

Public Functions

inline bool **operator()**(**const** *ConstNodePtr* &*a*, **const** *ConstNodePtr* &*b*)

class Progress

Public Functions

rmf_utils::optional<*Plan*> **step**()
Step the planner forward one time. This will expand the current highest priority *Node* in the queue and move it to the back of expanded_nodes. The nodes that result from the expansion will all be added to the queue.

inline operator bool() **const**
Implicitly cast the *Progress* instance to a boolean. The value will be true if the plan can keep expanding, and it will be false if it cannot expand any further.

After finding a solution, it may be possible to continue expanding, but there is no point because the first solution returned is guaranteed to be the optimal one.

const *Node*::*SearchQueue* &**queue**() **const**
A priority queue of unexpanded Nodes. They are sorted based on g(n)+h(n) in ascending order (see *Node::Compare*).

const *Node*::*Vector* &**expanded_nodes**() **const**
The set of Nodes that have been expanded. They are sorted in the order that they were chosen for expansion.

const *Node*::*Vector* &**terminal_nodes**() **const**
The set of Nodes which terminated, meaning it was not possible to expand from them.

Class Debug::Progress

- Defined in file_latest_rmf_traffic_include_rmf_traffic_agv_debug_debug_Planner.hpp

Nested Relationships

This class is a nested type of *Class Planner::Debug*.

Class Documentation

class rmf_traffic::agv::*Planner*::*Debug*::**Progress**

Public Functions

rmf_utils::optional<*Plan*> **step**()
Step the planner forward one time. This will expand the current highest priority *Node* in the queue and move it to the back of expanded_nodes. The nodes that result from the expansion will all be added to the queue.

inline operator bool() **const**
Implicitly cast the *Progress* instance to a boolean. The value will be true if the plan can keep expanding, and it will be false if it cannot expand any further.

After finding a solution, it may be possible to continue expanding, but there is no point because the first solution returned is guaranteed to be the optimal one.

const *Node*::*SearchQueue* &**queue**() **const**
A priority queue of unexpanded Nodes. They are sorted based on g(n)+h(n) in ascending order (see *Node::Compare*).

const *Node*::*Vector* &**expanded_nodes**() **const**
The set of Nodes that have been expanded. They are sorted in the order that they were chosen for expansion.

const *Node*::*Vector* &**terminal_nodes**() **const**
The set of Nodes which terminated, meaning it was not possible to expand from them.

Class Planner::Goal

- Defined in file_latest_rmf_traffic_include_rmf_traffic_agv_Planner.hpp

Nested Relationships

This class is a nested type of *Class Planner*.

Class Documentation

class rmf_traffic::agv::*Planner*::**Goal**
Describe the goal conditions of a plan.

Public Functions

Goal(std::size_t *goal_waypoint*)
Constructor

Note With this constructor, any final orientation will be accepted.

Parameters

- `[in] goal_waypoint`: The waypoint that the AGV needs to reach.

Goal(std::size_t *goal_waypoint*, double *goal_orientation*)
Constructor

Parameters

- `[in] goal_waypoint`: The waypoint that the AGV needs to reach.
- `[in] goal_orientation`: The orientation that the AGV needs to end with.

Goal(std::size_t *goal_waypoint*, std::optional<rmf_traffic::*Time*> *minimum_time*, std::optional<double> *goal_orientation* = std::nullopt)
Constructor

Parameters

- `[in] goal_waypoint`: The waypoint that the AGV needs to reach.
- `[in] minimum_time`: The AGV must be on the goal waypoint at or after this time for the plan to be successful. This is useful if a robot needs to wait at a location, but you want it to give way for other robots.
- `[in] goal_orientation`: An optional goal orientation that the AGV needs to end with.

Goal &**waypoint**(std::size_t *goal_waypoint*)
Set the goal waypoint.

std::size_t **waypoint**() **const**
Get the goal waypoint.

Goal &**orientation**(double *goal_orientation*)
Set the goal orientation.

Goal &**any_orientation**()
Accept any orientation for the final goal.

const double ***orientation**() **const**
Get a reference to the goal orientation (or a nullptr if any orientation is acceptable).

Goal &**minimum_time**(std::optional<rmf_traffic::*Time*> *value*)
Set the minimum time for the goal. Pass in a nullopt to remove the minimum time.

std::optional<rmf_traffic::*Time*> **minimum_time**() **const**
Get the minimum time for the goal (or a nullopt is there is no minimum time).

Class Planner::Options

- Defined in file_latest_rmf_traffic_include_rmf_traffic_agv_Planner.hpp

Nested Relationships

This class is a nested type of *Class Planner*.

Class Documentation

class rmf_traffic::agv::*Planner*::**Options**

The *Options* class contains planning parameters that can change between each planning attempt.

Public Functions

Options(rmf_utils::clone_ptr<*RouteValidator*> *validator*, *Duration* *min_hold_time* = *DefaultMinHoldingTime*, std::shared_ptr<**const** std::atomic_bool> *interrupt_flag* = nullptr, std::optional<double> *maximum_cost_estimate* = std::nullopt, std::optional<std::size_t> *saturation_limit* = std::nullopt)

Constructor

Parameters

- `[in] validator`: A validator to check the validity of the planner's branching options.
- `[in] min_hold_time`: The minimum amount of time that the planner should spend waiting at holding points. Smaller values will make the plan more aggressive about being time-optimal, but the plan may take longer to produce. Larger values will add some latency to the execution of the plan as the robot may wait at a holding point longer than necessary, but the plan will usually be generated more quickly.
- `[in] interrupt_flag`: A pointer to a flag that should be used to interrupt the planner if it has been running for too long. If the planner should run indefinitely, then pass in a nullptr.
- `[in] maximum_cost_estimate`: A cap on how high the best possible solution's cost can be. If the cost of the best possible solution ever exceeds this value, then the planner will interrupt itself, no matter what the state of the interrupt_flag is. Set this to nullopt to specify that there should not be a cap.
- `[in] saturation_limit`: A cap on how many search nodes the planner is allowed to produce.

Options(rmf_utils::clone_ptr<*RouteValidator*> *validator*, *Duration* *min_hold_time*, std::function<bool) > *interrupter*std::optional<double> *maximum_cost_estimate* = std::nullopt, std::optional<std::size_t> *saturation_limit* = std::nulloptConstructor

Parameters

- `[in] validator`: A validator to check the validity of the planner's branching options.
- `[in] validator`: A validator to check the validity of the planner's branching options.
- `[in] min_hold_time`: The minimum amount of time that the planner should spend waiting at holding points. Smaller values will make the plan more aggressive about being time-optimal, but the plan may take longer to produce. Larger values will add some latency to the execution of the plan as the robot may wait at a holding point longer than necessary, but the plan will usually be generated more quickly.

- `[in] interrupter`: A function that can determine whether the planning should be interrupted. This is an alternative to using the interrupt_flag.
- `[in] maximum_cost_estimate`: A cap on how high the best possible solution's cost can be. If the cost of the best possible solution ever exceeds this value, then the planner will interrupt itself, no matter what the state of the interrupt_flag is. Set this to nullopt to specify that there should not be a cap.
- `[in] saturation_limit`: A cap on how many search nodes the planner is allowed to produce.

Options &**validator**(rmf_utils::clone_ptr<*RouteValidator*> *v*)
Set the route validator.

const rmf_utils::clone_ptr<*RouteValidator*> &**validator**() **const**
Get the route validator.

Options &**minimum_holding_time**(*Duration holding_time*)
Set the minimum amount of time to spend waiting at holding points.

Duration **minimum_holding_time**() **const**
Get the minimum amount of time to spend waiting at holding points.

Options &**interrupter**(std::function<bool)
> *cb*Set an interrupter callback that can indicate to the planner if it should stop trying to plan.

Warning Using this function will replace anything that was given to interrupt_flag, and it will nullify the *interrupt_flag()* field.

const std::function<bool()> &**interrupter**
constGet the interrupter that will be used in these *Options*.

Options &**interrupt_flag**(std::shared_ptr<**const** std::atomic_bool> *flag*)
Set an interrupt flag to stop this planner if it has run for too long.

Warning Using this function will replace anything that was given to interrupter.

const std::shared_ptr<**const** std::atomic_bool> &**interrupt_flag**() **const**
Get the interrupt flag that will stop this planner if it has run for too long.

Options &**maximum_cost_estimate**(std::optional<double> *value*)
Set the maximum cost estimate that the planner should allow. If the cost estimate of the best possible plan that the planner could produce ever exceeds this value, the planner will pause itself (but this will not be considered an interruption).

std::optional<double> **maximum_cost_estimate**() **const**
Get the maximum cost estimate that the planner will allow.

Options &**saturation_limit**(std::optional<std::size_t> *value*)
Set the saturation limit for the planner. If the planner produces more search nodes than this limit, then the planning will stop.

std::optional<std::size_t> **saturation_limit**() **const**
Get the saturation limit.

Options &**dependency_window**(std::optional<*Duration*> *value*)
Set the dependency window for generated plans. Any potential conflicts with the generated plan that happen within this window will be added as dependencies to the plan waypoints. If set to a nullopt, the plan will not have any dependencies.

std::optional<*Duration*> **dependency_window**() **const**
Dependency window for the planner.

Options &**dependency_resoution**(*Duration value*)
Set the dependency resolution for generated plans. To check for dependencies, the planner will step the generated routes back in time by this value and check for conflicts. Detected conflicts get added to the list of dependencies. This backstepping happens until dependency_window is reached. If dependency_window is nullopt, this value will not be used.

Duration **dependency_resolution**() **const**
Get the dependency resolution for generated plans.

Public Static Attributes

static constexpr *Duration* **DefaultMinHoldingTime** = std::chrono::seconds(1)

Class Planner::Result

- Defined in file_latest_rmf_traffic_include_rmf_traffic_agv_Planner.hpp

Nested Relationships

This class is a nested type of *Class Planner*.

Class Documentation

class rmf_traffic::agv::*Planner*::**Result**

Public Functions

bool **success**() **const**
True if a plan was found and this *Result* can be dereferenced to obtain a plan.

bool **disconnected**() **const**
True if there is no feasible path that connects the start to the goal. In this case, a plan will never be found.

operator bool() **const**
Implicitly cast the result to a boolean. It will return true if a plan was found, otherwise it will return false.

const *Plan* ***operator->**() **const**
If the *Result* was successful, drill into the plan.

const *Plan* &**operator***() **const** &
If the *Result* was successful, get a reference to the plan.

Plan &&**operator***() &&
If the *Result* was successful, move the plan.

const *Plan* &&**operator***() **const** &&
If the *Result* was successful, get a reference to the plan.

Result **replan**(**const** *Start* &*new_start*) **const**
Replan to the same goal from a new start location using the same options as before.

Parameters

- `[in] new_start`: The starting conditions that should be used for replanning.

Result **replan**(**const** *Start* &*new_start*, *Options new_options*) **const**
Replan to the same goal from a new start location using a new set of options.

Parameters

- `[in] new_start`: The starting conditions that should be used for replanning.
- `[in] new_options`: The options that should be used for replanning.

Result **replan**(**const** *StartSet* &*new_starts*) **const**
Replan to the same goal from a new set of start locations using the same options.

Parameters

- `[in] new_starts`: The set of starting conditions that should be used for replanning.

Result **replan**(**const** *StartSet* &*new_starts*, *Options new_options*) **const**
Replan to the same goal from a new set of start locations using a new set of options.

Parameters

- `[in] new_starts`: The set of starting conditions that should be used for replanning.
- `[in] new_options`: The options that should be used for replanning.

Result **setup**(**const** *Start* &*new_start*) **const**
Set up a new planning job to the same goal, but do not start iterating.

See replan(const Start&)

Result **setup**(**const** *Start* &*new_start*, *Options new_options*) **const**
Set up a new planning job to the same goal, but do not start iterating.

See replan(const Start&, Options)

Result **setup**(**const** *StartSet* &*new_starts*) **const**
Set up a new planning job to the same goal, but do not start iterating.

See replan(const StartSet&)

Result **setup**(**const** *StartSet* &*new_starts*, *Options new_options*) **const**
Set up a new planning job to the same goal, but do not start iterating.

See replan(const StartSet&, Options)

bool **resume**()
Resume planning if the planner was paused.

Return true if a plan has been found, false otherwise.

bool **resume**(std::shared_ptr<**const** std::atomic_bool> *interrupt_flag*)
Resume planning if the planner was paused.

Return true if a plan has been found, false otherwise.

Parameters

- `[in] interrupt_flag`: A new interrupt flag to listen to while planning.

Options &**options**()
Get a mutable reference to the options that will be used by this planning task.

const *Options* &**options**() **const**
Get the options that will be used by this planning task.

Result &**options**(*Options new_options*)
Change the options to be used by this planning task.

std::optional<double> **cost_estimate**() **const**
Get the best cost estimate of the current state of this planner result. This is the value of the lowest f(n)=g(n)+h(n) in the planner's queue. If the node queue of this planner result is empty, this will return a nullopt.

double **initial_cost_estimate**() **const**
Get the cost estimate that was initially computed for this plan. If no valid starts were provided, then this will return infinity.

std::optional<double> **ideal_cost**() **const**
Get the cost that this plan would have if there is no traffic. If the plan is impossible (e.g. the starts are disconnected from the goal) this will return a nullopt.

const std::vector<*Start*> &**get_starts**() **const**
Get the start conditions that were given for this planning task.

const *Goal* &**get_goal**() **const**
Get the goal for this planning task.

const *Configuration* &**get_configuration**() **const**
If this *Plan* is valid, this will return the *Planner::Configuration* that was used to produce it.

If *replan()* is called, this *Planner::Configuration* will be used to produce the new *Plan*.

bool **interrupted**() **const**
This will return true if the planning failed because it was interrupted. Otherwise it will return false.

bool **saturated**() **const**
This will return true if the planner has reached its saturation limit.

std::vector<schedule::*ParticipantId*> **blockers**() **const**
This is a list of schedule Participants who blocked the planning effort. Blockers do not necessarily prevent a solution from being found, but they do prevent the optimal solution from being available.

Class Planner::Start

- Defined in file_latest_rmf_traffic_include_rmf_traffic_agv_Planner.hpp

Nested Relationships

This class is a nested type of *Class Planner*.

Class Documentation

class `rmf_traffic::agv::Planner::Start`
Describe the starting conditions of a plan.

Public Functions

Start(*Time* *initial_time*, std::size_t *initial_waypoint*, double *initial_orientation*, std::optional<Eigen::Vector2d> *location* = std::nullopt, std::optional<std::size_t> *initial_lane* = std::nullopt)
Constructor

Parameters

- `[in] inital_time`: The starting time of the plan.
- `[in] initial_waypoint`: The waypoint index that the plan will begin from.
- `[in] initial_orientation`: The orientation that the AGV will start with.
- `[in] initial_location`: Optional field to specify if the robot is not starting directly on the initial_waypoint location. When planning from this initial_location to the initial_waypoint the planner will assume it has an unconstrained lane.
- `[in] initial_lane`: Optional field to specify if the robot is starting in a certain lane. This will only be used if an initial_location is specified.

Start &**time**(*Time* *initial_time*)
Set the starting time of a plan.

Time **time**() **const**
Get the starting time.

Start &**waypoint**(std::size_t *initial_waypoint*)
Set the starting waypoint of a plan.

std::size_t **waypoint**() **const**
Get the starting waypoint.

Start &**orientation**(double *initial_orientation*)
Set the starting orientation of a plan.

double **orientation**() **const**
Get the starting orientation.

const std::optional<Eigen::Vector2d> &**location**() **const**
Get the starting location, if one was specified.

Start &**location**(std::optional<Eigen::Vector2d> *initial_location*)
Set the starting location, or remove it by using std::nullopt.

const std::optional<std::size_t> &**lane**() **const**
Get the starting lane, if one was specified.

Start &**lane**(std::optional<std::size_t> *initial_lane*)
Set the starting lane, or remove it by using std::nullopt.

Class Rollout

- Defined in file_latest_rmf_traffic_include_rmf_traffic_agv_Rollout.hpp

Class Documentation

class rmf_traffic::agv::**Rollout**
The *Rollout* class complements the *Planner* class. The *Planner* class may sometimes fail to find a feasible plan because of other traffic participants blocking the way. The rollout class can take a *Planner::Result* and expand sets of alternative routes that would be feasible in the absence of a blocker. Given these sets of alternatives,

Public Functions

Rollout(*Planner*::*Result* *result*)
Constructor

Parameters

- [in] result: The Planning Result that should be rolled out.

std::vector<schedule::*Itinerary*> **expand**(schedule::*ParticipantId* *blocker*, rmf_traffic::*Duration* *span*, **const** *Planner*::*Options* &*options*, rmf_utils::optional<std::size_t> *max_rollouts* = rmf_utils::nullopt) **const**
Expand the Planning Result through the specified blocker.

Return a collection of itineraries from the original Planning Result's starts past the blockages that were caused by the specified blocker.

Parameters

- [in] blocker: The blocking participant that should be expanded through. If this participant wasn't actually blocking, then the returned vector will be empty.
- [in] span: How far into the future the rollout should continue. Once a rollout extends this far, it will stop wherever it is.
- [in] options: The options to use while expanding. NOTE: It is important to provide a *RouteValidator* that will ignore the blocker, otherwise the expansion might not give back any useful results.
- [in] max_rollouts: The maximum number of rollouts to produce.

std::vector<schedule::*Itinerary*> **expand**(schedule::*ParticipantId* *blocker*, rmf_traffic::*Duration* *span*, rmf_utils::optional<std::size_t> *max_rollouts* = rmf_utils::nullopt) **const**

Expand the Planning Result through the specified behavior. Use the Options that are already tied to the Planning Result.

Warning It is critical to change the validator in the *Planner* Result Options before giving it to the *Rollout* if you want to use this method. Otherwise there will not be any expansion through the blocker.

Return a collection of itineraries from the original Planning Result's starts past the blockages that were caused by the specified blocker.

Parameters

- `[in] blocker`: The blocking participant that should be expanded through. If this participant wasn't actually blocking, then the returned vector will be empty.
- `[in] span`: How far into the future the rollout should continue. Once a rollout extends this far, it will stop wherever it is.
- `[in] max_rollouts`: The maximum number of rollouts to produce.

Class RouteValidator

- Defined in file_latest_rmf_traffic_include_rmf_traffic_agv_RouteValidator.hpp

Nested Relationships

Nested Types

- *Struct RouteValidator::Conflict*

Inheritance Relationships

Derived Types

- `public rmf_traffic::agv::NegotiatingRouteValidator` (*Class NegotiatingRouteValidator*)
- `public rmf_traffic::agv::ScheduleRouteValidator` (*Class ScheduleRouteValidator*)

Class Documentation

class `rmf_traffic::agv::`**RouteValidator**

The *RouteValidator* class provides an interface for identifying whether a given route can be considered valid.

Subclassed by *rmf_traffic::agv::NegotiatingRouteValidator*, *rmf_traffic::agv::ScheduleRouteValidator*

Public Types

using ParticipantId = schedule::*ParticipantId*

using Route = rmf_traffic::*Route*

Public Functions

virtual std::optional<*Conflict*> **find_conflict**(**const** *Route* &*route*) **const** = 0

If the specified route has a conflict with another participant, this will return the participant ID for the first conflict that gets identified. Otherwise it will return a nullopt.

Parameters

- `[in] route`: The route that is being checked.

virtual std::unique_ptr<*RouteValidator*> **clone**() **const** = 0

Create a clone of the underlying *RouteValidator* object.

virtual ~RouteValidator() = default

struct Conflict

Public Members

Dependency **dependency**

Time **time**

std::shared_ptr<**const** rmf_traffic::*Route*> **route**

Class ScheduleRouteValidator

- Defined in file_latest_rmf_traffic_include_rmf_traffic_agv_RouteValidator.hpp

Inheritance Relationships

Base Type

- `public rmf_traffic::agv::RouteValidator` (*Class RouteValidator*)

Class Documentation

class `rmf_traffic::agv::`**ScheduleRouteValidator** : **public** rmf_traffic::agv::*RouteValidator*

Public Functions

ScheduleRouteValidator(**const** schedule::*Viewer* &*viewer*, schedule::*ParticipantId* *participant_id*, *Profile* *profile*)

Constructor

Warning You are expected to maintain the lifetime of the schedule viewer for as long as this *ScheduleRouteValidator* instance is alive. This object will only retain a reference to the viewer, not a copy of it.

Parameters

- `[in] viewer`: The schedule viewer which will be used to check for conflicts
- `[in] participant`: The ID of the participant whose route is being validated. Any routes for this participant on the schedule will be ignored while validating.
- `[in] profile`: The profile for the participant. This is not inferred from the viewer because the viewer might not be synced with the schedule by the time this validator is being used.

ScheduleRouteValidator(std::shared_ptr<**const** schedule::*Viewer*> *viewer*, schedule::*ParticipantId* *participant_id*, *Profile* *profile*)

Constructor

This constructor will use the profile given to it for the participant that is being planned for. This is safe to use, even if the participant is not registered in the schedule yet.

Parameters

- `[in] viewer`: The schedule viewer which will be used ot check for conflicts. The reference to the viewer will be kept alive.
- `[in] participant_id`: The ID for the participant that is being validated.
- `[in] profile`: The profile for the participant.

ScheduleRouteValidator &**schedule_viewer**(**const** schedule::*Viewer* &*viewer*)

Change the schedule viewer to use for planning.

Warning The Options instance will store a reference to the viewer; it will not store a copy. Therefore you are responsible for keeping the schedule viewer alive while this Options class is being used.

const schedule::*Viewer* &**schedule_viewer**() **const**

Get a const reference to the schedule viewer that will be used for planning. It is undefined behavior to call this function is called after the schedule viewer has been destroyed.

ScheduleRouteValidator &**participant**(schedule::*ParticipantId* *p*)

Set the ID of the participant that is being validated.

schedule::*ParticipantId* **participant**() **const**

Get the ID of the participant that is being validated.

virtual std::optional<Conflict> **find_conflict**(**const** *Route* &*route*) **const final**

If the specified route has a conflict with another participant, this will return the participant ID for the first conflict that gets identified. Otherwise it will return a nullopt.

Parameters

- `[in] route`: The route that is being checked.

virtual std::unique_ptr<*RouteValidator*> **clone** () **const final**
Create a clone of the underlying *RouteValidator* object.

Public Static Functions

template<typename ...**Args**>
static inline rmf_utils::clone_ptr<*ScheduleRouteValidator*> **make** (*Args*&&... *args*)
Make the *ScheduleRouteValidator* as a clone_ptr.

Class SimpleNegotiator

- Defined in file_latest_rmf_traffic_include_rmf_traffic_agv_SimpleNegotiator.hpp

Nested Relationships

Nested Types

- *Class SimpleNegotiator::Debug*
- *Class SimpleNegotiator::Options*

Inheritance Relationships

Base Type

- `public rmf_traffic::schedule::Negotiator` (*Class Negotiator*)

Class Documentation

class `rmf_traffic::agv::`**SimpleNegotiator** : **public** rmf_traffic::schedule::*Negotiator*
A simple implementation of the *schedule::Negotiator* class. It uses an *agv::Planner* to try to find a solution that fits on the negotiation table.

Public Functions

SimpleNegotiator (schedule::Participant::AssignIDPtr *assign_id*, *Planner*::*Start* *start*, *Planner*::*Goal* *goal*, *Planner*::*Configuration* *planner_configuration*, *Options* *options* = *Options*())
Constructor

Parameters

- `[in] assign_id`: The ID assignment tool for the participant
- `[in] start`: The desired start for the plan.
- `[in] goal`: The desired goal for the plan.

- `[in] planner_configuration`: The configuration that will be used by the planner underlying this Negotiator.
- `[in] options`: Additional options that will be used by the Negotiator.

SimpleNegotiator(schedule::Participant::AssignIDPtr *assign_id*, std::vector<*Planner*::*Start*> *starts*, *Planner*::*Goal* *goal*, *Planner*::*Configuration* *planner_configuration*, *Options* *options* = *Options*())

Constructor

Parameters

- `[in] assign_id`: The ID assignment tool for the participant
- `[in] start`: A set of starts that can be used.
- `[in] goal`: The desired goal for the plan.
- `[in] planner_configuration`: The configuration that will be used by the planner underlying this Negotiator.
- `[in] options`: Additional options that will be used by the Negotiator.

SimpleNegotiator(schedule::Participant::AssignIDPtr *assign_id*, std::vector<*Planner*::*Start*> *starts*, *Planner*::*Goal* *goal*, std::shared_ptr<**const** *Planner*> *planner*, *Options* *options* = *Options*())

Constructor

Parameters

- `[in] assign_id`: The ID assignment tool for the participant
- `[in] starts`: A set of starts that can be used.
- `[in] goal`: The desired goal for the plan.
- `[in] planner`: The planner to use
- `[in] options`: Additional options that will be used by the negotiator

virtual void **respond**(**const** schedule::*Negotiation*::*Table*::*ViewerPtr* &*table_viewer*, **const** ResponderPtr &*responder*) **final**

Have the Negotiator respond to an attempt to negotiate.

Parameters

- `[in] table`: The Negotiation::Table that is being used for the negotiation.
- `[in] responder`: The Responder instance that the negotiator should use when a response is ready.
- `[in] interrupt_flag`: A pointer to a flag that can be used to interrupt the negotiator if it has been running for too long. If the planner should run indefinitely, then pass a nullptr.

class Debug

Public Static Functions

static *SimpleNegotiator* &**enable_debug_print**(*SimpleNegotiator* &*negotiator*)

class Options

A class to specify user-defined options for the Negotiator.

Public Types

using ApprovalCallback = std::function<Responder::UpdateVersion (rmf_traffic::agv::*Plan*) >

Public Functions

Options(*ApprovalCallback* *approval_cb* = nullptr, std::shared_ptr<**const** bool> *interrupt_flag* = nullptr, std::optional<double> *maximum_cost_leeway* = *DefaultMaxCostLeeway*, std::optional<std::size_t> *maximum_alts* = std::nullopt, *Duration* *min_hold_time* = *Planner*::*Options*::*DefaultMinHoldingTime*)

Constructor

Parameters

- `[in] approval_cb`: The callback that will be triggered if the proposal is approved.
- `[in] maximum_cost_leeway`: The initial cost estimate for each planning attempt will be multiplied by this factor to determine the maximum cost estimate that will be allowed for a plan before giving up.
- `[in] maximum_alts`: The maximum number of alternatives to produce when rejecting a proposal from another negotiator.
- `[in] min_hold_time`: The minimum amount of time that the planner should spend waiting at holding points. See *Planner::Options* for more information.

Options &**approval_callback**(*ApprovalCallback* *cb*)

Set the approval callback.

Options &**interrupt_flag**(std::shared_ptr<**const** bool> *flag*)

Set the interrupt flag.

const std::shared_ptr<**const** bool> &**interrupt_flag**() **const**

Get the interrupt flag.

Options &**maximum_cost_leeway**(std::optional<double> *leeway*)

Set the maximum cost leeway.

std::optional<double> **maximum_cost_leeway**() **const**

Get the maximum cost leeway.

Options &**minimum_cost_threshold**(std::optional<double> *cost*)

Set the minimum cost threshold. When this and maximum_cost_leeway are both set, the maximum cost estimate will be chosen by std::max(minimum_cost_threshold, initial_cost_estimate * maximum_cost_leeway)

By default, this is DefaultMinCostThreshold.

std::optional<double> **minimum_cost_threshold**() **const**

Get the minimum cost threshold.

Options &**maximum_cost_threshold**(std::optional<double> *cost*)

Set the maximum cost threshold. When this is set, the cost will not be allowed to exceed it, even if the maximum cost leeway would allow it. By default, this is nullopt.

std::optional<double> **maximum_cost_threshold**() **const**
Get the maximum cost threshold.

Options &**maximum_alternatives**(rmf_utils::optional<std::size_t> *num*)

std::optional<std::size_t> **maximum_alternatives**() **const**

Options &**minimum_holding_time**(*Duration holding_time*)
Set the minimum amount of time to spend waiting at holding points.

Duration **minimum_holding_time**() **const**
Get the minimum amount of time to spend waiting at holding points.

Public Static Attributes

static constexpr double **DefaultMaxCostLeeway** = 1.5

static constexpr double **DefaultMinCostThreshold** = 30.0

Class SimpleNegotiator::Debug

- Defined in file_latest_rmf_traffic_include_rmf_traffic_agv_debug_debug_Negotiator.hpp

Nested Relationships

This class is a nested type of *Class SimpleNegotiator*.

Class Documentation

class rmf_traffic::agv::*SimpleNegotiator*::**Debug**

Public Static Functions

static *SimpleNegotiator* &**enable_debug_print**(*SimpleNegotiator* &*negotiator*)

Class SimpleNegotiator::Options

- Defined in file_latest_rmf_traffic_include_rmf_traffic_agv_SimpleNegotiator.hpp

Nested Relationships

This class is a nested type of *Class SimpleNegotiator*.

Class Documentation

class rmf_traffic::agv::*SimpleNegotiator*::**Options**

A class to specify user-defined options for the Negotiator.

Public Types

using ApprovalCallback = std::function<Responder::UpdateVersion (rmf_traffic::agv::*Plan*) >

Public Functions

Options (*ApprovalCallback* *approval_cb* = nullptr, std::shared_ptr<**const** bool> *interrupt_flag* = nullptr, std::optional<double> *maximum_cost_leeway* = *DefaultMaxCostLeeway*, std::optional<std::size_t> *maximum_alts* = std::nullopt, *Duration* *min_hold_time* = *Planner*::*Options*::*DefaultMinHoldingTime*)

Constructor

Parameters

- `[in] approval_cb`: The callback that will be triggered if the proposal is approved.
- `[in] maximum_cost_leeway`: The initial cost estimate for each planning attempt will be multiplied by this factor to determine the maximum cost estimate that will be allowed for a plan before giving up.
- `[in] maximum_alts`: The maximum number of alternatives to produce when rejecting a proposal from another negotiator.
- `[in] min_hold_time`: The minimum amount of time that the planner should spend waiting at holding points. See *Planner::Options* for more information.

Options &**approval_callback** (*ApprovalCallback* *cb*)

Set the approval callback.

Options &**interrupt_flag** (std::shared_ptr<**const** bool> *flag*)

Set the interrupt flag.

const std::shared_ptr<**const** bool> &**interrupt_flag** () **const**

Get the interrupt flag.

Options &**maximum_cost_leeway** (std::optional<double> *leeway*)

Set the maximum cost leeway.

std::optional<double> **maximum_cost_leeway** () **const**

Get the maximum cost leeway.

Options &**minimum_cost_threshold** (std::optional<double> *cost*)

Set the minimum cost threshold. When this and maximum_cost_leeway are both set, the maximum cost estimate will be chosen by std::max(minimum_cost_threshold, initial_cost_estimate * maximum_cost_leeway)

By default, this is DefaultMinCostThreshold.

std::optional<double> **minimum_cost_threshold** () **const**

Get the minimum cost threshold.

Options &**maximum_cost_threshold**(std::optional<double> *cost*)
Set the maximum cost threshold. When this is set, the cost will not be allowed to exceed it, even if the maximum cost leeway would allow it. By default, this is nullopt.

std::optional<double> **maximum_cost_threshold**() **const**
Get the maximum cost threshold.

Options &**maximum_alternatives**(rmf_utils::optional<std::size_t> *num*)

std::optional<std::size_t> **maximum_alternatives**() **const**

Options &**minimum_holding_time**(*Duration holding_time*)
Set the minimum amount of time to spend waiting at holding points.

Duration **minimum_holding_time**() **const**
Get the minimum amount of time to spend waiting at holding points.

Public Static Attributes

static constexpr double **DefaultMaxCostLeeway** = 1.5

static constexpr double **DefaultMinCostThreshold** = 30.0

Class VehicleTraits

- Defined in file_latest_rmf_traffic_include_rmf_traffic_agv_VehicleTraits.hpp

Nested Relationships

Nested Types

- *Class VehicleTraits::Differential*
- *Class VehicleTraits::Holonomic*
- *Class VehicleTraits::Limits*

Class Documentation

class rmf_traffic::agv::**VehicleTraits**

Public Types

enum Steering
Values:

enumerator Differential
The vehicle uses differential steering, making it impossible to move laterally.

enumerator Holonomic
The vehicle can move holonomically, so it has no limitations about how it steers.

Public Functions

VehicleTraits(*Limits* *linear*, *Limits* *angular*, *Profile* *profile*, *Differential* *steering* = *Differential*())
Constructor.

Limits &**linear**()

const *Limits* &**linear**() **const**

Limits &**rotational**()

const *Limits* &**rotational**() **const**

Profile &**profile**()

const *Profile* &**profile**() **const**

Steering **get_steering**() **const**

Differential &**set_differential**(*Differential* *parameters* = *Differential*())

Differential ***get_differential**()

const *Differential* ***get_differential**() **const**

Holonomic &**set_holonomic**(*Holonomic* *parameters*)

Holonomic ***get_holonomic**()

const *Holonomic* ***get_holonomic**() **const**

bool **valid**() **const**
Returns true if the values of the traits are valid. For example, this means that all velocity and acceleration values are greater than zero.

class Differential

Public Functions

Differential(Eigen::Vector2d *forward* = Eigen::Vector2d::UnitX(), bool *reversible* = true)

Differential &**set_forward**(Eigen::Vector2d *forward*)

const Eigen::Vector2d &**get_forward**() **const**

Differential &**set_reversible**(bool *reversible*)

bool **is_reversible**() **const**

bool **valid**() **const**
Returns true if the length of the forward vector is not too close to zero. If it is too close to zero, then the direction of the forward vector cannot be reliably interpreted. Ideally the forward vector should have unit length.

class Holonomic

Public Functions

Holonomic()

class Limits

Public Functions

Limits(double *velocity* = 0.0, double *acceleration* = 0.0)

Limits &**set_nominal_velocity**(double *nom_vel*)

double **get_nominal_velocity**() **const**

Limits &**set_nominal_acceleration**(double *nom_accel*)

double **get_nominal_acceleration**() **const**

bool **valid**() **const**

Returns true if the values of these limits are valid, i.e. greater than zero.

Class VehicleTraits::Differential

- Defined in file_latest_rmf_traffic_include_rmf_traffic_agv_VehicleTraits.hpp

Nested Relationships

This class is a nested type of *Class VehicleTraits*.

Class Documentation

class rmf_traffic::agv::*VehicleTraits*::**Differential**

Public Functions

Differential(Eigen::Vector2d *forward* = Eigen::Vector2d::UnitX(), bool *reversible* = true)

Differential &**set_forward**(Eigen::Vector2d *forward*)

const Eigen::Vector2d &**get_forward**() **const**

Differential &**set_reversible**(bool *reversible*)

bool **is_reversible**() **const**

bool **valid**() **const**

Returns true if the length of the forward vector is not too close to zero. If it is too close to zero, then the direction of the forward vector cannot be reliably interpreted. Ideally the forward vector should have unit length.

Class VehicleTraits::Holonomic

- Defined in file_latest_rmf_traffic_include_rmf_traffic_agv_VehicleTraits.hpp

Nested Relationships

This class is a nested type of *Class VehicleTraits*.

Class Documentation

class rmf_traffic::agv::*VehicleTraits*::**Holonomic**

Public Functions

Holonomic()

Class VehicleTraits::Limits

- Defined in file_latest_rmf_traffic_include_rmf_traffic_agv_VehicleTraits.hpp

Nested Relationships

This class is a nested type of *Class VehicleTraits*.

Class Documentation

class rmf_traffic::agv::*VehicleTraits*::**Limits**

Public Functions

Limits(double *velocity* = 0.0, double *acceleration* = 0.0)

Limits &**set_nominal_velocity**(double *nom_vel*)

double **get_nominal_velocity**() **const**

Limits &**set_nominal_acceleration**(double *nom_accel*)

double **get_nominal_acceleration**() **const**

bool **valid**() **const**

Returns true if the values of these limits are valid, i.e. greater than zero.

Class Moderator

- Defined in file_latest_rmf_traffic_include_rmf_traffic_blockade_Moderator.hpp

Nested Relationships

Nested Types

- *Class Moderator::Assignments*

Inheritance Relationships

Base Type

- `public rmf_traffic::blockade::Writer` (*Class Writer*)

Class Documentation

class `rmf_traffic::blockade::`**Moderator** : **public** rmf_traffic::blockade::*Writer*

Public Functions

virtual void **set**(*ParticipantId participant_id*, *ReservationId reservation_id*, **const** Reservation *&reservation*) **final**

Set the path reservation of a participant.

If reservation_id is (modularly) less than or equal to the last reservation_id value given for this participant_id, then this function call will be ignored.

Any previous path reservation will be considered canceled.

virtual void **ready**(*ParticipantId participant_id*, *ReservationId reservation_id*, *CheckpointId checkpoint*) **final**

Indicate when a participant is ready at a checkpoint.

If reservation_id is not equal to the last reservation_id value given to *set()* for this participant_id, then this function call will be ignored.

virtual void **release**(*ParticipantId participant_id*, *ReservationId reservation_id*, *CheckpointId checkpoint*) **final**

Release a checkpoint (and all checkpoints that come after it) from ready status if the participant has not departed from it yet.

virtual void **reached**(*ParticipantId participant_id*, *ReservationId reservation_id*, *CheckpointId checkpoint*) **final**

Indicate when a participant has reached a checkpoint.

If reservation_id is not equal to the last reservation_id value given to *set()* for this participant_id, then this function call will be ignored.

virtual void **cancel**(*ParticipantId participant_id*, *ReservationId reservation_id*) **final**

Indicate that a path reservation is canceled if reservation_id is (modularly) greater than or equal to the last reservation_id value given to *set()* for this participant_id.

virtual void **cancel**(*ParticipantId participant_id*) **final**
Indicate that all path reservations for this participant_id are canceled.

Moderator(std::function<void) std::string
> *info_logger* = nullptr, std::function<voidstd::string> *debug_logger* = nullptr, double *min_conflict_angle* = 5.0 * M_PI / 180.0Default constructor

Parameters

- `[in] info_logger`: Provide a callback for logging informational updates about changes in the blockades, e.g. when a new path arrives, when a checkpoint is reached, or when one is ready.
- `[in] debug_logger`: Provide a callback for logging debugging information, e.g. which constraints are blocking a participant from advancing.
- `[in] min_conflict_angle`: If the angle between two path segments is greater than this value (radians), then the segments are considered to be in conflict. The default value for this parameter is 5-degrees. Something larger than 0 is recommended to help deal with numerical precision concerns.

double **minimum_conflict_angle**() **const**
Get the minimum angle that will trigger a conflict.

Moderator &**minimum_conflict_angle**(double *new_value*)
Set the minimum angle that will trigger a conflict.

Moderator &**info_logger**(std::function<void) std::string
> *info*Set the information logger for this *Moderator*. Pass in a nullptr to disable any information logging.

Moderator &**debug_logger**(std::function<void) std::string
> *debug*Set the debug logger for this *Moderator*. Pass in a nullptr to disable any debug logging.

const *Assignments* &**assignments**() **const**
Get the current set of assignments.

const std::unordered_map<*ParticipantId*, *Status*> &**statuses**() **const**
Get the current known statuses of each participant.

bool **has_gridlock**() **const**
Return true if the system is experiencing a gridlock.

class Assignments
This class indicates the range of each reservation that the blockade moderator has assigned as active. Each robot is allowed to move at will from the begin checkpoint to the end checkpoint in the range assigned for it.

Public Functions

std::size_t **version**() **const**
Get the version of the current assignment sets. The version number will increase by at least 1 each time the assignments change. This can be used to identify when new assignment notifications are necessary.

const std::unordered_map<*ParticipantId*, *ReservedRange*> &**ranges**() **const**
Get the ranges that are assigned to each participant.

Class Moderator::Assignments

- Defined in file_latest_rmf_traffic_include_rmf_traffic_blockade_Moderator.hpp

Nested Relationships

This class is a nested type of *Class Moderator*.

Class Documentation

class rmf_traffic::blockade::*Moderator*::**Assignments**

This class indicates the range of each reservation that the blockade moderator has assigned as active. Each robot is allowed to move at will from the begin checkpoint to the end checkpoint in the range assigned for it.

Public Functions

std::size_t **version**() **const**

Get the version of the current assignment sets. The version number will increase by at least 1 each time the assignments change. This can be used to identify when new assignment notifications are necessary.

const std::unordered_map<*ParticipantId*, *ReservedRange*> &**ranges**() **const**

Get the ranges that are assigned to each participant.

Class ModeratorRectificationRequesterFactory

- Defined in file_latest_rmf_traffic_include_rmf_traffic_blockade_Rectifier.hpp

Inheritance Relationships

Base Type

- public rmf_traffic::blockade::RectificationRequesterFactory (*Class RectificationRequesterFactory*)

Class Documentation

class rmf_traffic::blockade::**ModeratorRectificationRequesterFactory** : **public** rmf_traffic::blockade::*I*

This class provides a simple implementation of a *RectificationRequesterFactory* that just hooks directly into a *Moderator* instance and issues rectification requests when told to based on the current inconsistencies in the Database.

Public Functions

ModeratorRectificationRequesterFactory(std::shared_ptr<*Moderator*> *moderator*)
Constructor

Parameters

- `[in] moderator`: The moderator object that this will rectify for.

virtual std::unique_ptr<*RectificationRequester*> **make**(*Rectifier* *rectifier*, *ParticipantId* *participant_id*) **final**
Create a *RectificationRequester* to be held by a *Participant*

Parameters

- `[in] rectifier`: This rectifier can be used by the *RectificationRequester* to ask the participant to check its status.
- `[in] participant_id`: The ID of the participant that will hold onto this *RectificationRequester*. This is the same participant that the rectifier will request for checks.

void **rectify**()
Call this function to instruct all the RectificationRequesters produced by this factory to perform their rectifications.

Class Participant

- Defined in file_latest_rmf_traffic_include_rmf_traffic_blockade_Participant.hpp

Class Documentation

class `rmf_traffic::blockade::`**Participant**

Public Functions

void **radius**(double *new_radius*)
Change the radius for this participant. This will only take effect when a new path is set using the *set()* function.

double **radius**() **const**
Get the radius that's being used for this participant.

void **set**(std::vector<*Writer*::*Checkpoint*> *path*)
Set the path for this participant.

Parameters

- `[in] path`: The path that this participant intends to follow.

const std::vector<*Writer*::*Checkpoint*> &**path**() **const**
Get the current path for this participant.

void **ready**(*CheckpointId* *checkpoint*)
Tell the blockade writer that the participant is ready to depart from the given checkpoint.

void **release**(*CheckpointId checkpoint*)
Tell the blockade writer that the participant is releasing its departure from the given checkpoint.

std::optional<*CheckpointId*> **last_ready**() **const**
Get the last checkpoint that this participant said it is ready to depart from.

void **reached**(*CheckpointId checkpoint*)
Tell the blockade writer that the participant has reached the given checkpoint.

void **cancel**()
Cancel the current path entirely. Note that if a path is canceled while the robot is in space that it needs to share with other robots, a permanent deadlock could result.

CheckpointId **last_reached**() **const**
Get the last checkpoint that this participant said it has reached.

ParticipantId **id**() **const**
Get the ID that was assigned to this participant.

std::optional<*ReservationId*> **reservation_id**() **const**
Get the current reservation ID.

Class RectificationRequester

- Defined in file_latest_rmf_traffic_include_rmf_traffic_blockade_Rectifier.hpp

Class Documentation

class rmf_traffic::blockade::**RectificationRequester**
RectificationRequester is a pure abstract class which should be implemented for any middlewares that intend to act as transport layers for the scheduling system.

Classes that derive from *RectificationRequester* do not need to implement any interfaces, but they should practice RAII. The lifecycle of the *RectificationRequester* will be tied to the *Participant* that it was created for.

When a schedule database reports an inconsistency for the participant tied to a *RectificationRequester* instance, the instance should call *Rectifier::check()* on the *Rectifier* that was assigned to it.

Public Functions

virtual **~RectificationRequester**() = 0
This destructor is pure virtual to ensure that a derived class is instantiated.

Class RectificationRequesterFactory

- Defined in file_latest_rmf_traffic_include_rmf_traffic_blockade_Rectifier.hpp

Inheritance Relationships

Derived Type

- `public rmf_traffic::blockade::ModeratorRectificationRequesterFactory` (*Class ModeratorRectificationRequesterFactory*)

Class Documentation

class `rmf_traffic::blockade::`**RectificationRequesterFactory**

The *RectificationRequesterFactory* is a pure abstract interface class which should be implemented for any middlewares that intend to act as transport layers for the blockade system.

Subclassed by *rmf_traffic::blockade::ModeratorRectificationRequesterFactory*

Public Functions

virtual std::unique_ptr<*RectificationRequester*> **make**(*Rectifier* *rectifier*, *ParticipantId* *participant_id*) = 0

Create a *RectificationRequester* to be held by a *Participant*

Parameters

- `[in] rectifier`: This rectifier can be used by the *RectificationRequester* to ask the participant to check its status.
- `[in] participant_id`: The ID of the participant that will hold onto this *RectificationRequester*. This is the same participant that the rectifier will request for checks.

virtual ~RectificationRequesterFactory() = default

Class Rectifier

- Defined in file_latest_rmf_traffic_include_rmf_traffic_blockade_Rectifier.hpp

Class Documentation

class `rmf_traffic::blockade::`**Rectifier**

The *Rectifier* class provides an interface for telling a *Participant* to rectify an inconsistency in the information received by a moderator. This rectification protocol is important when the blockades are being managed over an unreliable network.

The *Rectifier* class can be used by a RectifierRequester to ask a participant to retransmit a range of its past status changes.

Only the *Participant* class is able to create a *Rectifier* instance. Users of rmf_traffic cannot instantiate a *Rectifier*.

Public Functions

void **check**(**const** *Status* &*status*)
Check that the given status is up to date, and retransmit if any information is out of sync.

void **check**()
Check that there should not be a status for this participant. If that is a mistake and this participant *should* have a status, then retransmit the necessary information.

Class Writer

- Defined in file_latest_rmf_traffic_include_rmf_traffic_blockade_Writer.hpp

Nested Relationships

Nested Types

- *Struct Writer::Checkpoint*
- *Struct Writer::Reservation*

Inheritance Relationships

Derived Type

- `public rmf_traffic::blockade::Moderator` (*Class Moderator*)

Class Documentation

class `rmf_traffic::blockade::`**Writer**
Subclassed by *rmf_traffic::blockade::Moderator*

Public Functions

virtual void **set**(*ParticipantId participant_id*, *ReservationId reservation_id*, **const** *Reservation &reservation*) = 0
Set the path reservation of a participant.

If reservation_id is (modularly) less than or equal to the last reservation_id value given for this participant_id, then this function call will be ignored.

Any previous path reservation will be considered canceled.

virtual void **ready**(*ParticipantId participant_id*, *ReservationId reservation_id*, *CheckpointId checkpoint*) = 0
Indicate when a participant is ready at a checkpoint.

If reservation_id is not equal to the last reservation_id value given to *set()* for this participant_id, then this function call will be ignored.

virtual void **release**(*ParticipantId participant_id*, *ReservationId reservation_id*, *CheckpointId checkpoint*) = 0
Release a checkpoint (and all checkpoints that come after it) from ready status if the participant has not departed from it yet.

virtual void **reached**(*ParticipantId participant_id*, *ReservationId reservation_id*, *CheckpointId checkpoint*) = 0
Indicate when a participant has reached a checkpoint.

If reservation_id is not equal to the last reservation_id value given to *set()* for this participant_id, then this function call will be ignored.

virtual void **cancel**(*ParticipantId participant_id*, *ReservationId reservation_id*) = 0
Indicate that a path reservation is canceled if reservation_id is (modularly) greater than or equal to the last reservation_id value given to *set()* for this participant_id.

virtual void **cancel**(*ParticipantId participant_id*) = 0
Indicate that all path reservations for this participant_id are canceled.

virtual ~Writer() = default

struct Checkpoint

Public Members

Eigen::Vector2d **position**

std::string **map_name**

bool **can_hold**

struct Reservation

Public Members

std::vector<*Checkpoint*> **path**

double **radius**

Class Plumber

- Defined in file_latest_rmf_traffic_include_rmf_traffic_debug_Plumber.hpp

Class Documentation

class rmf_traffic::debug::**Plumber**

Public Functions

Plumber(std::string *name*)

~Plumber()

Class DependsOnPlan

- Defined in file_latest_rmf_traffic_include_rmf_traffic_Route.hpp

Nested Relationships

Nested Types

- *Struct DependsOnPlan::Dependency*

Class Documentation

class rmf_traffic::**DependsOnPlan**

Express a dependency on the plan of another traffic participant.

Public Functions

DependsOnPlan()

Default constructor. There will be no dependency.

DependsOnPlan(*PlanId* *plan*, *DependsOnRoute* *routes*)

There will be a dependency on the specified plan.

DependsOnPlan &**plan**(std::optional<*PlanId*> *plan*)

Set the plan that there is a dependency on.

std::optional<*PlanId*> **plan**() **const**

Get the plan that there is a dependency on.

DependsOnPlan &**routes**(*DependsOnRoute* *routes*)

Set the routes that there is a dependency on.

DependsOnRoute &**routes**()

Get the routes that there is a dependency on.

const *DependsOnRoute* &**routes**() **const**

Get the routes that there is a dependency on.

DependsOnPlan &**add_dependency**(*CheckpointId* *dependent_checkpoint*, *Dependency* *dependency*)

Add a dependency.

struct Dependency

Public Members

RouteId **on_route**

CheckpointId **on_checkpoint**

Template Class bidirectional_iterator

- Defined in file_latest_rmf_traffic_include_rmf_traffic_detail_bidirectional_iterator.hpp

Class Documentation

template<typename **ElementType**, typename **ImplementationType**, typename **Friend**>
class `rmf_traffic::detail::bidirectional_iterator`

This class is used so we can provide iterators for various container classes without exposing any implementation details about what kind of STL container we are using inside of our container class. This allows us to guarantee ABI stability, even if we decide to change what STL container we use inside of our implementation.

This class is designed to offer only the most basic features of a bidirectional iterator.

Public Types

using Element = *ElementType*

using Implementation = *ImplementationType*

Public Functions

Element &**operator***() **const**

Dereference operator.

Element ***operator->**() **const**

Drill-down operator.

bidirectional_iterator &**operator++**()

Pre-increment operator: ++it

Note This is more efficient than the post-increment operator.

Return a reference to the iterator that was operated on

bidirectional_iterator &**operator--**()

Pre-decrement operator: it

Note This is more efficient than the post-decrement operator

Return a reference to the iterator that was operated on

bidirectional_iterator **operator++**(int)

Post-increment operator: it++

Return a copy of the iterator before it was incremented

bidirectional_iterator **operator--**(int)
Post-decrement operator: it

Return a copy of the iterator before it was decremented

bool **operator==**(**const** *bidirectional_iterator* &*other*) **const**
Equality comparison operator.

bool **operator!=**(**const** *bidirectional_iterator* &*other*) **const**
Inequality comparison operator.

operator bidirectional_iterator<const Element, Implementation, Friend>() **const**

bidirectional_iterator(**const** *bidirectional_iterator*&) = default

bidirectional_iterator(*bidirectional_iterator*&&) = default

bidirectional_iterator &**operator=**(**const** *bidirectional_iterator*&) = default

bidirectional_iterator &**operator=**(*bidirectional_iterator*&&) = default

bidirectional_iterator()

Template Class forward_iterator

- Defined in file_latest_rmf_traffic_include_rmf_traffic_detail_forward_iterator.hpp

Class Documentation

template<typename **ElementType**, typename **ImplementationType**, typename **Friend**>
class rmf_traffic::detail::**forward_iterator**
This class is used so we can provide iterators for various container classes without exposing any implementation details about what kind of STL container we are using inside of our container class. This allows us to guarantee ABI stability, even if we decide to change what STL container we use inside of our implementation.

This class is designed to offer only the most basic features of a forward iterator.

Public Types

using Element = *ElementType*

using Implementation = *ImplementationType*

Public Functions

Element &**operator***() **const**
Dereference operator.

Element ***operator->**() **const**
Drill-down operator.

forward_iterator &**operator++**()
Pre-increment operator: ++it

Note This is more efficient than the post-increment operator.

Return a reference to the iterator that was operated on

forward_iterator **operator++**(int)
Post-increment operator: it++

Return a copy of the iterator before it was incremented

bool **operator==**(**const** *forward_iterator* &*other*) **const**
Equality comparison operator.

bool **operator!=**(**const** *forward_iterator* &*other*) **const**
Inequality comparison operator.

operator forward_iterator<const Element, Implementation, Friend>() **const**

forward_iterator(**const** *forward_iterator*&) = default

forward_iterator(*forward_iterator*&&) = default

forward_iterator &**operator=**(**const** *forward_iterator*&) = default

forward_iterator &**operator=**(*forward_iterator*&&) = default

forward_iterator()

Class DetectConflict

- Defined in file_latest_rmf_traffic_include_rmf_traffic_DetectConflict.hpp

Nested Relationships

Nested Types

- *Struct DetectConflict::Conflict*

Class Documentation

class rmf_traffic::**DetectConflict**

Public Types

enum Interpolate
Values:

enumerator CubicSpline

Public Static Functions

static std::optional<*Conflict*> **between**(**const** *Profile* &*profile_a*, **const** *Trajectory* &*trajectory_a*, **const** *DependsOnCheckpoint* *dependencies_of_a_on_b, **const** *Profile* &*profile_b*, **const** *Trajectory* &*trajectory_b*, **const** *DependsOnCheckpoint* *dependencies_of_b_on_a, *Interpolate* *interpolation* = *Interpolate*::*CubicSpline*)

Checks if there are any conflicts between the two trajectories.

Return true if a conflict exists between the trajectories, false otherwise.

Parameters

- `[in] profile_a`: The profile of agent A
- `[in] trajectory_a`: The trajectory of agent A
- `[in] dependencies_of_a_on_b`: The dependencies that agent A has on the given trajectory of agent B
- `[in] profile_b`: The profile of agent B
- `[in] trajectory_b`: The trajectory of agent B
- `[in] dependencies_of_b_on_a`: The dependencies that agent B has on the given trajectory of agent A

struct Conflict

Public Members

Trajectory::*const_iterator* **a_it**

Trajectory::*const_iterator* **b_it**

Time **time**

Class Circle

- Defined in file_latest_rmf_traffic_include_rmf_traffic_geometry_Circle.hpp

Inheritance Relationships

Base Type

- `public rmf_traffic::geometry::ConvexShape` (*Class ConvexShape*)

Class Documentation

class rmf_traffic::geometry::**Circle** : **public** rmf_traffic::geometry::*ConvexShape*

This class represent a circle shape which can be added into a Zone or *Trajectory*.

Public Functions

Circle(double *radius*)

Circle(**const** *Circle* &*other*)

Circle &**operator=**(**const** *Circle* &*other*)

void **set_radius**(double *r*)

double **get_radius**() **const**

virtual *FinalShape* **finalize**() **const final**

Finalize the shape so that it can be given to a Trajectory::Profile or a Zone.

virtual *FinalConvexShape* **finalize_convex**() **const final**

Finalize the shape more specifically as a *ConvexShape*.

Class ConvexShape

- Defined in file_latest_rmf_traffic_include_rmf_traffic_geometry_ConvexShape.hpp

Inheritance Relationships

Base Type

- public rmf_traffic::geometry::Shape (*Class Shape*)

Derived Type

- public rmf_traffic::geometry::Circle (*Class Circle*)

Class Documentation

class rmf_traffic::geometry::**ConvexShape** : **public** rmf_traffic::geometry::*Shape*

This class is a more specific type of *Shape*. The Zone class can consume any kind of *Shape*, but the *Trajectory* class can only consume *ConvexShape* types.

See Box, *Circle*

Subclassed by *rmf_traffic::geometry::Circle*

Public Functions

virtual *FinalConvexShape* **finalize_convex**() **const** = 0
Finalize the shape more specifically as a *ConvexShape*.

Protected Functions

ConvexShape(std::unique_ptr<*Shape*::Internal> *internal*)

Class FinalConvexShape

- Defined in file_latest_rmf_traffic_include_rmf_traffic_geometry_ConvexShape.hpp

Inheritance Relationships

Base Type

- `public rmf_traffic::geometry::FinalShape` (*Class FinalShape*)

Class Documentation

class `rmf_traffic::geometry::`**FinalConvexShape** : **public** rmf_traffic::geometry::*FinalShape*
This is a finalized *ConvexShape* whose parameters can no longer be mutated.

Protected Functions

FinalConvexShape()

Class FinalShape

- Defined in file_latest_rmf_traffic_include_rmf_traffic_geometry_Shape.hpp

Inheritance Relationships

Derived Type

- `public rmf_traffic::geometry::FinalConvexShape` (*Class FinalConvexShape*)

Class Documentation

class rmf_traffic::geometry::**FinalShape**

This is a finalized shape whose parameters can no longer be mutated.

Subclassed by *rmf_traffic::geometry::FinalConvexShape*

Public Functions

const *Shape* &**source**() **const**

Look at the source of this *FinalShape* to inspect its parameters.

double **get_characteristic_length**() **const**

Get the characteristic length of this *FinalShape*.

virtual ~FinalShape() = default

bool **operator==**(**const** *FinalShape* &*other*) **const**

Equality operator.

bool **operator!=**(**const** *FinalShape* &*other*) **const**

Non-equality operator.

Protected Functions

FinalShape()

Protected Attributes

rmf_utils::impl_ptr<Implementation> **_pimpl**

Class Shape

- Defined in file_latest_rmf_traffic_include_rmf_traffic_geometry_Shape.hpp

Inheritance Relationships

Derived Type

- public rmf_traffic::geometry::ConvexShape (*Class ConvexShape*)

Class Documentation

class rmf_traffic::geometry::**Shape**

This is the base class of different shape classes that can be used by the rmf_traffic library. This cannot (currently) be extended by downstream libraries; instead, users must choose one of the pre-defined shape types belonging to this library.

See Box, *Circle*, Polygon

Subclassed by *rmf_traffic::geometry::ConvexShape*

Public Functions

virtual *FinalShape* **finalize**() **const** = 0
Finalize the shape so that it can be given to a Trajectory::Profile or a Zone.

Shape(*Shape*&&) = delete

Shape &**operator=**(*Shape*&&) = delete

virtual ~Shape()

Protected Functions

Internal *__get_internal__()

const Internal *__get_internal__() **const**

Shape(std::unique_ptr<Internal> *internal*)

Class Space

- Defined in file_latest_rmf_traffic_include_rmf_traffic_geometry_Space.hpp

Class Documentation

class rmf_traffic::geometry::**Space**

Public Functions

Space(geometry::*ConstFinalShapePtr* *shape*, Eigen::Isometry2d *tf*)

const geometry::*ConstFinalShapePtr* &**get_shape**() **const**

Space &**set_shape**(geometry::*ConstFinalShapePtr* *shape*)

const Eigen::Isometry2d &**get_pose**() **const**

Space &**set_pose**(Eigen::Isometry2d *tf*)

Class invalid_trajectory_error

- Defined in file_latest_rmf_traffic_include_rmf_traffic_DetectConflict.hpp

Inheritance Relationships

Base Type

- public exception

Class Documentation

class rmf_traffic::**invalid_trajectory_error** : **public** exception

Public Functions

const char ***what**() **const noexcept override**

Class Motion

- Defined in file_latest_rmf_traffic_include_rmf_traffic_Motion.hpp

Class Documentation

class rmf_traffic::**Motion**

Pure abstract interface for describing a continuous motion.

Public Functions

virtual *Time* **start_time**() **const** = 0

Get the lower bound on the time range where this motion is valid.

virtual *Time* **finish_time**() **const** = 0

Get the upper bound on the time range where this motion is valid.

virtual Eigen::Vector3d **compute_position**(*Time t*) **const** = 0

Get the position of this motion at a point in time.

Parameters

- [in] t: The time of interest. This time must be in the range [*start_time()*, *finish_time()*], or else the output is undefined and may result in an exception.

virtual Eigen::Vector3d **compute_velocity**(*Time t*) **const** = 0

Get the velocity of this motion at a point in time.

Parameters

- [in] t: The time of interest. This time must be in the range [*start_time()*, *finish_time()*], or else the output is undefined and may result in an exception.

virtual Eigen::Vector3d **compute_acceleration**(*Time t*) **const** = 0

Get the acceleration of this motion at a point in time.

Parameters

- [in] t: The time of interest. This time must be in the range [*start_time()*, *finish_time()*], or else the output is undefined and may result in an exception.

virtual ~Motion() = default

Public Static Functions

static std::unique_ptr<*Motion*> **compute_cubic_splines**(**const** *Trajectory*::*const_iterator* &*begin*, **const** *Trajectory*::*const_iterator* &*end*)

Compute a piecewise cubic spline motion object for a *Trajectory* from the begin iterator up to (but not including) the end iterator.

Parameters

- `[in] begin`: The iterator of the first waypoint to include in the motion. It is undefined behavior to pass in *Trajectory::end()* for this argument.
- `[in] end`: The iterator of the first waypoint to exclude from the motion. To include all the way to the end of the trajectory, pass in *Trajectory::end()*. An exception will be thrown if begin == end.

static std::unique_ptr<*Motion*> **compute_cubic_splines**(**const** *Trajectory* &*trajectory*)

Compute a piecewise cubic spline motion object for an entire *Trajectory*.

Class Profile

- Defined in file_latest_rmf_traffic_include_rmf_traffic_Profile.hpp

Class Documentation

class rmf_traffic::**Profile**

Public Functions

Profile(geometry::*ConstFinalConvexShapePtr* *footprint*, geometry::*ConstFinalConvexShapePtr* *vicinity* = nullptr)

Constructor

Parameters

- `[in] footprint`: An estimate of the space that this participant occupies.
- `[in] vicinity`: An estimate of the vicinity around the participant in which the presence of other traffic would disrupt its operations. If a nullptr is used for this, the footprint shape will be used as the vicinity.

bool **operator==**(**const** *Profile* &*rhs*) **const**

Equality operator.

Profile &**footprint**(geometry::*ConstFinalConvexShapePtr* *shape*)

Set the footprint of the participant.

const geometry::*ConstFinalConvexShapePtr* &**footprint**() **const**

Get the footprint of the participant.

Profile &**vicinity**(geometry::*ConstFinalConvexShapePtr* *shape*)

Set the vicinity of this participant.

const geometry::*ConstFinalConvexShapePtr* &**vicinity**() **const**
Get the vicinity of this participant.

Class Region

- Defined in file_latest_rmf_traffic_include_rmf_traffic_Region.hpp

Class Documentation

class rmf_traffic::**Region**
A class to describe a region within spacetime.

This specifies the map whose coordinates should be used, a lower and upper bound to define a time range, and a set of *geometry::Space* objects to define regions with space.

For the *geometry::Space* objects, this class acts like an STL container and provides an iterator interface to specify, access, and modify them.

Public Types

using Space = geometry::*Space*

using base_iterator = rmf_traffic::detail::*bidirectional_iterator*<E, I, F>

using iterator = *base_iterator*<*Space*, IterImpl, *Region*>

using const_iterator = *base_iterator*<**const** *Space*, IterImpl, *Region*>

Public Functions

Region(std::string *map*, *Time lower_bound*, *Time upper_bound*, std::vector<*Space*> *spaces*)
Construct a region given the parameters.

Parameters

- [in] map: The map whose coordinates will be used to define the regions in space.
- [in] lower_bound: The lower bound for the time range.
- [in] upper_bound: The upper bound for the time range.
- [in] spaces: A vector of *geometry::Space* objects to define the desired regions in space.

Region(std::string *map*, std::vector<*Space*> *spaces*)
Construct a region with no time constraints.

Parameters

- [in] map: The map whose coordinates will be used to define the regions in space.
- [in] spaces: A vector of *geometry::Space* objects to define the desired regions in space.

const std::string &**get_map**() **const**
Get the name of the map that this Spacetime refers to.

Region &**set_map**(std::string *map*)
Set the name of the map that this Spacetime refers to.

const *Time* ***get_lower_time_bound**() **const**
Get the lower bound for the time range.

If there is no lower bound for the time range, then this returns a nullptr.

Region &**set_lower_time_bound**(*Time time*)
Set the lower bound for the time range.

Region &**remove_lower_time_bound**()
Remove the lower bound for the time range.

const *Time* ***get_upper_time_bound**() **const**
Get the upper bound for the time range.

If there is no upper bound for the time range, then this returns a nullptr.

Region &**set_upper_time_bound**(*Time time*)
Set the upper bound for the time range.

Region &**remove_upper_time_bound**()
Remove the upper bound for the time range.

void **push_back**(*Space space*)
Add a region of space.

void **pop_back**()
Remove the last region of space that was added.

iterator **erase**(*iterator it*)
Erase a specific region of space based on its iterator.

iterator **erase**(*iterator first*, *iterator last*)
Erase a specific sets of regions of space based on their iterators.

iterator **begin**()
Get the beginning iterator for the regions of space.

const_iterator **begin**() **const**
const-qualified *begin()*

const_iterator **cbegin**() **const**
Explicitly const-qualified alternative for *begin()*

iterator **end**()
Get the one-past-the-end iterator for the regions of space.

const_iterator **end**() **const**
const-qualified *end()*

const_iterator **cend**() **const**
Explicitly const-qualified alternative for *end()*

std::size_t **num_spaces**() **const**
Get the number of Space regions in this Spacetime region.

Class Route

- Defined in file_latest_rmf_traffic_include_rmf_traffic_Route.hpp

Class Documentation

class rmf_traffic::**Route**

A route on the schedule. This is used as a component of a schedule participant's itinerary.

Public Functions

Route(std::string *map*, *Trajectory* *trajectory*)

Constructor

Parameters

- [in] map: The map that the trajectory is on
- [in] trajectory: The scheduled trajectory

Route &**map**(std::string *value*)

Set the map for this route.

const std::string &**map**() **const**

Get the map for this route.

Route &**trajectory**(*Trajectory* *value*)

Set the trajectory for this route.

Trajectory &**trajectory**()

Get the trajectory for this route.

const *Trajectory* &**trajectory**() **const**

Get the trajectory for this immutable route.

Route &**checkpoints**(std::set<uint64_t> *value*)

Set the checkpoints for this route. A checkpoint is a waypoint within this route which will explicitly trigger an traffic event update when it is reached.

std::set<uint64_t> &**checkpoints**()

Get the checkpoints for this route.

const std::set<uint64_t> &**checkpoints**() **const**

Get the checkpoints for this immutable route.

Route &**dependencies**(*DependsOnParticipant* *value*)

Set the dependencies of the route.

DependsOnParticipant &**dependencies**()

Get the dependencies of the route.

const *DependsOnParticipant* &**dependencies**() **const**

Get the dependencies of the immutable route.

Route &**add_dependency**(*CheckpointId* *dependent_checkpoint*, *Dependency* *dependency*)

Tell this route that it has a dependency on the checkpoint of another participant's route.

Parameters

- `[in] dependent_checkpoint`: The checkpoint inside of this route which has a dependency on the other participant's route.
- `[in] on_participant`: The other participant which this route is depending on.
- `[in] on_plan`: The ID of the other participant's plan that this route is depending on.
- `[in] on_route`: The ID of the other participant's route that this robot is depending on.
- `[in] on_checkpoint`: The ID of the checkpoint

bool **should_ignore**(*ParticipantId participant*, *PlanId plan*) **const**

True if this route should ignore information about the given (participant, plan) pair. If this route has a dependency on a plan from this participant with a higher ID value, then this will return true. Otherwise it returns false.

const *DependsOnCheckpoint* ***check_dependencies**(*ParticipantId on_participant*, *PlanId on_plan*, *RouteId on_route*) **const**

Get any dependencies that this route has on the given route of another participant.

Return A pointer to the relevant dependencies, if any exist. If there is no dependency relevant to the specified route of the participant, then this will be a nullptr.

Parameters

- `[in] on_participant`: The ID of the other participant of interest
- `[in] on_plan`: The ID of the other participant's current plan
- `[in] on_route`: The ID of the other participant's route that is being considered

Class Change

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_Change.hpp

Nested Relationships

Nested Types

- *Class Change::Add*
- *Struct Add::Item*
- *Class Change::Cull*
- *Class Change::Delay*
- *Class Change::Erase*
- *Class Change::Progress*
- *Class Change::RegisterParticipant*
- *Class Change::UnregisterParticipant*
- *Class Change::UpdateParticipantInfo*

Class Documentation

class rmf_traffic::schedule::**Change**

A class that describes a change within the schedule.

class Add

The API for an *Add* change.

Public Functions

Add(*PlanId plan*, std::vector<*Item*> *additions*)

Add a set of routes.

const std::vector<*Item*> &**items**() **const**

A reference to the *Trajectory* that was inserted.

PlanId **plan_id**() **const**

The plan ID that these routes are being added for.

struct Item

A description of an addition.

Public Members

RouteId **route_id**

The ID of the route being added, relative to the plan it belongs to.

StorageId **storage_id**

The storage ID of the route.

ConstRoutePtr **route**

The information for the route being added.

class Cull

A class that describes a culling.

Public Functions

Cull(*Time time*)

Constructor

Parameters

- [in] time: The time before which all routes should be culled

Time **time**() **const**

class Delay

The API for a *Delay* change.

Public Functions

Delay(*Duration duration*)
Add a delay

Parameters
- `[in] duration`: The duration of that delay.

Duration **duration**() **const**
The duration of the delay.

class Erase
A class that describes an erasing change.

Public Functions

Erase(std::vector<*StorageId*> *ids*)
Constructor

Parameters
- `[in] id`: The ID of the route that should be erased

const std::vector<*StorageId*> &**ids**() **const**

class Progress
A class that provides an update on itinerary progression.

Public Functions

Progress(*ProgressVersion version*, std::vector<*CheckpointId*> *checkpoints*)
Constructor.

ProgressVersion **version**() **const**

const std::vector<*CheckpointId*> &**checkpoints**() **const**

class RegisterParticipant
A class that describes a participant registration.

Public Functions

RegisterParticipant(*ParticipantId id*, *ParticipantDescription description*)
Constructor

Parameters
- `[in] id`: The ID of the participant
- `[in] description`: The description of the participant

ParticipantId **id**() **const**
The ID for the participant.

const *ParticipantDescription* &**description**() **const**
The description of the participant.

class UnregisterParticipant
A class that specifies a participant to unregister.

Public Functions

UnregisterParticipant(*ParticipantId id*)
Constructor

Parameters
- `[in] id`: The ID of the participant that is being unregistered.

ParticipantId **id**() **const**
The ID for the participant.

class UpdateParticipantInfo
A class that describes update in the participant info.

Public Functions

UpdateParticipantInfo(*ParticipantId id*, *ParticipantDescription desc*)
Constructor

Parameters
- `[in] id`: The ID of the participant that is being unregistered.

ParticipantId **id**() **const**
The ID for the participant.

ParticipantDescription **description**() **const**
Description for participants.

Class Change::Add

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_Change.hpp

Nested Relationships

This class is a nested type of *Class Change*.

Nested Types

- *Struct Add::Item*

Class Documentation

class rmf_traffic::schedule::*Change*::**Add**
The API for an *Add* change.

Public Functions

Add(*PlanId plan*, std::vector<*Item*> *additions*)
Add a set of routes.

const std::vector<*Item*> &**items**() **const**
A reference to the *Trajectory* that was inserted.

PlanId **plan_id**() **const**
The plan ID that these routes are being added for.

struct Item
A description of an addition.

Public Members

RouteId **route_id**
The ID of the route being added, relative to the plan it belongs to.

StorageId **storage_id**
The storage ID of the route.

ConstRoutePtr **route**
The information for the route being added.

Class Change::Cull

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_Change.hpp

Nested Relationships

This class is a nested type of *Class Change*.

Class Documentation

class rmf_traffic::schedule::*Change*::**Cull**
A class that describes a culling.

Public Functions

Cull(*Time time*)
Constructor

Parameters

- `[in] time`: The time before which all routes should be culled

Time **time**() **const**

Class Change::Delay

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_Change.hpp

Nested Relationships

This class is a nested type of *Class Change*.

Class Documentation

class rmf_traffic::schedule::*Change*::**Delay**
The API for a *Delay* change.

Public Functions

Delay(*Duration duration*)
Add a delay

Parameters

- `[in] duration`: The duration of that delay.

Duration **duration**() **const**
The duration of the delay.

Class Change::Erase

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_Change.hpp

Nested Relationships

This class is a nested type of *Class Change*.

Class Documentation

class rmf_traffic::schedule::*Change*::**Erase**
A class that describes an erasing change.

Public Functions

Erase(std::vector<*StorageId*> *ids*)
Constructor

Parameters

- [in] id: The ID of the route that should be erased

const std::vector<*StorageId*> &**ids**() **const**

Class Change::Progress

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_Change.hpp

Nested Relationships

This class is a nested type of *Class Change*.

Class Documentation

class rmf_traffic::schedule::*Change*::**Progress**
A class that provides an update on itinerary progression.

Public Functions

Progress(*ProgressVersion version*, std::vector<*CheckpointId*> *checkpoints*)
Constructor.

ProgressVersion **version**() **const**

const std::vector<*CheckpointId*> &**checkpoints**() **const**

Class Change::RegisterParticipant

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_Change.hpp

Nested Relationships

This class is a nested type of *Class Change*.

Class Documentation

class `rmf_traffic::schedule::`*`Change`*`::`**`RegisterParticipant`**
A class that describes a participant registration.

Public Functions

`RegisterParticipant`(*ParticipantId* *id*, *ParticipantDescription* *description*)
Constructor

Parameters

- `[in] id`: The ID of the participant
- `[in] description`: The description of the participant

ParticipantId **`id`**`()` **`const`**
The ID for the participant.

`const` *ParticipantDescription* &**`description`**`()` **`const`**
The description of the participant.

Class Change::UnregisterParticipant

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_Change.hpp

Nested Relationships

This class is a nested type of *Class Change*.

Class Documentation

class `rmf_traffic::schedule::`*`Change`*`::`**`UnregisterParticipant`**
A class that specifies a participant to unregister.

Public Functions

UnregisterParticipant(*ParticipantId id*)
Constructor

Parameters

- `[in] id`: The ID of the participant that is being unregistered.

ParticipantId **id**() **const**
The ID for the participant.

Class Change::UpdateParticipantInfo

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_Change.hpp

Nested Relationships

This class is a nested type of *Class Change*.

Class Documentation

class rmf_traffic::schedule::*Change*::**UpdateParticipantInfo**
A class that describes update in the participant info.

Public Functions

UpdateParticipantInfo(*ParticipantId id*, *ParticipantDescription desc*)
Constructor

Parameters

- `[in] id`: The ID of the participant that is being unregistered.

ParticipantId **id**() **const**
The ID for the participant.

ParticipantDescription **description**() **const**
Description for participants.

Class Database

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_Database.hpp

Inheritance Relationships

Base Types

- `public rmf_traffic::schedule::ItineraryViewer` (*Class ItineraryViewer*)
- `public rmf_traffic::schedule::Writer` (*Class Writer*)
- `public rmf_traffic::schedule::Snappable` (*Class Snappable*)

Class Documentation

class `rmf_traffic::schedule::`**Database** : **public** rmf_traffic::schedule::*ItineraryViewer*, **public** rmf_traffic::schedul

A class that maintains a database of scheduled Trajectories. This class is intended to be used only for the canonical RMF traffic schedule database.

The *Viewer* API can be queried to find Trajectories that match certain criteria.

You can also retrieve update patches from a database. To apply those patches to a downstream *Viewer*, it is strongly advised to use the *rmf_traffic::schedule::Mirror* class.

Public Functions

virtual void **set**(*ParticipantId participant*, *PlanId plan*, **const** *Itinerary* &*itinerary*, *StorageId storage_base*, *ItineraryVersion version*) **final**

Set a brand new itinerary for a participant. This will replace any itinerary that is already in the schedule for the participant.

Parameters

- `[in] participant`: The ID of the participant whose itinerary is being updated.
- `[in] plan`: The ID of the plan that this new itinerary belongs to.
- `[in] itinerary`: The new itinerary of the participant.
- `[in] storage_base`: The storage index offset that the database should use for this plan. This should generally be the integer number of total routes that the participant has ever given to the writer prior to setting this new itinerary. This value helps ensure consistent unique IDs for every route, even after a database has failed over or restarted.
- `[in] version`: The version for this itinerary change.

virtual void **extend**(*ParticipantId participant*, **const** *Itinerary* &*routes*, *ItineraryVersion version*) **final**

Add a set of routes to the itinerary of this participant.

Parameters

- `[in] participant`: The ID of the participant whose itinerary is being updated.
- `[in] routes`: The set of routes that should be added to the itinerary.
- `[in] version`: The version for this itinerary change

virtual void **delay**(*ParticipantId participant*, *Duration delay*, *ItineraryVersion version*) **final**
Add a delay to the itinerary from the specified Time.

Nothing about the routes in the itinerary will be changed except that waypoints will shifted through time.

Parameters

- `[in] participant`: The ID of the participant whose itinerary is being delayed.
- `[in] delay`: This is the duration of time to delay all qualifying *Trajectory* Waypoints.
- `[in] version`: The version for this itinerary change

virtual void **reached**(*ParticipantId participant*, *PlanId plan*, **const** std::vector<*CheckpointId*> *&reached_checkpoints*, *ProgressVersion version*) **final**
Indicate that a participant has reached certain checkpoints.

Parameters

- `[in] participant`: The ID of the participant whose progress is being set.
- `[in] plan`: The ID of the plan which progress has been made for.
- `[in] reached_checkpoints`: The set of checkpoints that have been reached. The indices in the vector must correspond to the RouteIds of the plan.
- `[in] version`: The version number for this progress.

virtual void **clear**(*ParticipantId participant*, *ItineraryVersion version*) **final**
Erase an itinerary from this database.

Parameters

- `[in] participant`: The ID of the participant whose itinerary is being erased.
- `[in] version`: The version for this itinerary change

virtual Registration **register_participant**(*ParticipantDescription participant_info*) **final**
Register a new participant.

Return result of registering the new participant.

Parameters

- `[in] participant_info`: Information about the new participant.
- `[in] time`: The time at which the registration is being requested.

virtual void **update_description**(*ParticipantId participant*, *ParticipantDescription desc*) **final**
Updates a participants footprint

Parameters

- `[in] participant`: The ID of the participant to update
- `[in] desc`: The participant description

virtual void **unregister_participant**(*ParticipantId participant*) **final**
Before calling this function on a *Database*, you should set the current time for the database by calling *set_current_time()*. This will allow the database to cull this participant after a reasonable amount of time has passed.

virtual View **query**(**const** *Query* &*parameters*) **const final**
Query this *Viewer* to get a View of the Trajectories inside of it that match the *Query* parameters.

virtual View **query**(**const** *Query*::*Spacetime* &*spacetime*, **const** *Query*::*Participants* &*participants*) **const final**
Alternative signature for *query()*

virtual const std::unordered_set<*ParticipantId*> &**participant_ids**() **const final**
Get the set of active participant IDs.

std::shared_ptr<**const** *ParticipantDescription*> **get_participant**(std::size_t *participant_id*) **const final**

virtual *Version* **latest_version**() **const final**
Get the latest version number of this *Database*.

std::optional<*ItineraryView*> **get_itinerary**(std::size_t *participant_id*) **const final**

std::optional<*PlanId*> **get_current_plan_id**(std::size_t *participant_id*) **const final**

virtual const std::vector<*CheckpointId*> ***get_current_progress**(*ParticipantId participant_id*) **const final**
Get the current progress of a specific participant. If a participant with the specified ID is not registered with the schedule or has never made progress, then this will return a nullptr.

virtual *ProgressVersion* **get_current_progress_version**(*ParticipantId participant_id*) **const final**
Get the current known progress of a specific participant along its current plan. If no progress has been made, this will have a value of 0.

virtual DependencySubscription **watch_dependency**(*Dependency dependency*, std::function<void) > *on_reached*std::function<void> *on_deprecated* **const final**Watch a traffic dependency. When a relevant event happens for the dependency, the on_reached or on_deprecated will be triggered. If the event had already come to pass before this function is called, then the relevant callback will be triggered right away, within the scope of this function.

Only one of the callbacks will ever be triggered, and it will only be triggered at most once.

Return an object that maintains the dependency for the viewer.

Parameters

- `[in] on_reached`: If the dependency is reached, this will be triggered. on_changed will never be triggered afterwards.
- `[in] on_deprecated`: If the plan of the participant changed before it reached this dependency then the dependency is deprecated and this callback will be triggered. on_reached will never be triggered afterwards.

virtual std::shared_ptr<**const** *Snapshot*> **snapshot**() **const final**
Get a snapshot of the schedule.

Database()
Initialize a *Database*.

const *Inconsistencies* &**inconsistencies**() **const**

A description of all inconsistencies currently present in the database. *Inconsistencies* are isolated between Participants.

To fix the inconsistency, the Participant should resend every Itinerary change that was missing from every range, or else send a change that nullifies all previous changes, such as a set(~) or erase(ParticipantId).

Patch **changes**(**const** *Query* &*parameters*, std::optional<*Version*> *after*) **const**

Get the changes in this *Database* that match the given *Query* parameters. If a version number is specified, then the returned *Patch* will reflect the changes that occurred from the specified version to the current version of the schedule.

To get a consistent reflection of the schedule when specifying a base version, it is important that the query parameters are not changed in between calls.

Return A *Patch* of schedule changes that are relevant to the specified query parameters.

Parameters

- `[in] parameters`: The parameters describing what types of schedule entries the mirror cares about.
- `[in] after`: Specify that only changes which come after this version number are desired. If you give a nullopt for this argument, then all changes will be provided.

View **query**(**const** *Query* &*parameters*, *Version* *after*) **const**

View the routes that match the parameters and have changed (been added or delayed) since the specified version. This is useful for viewing incremental changes.

Return a view of the routes that are different since the specified version.

Parameters

- `[in] parameters`: The parameters describing what types of schedule entries are relevant.
- `[in] after`: Specify that only routes which changed after this version number are desired.

Version **cull**(*Time* *time*)

Throw away all itineraries up to the specified time.

Return The new version of the schedule database. If nothing was culled, this version number will remain the same.

Parameters

- `[in] time`: All Trajectories that finish before this time will be culled from the *Database*. Their data will be completely deleted from this *Database* object.

void **set_current_time**(*Time* *time*)

Set the current time on the database. This should be used immediately before calling *unregister_participant()* so that the database can cull the existence of the participant at an appropriate time. There's no need to call this function for any other purpose.

ItineraryVersion **itinerary_version**(*ParticipantId* *participant*) **const**

Get the current itinerary version for the specified participant.

PlanId **latest_plan_id**(*ParticipantId* *participant*) **const**

Get the last Plan ID used by this participant.

This provides the same information as get_current_plan_id, except it throws an exception instead of returning an optional if the participant does not exist.

StorageId **next_storage_base**(*ParticipantId participant*) **const**
Get the last Storage ID used by this participant.

Class DatabaseRectificationRequesterFactory

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_Rectifier.hpp

Inheritance Relationships

Base Type

- `public rmf_traffic::schedule::RectificationRequesterFactory` (*Class RectificationRequesterFactory*)

Class Documentation

class `rmf_traffic::schedule::`**DatabaseRectificationRequesterFactory** : **public** rmf_traffic::schedule::*Re*
This class provides a simple implementation of a *RectificationRequesterFactory* that just hooks directly into a *Database* instance and issues rectification requests when told to based on the current inconsistencies in the *Database*.

Public Functions

DatabaseRectificationRequesterFactory(std::shared_ptr<*Database*> *database*)
This accepts a const-reference to a *Database* instance. Note that this class will store a reference to this *Database*, so its lifecycle is implicitly dependent on the *Database*'s lifecycle.

virtual std::unique_ptr<*RectificationRequester*> **make**(*Rectifier rectifier*, *ParticipantId participant_id*) **final**
Create a *RectificationRequester* to be held by a Participant

Parameters

- `[in] rectifier`: This rectifier can be used by the *RectificationRequester* to ask the participant to retransmit some of its changes.
- `[in] participant_id`: The ID of the participant that will hold onto this *RectificationRequester*. This is the same participant that the rectifier will request retransmissions to.

void **rectify**()
Call this function to instruct all the RectificationRequestors produced by this factory to perform their rectifications.

void **change_database**(std::shared_ptr<*Database*> *new_database*)
Change the database that will be getting rectified. This can be used to switch to rectifying a new database fork.

Class Inconsistencies

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_Inconsistencies.hpp

Nested Relationships

Nested Types

- *Struct Inconsistencies::Element*
- *Class Inconsistencies::Ranges*
- *Struct Ranges::Range*

Class Documentation

class rmf_traffic::schedule::**Inconsistencies**

An Inconsistency occurs when one or more ItineraryVersion values get skipped by the inputs into the database. This container expresses the ranges of which ItineraryVersions were skipped for a single Participant.

Iterators

Public Types

using base_iter = rmf_traffic::detail::*forward_iterator*<E, I, F>

using const_iterator = *base_iter*<**const** *Element*, IterImpl, *Inconsistencies*>

Public Functions

const_iterator **begin**() **const**

Get the beginning iterator.

const_iterator **cbegin**() **const**

Explicitly const-qualified alternative for *begin()*

const_iterator **end**() **const**

Get the one-past-the-end iterator.

const_iterator **cend**() **const**

Explicitly const-qualified alternative for *end()*

const_iterator **find**(*ParticipantId id*) **const**

Get the iterator for this ParticipantId.

std::size_t **size**() **const**

Get the number of participants with inconsistencies.

struct Element

An element of the *Inconsistencies* container. This tells the ranges of inconsistencies that are present for the specified Participant.

Public Members

ParticipantId **participant**

Ranges **ranges**

class Ranges

A container of the ranges of inconsistencies for a single participant.

Public Types

using const_iterator = *base_iter*<**const** *Range*, IterImpl, *Ranges*>

Public Functions

const_iterator **begin**() **const**

Get the beginning iterator.

const_iterator **cbegin**() **const**

Explicitly const-qualified alternative for *begin()*

const_iterator **end**() **const**

Get the one-past-the-end iterator.

const_iterator **cend**() **const**

Explicitly const-qualified alternative for *end()*

std::size_t **size**() **const**

Get the number of ranges in this container.

ItineraryVersion **last_known_version**() **const**

Get the value of the last itinerary version that has been received.

struct Range

A single range of inconsistencies within a participant.

Every version between (and including) the lower and upper versions have not been received by the *Database*.

Public Members

ItineraryVersion **lower**

ItineraryVersion **upper**

Class Inconsistencies::Ranges

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_Inconsistencies.hpp

Nested Relationships

This class is a nested type of *Class Inconsistencies*.

Nested Types

- *Struct Ranges::Range*

Class Documentation

class rmf_traffic::schedule::*Inconsistencies*::**Ranges**

A container of the ranges of inconsistencies for a single participant.

Public Types

using const_iterator = *base_iter*<**const** *Range*, IterImpl, *Ranges*>

Public Functions

const_iterator **begin**() **const**

Get the beginning iterator.

const_iterator **cbegin**() **const**

Explicitly const-qualified alternative for *begin()*

const_iterator **end**() **const**

Get the one-past-the-end iterator.

const_iterator **cend**() **const**

Explicitly const-qualified alternative for *end()*

std::size_t **size**() **const**

Get the number of ranges in this container.

ItineraryVersion **last_known_version**() **const**

Get the value of the last itinerary version that has been received.

struct Range

A single range of inconsistencies within a participant.

Every version between (and including) the lower and upper versions have not been received by the *Database*.

Public Members

ItineraryVersion **lower**

ItineraryVersion **upper**

Class ItineraryViewer

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_Viewer.hpp

Nested Relationships

Nested Types

- *Class ItineraryViewer::DependencySubscription*

Inheritance Relationships

Base Type

- `public rmf_traffic::schedule::Viewer` (*Class Viewer*)

Derived Types

- `public rmf_traffic::schedule::Database` (*Class Database*)
- `public rmf_traffic::schedule::Mirror` (*Class Mirror*)

Class Documentation

class `rmf_traffic::schedule::`**ItineraryViewer** : **public virtual** rmf_traffic::schedule::*Viewer*

A pure abstract interface class that extends *Viewer* to allow users to explicitly request the itinerary of a specific participant.

Note This interface class is separate from *Viewer* because it is not generally needed by the traffic planning or negotiation systems, and the *Snapshot* class can perform better if it does not need to provide this function.

Subclassed by *rmf_traffic::schedule::Database*, *rmf_traffic::schedule::Mirror*

Public Functions

virtual std::optional<*ItineraryView*> **get_itinerary**(*ParticipantId participant_id*) **const** = 0

Get the itinerary of a specific participant if it is available. If a participant with the specified ID is not registered with the schedule or has never submitted an itinerary, then this will return a nullopt.

virtual std::optional<*PlanId*> **get_current_plan_id**(*ParticipantId participant_id*) **const** = 0

Get the current plan ID of a specific participant if it is available. If a participant with the specified ID is not registered with the schedule, then this will return a nullopt.

virtual const std::vector<*CheckpointId*> ***get_current_progress**(*ParticipantId participant_id*) **const** = 0

Get the current progress of a specific participant. If a participant with the specified ID is not registered with the schedule or has never made progress, then this will return a nullptr.

virtual *ProgressVersion* **get_current_progress_version**(*ParticipantId* *participant_id*) **const** = 0

Get the current known progress of a specific participant along its current plan. If no progress has been made, this will have a value of 0.

virtual *DependencySubscription* **watch_dependency**(*Dependency* *dependency*, std::function<void()> *on_reached*std::function<void> *on_deprecated* **const** = 0Watch a traffic dependency. When a relevant event happens for the dependency, the on_reached or on_deprecated will be triggered. If the event had already come to pass before this function is called, then the relevant callback will be triggered right away, within the scope of this function.

Only one of the callbacks will ever be triggered, and it will only be triggered at most once.

Return an object that maintains the dependency for the viewer.

Parameters

- `[in] on_reached`: If the dependency is reached, this will be triggered. on_changed will never be triggered afterwards.
- `[in] on_deprecated`: If the plan of the participant changed before it reached this dependency then the dependency is deprecated and this callback will be triggered. on_reached will never be triggered afterwards.

virtual ~ItineraryViewer() = default

class DependencySubscription

A handle for maintaining a dependency on the progress of an itinerary.

Public Functions

bool **reached**() **const**

The dependency was reached by the participant.

bool **deprecated**() **const**

The plan of the participant changed before it ever reached the dependency

bool **finished**() **const**

Equivalent to *reached()* || *deprecated()*

Dependency **dependency**() **const**

Check what dependency this is subscribed to.

Class ItineraryViewer::DependencySubscription

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_Viewer.hpp

Nested Relationships

This class is a nested type of *Class ItineraryViewer*.

Class Documentation

class rmf_traffic::schedule::*ItineraryViewer*::**DependencySubscription**
A handle for maintaining a dependency on the progress of an itinerary.

Public Functions

bool **reached**() **const**
The dependency was reached by the participant.

bool **deprecated**() **const**
The plan of the participant changed before it ever reached the dependency

bool **finished**() **const**
Equivalent to *reached()* || *deprecated()*

Dependency **dependency**() **const**
Check what dependency this is subscribed to.

Class Mirror

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_Mirror.hpp

Inheritance Relationships

Base Types

- public rmf_traffic::schedule::ItineraryViewer (*Class ItineraryViewer*)
- public rmf_traffic::schedule::Snappable (*Class Snappable*)

Class Documentation

class rmf_traffic::schedule::**Mirror** : **public** rmf_traffic::schedule::*ItineraryViewer*, **public** rmf_traffic::schedule::
A class that maintains a mirror of a *Database* of scheduled Trajectories. This class is intended to provide a cache of the scheduled Trajectories to processes or threads that do not contain the original upstream copy of the *rmf_traffic::schedule::Database*.

The *Mirror* is designed to mirror a relevant subset of the schedule database.

Public Functions

virtual View **query**(**const** *Query* &*parameters*) **const final**
Query this *Viewer* to get a View of the Trajectories inside of it that match the *Query* parameters.

virtual View **query**(**const** *Query*::*Spacetime* &*spacetime*, **const** *Query*::*Participants* &*participants*) **const final**
Alternative signature for *query()*

virtual const std::unordered_set<*ParticipantId*> &**participant_ids**() **const final**
Get the set of active participant IDs.

std::shared_ptr<**const** *ParticipantDescription*> **get_participant**(std::size_t *participant_id*) **const final**

std::optional<*ItineraryView*> **get_itinerary**(std::size_t *participant_id*) **const final**

virtual *Version* **latest_version**() **const final**
Get the latest version number of this *Database*.

virtual std::optional<*PlanId*> **get_current_plan_id**(*ParticipantId* *participant_id*) **const final**
Get the current plan ID of a specific participant if it is available. If a participant with the specified ID is not registered with the schedule, then this will return a nullopt.

virtual const std::vector<*CheckpointId*> ***get_current_progress**(*ParticipantId* *participant_id*) **const final**
Get the current progress of a specific participant. If a participant with the specified ID is not registered with the schedule or has never made progress, then this will return a nullptr.

virtual *ProgressVersion* **get_current_progress_version**(*ParticipantId* *participant_id*) **const final**
Get the current known progress of a specific participant along its current plan. If no progress has been made, this will have a value of 0.

virtual DependencySubscription **watch_dependency**(*Dependency* *dependency*, std::function<void()> *on_reached*std::function<void> *on_deprecated* **const final**Watch a traffic dependency. When a relevant event happens for the dependency, the on_reached or on_deprecated will be triggered. If the event had already come to pass before this function is called, then the relevant callback will be triggered right away, within the scope of this function.

Only one of the callbacks will ever be triggered, and it will only be triggered at most once.

Return an object that maintains the dependency for the viewer.

Parameters

- `[in] on_reached`: If the dependency is reached, this will be triggered. on_changed will never be triggered afterwards.
- `[in] on_deprecated`: If the plan of the participant changed before it reached this dependency then the dependency is deprecated and this callback will be triggered. on_reached will never be triggered afterwards.

virtual std::shared_ptr<**const** *Snapshot*> **snapshot**() **const final**
Get a snapshot of the schedule.

Mirror()
Create a database mirror.

void **update_participants_info**(**const** *ParticipantDescriptionsMap* &*participants*)
Update the known participants and their descriptions.

bool **update**(**const** *Patch* &*patch*)
Update this mirror.

Return true if this update is okay. false if the base version of the patch does not match

Database **fork**() **const**
Fork a new database off of this *Mirror*. The state of the new database will match the last state of the upstream database that this *Mirror* knows about.

Class Negotiation

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_Negotiation.hpp

Nested Relationships

Nested Types

- *Class Negotiation::Evaluator*
- *Template Struct Negotiation::SearchResult*
- *Struct Negotiation::Submission*
- *Class Negotiation::Table*
- *Class Table::Viewer*
- *Class Viewer::Endpoint*
- *Struct Negotiation::VersionedKey*

Class Documentation

class rmf_traffic::schedule::**Negotiation**

Public Types

enum SearchStatus
This enumeration describes the status of a search attempt.

Values:

enumerator Deprecated
The requested *Table* existed, but the requested version is out of date.

enumerator Absent
The requested version of this *Table* has never been seen by this *Negotiation*.

enumerator Found
The requested *Table* has been found.

using VersionedKeySequence = std::vector<*VersionedKey*>
The versioned key sequence can be used to select tables while demanding specific versions for those tables.

using Proposal = std::vector<*Submission*>

using Alternatives = std::vector<*Itinerary*>

using TablePtr = std::shared_ptr<*Table*>

using ConstTablePtr = std::shared_ptr<**const** *Table*>

Public Functions

const std::unordered_set<*ParticipantId*> &**participants**() **const**
Get the participants that are currently involved in this negotiation.

void **add_participant**(*ParticipantId p*)
Add a new participant to the negotiation. This participant will become involved in the negotiation, and must give its consent for any agreement to be finalized.

bool **ready**() **const**
Returns true if at least one proposal is available that has the consent of every participant.

bool **complete**() **const**
Returns true if all possible proposals have been received and are ready to be evaluated.

Note that *ready()* may still be false if *complete()* is true, in the event that all proposals have been rejected.

TablePtr **table**(*ParticipantId for_participant*, **const** std::vector<*ParticipantId*> &*to_accommodate*)
Get a *Negotiation::Table* that provides a view into what participants are proposing.

This function does not care about table versioning.

See *find()*

Parameters

- `[in] for_participant`: The participant that is supposed to be viewing this *Table*. The itineraries of this participant will be left off of the *Table*.
- `[in] to_accommodate`: The set of participants who are being accommodated at this *Table*. The ordering of the participants in this set is hierarchical where each participant is accommodating all of the participants that come before it.

ConstTablePtr **table**(*ParticipantId for_participant*, **const** std::vector<*ParticipantId*> &*to_accommodate*) **const**

TablePtr **table**(**const** std::vector<*ParticipantId*> &*sequence*)
Get a *Negotiation::Table* that corresponds to the given participant sequence. For a table in terms of for_participant and to_accomodate, you would call: table([to_accommodate..., for_participant])

This function does not care about table versioning.

See *find()*

Parameters

- `[in] sequence`: The participant sequence that corresponds to the desired table. This is equivalent to [to_accommodate..., for_participant]

ConstTablePtr **table**(**const** std::vector<*ParticipantId*> &*sequence*) **const**

SearchResult<*TablePtr*> **find**(*ParticipantId for_participant*, **const** *VersionedKeySequence &to_accommodate*)
Find a table, requesting specific versions

See *table()*

SearchResult<*ConstTablePtr*> **find**(*ParticipantId for_participant*, **const** *VersionedKeySequence &to_accommodate*) **const**
const-qualified *find()*

SearchResult<*TablePtr*> **find**(**const** *VersionedKeySequence &sequence*)
Find a table, requesting specific versions

See *table()*

SearchResult<*ConstTablePtr*> **find**(**const** *VersionedKeySequence &sequence*) **const**
const-qualified *find()*

ConstTablePtr **evaluate**(**const** *Evaluator &evaluator*) **const**
Evaluate the proposals that are available.

Return the negotiation table that was considered the best. Call *Table::proposal()* on this return value to see the full proposal. If there was no

Public Static Functions

static rmf_utils::optional<*Negotiation*> **make**(std::shared_ptr<**const** *Viewer*> *schedule_viewer*, std::vector<*ParticipantId*> *participants*)
Begin a negotiation.

Return a negotiation between the given participants. If the *Viewer* is missing a description of any of the participants, then a nullopt will be returned instead.

See *make_shared()*

Parameters

- `[in] viewer`: A reference to the schedule viewer that represents the most up-to-date schedule.
- `[in] participants`: The participants who are involved in the schedule negotiation.

static std::shared_ptr<*Negotiation*> **make_shared**(std::shared_ptr<**const** *Viewer*> *schedule_viewer*, std::vector<*ParticipantId*> *participants*)
Begin a negotiation.

Return a negotiation between the given participants. If the *Viewer* is missing a description of any of the participants, then a nullptr will be returned instead.

See *make()*

Parameters

- `[in] viewer`: A reference to the schedule viewer that represents the most up-to-date schedule.
- `[in] participants`: The participants who are involved in the schedule negotiation.

class Evaluator

A pure abstract interface class for choosing the best proposal.

Subclassed by *rmf_traffic::schedule::QuickestFinishEvaluator*

Public Functions

virtual std::size_t **choose**(**const** std::vector<**const** *Proposal**> &*proposals*) **const** = 0

Given a set of proposals, choose the one that is the "best". It is up to the implementation of the *Evaluator* to decide how to rank proposals.

virtual ~Evaluator() = default

template<typename **Ptr**>
struct SearchResult

Public Functions

inline bool **deprecated**() **const**

inline bool **absent**() **const**

inline bool **found**() **const**

inline operator bool() **const**

Public Members

SearchStatus **status**

The status of the search.

Ptr **table**

The *Table* that was searched for (or nullptr if status is Deprecated or Absent)

struct Submission

Public Members

ParticipantId **participant**

PlanId **plan**

Itinerary **itinerary**

class Table : **public** std::enable_shared_from_this<*Table*>

The *Negotiation::Table* class gives a view of what the other negotiation participants have proposed.

A *Table* instance is meant to be viewed by a specific participant and displays the proposals of other participants for a specific hierarchies of accommodations. See the documentation of *Negotiation::table()*.

Alongside the views of the other *Negotiation* participants, the View provided by the *Table* instance will show the itineraries of schedule participants that are not part of the *Negotiation*. That way the external itineraries can also be accounted for when planning a submission based on this *Table*.

Public Types

using ViewerPtr = std::shared_ptr<**const** *Viewer*>

Public Functions

ViewerPtr **viewer**() **const**
Get a viewer for this *Table*. The *Viewer* can be safely used across multiple threads.

const *Itinerary* ***submission**() **const**
Return the submission on this *Negotiation Table* if it has one.

Version **version**() **const**
The a pointer to the latest itinerary version that was submitted to this table, if one was submitted at all.

const *Proposal* &**proposal**() **const**
The proposal on this table so far. This will include the latest itinerary that has been submitted to this *Table* if anything has been submitted. Otherwise it will only include the submissions that underlie this table.

ParticipantId **participant**() **const**
The participant that is meant to submit to this *Table*.

const *VersionedKeySequence* &**sequence**() **const**
The sequence key that refers to this table. This is equivalent to [to_accommodate..., for_participant]

std::vector<*ParticipantId*> **unversioned_sequence**() **const**
The versioned sequence key that refers to this table.

bool **submit**(*PlanId plan_id*, std::vector<*Route*> *itinerary*, *Version version*)
Submit a proposal for a participant that accommodates some of the other participants in the negotiation (or none if an empty vector is given for the to_accommodate argument).

Return True if the submission was accepted. False if the version was out of date and nothing changed in the negotiation.

Parameters
- `[in] plan_id`: A unique identifier for this plan. If this plan is selected by the negotiation, then this ID will be submitted to the traffic schedule as the PlanId for this participant.
- `[in] itinerary`: The itinerary that is being submitted by this participant.
- `[in] version`: A version number assigned to the submission. If this is less or equal to the last version number given, then nothing will change.

bool **reject**(*Version version*, *ParticipantId rejected_by*, *Alternatives alternatives*)
Reject the submission of this *Negotiation::Table*. This indicates that the underlying proposals are infeasible for the Participant of this *Table* to accommodate. The rejecter should give a set of alternative rollouts that it is capable of. That way the proposer for this *Table* can submit an itinerary that accommodates it.

Return True if the rejection was accepted. False if the version was out of date and nothing changed in the negotiation.

Parameters
- `[in] version`: A version number assigned to the submission. If this is equal to or greater than the last version number given, then this table will be put into a rejected state until a higher proposal version is submitted.
- `[in] rejected_by`: The participant who is rejecting this proposal

- `[in] alternatives`: A set of rollouts that could be used by the participant that is rejecting this proposal. The proposer should use this information to offer a proposal that can accommodate at least one of these rollouts.

bool **`rejected`**`() const`
Returns true if the proposal put on this *Table* has been rejected.

void **`forfeit`**(*Version version*)
Give up on this *Negotiation Table*. This should be called when the participant that is supposed to submit to this *Table* is unable to find a feasible proposal.

bool **`forfeited`**`() const`
Returns true if the proposer for this *Table* has forfeited.

bool **`defunct`**`() const`
Returns true if any of this table's ancestors were rejected or forfeited. When that happens, this *Table* will no longer have any effect on the *Negotiation*.

TablePtr **`respond`**(*ParticipantId by_participant*)
If by_participant can respond to this table, then this will return a TablePtr that by_participant can submit a proposal to.

If this function is called before anything has been submitted to this *Table*, then it will certainly return a nullptr.

ConstTablePtr **`respond`**(*ParticipantId by_participant*) **`const`**

TablePtr **`parent`**`()`
Get the parent *Table* of this *Table* if it has a parent.

ConstTablePtr **`parent`**`() const`

std::vector<*TablePtr*> **`children`**`()`
Get the children of this *Table* if any children exist.

std::vector<*ConstTablePtr*> **`children`**`() const`

bool **`ongoing`**`() const`
Return true if the negotiation is ongoing (i.e. the *Negotiation* instance that created this table is still alive). When the *Negotiation* instance that this *Table* belongs to has destructed, this will begin to return false.

`class Viewer`

Public Types

`using View` = schedule::*Viewer*::*View*

`using AlternativeMap` = std::unordered_map<*ParticipantId*, std::shared_ptr<*Alternatives*>>

Public Functions

View **query**(**const** *Query*::*Spacetime* &*parameters*, **const** *VersionedKeySequence* &*alternatives*) **const**
View this table with the given parameters.

Parameters
- `[in] parameters`: The spacetime parameters to filter irrelevant routes out of the view
- `[in] rollouts`: The selection of which rollout alternatives should be viewed for the participants who have rejected this proposal in the past.

std::unordered_map<*ParticipantId*, *Endpoint*> **initial_endpoints**(**const** *VersionedKeySequence* &*alternatives*) **const**
Get the set of initial waypoints for the negotiation participants.

std::unordered_map<*ParticipantId*, *Endpoint*> **final_endpoints**(**const** *VersionedKeySequence* &*alterantives*) **const**
Get the set of final waypoints for the negotiation participants.

const *AlternativeMap* &**alternatives**() **const**
When a *Negotiation::Table* is rejected by one of the participants who is supposed to respond, they can offer a set of rollout alternatives. If the proposer can accommodate one of the alternatives for each responding participant, then the negotiation might be able to proceed. This map gives the alternatives for each participant that has provided them.

const *Proposal* &**base_proposals**() **const**
The proposals submitted to the predecessor tables.

std::shared_ptr<**const** *ParticipantDescription*> **get_description**(*ParticipantId* *participant_id*) **const**
Get the description of a participant in this *Viewer*.

ParticipantId **participant_id**() **const**
Get the Participant ID of the participant who should submit to this table.

rmf_utils::optional<*ParticipantId*> **parent_id**() **const**
If the *Table* has a parent, get its Participant ID.

const *VersionedKeySequence* &**sequence**() **const**
The sequence of the table that is being viewed.

bool **defunct**() **const**
Returns true if the table of this viewer is no longer relevant. Unlike the other fields of the *Viewer*, this is not a snapshot of the table's state when the *Viewer* was created; instead this defunct status will remain in sync with the state of the source *Table*.

bool **rejected**() **const**
Returns true if the proposal put on this *Table* has been rejected.

bool **forfeited**() **const**
Returns true if the proposer for this *Table* has forfeited.

const *Itinerary* ***submission**() **const**
Return the submission on this *Negotiation Table* if it has one.

std::optional<rmf_traffic::*Time*> **earliest_base_proposal_time**() **const**
The earliest start time of any of the proposals in the table.

std::optional<rmf_traffic::*Time*> **latest_base_proposal_time**() **const**
The latest finish time of any of the proposals in the table.

class Endpoint
View the first or last (depending on context) waypoint in a negotiation participant's itinerary or alternative.

Public Functions

ParticipantId **participant**() **const**
The ID of the participant.

PlanId **plan_id**() **const**
The ID of the plan for this endpoint.

RouteId **route_id**() **const**
The ID of the route for this endpoint.

const rmf_traffic::*Trajectory*::*Waypoint* &**waypoint**() **const**
The first or last (depending on context) waypoint.

const std::string &**map**() **const**
The map that the endpoint is on.

const *ParticipantDescription* &**description**() **const**
The description of the participant.

struct VersionedKey
This struct is used to select a child table, demaning a specific version.

Public Functions

inline bool **operator==**(**const** *VersionedKey* &*other*) **const**

inline bool **operator!=**(**const** *VersionedKey* &*other*) **const**

Public Members

ParticipantId **participant**

Version **version**

Class Negotiation::Evaluator

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_Negotiation.hpp

Nested Relationships

This class is a nested type of *Class Negotiation*.

Inheritance Relationships

Derived Type

- `public rmf_traffic::schedule::QuickestFinishEvaluator` (*Class QuickestFinishEvaluator*)

Class Documentation

class `rmf_traffic::schedule::`*`Negotiation`*`::`**`Evaluator`**

A pure abstract interface class for choosing the best proposal.

Subclassed by *rmf_traffic::schedule::QuickestFinishEvaluator*

Public Functions

virtual std::size_t **choose**(**const** std::vector<**const** *Proposal**> &*proposals*) **const** = 0

Given a set of proposals, choose the one that is the "best". It is up to the implementation of the *Evaluator* to decide how to rank proposals.

virtual **~Evaluator**() = default

Class Negotiation::Table

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_Negotiation.hpp

Nested Relationships

This class is a nested type of *Class Negotiation*.

Nested Types

- *Class Table::Viewer*
- *Class Viewer::Endpoint*

Inheritance Relationships

Base Type

- `public std::enable_shared_from_this< Table >`

Class Documentation

class `rmf_traffic::schedule::Negotiation::Table` : **public** std::enable_shared_from_this<*Table*>

The *Negotiation::Table* class gives a view of what the other negotiation participants have proposed.

A *Table* instance is meant to be viewed by a specific participant and displays the proposals of other participants for a specific hierarchies of accommodations. See the documentation of *Negotiation::table()*.

Alongside the views of the other *Negotiation* participants, the View provided by the *Table* instance will show the itineraries of schedule participants that are not part of the *Negotiation*. That way the external itineraries can also be accounted for when planning a submission based on this *Table*.

Public Types

using ViewerPtr = std::shared_ptr<**const** *Viewer*>

Public Functions

ViewerPtr **viewer**() **const**

Get a viewer for this *Table*. The *Viewer* can be safely used across multiple threads.

const *Itinerary* ***submission**() **const**

Return the submission on this *Negotiation Table* if it has one.

Version **version**() **const**

The a pointer to the latest itinerary version that was submitted to this table, if one was submitted at all.

const *Proposal* &**proposal**() **const**

The proposal on this table so far. This will include the latest itinerary that has been submitted to this *Table* if anything has been submitted. Otherwise it will only include the submissions that underlie this table.

ParticipantId **participant**() **const**

The participant that is meant to submit to this *Table*.

const *VersionedKeySequence* &**sequence**() **const**

The sequence key that refers to this table. This is equivalent to [to_accommodate..., for_participant]

std::vector<*ParticipantId*> **unversioned_sequence**() **const**

The versioned sequence key that refers to this table.

bool **submit**(*PlanId plan_id*, std::vector<*Route*> *itinerary*, *Version version*)

Submit a proposal for a participant that accommodates some of the other participants in the negotiation (or none if an empty vector is given for the to_accommodate argument).

Return True if the submission was accepted. False if the version was out of date and nothing changed in the negotiation.

Parameters

- `[in] plan_id`: A unique identifier for this plan. If this plan is selected by the negotiation, then this ID will be submitted to the traffic schedule as the PlanId for this participant.
- `[in] itinerary`: The itinerary that is being submitted by this participant.
- `[in] version`: A version number assigned to the submission. If this is less or equal to the last version number given, then nothing will change.

bool **reject**(*Version version*, *ParticipantId rejected_by*, *Alternatives alternatives*)
Reject the submission of this *Negotiation::Table*. This indicates that the underlying proposals are infeasible for the Participant of this *Table* to accommodate. The rejecter should give a set of alternative rollouts that it is capable of. That way the proposer for this *Table* can submit an itinerary that accommodates it.

Return True if the rejection was accepted. False if the version was out of date and nothing changed in the negotiation.

Parameters

- `[in] version`: A version number assigned to the submission. If this is equal to or greater than the last version number given, then this table will be put into a rejected state until a higher proposal version is submitted.
- `[in] rejected_by`: The participant who is rejecting this proposal
- `[in] alternatives`: A set of rollouts that could be used by the participant that is rejecting this proposal. The proposer should use this information to offer a proposal that can accommodate at least one of these rollouts.

bool **rejected**() **const**
Returns true if the proposal put on this *Table* has been rejected.

void **forfeit**(*Version version*)
Give up on this *Negotiation Table*. This should be called when the participant that is supposed to submit to this *Table* is unable to find a feasible proposal.

bool **forfeited**() **const**
Returns true if the proposer for this *Table* has forfeited.

bool **defunct**() **const**
Returns true if any of this table's ancestors were rejected or forfeited. When that happens, this *Table* will no longer have any effect on the *Negotiation*.

TablePtr **respond**(*ParticipantId by_participant*)
If by_participant can respond to this table, then this will return a TablePtr that by_participant can submit a proposal to.

If this function is called before anything has been submitted to this *Table*, then it will certainly return a nullptr.

ConstTablePtr **respond**(*ParticipantId by_participant*) **const**

TablePtr **parent**()
Get the parent *Table* of this *Table* if it has a parent.

ConstTablePtr **parent**() **const**

std::vector<*TablePtr*> **children**()
Get the children of this *Table* if any children exist.

std::vector<*ConstTablePtr*> **children**() **const**

bool **ongoing**() **const**
Return true if the negotiation is ongoing (i.e. the *Negotiation* instance that created this table is still alive). When the *Negotiation* instance that this *Table* belongs to has destructed, this will begin to return false.

class Viewer

Public Types

using View = schedule::*Viewer*::*View*

using AlternativeMap = std::unordered_map<*ParticipantId*, std::shared_ptr<*Alternatives*>>

Public Functions

View **query**(**const** *Query*::*Spacetime* &*parameters*, **const** *VersionedKeySequence* &*alternatives*) **const**
View this table with the given parameters.

Parameters
- `[in] parameters`: The spacetime parameters to filter irrelevant routes out of the view
- `[in] rollouts`: The selection of which rollout alternatives should be viewed for the participants who have rejected this proposal in the past.

std::unordered_map<*ParticipantId*, *Endpoint*> **initial_endpoints**(**const** *VersionedKeySequence* &*alternatives*) **const**
Get the set of initial waypoints for the negotiation participants.

std::unordered_map<*ParticipantId*, *Endpoint*> **final_endpoints**(**const** *VersionedKeySequence* &*alterantives*) **const**
Get the set of final waypoints for the negotiation participants.

const *AlternativeMap* &**alternatives**() **const**
When a *Negotiation::Table* is rejected by one of the participants who is supposed to respond, they can offer a set of rollout alternatives. If the proposer can accommodate one of the alternatives for each responding participant, then the negotiation might be able to proceed. This map gives the alternatives for each participant that has provided them.

const *Proposal* &**base_proposals**() **const**
The proposals submitted to the predecessor tables.

std::shared_ptr<**const** *ParticipantDescription*> **get_description**(*ParticipantId* *participant_id*) **const**
Get the description of a participant in this *Viewer*.

ParticipantId **participant_id**() **const**
Get the Participant ID of the participant who should submit to this table.

rmf_utils::optional<*ParticipantId*> **parent_id**() **const**
If the *Table* has a parent, get its Participant ID.

const *VersionedKeySequence* &**sequence**() **const**
The sequence of the table that is being viewed.

bool **defunct**() **const**
Returns true if the table of this viewer is no longer relevant. Unlike the other fields of the *Viewer*, this is not a snapshot of the table's state when the *Viewer* was created; instead this defunct status will remain in sync with the state of the source *Table*.

bool **rejected**() **const**
Returns true if the proposal put on this *Table* has been rejected.

bool **forfeited**() **const**
Returns true if the proposer for this *Table* has forfeited.

const *Itinerary* ***submission**() **const**
Return the submission on this *Negotiation Table* if it has one.

std::optional<rmf_traffic::*Time*> **earliest_base_proposal_time**() **const**
The earliest start time of any of the proposals in the table.

std::optional<rmf_traffic::*Time*> **latest_base_proposal_time**() **const**
The latest finish time of any of the proposals in the table.

class Endpoint
View the first or last (depending on context) waypoint in a negotiation participant's itinerary or alternative.

Public Functions

ParticipantId **participant**() **const**
The ID of the participant.

PlanId **plan_id**() **const**
The ID of the plan for this endpoint.

RouteId **route_id**() **const**
The ID of the route for this endpoint.

const rmf_traffic::*Trajectory*::*Waypoint* &**waypoint**() **const**
The first or last (depending on context) waypoint.

const std::string &**map**() **const**
The map that the endpoint is on.

const *ParticipantDescription* &**description**() **const**
The description of the participant.

Class Table::Viewer

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_Negotiation.hpp

Nested Relationships

This class is a nested type of *Class Negotiation::Table*.

Nested Types

- *Class Viewer::Endpoint*

Class Documentation

class rmf_traffic::schedule::*Negotiation*::*Table*::**Viewer**

Public Types

using View = schedule::*Viewer*::*View*

using AlternativeMap = std::unordered_map<*ParticipantId*, std::shared_ptr<*Alternatives*>>

Public Functions

View **query**(**const** *Query*::*Spacetime* &*parameters*, **const** *VersionedKeySequence* &*alternatives*) **const**

View this table with the given parameters.

Parameters

- `[in] parameters`: The spacetime parameters to filter irrelevant routes out of the view
- `[in] rollouts`: The selection of which rollout alternatives should be viewed for the participants who have rejected this proposal in the past.

std::unordered_map<*ParticipantId*, *Endpoint*> **initial_endpoints**(**const** *VersionedKeySequence* &*alternatives*) **const**

Get the set of initial waypoints for the negotiation participants.

std::unordered_map<*ParticipantId*, *Endpoint*> **final_endpoints**(**const** *VersionedKeySequence* &*alterantives*) **const**

Get the set of final waypoints for the negotiation participants.

const *AlternativeMap* &**alternatives**() **const**

When a *Negotiation::Table* is rejected by one of the participants who is supposed to respond, they can offer a set of rollout alternatives. If the proposer can accommodate one of the alternatives for each responding participant, then the negotiation might be able to proceed. This map gives the alternatives for each participant that has provided them.

const *Proposal* &**base_proposals**() **const**

The proposals submitted to the predecessor tables.

std::shared_ptr<**const** *ParticipantDescription*> **get_description**(*ParticipantId* *participant_id*) **const**

Get the description of a participant in this *Viewer*.

ParticipantId **participant_id**() **const**

Get the Participant ID of the participant who should submit to this table.

rmf_utils::optional<*ParticipantId*> **parent_id**() **const**

If the *Table* has a parent, get its Participant ID.

const *VersionedKeySequence* &**sequence**() **const**

The sequence of the table that is being viewed.

bool **defunct**() **const**
: Returns true if the table of this viewer is no longer relevant. Unlike the other fields of the *Viewer*, this is not a snapshot of the table's state when the *Viewer* was created; instead this defunct status will remain in sync with the state of the source *Table*.

bool **rejected**() **const**
: Returns true if the proposal put on this *Table* has been rejected.

bool **forfeited**() **const**
: Returns true if the proposer for this *Table* has forfeited.

const *Itinerary* ***submission**() **const**
: Return the submission on this *Negotiation Table* if it has one.

std::optional<rmf_traffic::*Time*> **earliest_base_proposal_time**() **const**
: The earliest start time of any of the proposals in the table.

std::optional<rmf_traffic::*Time*> **latest_base_proposal_time**() **const**
: The latest finish time of any of the proposals in the table.

class Endpoint
: View the first or last (depending on context) waypoint in a negotiation participant's itinerary or alternative.

Public Functions

ParticipantId **participant**() **const**
: The ID of the participant.

PlanId **plan_id**() **const**
: The ID of the plan for this endpoint.

RouteId **route_id**() **const**
: The ID of the route for this endpoint.

const rmf_traffic::*Trajectory*::*Waypoint* &**waypoint**() **const**
: The first or last (depending on context) waypoint.

const std::string &**map**() **const**
: The map that the endpoint is on.

const *ParticipantDescription* &**description**() **const**
: The description of the participant.

Class Viewer::Endpoint

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_Negotiation.hpp

Nested Relationships

This class is a nested type of *Class Table::Viewer*.

Class Documentation

class rmf_traffic::schedule::*Negotiation*::*Table*::*Viewer*::**Endpoint**

View the first or last (depending on context) waypoint in a negotiation participant's itinerary or alternative.

Public Functions

ParticipantId **participant**() **const**

The ID of the participant.

PlanId **plan_id**() **const**

The ID of the plan for this endpoint.

RouteId **route_id**() **const**

The ID of the route for this endpoint.

const rmf_traffic::*Trajectory*::*Waypoint* &**waypoint**() **const**

The first or last (depending on context) waypoint.

const std::string &**map**() **const**

The map that the endpoint is on.

const *ParticipantDescription* &**description**() **const**

The description of the participant.

Class Negotiator

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_Negotiator.hpp

Nested Relationships

Nested Types

- *Class Negotiator::Responder*

Inheritance Relationships

Derived Types

- public rmf_traffic::agv::SimpleNegotiator (*Class SimpleNegotiator*)
- public rmf_traffic::schedule::StubbornNegotiator (*Class StubbornNegotiator*)

Class Documentation

class rmf_traffic::schedule::**Negotiator**

A pure abstract interface class that facilitates negotiating a resolution to a schedule conflict. An example implementation of this class can be found as rmf_traffic::agv::Negotiator.

Subclassed by *rmf_traffic::agv::SimpleNegotiator*, *rmf_traffic::schedule::StubbornNegotiator*

Public Types

using TableViewerPtr = *Negotiation*::*Table*::*ViewerPtr*

using ResponderPtr = std::shared_ptr<**const** *Responder*>

Public Functions

virtual void **respond**(**const** *TableViewerPtr* &*table_viewer*, **const** *ResponderPtr* &*responder*) = 0

Have the *Negotiator* respond to an attempt to negotiate.

Parameters

- `[in] table`: The *Negotiation::Table* that is being used for the negotiation.
- `[in] responder`: The *Responder* instance that the negotiator should use when a response is ready.
- `[in] interrupt_flag`: A pointer to a flag that can be used to interrupt the negotiator if it has been running for too long. If the planner should run indefinitely, then pass a nullptr.

virtual ~Negotiator() = default

class Responder

A pure abstract interface class that allows the *Negotiator* to respond to other Negotiators.

Subclassed by *rmf_traffic::schedule::SimpleResponder*

Public Types

using ParticipantId = rmf_traffic::schedule::*ParticipantId*

using ItineraryVersion = rmf_traffic::schedule::*ItineraryVersion*

using UpdateVersion = rmf_utils::optional<*ItineraryVersion*>

using ApprovalCallback = std::function<*UpdateVersion*()>

using Alternatives = *Negotiation*::*Alternatives*

Public Functions

virtual void **submit**(*PlanId plan_id*, std::vector<*Route*> *itinerary*, *ApprovalCallback approval_callback* = nullptr) **const** = 0

The negotiator will call this function when it has an itinerary to submit in response to a negotiation.

Parameters

- `[in] plan_id`: A unique ID that refers to the plan that is being submitted.
- `[in] itinerary`: The itinerary that is being proposed
- `[in] approval_callback`: This callback will get triggered if this submission gets approved. The return value of the callback should be the itinerary version of the participant update that will follow the resolution of this negotiation (or a nullopt if no update will be performed). Pass in a nullptr if an approval callback is not necessary.

virtual void **reject**(**const** *Alternatives* &*alternatives*) **const** = 0

The negotiator will call this function if it has decided to reject an attempt to negotiate. It must supply a set of alternatives for the parent negotiator to consider for its next proposal.

virtual void **forfeit**(**const** std::vector<*ParticipantId*> &*blockers*) **const** = 0

The negotiator will call this function if it cannot find any feasible proposal or alternative that can be accommodated by the parent.

Parameters

- `[in] blockers`: Give the set of schedule participants that are blocking a solution from being found.

virtual **~Responder**() = default

Class Negotiator::Responder

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_Negotiator.hpp

Nested Relationships

This class is a nested type of *Class Negotiator*.

Inheritance Relationships

Derived Type

- `public rmf_traffic::schedule::SimpleResponder` (*Class SimpleResponder*)

Class Documentation

class rmf_traffic::schedule::*Negotiator*::**Responder**

A pure abstract interface class that allows the *Negotiator* to respond to other Negotiators.

Subclassed by *rmf_traffic::schedule::SimpleResponder*

Public Types

using ParticipantId = rmf_traffic::schedule::*ParticipantId*

using ItineraryVersion = rmf_traffic::schedule::*ItineraryVersion*

using UpdateVersion = rmf_utils::optional<*ItineraryVersion*>

using ApprovalCallback = std::function<*UpdateVersion* ()>

using Alternatives = *Negotiation*::*Alternatives*

Public Functions

virtual void **submit**(*PlanId plan_id*, std::vector<*Route*> *itinerary*, *ApprovalCallback approval_callback* = nullptr) **const** = 0

The negotiator will call this function when it has an itinerary to submit in response to a negotiation.

Parameters

- `[in] plan_id`: A unique ID that refers to the plan that is being submitted.
- `[in] itinerary`: The itinerary that is being proposed
- `[in] approval_callback`: This callback will get triggered if this submission gets approved. The return value of the callback should be the itinerary version of the participant update that will follow the resolution of this negotiation (or a nullopt if no update will be performed). Pass in a nullptr if an approval callback is not necessary.

virtual void **reject**(**const** *Alternatives* &*alternatives*) **const** = 0

The negotiator will call this function if it has decided to reject an attempt to negotiate. It must supply a set of alternatives for the parent negotiator to consider for its next proposal.

virtual void **forfeit**(**const** std::vector<*ParticipantId*> &*blockers*) **const** = 0

The negotiator will call this function if it cannot find any feasible proposal or alternative that can be accommodated by the parent.

Parameters

- `[in] blockers`: Give the set of schedule participants that are blocking a solution from being found.

virtual ~Responder() = default

Class ParticipantDescription

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_ParticipantDescription.hpp

Class Documentation

class rmf_traffic::schedule::**ParticipantDescription**

Public Types

enum Rx
Enumeration for responsiveness.

Values:

enumerator Invalid
This responsiveness type is illegal and will always be rejected by the schedule verifier. Having this movement type implies a major bug in the code and should be reported immediately.

enumerator Unresponsive
The participant will not respond to any conflicts.

enumerator Responsive
The participant will try to respond to conflicts.

Public Functions

ParticipantDescription(std::string *name*, std::string *owner*, *Rx responsiveness*, *Profile profile*)
Constructor

Parameters

- `[in] name`: The name of the object participating in the schedule.
- `[in] owner`: The name of the "owner" of this participant. This does not currently have a formal definition, but for most vehicles it should be the name of the fleet that the vehicle belongs to.
- `[in] responsiveness`: What category of responsiveness this participant has. A Responsive participant might be able to react to a conflict or a request for accommodations.

bool **operator==**(**const** *ParticipantDescription* &*rhs*) **const**
Equality operator.

bool **operator!=**(**const** *ParticipantDescription* &*rhs*) **const**
Inequality operator.

ParticipantDescription &**name**(std::string *value*)
Set the name of the participant.

const std::string &**name**() **const**
Get the name of the participant.

ParticipantDescription &**owner**(std::string *value*)
Set the name of the "owner" of the participant.

const std::string &**owner**() **const**
Get the name of the "owner" of the participant.

ParticipantDescription &**responsiveness**(*Rx value*)
Set the responsiveness of the participant.

Rx **responsiveness**() **const**
Get the responsiveness of the participant.

ParticipantDescription &**profile**(*Profile new_profile*)
Set the profile of the participant.

const *Profile* &**profile**() **const**
Get the profile of the participant.

Class Patch

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_Patch.hpp

Nested Relationships

Nested Types

- *Class Patch::Participant*

Class Documentation

class rmf_traffic::schedule::**Patch**
A container of *Database* changes.

Public Types

using base_iterator = rmf_traffic::detail::*bidirectional_iterator*<E, I, F>

using const_iterator = *base_iterator*<**const** *Participant*, IterImpl, *Patch*>

Public Functions

Patch(std::vector<*Participant*> *changes*, rmf_utils::optional<*Change*::*Cull*> *cull*, std::optional<*Version*> *base_version*, *Version latest_version*)
Constructor. Mirrors should evaluate the fields of the *Patch* class in the order of these constructor arguments.

Parameters

- `[in] changes`: Information about how the participants have changed since the last update.
- `[in] cull`: Information about how the database has culled old data since the last update.
- `[in] base_version`: The base version of the database that this *Patch* builds on top of.
- `[in] latest_version`: The lastest version of the database that this *Patch* represents.

const_iterator **begin**() **const**
Returns an iterator to the first element of the *Patch*.

const_iterator **end**() **const**
Returns an iterator to the element following the last element of the *Patch*. This iterator acts as a placeholder; attempting to dereference it results in undefined behavior.

std::size_t **size**() **const**
Get the number of elements in this *Patch*.

const *Change*::*Cull* ***cull**() **const**
Get the cull information for this patch if a cull has occurred.

std::optional<*Version*> **base_version**() **const**
Get the base version of the *Database* that this patch builds on.

If this is a nullopt, then this patch does not need to build off of any base version.

Version **latest_version**() **const**
Get the latest version of the *Database* that informed this *Patch*.

class Participant

Public Functions

Participant(*ParticipantId id*, *ItineraryVersion itinerary_version*, *Change*::*Erase erasures*, std::vector<*Change*::*Delay*> *delays*, *Change*::*Add additions*, std::optional<*Change*::*Progress*> *progress*)
Constructor

Parameters
- `[in] id`: The ID of the participant that is being changed
- `[in] itinerary_version`: The version of this participant's itinerary that results from applying this patch
- `[in] erasures`: The information about which routes to erase
- `[in] delays`: The information about what delays have occurred
- `[in] additions`: The information about which routes to add
- `[in] progress`: Information about progress that the participant has made since the last change, if any.

ParticipantId **participant_id**() **const**
The ID of the participant that this set of changes will patch.

ItineraryVersion **itinerary_version**() **const**
The itinerary version that results from this patch.

const *Change*::*Erase* &**erasures**() **const**
The route erasures to perform.

These erasures should be performed before any other changes.

const std::vector<*Change*::*Delay*> &**delays**() **const**
The sequence of delays to apply.

These delays should be applied in sequential order after the erasures are performed, and before any additions are performed.

const *Change*::*Add* &**additions**() **const**
The set of additions to perfom.

These additions should be applied after all other changes.

const std::optional<*Change*::*Progress*> &**progress**() **const**
Progress that this participant made since the last version, if any.

Class Patch::Participant

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_Patch.hpp

Nested Relationships

This class is a nested type of *Class Patch*.

Class Documentation

class rmf_traffic::schedule::*Patch*::**Participant**

Public Functions

Participant(*ParticipantId id*, *ItineraryVersion itinerary_version*, *Change*::*Erase erasures*, std::vector<*Change*::*Delay*> *delays*, *Change*::*Add additions*, std::optional<*Change*::*Progress*> *progress*)
Constructor

Parameters

- [in] id: The ID of the participant that is being changed
- [in] itinerary_version: The version of this participant's itinerary that results from applying this patch
- [in] erasures: The information about which routes to erase
- [in] delays: The information about what delays have occurred
- [in] additions: The information about which routes to add
- [in] progress: Information about progress that the participant has made since the last change, if any.

ParticipantId **participant_id**() **const**
The ID of the participant that this set of changes will patch.

ItineraryVersion **itinerary_version**() **const**
The itinerary version that results from this patch.

const *Change*::*Erase* &**erasures**() **const**
The route erasures to perform.

These erasures should be performed before any other changes.

const std::vector<*Change*::*Delay*> &**delays**() **const**
The sequence of delays to apply.

These delays should be applied in sequential order after the erasures are performed, and before any additions are performed.

const *Change*::*Add* &**additions**() **const**
The set of additions to perfom.

These additions should be applied after all other changes.

const std::optional<*Change*::*Progress*> &**progress**() **const**
Progress that this participant made since the last version, if any.

Class Query

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_Query.hpp

Nested Relationships

Nested Types

- *Class Query::Participants*
- *Class Participants::All*
- *Class Participants::Exclude*
- *Class Participants::Include*
- *Class Query::Spacetime*
- *Class Spacetime::All*
- *Class Spacetime::Regions*
- *Class Spacetime::Timespan*

Class Documentation

class rmf_traffic::schedule::**Query**
A class to define a query into a schedule database.

Public Types

using **base_iterator** = rmf_traffic::detail::*bidirectional_iterator*<E, I, F>

Public Functions

Spacetime &**spacetime**()
Get the *Spacetime* component of this *Query*.

const *Spacetime* &**spacetime**() **const**
const-qualified *spacetime()*

Participants &**participants**()
Get the *Participants* component of this *Query*.

const *Participants* &**participants**() **const**
const-qualified *participants()*

class Participants

A class to describe a filter on which schedule participants to pay attention to.

Public Types

enum Mode

Values:

enumerator Invalid

Invalid mode, behavior is undefined.

enumerator All

Get all participants.

enumerator Include

Get only the participants listed.

enumerator Exclude

Get all participants except the ones listed.

Public Functions

Participants()

Default constructor, uses *All* mode.

Mode **get_mode**() **const**

Get the mode for this *Participants* filter.

All ***all**()

Get the *All* interface if this *Participants* filter is in *All* mode, otherwise get a nullptr.

const *All* ***all**() **const**

const-qualified *all()*

Include ***include**()

Get the *Include* interface if this *Participants* filter is in *Include* mode, otherwise get a nullptr.

const *Include* ***include**() **const**

const-qualified *include()*

Participants &**include**(std::vector<*ParticipantId*> *ids*)

Change this filter to *Include* mode, and include the specified participant IDs.

Exclude ***exclude**()

Get the *Exclude* interface if this *Participants* filter is in *Exclude* mode, otherwise get a nullptr.

const *Exclude* ***exclude**() **const**

const-qualified *exclude()*

Participants &**exclude**(std::vector<*ParticipantId*> *ids*)

Change this filter to *Exclude* mode, and exclude the specified participant IDs.

Public Static Functions

static const *Participants* &**make_all**()
Constructor to use *All* mode.

static *Participants* **make_only**(std::vector<*ParticipantId*> *ids*)
Constructor to use *Include* mode.

Parameters
- `[in] ids`: The IDs of the participants that should be included in the query.

static *Participants* **make_all_except**(std::vector<*ParticipantId*> *ids*)
Constructor to use *Exclude* mode.

Parameters
- `[in] ids`: The IDs of the participants that should be excluded from the query.

class All
This is a placeholder class in case we ever want to extend the features of the `All` mode.

class Exclude
The interface for the *Participants::Exclude* mode.

Public Functions

Exclude(std::vector<*ParticipantId*> *ids*)
Constructor.

const std::vector<*ParticipantId*> &**get_ids**() **const**
Get the IDs of the participants that should be excluded.

Exclude &**set_ids**(std::vector<*ParticipantId*> *ids*)
Set the IDs of the participants that should be excluded.

class Include
The interface for the *Participants::Include* mode.

Public Functions

Include(std::vector<*ParticipantId*> *ids*)
Constructor.

const std::vector<*ParticipantId*> &**get_ids**() **const**
Get the IDs of the participants that should be included.

Include &**set_ids**(std::vector<*ParticipantId*> *ids*)
Set the IDs of the participants that should be included.

class Spacetime
A class to describe spacetime filters for a schedule *Query*.

Public Types

enum Mode
This enumerator determines what *Spacetime* mode the query will be in.

Values:

enumerator Invalid
Invalid mode, behavior is undefined.

enumerator All
Request trajectories throughout all of space and time. This will still be constrained by the version field.

enumerator Regions
Request trajectories in specific regions spacetime regions.

enumerator Timespan
Request trajectories that are active in a specified timespan.

using Space = geometry::*Space*

Public Functions

Spacetime()
Default constructor, uses *All* mode.

Spacetime(std::vector<*Region*> *regions*)
Regions mode constructor.

Parameters
- `[in] regions`: The regions to use

Spacetime(std::vector<std::string> *maps*)
Timespan mode constructor.

This will query all trajectories across all time for the specified maps.

Parameters
- `[in] maps`: The maps to query from

Spacetime(std::vector<std::string> *maps*, *Time lower_bound*)
Timespan mode constructor.

This will query all trajectories that have at least one waypoint active after the lower bound on the specified maps.

Parameters
- `[in] maps`: The maps to query from
- `[in] lower_bound`: The lower bound on time

Spacetime(std::vector<std::string> *maps*, *Time lower_bound*, *Time upper_bound*)
Timespan mode constructor.

This will query all trajectories that have at least one waypoint active after the lower bound and before the upper bound on the specified maps.

Parameters

- `[in] maps`: The maps to query from
- `[in] lower_bound`: The lower bound on time
- `[in] upper_bound`: The upper bound on time

Mode **get_mode**() **const**
Get the current *Spacetime* Mode of this query.

All &**query_all**()
Set the mode of this *Spacetime* to query for *All* Trajectories throughout *Spacetime*.

Regions &**query_regions**(std::vector<*Region*> *regions* = {})
Set the mode of this *Spacetime* to query for specific *Regions*.

Parameters
- `[in] regions`: Specify the regions of *Spacetime* to use.

Regions ***regions**()
Get the *Regions* of *Spacetime* to use for this *Query*. If this *Spacetime* is not in *Regions* mode, then this will return a nullptr.

const *Regions* ***regions**() **const**
const-qualified *regions()*

Timespan &**query_timespan**(std::vector<std::string> *maps*, *Time* *lower_bound*, *Time* *upper_bound*)
Query a timespan between two bounds for a set of maps.

Timespan &**query_timespan**(std::vector<std::string> *maps*, *Time* *lower_bound*)
Query from a lower bound in time for a set of maps.

Timespan &**query_timespan**(std::vector<std::string> *maps*)
Query for all trajectories on a set of maps.

Timespan &**query_timespan**(bool *query_all_maps* = true)
Switch to timespan mode, and specify whether or not to use all maps.

Timespan ***timespan**()
Get the *Timespan* of *Spacetime* to use for this *Query*. If this *Spacetime* is not in *Timespan* mode, then this will return a nullptr.

const *Timespan* ***timespan**() **const**
const-qualified *timespan()*

class All
This is a placeholder class in case we ever want to extend the features of the `All` mode.

class Regions
A container class for *rmf_traffic::Region* instances. Using *Regions* mode will query for Trajectories that intersect the specified regions.

Public Types

using iterator = *base_iterator*<*Region*, IterImpl, *Regions*>

using const_iterator = *base_iterator*<**const** *Region*, IterImpl, *Regions*>

Public Functions

void **push_back**(*Region region*)
Add a *Region* to this container.

void **pop_back**()
Remove the last *Region* that was added to this container.

iterator **erase**(*iterator it*)
Erase a *Region* based on its iterator.

iterator **erase**(*iterator first*, *iterator last*)
Erase a range of *Regions* based on their iterators.

iterator **begin**()
Get the beginning iterator of this container.

const_iterator **begin**() **const**
const-qualified *begin()*

const_iterator **cbegin**() **const**
Explicitly const-qualified alternative to *begin()*

iterator **end**()
Get the one-past-the-end iterator of this container.

const_iterator **end**() **const**
const-qualified *end()*

const_iterator **cend**() **const**
Explicitly const-qualified alternative to *end()*

std::size_t **size**() **const**
Get the number of *Spacetime Region* elements in this container.

class Timespan
A class for specifying a timespan.

Public Functions

const std::unordered_set<std::string> &**maps**() **const**
Get the maps that will be queried.

Timespan &**add_map**(std::string *map_name*)
Add a map to the query.

Timespan &**remove_map**(**const** std::string &*map_name*)
Remove a map from the query.

Timespan &**clear_maps**()
Remove all maps from the query.

bool **all_maps**() **const**
Returns true if all maps should be queried. If true, the set of maps mentioned above will be ignored.

Timespan &**all_maps**(bool *query_all_maps*)
Set whether all maps should be queried. When true, the set of maps above will be ignored. When false, only the maps in the set above will be included in the query.

const *Time* ***get_lower_time_bound**() **const**
Get the lower bound for the time range.

If there is no lower bound for the time range, then this returns a nullptr.

Timespan &**set_lower_time_bound**(*Time time*)
Set the lower bound fore the time range.

Timespan &**remove_lower_time_bound**()
Remove the lower bound for the time range.

const *Time* ***get_upper_time_bound**() **const**
Get the upper bound for the time range.

If there is no upper bound for the time range, then this returns a nullptr.

Timespan &**set_upper_time_bound**(*Time time*)
Set the upper bound for the time range.

Timespan &**remove_upper_time_bound**()
Remove the upper bound for the time range.

Class Query::Participants

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_Query.hpp

Nested Relationships

This class is a nested type of *Class Query*.

Nested Types

- *Class Participants::All*
- *Class Participants::Exclude*
- *Class Participants::Include*

Class Documentation

class rmf_traffic::schedule::*Query*::**Participants**
A class to describe a filter on which schedule participants to pay attention to.

Public Types

enum Mode
Values:

enumerator Invalid
Invalid mode, behavior is undefined.

enumerator All
Get all participants.

enumerator Include
Get only the participants listed.

enumerator Exclude
Get all participants except the ones listed.

Public Functions

Participants()
Default constructor, uses *All* mode.

Mode **get_mode**() **const**
Get the mode for this *Participants* filter.

All ***all**()
Get the *All* interface if this *Participants* filter is in *All* mode, otherwise get a nullptr.

const *All* ***all**() **const**
const-qualified *all()*

Include ***include**()
Get the *Include* interface if this *Participants* filter is in *Include* mode, otherwise get a nullptr.

const *Include* ***include**() **const**
const-qualified *include()*

Participants &**include**(std::vector<*ParticipantId*> *ids*)
Change this filter to *Include* mode, and include the specified participant IDs.

Exclude ***exclude**()
Get the *Exclude* interface if this *Participants* filter is in *Exclude* mode, otherwise get a nullptr.

const *Exclude* ***exclude**() **const**
const-qualified *exclude()*

Participants &**exclude**(std::vector<*ParticipantId*> *ids*)
Change this filter to *Exclude* mode, and exclude the specified participant IDs.

Public Static Functions

static const *Participants* &**make_all**()
Constructor to use *All* mode.

static *Participants* **make_only**(std::vector<*ParticipantId*> *ids*)
Constructor to use *Include* mode.

Parameters

- `[in] ids`: The IDs of the participants that should be included in the query.

static *Participants* **make_all_except** (std::vector<*ParticipantId*> *ids*)
Constructor to use *Exclude* mode.

Parameters

- `[in] ids`: The IDs of the participants that should be excluded from the query.

class All
This is a placeholder class in case we ever want to extend the features of the `All` mode.

class Exclude
The interface for the *Participants::Exclude* mode.

Public Functions

Exclude (std::vector<*ParticipantId*> *ids*)
Constructor.

const std::vector<*ParticipantId*> &**get_ids** () **const**
Get the IDs of the participants that should be excluded.

Exclude &**set_ids** (std::vector<*ParticipantId*> *ids*)
Set the IDs of the participants that should be excluded.

class Include
The interface for the *Participants::Include* mode.

Public Functions

Include (std::vector<*ParticipantId*> *ids*)
Constructor.

const std::vector<*ParticipantId*> &**get_ids** () **const**
Get the IDs of the participants that should be included.

Include &**set_ids** (std::vector<*ParticipantId*> *ids*)
Set the IDs of the participants that should be included.

Class Participants::All

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_Query.hpp

Nested Relationships

This class is a nested type of *Class Query::Participants*.

Class Documentation

class All
This is a placeholder class in case we ever want to extend the features of the *All* mode.

Class Participants::Exclude

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_Query.hpp

Nested Relationships

This class is a nested type of *Class Query::Participants*.

Class Documentation

class rmf_traffic::schedule::*Query*::*Participants*::**Exclude**
The interface for the *Participants::Exclude* mode.

Public Functions

Exclude(std::vector<*ParticipantId*> *ids*)
Constructor.

const std::vector<*ParticipantId*> &**get_ids**() **const**
Get the IDs of the participants that should be excluded.

Exclude &**set_ids**(std::vector<*ParticipantId*> *ids*)
Set the IDs of the participants that should be excluded.

Class Participants::Include

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_Query.hpp

Nested Relationships

This class is a nested type of *Class Query::Participants*.

Class Documentation

class rmf_traffic::schedule::*Query*::*Participants*::**Include**
The interface for the *Participants::Include* mode.

Public Functions

Include (std::vector<*ParticipantId*> *ids*)
Constructor.

const std::vector<*ParticipantId*> &**get_ids** () **const**
Get the IDs of the participants that should be included.

Include &**set_ids** (std::vector<*ParticipantId*> *ids*)
Set the IDs of the participants that should be included.

Class Query::Spacetime

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_Query.hpp

Nested Relationships

This class is a nested type of *Class Query*.

Nested Types

- *Class Spacetime::All*
- *Class Spacetime::Regions*
- *Class Spacetime::Timespan*

Class Documentation

class rmf_traffic::schedule::*Query*::**Spacetime**
A class to describe spacetime filters for a schedule *Query*.

Public Types

enum Mode
This enumerator determines what *Spacetime* mode the query will be in.

Values:

enumerator Invalid
Invalid mode, behavior is undefined.

enumerator All
Request trajectories throughout all of space and time. This will still be constrained by the version field.

enumerator Regions
Request trajectories in specific regions spacetime regions.

enumerator Timespan
Request trajectories that are active in a specified timespan.

using Space = geometry::*Space*

Public Functions

Spacetime()
Default constructor, uses *All* mode.

Spacetime(std::vector<*Region*> *regions*)
Regions mode constructor.

Parameters

- `[in] regions`: The regions to use

Spacetime(std::vector<std::string> *maps*)
Timespan mode constructor.

This will query all trajectories across all time for the specified maps.

Parameters

- `[in] maps`: The maps to query from

Spacetime(std::vector<std::string> *maps*, *Time lower_bound*)
Timespan mode constructor.

This will query all trajectories that have at least one waypoint active after the lower bound on the specified maps.

Parameters

- `[in] maps`: The maps to query from
- `[in] lower_bound`: The lower bound on time

Spacetime(std::vector<std::string> *maps*, *Time lower_bound*, *Time upper_bound*)
Timespan mode constructor.

This will query all trajectories that have at least one waypoint active after the lower bound and before the upper bound on the specified maps.

Parameters

- `[in] maps`: The maps to query from
- `[in] lower_bound`: The lower bound on time
- `[in] upper_bound`: The upper bound on time

Mode **get_mode**() **const**
Get the current *Spacetime* Mode of this query.

All &**query_all**()
Set the mode of this *Spacetime* to query for *All* Trajectories throughout *Spacetime*.

Regions &**query_regions**(std::vector<*Region*> *regions* = {})
Set the mode of this *Spacetime* to query for specific *Regions*.

Parameters

- `[in] regions`: Specify the regions of *Spacetime* to use.

Regions ***regions**()
Get the *Regions* of *Spacetime* to use for this *Query*. If this *Spacetime* is not in *Regions* mode, then this will return a nullptr.

const *Regions* ***regions**() **const**
const-qualified *regions()*

Timespan &**query_timespan**(std::vector<std::string> *maps*, *Time lower_bound*, *Time upper_bound*)
Query a timespan between two bounds for a set of maps.

Timespan &**query_timespan**(std::vector<std::string> *maps*, *Time lower_bound*)
Query from a lower bound in time for a set of maps.

Timespan &**query_timespan**(std::vector<std::string> *maps*)
Query for all trajectories on a set of maps.

Timespan &**query_timespan**(bool *query_all_maps* = true)
Switch to timespan mode, and specify whether or not to use all maps.

Timespan ***timespan**()
Get the *Timespan* of *Spacetime* to use for this *Query*. If this *Spacetime* is not in *Timespan* mode, then this will return a nullptr.

const *Timespan* ***timespan**() **const**
const-qualified *timespan()*

class All
This is a placeholder class in case we ever want to extend the features of the `All` mode.

class Regions
A container class for *rmf_traffic::Region* instances. Using *Regions* mode will query for Trajectories that intersect the specified regions.

Public Types

using iterator = *base_iterator*<*Region*, IterImpl, *Regions*>

using const_iterator = *base_iterator*<**const** *Region*, IterImpl, *Regions*>

Public Functions

void **push_back**(*Region region*)
Add a *Region* to this container.

void **pop_back**()
Remove the last *Region* that was added to this container.

iterator **erase**(*iterator it*)
Erase a *Region* based on its iterator.

iterator **erase**(*iterator first*, *iterator last*)
Erase a range of *Regions* based on their iterators.

iterator **begin**()
Get the beginning iterator of this container.

const_iterator **begin**() **const**
const-qualified *begin()*

const_iterator **cbegin**() **const**
Explicitly const-qualified alternative to *begin()*

iterator **end**()
Get the one-past-the-end iterator of this container.

const_iterator **end**() **const**
const-qualified *end()*

const_iterator **cend**() **const**
Explicitly const-qualified alternative to *end()*

std::size_t **size**() **const**
Get the number of *Spacetime Region* elements in this container.

class Timespan
A class for specifying a timespan.

Public Functions

const std::unordered_set<std::string> &**maps**() **const**
Get the maps that will be queried.

Timespan &**add_map**(std::string *map_name*)
Add a map to the query.

Timespan &**remove_map**(**const** std::string &*map_name*)
Remove a map from the query.

Timespan &**clear_maps**()
Remove all maps from the query.

bool **all_maps**() **const**
Returns true if all maps should be queried. If true, the set of maps mentioned above will be ignored.

Timespan &**all_maps**(bool *query_all_maps*)
Set whether all maps should be queried. When true, the set of maps above will be ignored. When false, only the maps in the set above will be included in the query.

const *Time* ***get_lower_time_bound**() **const**
Get the lower bound for the time range.

If there is no lower bound for the time range, then this returns a nullptr.

Timespan &**set_lower_time_bound**(*Time time*)
Set the lower bound fore the time range.

Timespan &**remove_lower_time_bound**()
Remove the lower bound for the time range.

const *Time* ***get_upper_time_bound**() **const**
Get the upper bound for the time range.

If there is no upper bound for the time range, then this returns a nullptr.

Timespan &**set_upper_time_bound**(*Time time*)
Set the upper bound for the time range.

Timespan &**remove_upper_time_bound**()
Remove the upper bound for the time range.

Class Spacetime::All

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_Query.hpp

Nested Relationships

This class is a nested type of *Class Query::Spacetime*.

Class Documentation

class All
This is a placeholder class in case we ever want to extend the features of the `All` mode.

Class Spacetime::Regions

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_Query.hpp

Nested Relationships

This class is a nested type of *Class Query::Spacetime*.

Class Documentation

class rmf_traffic::schedule::*Query*::*Spacetime*::**Regions**
A container class for *rmf_traffic::Region* instances. Using *Regions* mode will query for Trajectories that intersect the specified regions.

Public Types

using iterator = *base_iterator*<*Region*, IterImpl, *Regions*>

using const_iterator = *base_iterator*<**const** *Region*, IterImpl, *Regions*>

Public Functions

void **push_back**(*Region region*)
Add a *Region* to this container.

void **pop_back**()
Remove the last *Region* that was added to this container.

iterator **erase**(*iterator it*)
Erase a *Region* based on its iterator.

iterator **erase**(*iterator first*, *iterator last*)
Erase a range of *Regions* based on their iterators.

iterator **begin**()
Get the beginning iterator of this container.

const_iterator **begin**() **const**
: const-qualified *begin()*

const_iterator **cbegin**() **const**
: Explicitly const-qualified alternative to *begin()*

iterator **end**()
: Get the one-past-the-end iterator of this container.

const_iterator **end**() **const**
: const-qualified *end()*

const_iterator **cend**() **const**
: Explicitly const-qualified alternative to *end()*

std::size_t **size**() **const**
: Get the number of *Spacetime Region* elements in this container.

Class Spacetime::Timespan

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_Query.hpp

Nested Relationships

This class is a nested type of *Class Query::Spacetime*.

Class Documentation

class rmf_traffic::schedule::*Query*::*Spacetime*::**Timespan**
: A class for specifying a timespan.

Public Functions

const std::unordered_set<std::string> &**maps**() **const**
: Get the maps that will be queried.

Timespan &**add_map**(std::string *map_name*)
: Add a map to the query.

Timespan &**remove_map**(**const** std::string &*map_name*)
: Remove a map from the query.

Timespan &**clear_maps**()
: Remove all maps from the query.

bool **all_maps**() **const**
: Returns true if all maps should be queried. If true, the set of maps mentioned above will be ignored.

Timespan &**all_maps**(bool *query_all_maps*)
: Set whether all maps should be queried. When true, the set of maps above will be ignored. When false, only the maps in the set above will be included in the query.

const *Time* ***get_lower_time_bound**() **const**
: Get the lower bound for the time range.

  If there is no lower bound for the time range, then this returns a nullptr.

Timespan &**set_lower_time_bound**(*Time time*)
Set the lower bound fore the time range.

Timespan &**remove_lower_time_bound**()
Remove the lower bound for the time range.

const *Time* ***get_upper_time_bound**() **const**
Get the upper bound for the time range.

If there is no upper bound for the time range, then this returns a nullptr.

Timespan &**set_upper_time_bound**(*Time time*)
Set the upper bound for the time range.

Timespan &**remove_upper_time_bound**()
Remove the upper bound for the time range.

Class QuickestFinishEvaluator

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_Negotiation.hpp

Inheritance Relationships

Base Type

- `public rmf_traffic::schedule::Negotiation::Evaluator` (*Class Negotiation::Evaluator*)

Class Documentation

class `rmf_traffic::schedule::`**QuickestFinishEvaluator** : **public** rmf_traffic::schedule::*Negotiation*::*Evaluator*
An implementation of an evaluator that chooses the proposal that minimizes net delays in completing the itineraries.

Public Functions

std::size_t **choose**(**const** std::vector<**const** *Negotiation*::*Proposal**> &*proposals*) **const final**

Class RectificationRequester

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_Rectifier.hpp

Class Documentation

class `rmf_traffic::schedule::`**RectificationRequester**
RectificationRequester is a pure abstract class which should be implemented for any middlewares that intend to act as transport layers for the scheduling system.

Classes that derive from *RectificationRequester* do not need to implement any interfaces, but they should practice RAII. The lifecycle of the *RectificationRequester* will be tied to the Participant that it was created for.

When a schedule database reports an inconsistency for the participant tied to a *RectificationRequester* instance, the instance should call *Rectifier::retransmit()* on the *Rectifier* that was assigned to it.

Public Functions

virtual ~RectificationRequester() = 0
This destructor is pure virtual to ensure that a derived class is instantiated.

Class RectificationRequesterFactory

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_Rectifier.hpp

Inheritance Relationships

Derived Type

- public rmf_traffic::schedule::DatabaseRectificationRequesterFactory (*Class DatabaseRectificationRequesterFactory*)

Class Documentation

class rmf_traffic::schedule::**RectificationRequesterFactory**
The *RectificationRequesterFactory* is a pure abstract interface class which should be implemented for any middlewares that intend to act as transport layers for the scheduling system.

Subclassed by *rmf_traffic::schedule::DatabaseRectificationRequesterFactory*

Public Functions

virtual std::unique_ptr<*RectificationRequester*> **make**(*Rectifier rectifier*, *ParticipantId participant_id*) = 0
Create a *RectificationRequester* to be held by a Participant

Parameters

- [in] rectifier: This rectifier can be used by the *RectificationRequester* to ask the participant to retransmit some of its changes.
- [in] participant_id: The ID of the participant that will hold onto this *RectificationRequester*. This is the same participant that the rectifier will request retransmissions to.

virtual ~RectificationRequesterFactory() = default

Class Rectifier

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_Rectifier.hpp

Nested Relationships

Nested Types

- *Struct Rectifier::Range*

Class Documentation

class rmf_traffic::schedule::**Rectifier**

The *Rectifier* class provides an interface for telling a Participant to rectify an inconsistency in the information received by a database. This rectification protocol is important when the schedule is being managed over an unreliable network.

The *Rectifier* class can be used by a RectifierRequester to ask a participant to retransmit a range of its past itinerary changes.

Only the Participant class is able to create a *Rectifier* instance. Users of rmf_traffic cannot instantiate a *Rectifier*.

Public Functions

void **retransmit**(**const** std::vector<*Range*> &*ranges*, *ItineraryVersion* *last_known_itinerary*, *ProgressVersion* *last_known_progress*)

Ask the participant to retransmit the specified range of its itinerary changes.

Parameters

- [in] ranges: The ranges of missing Itinerary IDs
- [in] last_known_itinerary: The last ItineraryVersion known upstream.
- [in] last_known_progress: The last ProgressVersion known upstream.

void **correct_id**(*ParticipantId* *new_id*)

Correct the ID of the participant.

std::optional<*ItineraryVersion*> **current_version**() **const**

Get the current ItineraryVersion of the Participant.

std::optional<*ParticipantId*> **get_id**() **const**

Get the ID of the Participant.

std::optional<*ParticipantDescription*> **get_description**() **const**

Get the description of the Participant.

struct Range

A range of itinerary change IDs that is currently missing from a database. All IDs from lower to upper are missing, including lower and upper themselves.

It is undefined behavior if the value given to upper is less than the value given to upper.

Public Members

ItineraryVersion **lower**
The ID of the first itinerary change in this range that is missing.

ItineraryVersion **upper**
The ID of the last itinerary change in this range that is missing.

Class SimpleResponder

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_Negotiator.hpp

Inheritance Relationships

Base Type

- `public rmf_traffic::schedule::Negotiator::Responder` (*Class Negotiator::Responder*)

Class Documentation

class `rmf_traffic::schedule::`**SimpleResponder** : **public** rmf_traffic::schedule::*Negotiator*::*Responder*
A simple implementation of a *Negotiator::Responder*. It simply passes the result along to the *Negotiation*.

Public Types

using ApprovalMap = std::unordered_map<*Negotiation*::*ConstTablePtr*, std::function<UpdateVersion ()>>

using BlockerSet = std::unordered_set<schedule::*ParticipantId*>

Public Functions

SimpleResponder(**const** *Negotiation*::*TablePtr* &*table*, std::vector<schedule::*ParticipantId*> **report_blockers* = nullptr)
Constructor

Parameters

- `[in] table`: The negotiation table that this *SimpleResponder* is tied to
- `[in] report_blockers`: If the blockers should be reported when a forfeit is given, provide a pointer to a vector of ParticipantIds.

SimpleResponder(**const** *Negotiation*::*TablePtr* &*table*, std::shared_ptr<*ApprovalMap*> *approval_map*, std::shared_ptr<*BlockerSet*> *blockers*)
Constructor

Parameters

- `[in] table`: The negotiation table that this *SimpleResponder* is tied to
- `[in] approval_map`: If provided, the responder will store the approval callback in this map

- `[in] blockers`: If provided, the responder will store any solution blockers in this set

void **submit**(*PlanId plan_id*, std::vector<*Route*> *itinerary*, std::function<UpdateVersion) > *approval_callback* = nullptr **const final**

virtual void **reject**(**const** *Negotiation*::*Alternatives* &*alternatives*) **const final**
The negotiator will call this function if it has decided to reject an attempt to negotiate. It must supply a set of alternatives for the parent negotiator to consider for its next proposal.

virtual void **forfeit**(**const** std::vector<*ParticipantId*> &*blockers*) **const final**
The negotiator will call this function if it cannot find any feasible proposal or alternative that can be accommodated by the parent.

Parameters

- `[in] blockers`: Give the set of schedule participants that are blocking a solution from being found.

const std::vector<*ParticipantId*> &**blockers**() **const**
Get the blockers that were reported by the *Negotiator*, if a forfeit was given.

Public Static Functions

template<typename ...**Args**>
static inline std::shared_ptr<*SimpleResponder*> **make**(*Args*&&... *args*)

Class Snappable

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_Snapshot.hpp

Inheritance Relationships

Derived Types

- `public rmf_traffic::schedule::Database` (*Class Database*)
- `public rmf_traffic::schedule::Mirror` (*Class Mirror*)

Class Documentation

class `rmf_traffic::schedule::`**Snappable**
This is a pure abstract interface class that can be inherited by any schedule *Viewer* that wants to be able to provide a frozen snapshot of its schedule.

Subclassed by *rmf_traffic::schedule::Database*, *rmf_traffic::schedule::Mirror*

Public Functions

virtual std::shared_ptr<**const** *Snapshot*> **snapshot** () **const** = 0
Get a snapshot of the schedule.

virtual **~Snappable** () = default

Class Snapshot

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_Snapshot.hpp

Inheritance Relationships

Base Type

- `public rmf_traffic::schedule::Viewer` (*Class Viewer*)

Class Documentation

class **Snapshot** : **public** rmf_traffic::schedule::*Viewer*

Class StubbornNegotiator

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_StubbornNegotiator.hpp

Inheritance Relationships

Base Type

- `public rmf_traffic::schedule::Negotiator` (*Class Negotiator*)

Class Documentation

class `rmf_traffic::schedule::`**StubbornNegotiator** : **public** rmf_traffic::schedule::*Negotiator*
A *StubbornNegotiator* will only accept plans that accommodate the current itinerary of the

Public Types

using **UpdateVersion** = rmf_utils::optional<*ItineraryVersion*>

Public Functions

StubbornNegotiator(**const** Participant &*participant*)
Constructor

```
StubbornNegotiator(participant).respond(table_view, responder);
```

Note We take a const-reference to the Participant with the expectation that the Participant instance will outlive this *StubbornNegotiator* instance. The *StubbornNegotiator* costs very little to construct, so it is okay to use a pattern like

Parameters

- `[in] participant`: The Participant who wants to be stubborn.

StubbornNegotiator(std::shared_ptr<**const** Participant> *participant*)
Owning Constructor

The *StubbornNegotiator* instance will now hold a shared reference to the participant to ensure it maintains its lifetime. This constructor should be used in cases where the *StubbornNegotiator* instance has a prolonged lifecycle.

Parameters

- `[in] participant`: The Participant who wants to be stubborn.

StubbornNegotiator &**acceptable_waits**(std::vector<*Duration*> *wait_times*, std::function<*UpdateVersion*(*Duration* wait_time)> *approval_cb* = nullptr)
Add a set of acceptable wait times.

Parameters

- `[in] wait_times`: A list of the wait times that would be accepted for negotiation
- `[in] approval_cb`: A callback that will be triggered when the negotiator decides that you need to wait for another participant. The callback will receive the chosen wait duration, and is expected to return the schedule version that will incorporate the given wait time.

StubbornNegotiator &**additional_margins**(std::vector<rmf_traffic::*Duration*> *margins*)
Add some timing margins that will be put into the negotiation submission. This effectively asks other robots to back off somewhat.

Parameters

- `[in] margins`: The margins to put into the proposal.

virtual void **respond**(**const** schedule::*Negotiation*::*Table*::*ViewerPtr* &*table_viewer*, **const** ResponderPtr &*responder*) **final**
Have the *Negotiator* respond to an attempt to negotiate.

Parameters

- `[in] table`: The *Negotiation::Table* that is being used for the negotiation.
- `[in] responder`: The Responder instance that the negotiator should use when a response is ready.
- `[in] interrupt_flag`: A pointer to a flag that can be used to interrupt the negotiator if it has been running for too long. If the planner should run indefinitely, then pass a nullptr.

Class Viewer

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_Viewer.hpp

Nested Relationships

Nested Types

- *Class Viewer::View*
- *Struct View::Element*

Inheritance Relationships

Derived Types

- `public rmf_traffic::schedule::ItineraryViewer` (*Class ItineraryViewer*)
- `public rmf_traffic::schedule::Snapshot` (*Class Snapshot*)

Class Documentation

class `rmf_traffic::schedule::`**Viewer**

A pure abstract interface class that allows users to query for itineraries that are in a schedule.

This class cannot be instantiated directly. To get a *Viewer*, you must instantiate an *rmf_traffic::schedule::Database* or an *rmf_traffic::schedule::Mirror* object.

Subclassed by *rmf_traffic::schedule::ItineraryViewer*, *rmf_traffic::schedule::Snapshot*

Public Functions

virtual *View* **query**(**const** *Query* &*parameters*) **const** = 0

Query this *Viewer* to get a *View* of the Trajectories inside of it that match the *Query* parameters.

virtual *View* **query**(**const** *Query*::*Spacetime* &*spacetime*, **const** *Query*::*Participants* &*participants*) **const** = 0

Alternative signature for *query()*

virtual const std::unordered_set<*ParticipantId*> &**participant_ids**() **const** = 0

Get the set of active participant IDs.

virtual std::shared_ptr<**const** *ParticipantDescription*> **get_participant**(*ParticipantId participant_id*) **const** = 0

Get the information of the specified participant if it is available. If a participant with the specified ID is not registered with the schedule, then this will return a nullptr.

virtual *Version* **latest_version**() **const** = 0

Get the latest version number of this *Database*.

virtual ~Viewer() = default

class View

A read-only view of some Trajectories in a *Database* or *Mirror*.

It is undefined behavior to modify a *Database* or patch a *Mirror* while reading Trajectories from this view. The user of this class is responsible for managing access to reads vs access to writes.

Public Types

using base_iterator = rmf_traffic::detail::*bidirectional_iterator*<E, I, F>

using const_iterator = *base_iterator*<**const** *Element*, IterImpl, *View*>

using iterator = *const_iterator*

Public Functions

const_iterator **begin**() **const**

Returns an iterator to the first element of the *View*.

const_iterator **end**() **const**

Returns an iterator to the element following the last element of the *View*.

std::size_t **size**() **const**

Returns the number of elements in this *View*.

struct Element

Public Members

const *ParticipantId* **participant**

const *PlanId* **plan_id**

const *RouteId* **route_id**

const std::shared_ptr<**const** *Route*> **route**

const *ParticipantDescription* &**description**

Class Viewer::View

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_Viewer.hpp

Nested Relationships

This class is a nested type of *Class Viewer*.

Nested Types

- *Struct View::Element*

Class Documentation

class rmf_traffic::schedule::*Viewer*::**View**

A read-only view of some Trajectories in a *Database* or *Mirror*.

It is undefined behavior to modify a *Database* or patch a *Mirror* while reading Trajectories from this view. The user of this class is responsible for managing access to reads vs access to writes.

Public Types

using base_iterator = rmf_traffic::detail::*bidirectional_iterator*<E, I, F>

using const_iterator = *base_iterator*<**const** *Element*, IterImpl, *View*>

using iterator = *const_iterator*

Public Functions

const_iterator **begin**() **const**

Returns an iterator to the first element of the *View*.

const_iterator **end**() **const**

Returns an iterator to the element following the last element of the *View*.

std::size_t **size**() **const**

Returns the number of elements in this *View*.

struct Element

Public Members

const *ParticipantId* **participant**

const *PlanId* **plan_id**

const *RouteId* **route_id**

const std::shared_ptr<**const** *Route*> **route**

const *ParticipantDescription* &**description**

Class Writer

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_Writer.hpp

Nested Relationships

Nested Types

- *Class Writer::Registration*

Inheritance Relationships

Derived Type

- `public rmf_traffic::schedule::Database` (*Class Database*)

Class Documentation

class `rmf_traffic::schedule::`**Writer**

A pure abstract interface class that defines an API for writing to the schedule database. This API is implemented by the *Database* class, but it should also be implemented for any middleware that intends to have a schedule participant write changes to a remote database.

Subclassed by *rmf_traffic::schedule::Database*

Public Types

using ParticipantId = rmf_traffic::schedule::*ParticipantId*

using ParticipantDescription = rmf_traffic::schedule::*ParticipantDescription*

using Itinerary = rmf_traffic::schedule::*Itinerary*

using ItineraryVersion = rmf_traffic::schedule::*ItineraryVersion*

using ProgressVersion = rmf_traffic::schedule::*ProgressVersion*

using PlanId = rmf_traffic::*PlanId*

using Duration = rmf_traffic::*Duration*

using RouteId = rmf_traffic::*RouteId*

using CheckpointId = rmf_traffic::*CheckpointId*

using StorageId = uint64_t

Public Functions

virtual void **set**(*ParticipantId participant*, *PlanId plan*, **const** *Itinerary* &*itinerary*, *StorageId storage_base*, *ItineraryVersion version*) = 0

Set a brand new itinerary for a participant. This will replace any itinerary that is already in the schedule for the participant.

Parameters

- `[in] participant`: The ID of the participant whose itinerary is being updated.
- `[in] plan`: The ID of the plan that this new itinerary belongs to.
- `[in] itinerary`: The new itinerary of the participant.
- `[in] storage_base`: The storage index offset that the database should use for this plan. This should generally be the integer number of total routes that the participant has ever given to the writer prior to setting this new itinerary. This value helps ensure consistent unique IDs for every route, even after a database has failed over or restarted.
- `[in] version`: The version for this itinerary change.

virtual void **extend**(*ParticipantId participant*, **const** *Itinerary* &*routes*, *ItineraryVersion version*) = 0

Add a set of routes to the itinerary of this participant.

Parameters

- `[in] participant`: The ID of the participant whose itinerary is being updated.
- `[in] routes`: The set of routes that should be added to the itinerary.
- `[in] version`: The version for this itinerary change

virtual void **delay**(*ParticipantId participant*, *Duration delay*, *ItineraryVersion version*) = 0

Add a delay to the itinerary from the specified Time.

Nothing about the routes in the itinerary will be changed except that waypoints will shifted through time.

Parameters

- `[in] participant`: The ID of the participant whose itinerary is being delayed.
- `[in] delay`: This is the duration of time to delay all qualifying *Trajectory* Waypoints.
- `[in] version`: The version for this itinerary change

virtual void **reached**(*ParticipantId participant*, *PlanId plan*, **const** std::vector<*CheckpointId*> &*reached_checkpoints*, *ProgressVersion version*) = 0

Indicate that a participant has reached certain checkpoints.

Parameters

- `[in] participant`: The ID of the participant whose progress is being set.
- `[in] plan`: The ID of the plan which progress has been made for.
- `[in] reached_checkpoints`: The set of checkpoints that have been reached. The indices in the vector must correspond to the RouteIds of the plan.

- `[in] version`: The version number for this progress.

virtual void **clear**(*ParticipantId participant*, *ItineraryVersion version*) = 0
Erase an itinerary from this database.

Parameters

- `[in] participant`: The ID of the participant whose itinerary is being erased.
- `[in] version`: The version for this itinerary change

virtual *Registration* **register_participant**(*ParticipantDescription participant_info*) = 0
Register a new participant.

Return result of registering the new participant.

Parameters

- `[in] participant_info`: Information about the new participant.
- `[in] time`: The time at which the registration is being requested.

virtual void **unregister_participant**(*ParticipantId participant*) = 0
Unregister an existing participant.

Return the new version of the schedule.

Parameters

- `[in] participant`: The ID of the participant to unregister.

virtual void **update_description**(*ParticipantId participant*, *ParticipantDescription desc*) = 0
Updates a participants footprint

Parameters

- `[in] participant`: The ID of the participant to update
- `[in] desc`: The participant description

virtual ~Writer() = default

class Registration
Information resulting from registering a participant.

Public Functions

Registration(*ParticipantId id*, *ItineraryVersion version*, *PlanId plan_id*, *StorageId storage_base*)
Constructor

Parameters

- `[in] id`: The ID for the registered participant
- `[in] version`: The last itinerary version for the registered participant
- `[in] plan_id`: The last plan_id for the registered participant
- `[in] storage_base`: The next storage base that the registered participant should use

ParticipantId **id**() **const**
The ID of the registered participant.

ItineraryVersion **last_itinerary_version**() **const**
The last itinerary version of the registered participant. New Participants will begin by adding up from this version when issuing schedule updates.

This value might vary for systems that enforce participant uniqueness. If this participant was registered in the past and is now being re-registered, then the version number will pick up where it previously left off.

PlanId **last_plan_id**() **const**
The last *Route* ID of the registered participant. New Participants will begin by adding up from this *Route* ID when issuing new schedule updates.

Similar to last_itinerary_version, this value might vary for systems that enforce participant uniqueness.

StorageId **next_storage_base**() **const**
The next storage base that the participant should use.

Class Writer::Registration

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_Writer.hpp

Nested Relationships

This class is a nested type of *Class Writer*.

Class Documentation

class rmf_traffic::schedule::*Writer*::**Registration**
Information resulting from registering a participant.

Public Functions

Registration(*ParticipantId* *id*, *ItineraryVersion* *version*, *PlanId* *plan_id*, *StorageId* *storage_base*)
Constructor

Parameters

- [in] id: The ID for the registered participant
- [in] version: The last itinerary version for the registered participant
- [in] plan_id: The last plan_id for the registered participant
- [in] storage_base: The next storage base that the registered participant should use

ParticipantId **id**() **const**
The ID of the registered participant.

ItineraryVersion **last_itinerary_version**() **const**
The last itinerary version of the registered participant. New Participants will begin by adding up from this version when issuing schedule updates.

This value might vary for systems that enforce participant uniqueness. If this participant was registered in the past and is now being re-registered, then the version number will pick up where it previously left off.

PlanId **last_plan_id**() **const**
The last *Route* ID of the registered participant. New Participants will begin by adding up from this *Route* ID when issuing new schedule updates.

Similar to last_itinerary_version, this value might vary for systems that enforce participant uniqueness.

StorageId **next_storage_base**() **const**
The next storage base that the participant should use.

Class Trajectory

- Defined in file_latest_rmf_traffic_include_rmf_traffic_Trajectory.hpp

Nested Relationships

Nested Types

- *Struct Trajectory::InsertionResult*
- *Class Trajectory::Waypoint*
- *Template Class Trajectory::base_iterator*

Class Documentation

class rmf_traffic::**Trajectory**

Public Types

using iterator = *base_iterator*<*Waypoint*>

using const_iterator = *base_iterator*<**const** *Waypoint*>

Public Functions

Trajectory()
Create an empty *Trajectory*.

Trajectory(**const** *Trajectory* &*other*)

Trajectory &**operator=**(**const** *Trajectory* &*other*)

Trajectory(*Trajectory*&&) = default

Warning After using the move constructor or move assignment operator, the *Trajectory* that was moved from will be unusable until a fresh *Trajectory* instance is assigned to it (using either the copy or move constructor). Attempting to use a *Trajectory* that was moved from will result in a segfault if you do not assign it a new instance.

Trajectory &**operator=**(*Trajectory*&&) = default

InsertionResult **insert** (*Time time*, Eigen::Vector3d *position*, Eigen::Vector3d *velocity*)
Add a *Waypoint* to this *Trajectory*.

The *Waypoint* will be inserted into the *Trajectory* according to its time, ensuring correct ordering of all Waypoints.

InsertionResult **insert** (**const** *Waypoint &other*)
Insert a copy of another *Trajectory*'s *Waypoint* into this one.

iterator **find** (*Time time*)
Find the *Waypoint* of this *Trajectory* that comes after or exactly on the given time.

Note This will return *Trajectory::end()* if the time is before the *Trajectory* starts or after the *Trajectory* finishes.

Return an iterator to the *Waypoint* that is active during the given time, or *Trajectory::end()* if the time falls outside the range of the *Trajectory*

Parameters

- `[in] time`: The time of interest.

const_iterator **find** (*Time time*) **const**
const-qualified version of *find()*

Waypoint &**operator[]** (std::size_t *index*)
Get a reference to the *Waypoint* at the specified index. No bounds checking is performed, so there will be undefined behavior if the index is out of bounds.

const *Waypoint* &**operator[]** (std::size_t *index*) **const**
Const-qualified index operator.

Waypoint &**at** (std::size_t *index*)
Get a reference to the *Waypoint* at the specified index. Bound checking will be performed, and an exception will be thrown if index is out of bounds.

const *Waypoint* &**at** (std::size_t *index*) **const**
Const-qualified *at()*

iterator **lower_bound** (*Time time*)
Get the first waypoint of this *Trajectory* that occurs at a time greater than or equal to the specified time. This is effectively the same as *find(Time)*, except it will return *Trajectory::begin()* if the time comes before the start of the *Trajectory*.

Return an iterator to the first *Waypoint* that occurs at a time on or after the given time, or *Trajectory::end()* if the time is after the end of the *Trajectory*.

Parameters

- `[in] time`: The inclusive lower bound on the time of interest.

const_iterator **lower_bound** (*Time time*) **const**
const-qualified version of *lower_bound()*

std::size_t **index_after** (*Time time*) **const**
Get the index of first waypoint that comes after the specified time. If the last waypoint in the trajectory comes before the specified time then *size()* will be returned.

iterator **erase** (*iterator waypoint*)
Erase the specified waypoint.

Return an iterator following the last removed element

iterator **erase**(*iterator first*, *iterator last*)
Erase the range of elements: [first, last).

Note The `last` element is not included in the range.

Return an iterator following the last removed element

iterator **begin**()
Returns an iterator to the fist *Waypoint* of the *Trajectory*.

If the *Trajectory* is empty, the returned iterator will be equal to *end()*.

const_iterator **begin**() **const**
const-qualified version of *begin()*

const_iterator **cbegin**() **const**
Explicitly call the const-qualified version of *begin()*

iterator **end**()
Returns an iterator to the element following the last *Waypoint* of the *Trajectory*. This iterator acts as a placeholder; attempting to dereference it results in undefined behavior.

Note In compliance with C++ standards, this is really a one-past-the-end iterator and must not be dereferenced. It should only be used to identify when an iteration must end. See: https://en.cppreference.com/w/cpp/container/list/end

const_iterator **end**() **const**
const-qualified version of *end()*

const_iterator **cend**() **const**
Explicitly call the const-qualified version of *end()*

Waypoint &**front**()
Get a mutable reference to the first *Waypoint* in this *Trajectory*.

Warning Calling this function on an empty trajectory is undefined.

const *Waypoint* &**front**() **const**
Get a const reference to the first *Waypoint* in this *Trajectory*.

Warning Calling this function on an empty trajectory is undefined.

Waypoint &**back**()
Get a mutable reference to the last *Waypoint* in this *Trajectory*.

Warning Calling this function on an empty trajectory is undefined.

const *Waypoint* &**back**() **const**
Get a const reference to the last *Waypoint* in this *Trajectory*.

Warning Calling this function on an empty trajectory is undefined.

const *Time* ***start_time**() **const**
Get the start time, if available. This will return a nullptr if the *Trajectory* is empty.

const *Time* ***finish_time**() **const**
Get the finish time of the *Trajectory*, if available. This will return a nullptr if the *Trajectory* is empty.

Duration **duration**() **const**
Get the duration of the *Trajectory*. This will be 0 if the *Trajectory* is empty or if it has only one *Waypoint*.

std::size_t **size**() **const**
Get the number of Waypoints in the *Trajectory*. To be used in conflict detection, the *Trajectory* must have a size of at least 2.

bool **empty**() **const**
Return true if the trajectory has no waypoints, false otherwise.

Friends

friend class internal::TrajectoryIteratorImplementation

template<typename **W**>
class base_iterator

Public Functions

W &**operator***() **const**
Dereference operator.

W ***operator->**() **const**
Drill-down operator.

base_iterator &**operator++**()
Pre-increment operator: ++it

Note This is more efficient than the post-increment operator.
Return a reference to the iterator that was operated on

base_iterator &**operator--**()
Pre-decrement operator: it

Note This is more efficient than the post-decrement operator
Return a reference to the iterator that was operated on

base_iterator **operator++**(int)
Post-increment operator: it++

Return a copy of the iterator before it was incremented

base_iterator **operator--**(int)
Post-decrement operator: it

Return a copy of the iterator before it was decremented

bool **operator==**(**const** *base_iterator* &*other*) **const**
Equality comparison operator.

bool **operator!=**(**const** *base_iterator* &*other*) **const**
Inequality comparison operator.

bool **operator<**(**const** *base_iterator* &*other*) **const**
Less-than comparison operator (the left-hand side is earlier in the trajectory than the right-hand side)

bool **operator>**(**const** *base_iterator* &*other*) **const**
Greater-than comparison operator (the left-hand side is later in the trajectory than the right-hand side)

bool **operator<=**(**const** *base_iterator* &*other*) **const**
Less-than-or-equal comparison operator.

bool **operator>=**(**const** *base_iterator* &*other*) **const**
Greater-than-or-equal comparison operator.

operator const_iterator() **const**

base_iterator(**const** *base_iterator* &*other*) = default

base_iterator(*base_iterator* &&*other*) = default

base_iterator &**operator=**(**const** *base_iterator* &*other*) = default

base_iterator &**operator=**(*base_iterator* &&*other*) = default

base_iterator()

Friends

friend class internal::TrajectoryIteratorImplementation

struct InsertionResult

Public Members

iterator **it**

bool **inserted**

class Waypoint

Public Functions

Eigen::Vector3d **position**() **const**
Get the intended physical location of the robot at the end of this *Trajectory Waypoint*.

This is a 2D homogeneous position. The first two values in the vector are x and y coordinates, while the third is rotation about the z-axis.

Waypoint &**position**(Eigen::Vector3d *new_position*)
Set the intended physical location of the robot at the end of this *Trajectory Waypoint*.

This is a 2D homogeneous position. The first two values in the vector are x and y coordinates, while the third is rotation about the z-axis.

Parameters
- `[in] new_position`: The new position for this *Trajectory Waypoint*.

Eigen::Vector3d **velocity**() **const**
Get the intended velocity of the robot at the end of this *Trajectory Waypoint*.

This is a 2D homogeneous position. The first two values in the vector are x and y velocities, while the third is rotational velocity about the z-axis.

Waypoint &**velocity**(Eigen::Vector3d *new_velocity*)
Set the intended velocity of the robot at the end of this *Trajectory Waypoint*.

This is a 2D homogeneous position. The first two values in the vector are x and y coordinates, while the third is rotation about the z-axis.

Parameters
- `[in] new_velocity`: The new velocity at this *Trajectory Waypoint*.

Time **time**() **const**
Get the time that the trajectory will reach this *Waypoint*.

std::size_t **index**() **const**
The index of this waypoint within its trajectory. Waypoints are indexed according to their chronological order. Adjusting the time of any waypoint in a trajectory could change its index and/or the index of other waypoints.

Waypoint &**change_time**(*Time new_time*)
Change the timing of this *Trajectory Waypoint*. Note that this function will only affect this waypoint, and may cause this waypoint to be reordered within the *Trajectory*.

To change the timing for this waypoint while preserving the relative times of all subsequent *Trajectory* Waypoints, use *adjust_times()* instead.

Warning If you change the time value of this *Waypoint* such that it falls directly on another *Waypoint*'s time, you will get a std::invalid_argument exception, because discontinuous jumps are not supported, and indicate a significant mishandling of trajectory data, which is most likely a serious bug that should be remedied.
Note If this *Waypoint*'s time crosses over another *Waypoint*'s time, that signficantly changes the topology of the *Trajectory*, because it will change the order in which the positions are traversed.
See adjust_times(Time new_time)
Parameters
- `[in] new_time`: The new timing for this *Trajectory Waypoint*.

void **adjust_times**(*Duration delta_t*)
Adjust the timing of this waypoint and all subsequent waypoints by the given duration. This is guaranteed to maintain the ordering of the *Trajectory* Waypoints, and is more efficient than changing all the times directly.

Warning If a negative delta_t is given, it must not cause this *Waypoint*'s time to be less than or equal to the time of its preceding *Waypoint*, or else a std::invalid_argument exception will be thrown.
See *change_time(Time new_time)*
Parameters
- `[in] delta_t`: How much to change the timing of this waypoint and all later waypoints. If negative, it must not cross over the time of the previous waypoint, or else a std::invalid_argument will be thrown.

Template Class Trajectory::base_iterator

- Defined in file_latest_rmf_traffic_include_rmf_traffic_Trajectory.hpp

Nested Relationships

This class is a nested type of *Class Trajectory*.

Class Documentation

template<typename **W**>
class rmf_traffic::*Trajectory*::**base_iterator**

Public Functions

W &**operator***() **const**
Dereference operator.

W ***operator->**() **const**
Drill-down operator.

base_iterator &**operator++**()
Pre-increment operator: ++it

Note This is more efficient than the post-increment operator.

Return a reference to the iterator that was operated on

base_iterator &**operator--**()
Pre-decrement operator: it

Note This is more efficient than the post-decrement operator

Return a reference to the iterator that was operated on

base_iterator **operator++**(int)
Post-increment operator: it++

Return a copy of the iterator before it was incremented

base_iterator **operator--**(int)
Post-decrement operator: it

Return a copy of the iterator before it was decremented

bool **operator==**(**const** *base_iterator* &*other*) **const**
Equality comparison operator.

bool **operator!=**(**const** *base_iterator* &*other*) **const**
Inequality comparison operator.

bool **operator<**(**const** *base_iterator* &*other*) **const**
Less-than comparison operator (the left-hand side is earlier in the trajectory than the right-hand side)

bool **operator>**(**const** *base_iterator* &*other*) **const**
Greater-than comparison operator (the left-hand side is later in the trajectory than the right-hand side)

bool **operator<=**(**const** *base_iterator* &*other*) **const**
Less-than-or-equal comparison operator.

bool **operator>=**(**const** *base_iterator* &*other*) **const**
Greater-than-or-equal comparison operator.

operator const_iterator() **const**

base_iterator(**const** *base_iterator* &*other*) = default

base_iterator(*base_iterator* &&*other*) = default

base_iterator &**operator=**(**const** *base_iterator* &*other*) = default

base_iterator &**operator=**(*base_iterator* &&*other*) = default

base_iterator()

Friends

friend class internal::TrajectoryIteratorImplementation

Class Trajectory::Waypoint

- Defined in file_latest_rmf_traffic_include_rmf_traffic_Trajectory.hpp

Nested Relationships

This class is a nested type of *Class Trajectory*.

Class Documentation

class rmf_traffic::*Trajectory*::**Waypoint**

Public Functions

Eigen::Vector3d **position**() **const**
Get the intended physical location of the robot at the end of this *Trajectory Waypoint*.

This is a 2D homogeneous position. The first two values in the vector are x and y coordinates, while the third is rotation about the z-axis.

Waypoint &**position**(Eigen::Vector3d *new_position*)
Set the intended physical location of the robot at the end of this *Trajectory Waypoint*.

This is a 2D homogeneous position. The first two values in the vector are x and y coordinates, while the third is rotation about the z-axis.

Parameters

- `[in] new_position`: The new position for this *Trajectory Waypoint*.

Eigen::Vector3d **velocity**() **const**
Get the intended velocity of the robot at the end of this *Trajectory Waypoint*.

This is a 2D homogeneous position. The first two values in the vector are x and y velocities, while the third is rotational velocity about the z-axis.

Waypoint &**velocity**(Eigen::Vector3d *new_velocity*)
Set the intended velocity of the robot at the end of this *Trajectory Waypoint*.

This is a 2D homogeneous position. The first two values in the vector are x and y coordinates, while the third is rotation about the z-axis.

Parameters

- `[in] new_velocity`: The new velocity at this *Trajectory Waypoint*.

Time **time**() **const**
Get the time that the trajectory will reach this *Waypoint*.

std::size_t **index**() **const**
The index of this waypoint within its trajectory. Waypoints are indexed according to their chronological order. Adjusting the time of any waypoint in a trajectory could change its index and/or the index of other waypoints.

Waypoint &**change_time**(*Time new_time*)
Change the timing of this *Trajectory Waypoint*. Note that this function will only affect this waypoint, and may cause this waypoint to be reordered within the *Trajectory*.

To change the timing for this waypoint while preserving the relative times of all subsequent *Trajectory* Waypoints, use *adjust_times()* instead.

Warning If you change the time value of this *Waypoint* such that it falls directly on another *Waypoint*'s time, you will get a std::invalid_argument exception, because discontinuous jumps are not supported, and indicate a significant mishandling of trajectory data, which is most likely a serious bug that should be remedied.

Note If this *Waypoint*'s time crosses over another *Waypoint*'s time, that signficantly changes the topology of the *Trajectory*, because it will change the order in which the positions are traversed.

See adjust_times(Time new_time)

Parameters

- `[in] new_time`: The new timing for this *Trajectory Waypoint*.

void **adjust_times**(*Duration delta_t*)
Adjust the timing of this waypoint and all subsequent waypoints by the given duration. This is guaranteed to maintain the ordering of the *Trajectory* Waypoints, and is more efficient than changing all the times directly.

Warning If a negative delta_t is given, it must not cause this *Waypoint*'s time to be less than or equal to the time of its preceding *Waypoint*, or else a std::invalid_argument exception will be thrown.

See *change_time(Time new_time)*

Parameters

- `[in] delta_t`: How much to change the timing of this waypoint and all later waypoints. If negative, it must not cross over the time of the previous waypoint, or else a std::invalid_argument will be thrown.

1.2.3 Functions

Function rmf_traffic::agv::compute_plan_starts

- Defined in file_latest_rmf_traffic_include_rmf_traffic_agv_Planner.hpp

Function Documentation

std::vector<*Plan*::*Start*> `rmf_traffic::agv::`**`compute_plan_starts`**(**`const`** rmf_traffic::agv::*Graph* &*graph*, **`const`** std::string &*map_name*, **`const`** Eigen::Vector3d *pose*, **`const`** rmf_traffic::*Time* *start_time*, **`const`** double *max_merge_waypoint_distance* = 0.1, **`const`** double *max_merge_lane_distance* = 1.0, **`const`** double *min_lane_length* = 1e-8)

Produces a set of possible starting waypoints and lanes in order to start planning. This method attempts to find the most suitable starting nodes within the provided graph for merging, planning and execution of plans, from the provided pose. If none of the waypoints in the graph fulfils the requirements, an empty vector will be returned.

Parameters

- `[in] graph`: *Graph* which the starting waypoints and lanes will be derived from.
- `[in] pose`: Current pose in terms of 2D coordinates, x and y, being the first and second element respectively, while the third element being the yaw.
- `[in] start_time`: The starting time that will be attributed to all the generated starts to compute a new plan. In some occasions, users will want to add small delays to the current time, in order to account for computation time or network delays.
- `[in] max_merge_waypoint_distance`: The maximum distance allowed to automatically merge onto a waypoint in the graph. Default value as 0.1 meters.
- `[in] max_merge_lane_distance`: The maximum distance allowed to automatically merge onto a lane, i.e. adding the lane's entry and exit waypoints as potential starts. Default value as 1.0 meters.
- `[in] min_lane_length`: The minimum length of a lane in the provided graph to be considered valid, any lanes shorter than this value will not be evaluated. Default value as 1e-8 meters.

Function rmf_traffic::agv::interpolate_time_along_quadratic_straight_line

- Defined in file_latest_rmf_traffic_include_rmf_traffic_agv_Interpolate.hpp

Function Documentation

TimeVelocity rmf_traffic::agv::**interpolate_time_along_quadratic_straight_line**(**const** *Trajectory* &*trajectory*, **const** Eigen::Vector2d &*position*, double *holding_point_tolerance* = 0.05)

This function only works correctly if the trajectory follows a straight line trajectory with zero jerk (cubic coefficient) and the position lies along the trajectory.

Function rmf_traffic::blockade::make_participant

- Defined in file_latest_rmf_traffic_include_rmf_traffic_blockade_Participant.hpp

Function Documentation

Participant rmf_traffic::blockade::**make_participant**(*ParticipantId* *participant_id*, double *radius*, std::shared_ptr<*Writer*> *writer*, std::shared_ptr<*RectificationRequesterFactory*> *rectifier_factory* = nullptr)

Make a blockade participant.

Parameters

- [in] participant_id: Every blockade participant must also be a schedule participant. Pass in the schedule participant ID here.
- [in] radius: The initial default radius to use for this participant's blockade.
- [in] writer: The writer that this participant should interact with.
- [in] rectifier_factory: The factory that this participant should use to create a rectifier for itself. If no factory is provided, we will assume the writer is always perfectly reliable.

Template Function rmf_traffic::geometry::make_final(Args&&...)

- Defined in file_latest_rmf_traffic_include_rmf_traffic_geometry_Shape.hpp

Function Documentation

template<typename **T**, typename ...**Args**>
FinalShapePtr `rmf_traffic::geometry::`**`make_final`**`(`*Args*`&&... args)`

Template Function rmf_traffic::geometry::make_final(const T&)

- Defined in file_latest_rmf_traffic_include_rmf_traffic_geometry_Shape.hpp

Function Documentation

template<typename **T**>
FinalShapePtr `rmf_traffic::geometry::`**`make_final`**`(`**`const`** *T* `&shape)`

Template Function rmf_traffic::geometry::make_final_convex(Args&&...)

- Defined in file_latest_rmf_traffic_include_rmf_traffic_geometry_ConvexShape.hpp

Function Documentation

template<typename **T**, typename ...**Args**>
FinalConvexShapePtr `rmf_traffic::geometry::`**`make_final_convex`**`(`*Args*`&&... args)`

Template Function rmf_traffic::geometry::make_final_convex(const T&)

- Defined in file_latest_rmf_traffic_include_rmf_traffic_geometry_ConvexShape.hpp

Function Documentation

template<typename **T**>
FinalConvexShapePtr `rmf_traffic::geometry::`**`make_final_convex`**`(`**`const`** *T* `&convex)`

Function rmf_traffic::geometry::operator!=(const Circle&, const Circle&)

- Defined in file_latest_rmf_traffic_include_rmf_traffic_geometry_Circle.hpp

Function Documentation

bool `rmf_traffic::geometry::operator!=`(**const** *Circle* &*lhs*, **const** *Circle* &*rhs*)
Non-equality operator for *Circle* objects.

Parameters

- `[in] lhs`: A const reference to the left-hand-side of the comparison.
- `[in] rhs`: A const reference to the right-hand-side of the comparison.

Function rmf_traffic::geometry::operator!=(const Space&, const Space&)

- Defined in file_latest_rmf_traffic_include_rmf_traffic_geometry_Space.hpp

Function Documentation

bool `rmf_traffic::geometry::operator!=`(**const** *Space* &*lhs*, **const** *Space* &*rhs*)
Non-equality operator for *Space* objects.

Parameters

- `[in] lhs`: A const reference to the left-hand-side of the comparison.
- `[in] rhs`: A const reference to the right-hand-side of the comparison.

Function rmf_traffic::geometry::operator==(const Circle&, const Circle&)

- Defined in file_latest_rmf_traffic_include_rmf_traffic_geometry_Circle.hpp

Function Documentation

bool `rmf_traffic::geometry::operator==`(**const** *Circle* &*lhs*, **const** *Circle* &*rhs*)
Equality operator for *Circle* objects.

Parameters

- `[in] lhs`: A const reference to the left-hand-side of the comparison.
- `[in] rhs`: A const reference to the right-hand-side of the comparison.

Function rmf_traffic::geometry::operator==(const Space&, const Space&)

- Defined in file_latest_rmf_traffic_include_rmf_traffic_geometry_Space.hpp

Function Documentation

bool `rmf_traffic::geometry::operator==`(**const** *Space* &*lhs*, **const** *Space* &*rhs*)
Equality operator for *Space* objects.

Parameters

- `[in] lhs`: A const reference to the left-hand-side of the comparison.
- `[in] rhs`: A const reference to the right-hand-side of the comparison.

Function rmf_traffic::operator!=

- Defined in file_latest_rmf_traffic_include_rmf_traffic_Region.hpp

Function Documentation

bool `rmf_traffic::operator!=`(**const** *Region* &*lhs*, **const** *Region* &*rhs*)
Non-equality operator for *Region* objects.

Parameters

- `[in] lhs`: A const reference to the left-hand-side of the comparison.
- `[in] rhs`: A const reference to the right-hand-side of the comparison.

Function rmf_traffic::operator==

- Defined in file_latest_rmf_traffic_include_rmf_traffic_Region.hpp

Function Documentation

bool `rmf_traffic::operator==`(**const** *Region* &*lhs*, **const** *Region* &*rhs*)
Equality operator for *Region* objects.

Parameters

- `[in] lhs`: A const reference to the left-hand-side of the comparison.
- `[in] rhs`: A const reference to the right-hand-side of the comparison.

Function rmf_traffic::schedule::make_query(std::vector<Region>)

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_Query.hpp

Function Documentation

Query `rmf_traffic::schedule::`**`make_query`**(std::vector<*Region*> *regions*)

Query for all Trajectories that intersect with this set of spacetime regions.

Parameters

- `[in] regions`: Only query Trajectories that intersect with the specified regions.

Function rmf_traffic::schedule::make_query(std::vector<std::string>, const Time *, const Time *)

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_Query.hpp

Function Documentation

Query `rmf_traffic::schedule::`**`make_query`**(std::vector<std::string> *maps*, **`const`** *Time* **start_time*, **`const`** *Time* **finish_time*)

Query for all Trajectories that fall within a time range.

Parameters

- `[in] start_time`: A pointer to the lower bound for the time range. Pass in a nullptr to indicate that there is no lower bound.
- `[in] finish_time`: A pointer to the upper bound for the time range. Pass in a nullptr to indicate that there is no upper bound.

Function rmf_traffic::schedule::operator!=

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_Query.hpp

Function Documentation

bool `rmf_traffic::schedule::`**`operator!=`**(**`const`** *Query* &*lhs*, **`const`** *Query* &*rhs*)

Non-equality operator for *Query* objects.

Parameters

- `[in] lhs`: A const reference to the left-hand-side of the comparison.
- `[in] rhs`: A const reference to the right-hand-side of the comparison.

Function rmf_traffic::schedule::operator==

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_Query.hpp

Function Documentation

bool `rmf_traffic::schedule::operator==`(**const** *Query* &*lhs*, **const** *Query* &*rhs*)
Equality operator for *Query* objects.

Parameters

- `[in] lhs`: A const reference to the left-hand-side of the comparison.
- `[in] rhs`: A const reference to the right-hand-side of the comparison.

Function rmf_traffic::schedule::query_all

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_Query.hpp

Function Documentation

Query `rmf_traffic::schedule::query_all`()
Query for all entries in a schedule database.

Function rmf_traffic::time::apply_offset

- Defined in file_latest_rmf_traffic_include_rmf_traffic_Time.hpp

Function Documentation

Time `rmf_traffic::time::apply_offset`(*Time* *start_time*, double *delta_seconds*)
Return the given start_time, offset by the number of seconds given.

Parameters

- `[in] start_time`: The time to start from
- `[in] delta_seconds`: The number of seconds to add to the start_time

Function rmf_traffic::time::from_seconds

- Defined in file_latest_rmf_traffic_include_rmf_traffic_Time.hpp

Function Documentation

Duration rmf_traffic::time::**from_seconds**(double *delta_t*)
Chance the given duration from a double-precision floating-point representation to a nanosecond count.

Function rmf_traffic::time::to_seconds

- Defined in file_latest_rmf_traffic_include_rmf_traffic_Time.hpp

Function Documentation

double rmf_traffic::time::**to_seconds**(*Duration delta_t*)
Change the given duration from a nanosecond count to a double-precision floating-point representation in seconds.

1.2.4 Defines

Define CAPTURE_LEAK

- Defined in file_latest_rmf_traffic_include_rmf_traffic_debug_Plumber.hpp

Define Documentation

CAPTURE_LEAK(*X*)

Define CAPTURE_LEAK_HERE

- Defined in file_latest_rmf_traffic_include_rmf_traffic_debug_Plumber.hpp

Define Documentation

CAPTURE_LEAK_HERE

Define CHECK_LEAK

- Defined in file_latest_rmf_traffic_include_rmf_traffic_debug_Plumber.hpp

Define Documentation

CHECK_LEAK(*X*)

1.2.5 Typedefs

Typedef rmf_traffic::blockade::CheckpointId

- Defined in file_latest_rmf_traffic_include_rmf_traffic_blockade_Status.hpp

Typedef Documentation

using rmf_traffic::blockade::**CheckpointId** = uint64_t

Typedef rmf_traffic::blockade::ParticipantId

- Defined in file_latest_rmf_traffic_include_rmf_traffic_blockade_Status.hpp

Typedef Documentation

using rmf_traffic::blockade::**ParticipantId** = uint64_t

Typedef rmf_traffic::blockade::ReservationId

- Defined in file_latest_rmf_traffic_include_rmf_traffic_blockade_Status.hpp

Typedef Documentation

using rmf_traffic::blockade::**ReservationId** = uint64_t

Typedef rmf_traffic::blockade::Version

- Defined in file_latest_rmf_traffic_include_rmf_traffic_blockade_Status.hpp

Typedef Documentation

using rmf_traffic::blockade::**Version** = uint64_t

Typedef rmf_traffic::CheckpointId

- Defined in file_latest_rmf_traffic_include_rmf_traffic_Route.hpp

Typedef Documentation

using rmf_traffic::**CheckpointId** = uint64_t

Typedef rmf_traffic::ConstRoutePtr

- Defined in file_latest_rmf_traffic_include_rmf_traffic_Route.hpp

Typedef Documentation

using rmf_traffic::**ConstRoutePtr** = std::shared_ptr<**const** *Route*>

Typedef rmf_traffic::Dependencies

- Defined in file_latest_rmf_traffic_include_rmf_traffic_Route.hpp

Typedef Documentation

using rmf_traffic::**Dependencies** = std::vector<*Dependency*>

Typedef rmf_traffic::DependsOnCheckpoint

- Defined in file_latest_rmf_traffic_include_rmf_traffic_Route.hpp

Typedef Documentation

using rmf_traffic::**DependsOnCheckpoint** = std::map<*CheckpointId*, *CheckpointId*>
The checkpoint in the value waits for the checkpoint in the key.

Typedef rmf_traffic::DependsOnParticipant

- Defined in file_latest_rmf_traffic_include_rmf_traffic_Route.hpp

Typedef Documentation

using rmf_traffic::**DependsOnParticipant** = std::unordered_map<*ParticipantId*, *DependsOnPlan*>
Express a dependency on a participant.

Typedef rmf_traffic::DependsOnRoute

- Defined in file_latest_rmf_traffic_include_rmf_traffic_Route.hpp

Typedef Documentation

using rmf_traffic::**DependsOnRoute** = std::unordered_map<*RouteId*, *DependsOnCheckpoint*>
The checkpoint dependencies relate to the route ID of the key.

Typedef rmf_traffic::Duration

- Defined in file_latest_rmf_traffic_include_rmf_traffic_Time.hpp

Typedef Documentation

using rmf_traffic::**Duration** = std::chrono::steady_clock::duration
Specifies a change in time, with nanosecond precision.

Typedef rmf_traffic::geometry::ConstConvexShapePtr

- Defined in file_latest_rmf_traffic_include_rmf_traffic_geometry_ConvexShape.hpp

Typedef Documentation

using rmf_traffic::geometry::**ConstConvexShapePtr** = std::shared_ptr<**const** *ConvexShape*>

Typedef rmf_traffic::geometry::ConstFinalConvexShapePtr

- Defined in file_latest_rmf_traffic_include_rmf_traffic_geometry_ConvexShape.hpp

Typedef Documentation

using rmf_traffic::geometry::**ConstFinalConvexShapePtr** = std::shared_ptr<**const** *FinalConvexShape*>

Typedef rmf_traffic::geometry::ConstFinalShapePtr

- Defined in file_latest_rmf_traffic_include_rmf_traffic_geometry_Shape.hpp

Typedef Documentation

using rmf_traffic::geometry::**ConstFinalShapePtr** = std::shared_ptr<**const** *FinalShape*>

Typedef rmf_traffic::geometry::ConstShapePtr

- Defined in file_latest_rmf_traffic_include_rmf_traffic_geometry_Shape.hpp

Typedef Documentation

using rmf_traffic::geometry::**ConstShapePtr** = std::shared_ptr<**const** *Shape*>

Typedef rmf_traffic::geometry::ConvexShapePtr

- Defined in file_latest_rmf_traffic_include_rmf_traffic_geometry_ConvexShape.hpp

Typedef Documentation

using rmf_traffic::geometry::**ConvexShapePtr** = std::shared_ptr<*ConvexShape*>

Typedef rmf_traffic::geometry::FinalConvexShapePtr

- Defined in file_latest_rmf_traffic_include_rmf_traffic_geometry_ConvexShape.hpp

Typedef Documentation

using rmf_traffic::geometry::**FinalConvexShapePtr** = std::shared_ptr<*FinalConvexShape*>

Typedef rmf_traffic::geometry::FinalShapePtr

- Defined in file_latest_rmf_traffic_include_rmf_traffic_geometry_Shape.hpp

Typedef Documentation

using rmf_traffic::geometry::**FinalShapePtr** = std::shared_ptr<*FinalShape*>

Typedef rmf_traffic::geometry::ShapePtr

- Defined in file_latest_rmf_traffic_include_rmf_traffic_geometry_Shape.hpp

Typedef Documentation

using rmf_traffic::geometry::**ShapePtr** = std::shared_ptr<*Shape*>

Typedef rmf_traffic::ParticipantId

- Defined in file_latest_rmf_traffic_include_rmf_traffic_Route.hpp

Typedef Documentation

using rmf_traffic::**ParticipantId** = uint64_t

Typedef rmf_traffic::PlanId

- Defined in file_latest_rmf_traffic_include_rmf_traffic_Route.hpp

Typedef Documentation

using rmf_traffic::**PlanId** = uint64_t

Typedef rmf_traffic::RouteId

- Defined in file_latest_rmf_traffic_include_rmf_traffic_Route.hpp

Typedef Documentation

using rmf_traffic::**RouteId** = uint64_t

Typedef rmf_traffic::RoutePtr

- Defined in file_latest_rmf_traffic_include_rmf_traffic_Route.hpp

Typedef Documentation

using rmf_traffic::**RoutePtr** = std::shared_ptr<*Route*>

Typedef rmf_traffic::schedule::Itinerary

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_Itinerary.hpp

Typedef Documentation

using rmf_traffic::schedule::**Itinerary** = std::vector<*Route*>

Typedef rmf_traffic::schedule::ItineraryVersion

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_Itinerary.hpp

Typedef Documentation

using rmf_traffic::schedule::**ItineraryVersion** = uint64_t

Typedef rmf_traffic::schedule::ItineraryView

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_Itinerary.hpp

Typedef Documentation

using rmf_traffic::schedule::**ItineraryView** = std::vector<std::shared_ptr<**const** *Route*>>

Typedef rmf_traffic::schedule::ParticipantDescriptionsMap

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_ParticipantDescription.hpp

Typedef Documentation

using rmf_traffic::schedule::**ParticipantDescriptionsMap** = std::unordered_map<*ParticipantId*, *ParticipantDe*

Typedef rmf_traffic::schedule::ParticipantId

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_ParticipantDescription.hpp

Typedef Documentation

using rmf_traffic::schedule::**ParticipantId** = uint64_t

Typedef rmf_traffic::schedule::ProgressVersion

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_Itinerary.hpp

Typedef Documentation

using rmf_traffic::schedule::**ProgressVersion** = uint64_t

Typedef rmf_traffic::schedule::StorageId

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_Change.hpp

Typedef Documentation

using rmf_traffic::schedule::**StorageId** = uint64_t

Typedef rmf_traffic::schedule::Version

- Defined in file_latest_rmf_traffic_include_rmf_traffic_schedule_Version.hpp

Typedef Documentation

using rmf_traffic::schedule::**Version** = uint64_t

The schedule version is represented by an unsigned 64-bit integer. This means that the schedule can identify over 9 quintillion database entries at any single moment in time (which is more than a server is likely to have enough RAM to store).

The Version number is used to identify the current version of a database.

As database entries become irrelevant (e.g. they refer to events that have already finished taking place) they will be culled from the database. The database will keep track of what its "oldest" known version number is. After a very long period of continuous operation, the version numbers could eventually wrap around and overflow the 64-bit unsigned integer. This is okay because modular arithmetic will be used to ensure that version values which are lower than the "oldest" version number will be evaluated as greater than any version numbers that are greater than the "oldest" version number.

Typedef rmf_traffic::Time

- Defined in file_latest_rmf_traffic_include_rmf_traffic_Time.hpp

Typedef Documentation

using rmf_traffic::**Time** = std::chrono::steady_clock::time_point

Specifies a specific point in time, with nanosecond precision.

Conventionally this will be represented relative to the Unix Epoch.

INDEX

S