Appendix H

Framework - Matlab Implementaiton

The presented framework covers a wide variety of the functionality to enable future DAA methods development. The framework enables to integrate any partial method into complex infrastructure. The user can play with reach set approximation and trajectory properties.

There is a possibility to add additional information sources, sensor reading assessments and reach set approximations. The scalability of the approach is possible thanks to movement automaton; there is only a requirement of the proper movement set implementation.

The amount of the work in this framework is wide; it has been developed over past two years, it started as a simple proof of concept[1] for optimal control report. The predictive properties of the *movement automaton* were added in need of *Predictive control report*[2]. The cooperation with Linkoping University yielded *Data fusion procedure*[3]. The final increment in the form of UTM implementation is summarized in this thesis. The initial build of theframework[4] has been released on 9$^{\text{th}}$ October 2018 for the public.

H.1 Functionality Description

Core Framework: The *core framework* reflects the implementation used in (ch. **??**).

`[Module]` *Vehicle* - UAS Model and Movement Automaton implementation (sec. **??**).

- `[Class]` *Vehicle* - UAS model with movement automaton.
  - `[Method]` *fly(MovementType)* - executes one movement for one second.
  - `[Method]` *plot()* - displays trajectory, UAS body and planned trajectory in GCF.
- `[Class]` *LinerizedModel* - used as movement automaton predictor.
- `[Class]` *State* (Class Role) - represent UAS system state evolution as a set of state-input time series.
  - `[Method]` *plot* - hows state evolution over time in figures.

[1] Optimal Control `https://github.com/logomo/Optimal-Control`
[2] Predictive Control `https://github.com/logomo/Predictive-control---Final-report`
[3] Data fusion procedure `https://github.com/logomo/Data-Fusion-Report`
[4] Feature-based ACAS `https://github.com/logomo/Feature-based-ACAS`

[Module] *AvoidanceGrid* - Avoidance Grid (sec. **??**) and Reach set approximation (sec. **??**).

- [Class] *AvoidanceGrid* - the space segmentation class containing the following methods:
  - [Method] *putObstacle(AbstractObstacle)* intersects static/map obstacle object simulates LiDAR reading in case of *static obstacle*.
  - [Method] *intersectAdversary(AbstractAdversary)* - insert intruder intersection into avoidance grid cells.
  - [Method] *applyConstraint(AbstractConstraint)* - insert constraint hard/soft static/moving into avoidance grid cells.
  - [Method] *precalculate[ReachSetType](PredictorNode)* - creates specific reach set type for initial node (offline/online).
  - [Method] *plotGridSlice(*)* - plots specific status of the cells on horizontal/vertical layer specified by cell indexes.
  - [Method] *recalculate(*)* - enforces data fusion procedure on avoidance grid cells and reach set approximation trajectories.
- [Class] *GridLayer* - represents one distance layer in avoidance grid, used in wave-front algorithm, contains methods for reach set estimation and rating calculations, no notable methods to describe.
- [Class] *GridCell* - represents one cell in avoidance grid, contains a set of passing trajectories, low-level support methods for reach set estimation, no notable methods to describe.
- [Class] *PredictorNode* - represent one piece-wise segment of trajectory between before-after unitary movement application it can be linked into tree/graph data structure:
  - [Method] *expand(AvoidanceGrid)* - applies full movement set to create a possible frontal expansion of the trajectories. Various expansion constraints are applied to enhance functionality.
  - [Method] *plotTrajectories(*)* - shows trajectories coming out from the *root* node.
  - [Method] *calculateCost(*)* - applies a cost function to calculate trajectory segment expenses, this is used later in avoidance path selection process.

[Module] *Obstacles* - Static/Map Obstacles (sec. **??**) and All Constraints types (sec. **??**).

- [Class] *AbstractConstraint* - abstract class representing constraint in the 3D environment the notable methods:
  - [Method] *getPoints(*)* - gets 3D point cloud in GCF, representing LiDAR reading of the constraint (constraint also be used as obstacles and vice-versa).
  - [Method] *dynamize(*)* - enables/disables movement property of constraint, this is used for weather behavior simulation, it can comprehend any kind of linear movement.
  - [Method] *applyMovement(*)* - applies the movement defined by *dynamize(*)* method. This is to force the movement of constraint if it is necessary.
  - [Method] *isIntersection(*)* - Description - calculates the impact on *UAS* operation space (threat verification method).

`[Class]` *PolyConstraint* - the standard weather map is represented as 2D flight level slices, where in leveled flight the weather constraints are 2D polygons. The 3D representation uses a 2D polygon for a horizontal boundary and altitude range for a vertical boundary. The interface methods of *AbstractConstraint* are implemented.

`[Class]` *ExamplePolyConstraint* - the set of example constraints/obstacles, buildings weather areas prototypes, refer to*ExamplePolygonType* enumeration, extension of the *PolyConstraint* class.

`[Class]` *AbstractObstacle* - abstract class representing a map or detected obstacle in a 3D environment, the complete listing of implementations can be found in [1], the notable methods:

`[Method]` *getPoints(*)* - get obstacle points valued with *ObstacleType* enumeration. The intersection algorithm is depending on this,

`[Method]` *isCollision(*)* - checks if there is an inevitable collision with an obstacle body.

`[Method]` *isIntersection(*)* - checks if there is an intersection with UAS operation space.

`[Class]` *SphereObstacle* - the *AbstractObstacle* implementation representing sphere with fixed point center and fixed radius in time.

`[Class]` *BarellObstacle* - the *AbstractObstacle* implementation representing sphere with fixed point center and circle constraint on horizontal GCF plane and altitude limitation on Y GCF axis.

`[Class]` *MazeMatrix* (Class Role) - helper class to create a city-like landscapes in a mesh grid, the plan is defined by *mazemap*, the notable methods:

`[Method]` *generateMazeObstacles(mazeMap)* - returns the set of *AbstractObstacle* implementations, representing the landscape of the maze.

`[Module]` *AdversaryVehicle* - Intruder Intersection Implementation (sec. **??**).

`[Class]` *AbstractAdversaryVehicle* (Class Role) - abstract intruder implementation defining base functions for registering and intruder and outlining intersection models representation, notable method:

`[Method]` *registerSelf(avoidanceGrid)* - the intruder register itself at the UAS avoidance grid.

`[Class]` *AdversaryVehicle* (Class Role) - the specification of *AbstractAdversaryVehicle* interface implementing space related intersection calculations, notable methods:

`[Method]` *findLinearIntersection(*)* - finds linear intersection with avoidance grid.

`[Class]` *TimedAdversaryVehicle* - the specification of *AdversaryVehicle* adding meeting time in avoidance grid cell aspect, notable methods:

`[Method]` *findLinearIntersection(*)* - timed linear intersection with avoidance grid search method.

`[Method]` *findIntersectionBalls(*)* - timed body volume intersection with avoidance grid search method.

[Method] *findIntersectionEllipseCells(*)* - timed uncertainty spread intersection implementation.

[Class] *IntersectionConfig* - intersection model configuration class containing setting class.

[Module] *MissionControl* - Avoidance/Navigation loop implementation (sec. **??**).

[Class] *MissionControl* - the control concept implementation of the *non-cooperative* avoidance for one UAS working with Obstacle/Intruders/Constraints, optimized for LiDAR/ADS-B equipment, notable methods:

[Method] *runOnce(*)* - mission control implementation with all data processing, hierarchies and event handling.

[Method] *runOnceWithPlot(*)* - mission control run with situation plot of mission dynamic content - plane position, flew trajectory, planned trajectory.

[Method] *findBestPath(*)* - avoidance grid run implementation, to search the best path for one time, one threat setup and fixed goal waypoint, underlying functions shows the logic of the trajectory selection.

[Method] *plotMissionStaticContent(*)* - plots static mission content - waypoints, goals initial position.

[Method] *plotObstacletoDistanceStatistic(*)* - plots and calculates the *crash distance to nearest threat* statistic during the mission.

[Method] *plotRealvsPlanTrajectoryStatistics(*)* - plots and calculates *trajectory tracking* statistics.

[Method] *plotAndCalculateComputationTime(*)* - plots and calculates *computational load* statistics.

[Method] *notifyIntruder(*)* - ADS-B notification method corpus, contains *Position-Notification* object creation and *Intruder* registration procedure.

[Method] *getVehiclePositionNotification(*)* - returns actual position notification for UTM implementation.

[Class] *FlightLog* - wrapper class for notable flight parameters snapshots on the *decision time*. The flight log is created by mission control *runOnce(*)* method.

[Class] *Intruder* (Class Role) - mission control non-cooperative intruder data structure (ADS-B like message).

[Class] *Waypoint* (Class Role) - the waypoint representation in 3D GCF, including the status flags.

[Module] *UTM* - UAS Traffic Management leveled flight traffic management services (sec. **??**) implementation in one airspace cluster.

[Class] *UTMControl* - the UTM core services implementation to cover necessary *Rules Of the Air* implementation.

[Method] *registerMission(*)* - registers the mission in active airspace segment.

[Method] *runSimulations(*)* - runs active mission controls for one *UTM decision* frame, this includes the mission dynamic content plot, the missions static content plot must be run prior to this function call.

[Method] *createCollisionCases(*)* - creates collision cases for active collisions, closes inactive cases and runs overall management of the ongoing situations.

[Method] *linkCollisionCase(*)* - when new collision case pops out, the ongoing collision cases are checked for a possible link, also issues the directives to active UAS systems.

[Method] *showCollisioncaseTrace(*)* - shows decision making trace for linked collision cases, the process is printed out in the console.

[Class] *CollisionCase* - collision case data structure wrapper.

[Class] *VehiclePositionNotification* - position notification with additional information data-wrapper.

[Module] *RuleEngine* - UTM directives implementation over *navigation loop* (sec. **??**) (MissionControl/AvoidanceGrid/PredictorNode).

[Class] *RuleEngine* - the *rule engine* implementation with notable functions:

[Method] *activateRule(jointPointCode,ruleCode)* -for given decision point in algorithm activate rule given by *RuleCode* enumeration member, the next trigger of decision point will invoke rule behaviour.

[Method] *deactiveRule(jointPointCode,ruleCode)* - for given decision point in algorithm deactivate rule given by *RuleCode* enumeration member, the running instances of the rule will finish, the new instance will not be invoked.

[Method] *invoke(rullable,jointPoint)* - invoke rule engine on specific joint (decision) point, the rullable is reference to *RullableObject* interface implementation.

[Method] *invokeRule(context,jointPointCode,ruleCode)* - forced rule invocation, the standard conditions are checked, the context can be modified. This method is used in rule chaining.

[Class] *AbstractRule* - the interface class of rule implementation, notable functions:

[Method] *parseContext(*)* - the context of *RullableObject* is passed as a map, the internal structures of the rule needs to initialized.

[Method] *testCondition(*)* - the parsed context is checked to triggering conditions, if the conditions are met, the rule continues with *invokeRuleBody()*.

[Method] *invokeRuleBody(*)* - the business logic of the rule, context changes, calculations, etc.

[Class] *TestRule* - extends *AbstractRule*, the template for new rule implementation.

[Class] *RulePriorRulesOfAir* - extends *AbstractRule*, gathers and verifies directives in the form of active collision cases, prepares the list of *active collision cases*.

[Class] *RuleCollisionCaseResulution* - extends *AbstractRule*, process the list of active collision cases, based on *avoidance role* invokes specific situation resolution commands.

[Class] *RuleConvergingManeuver* - extends *AbstractRule*, enforces converging role on UAS, to avoid collision point.

[Class] *RuleHeadOnApproachManeuver* - extends *AbstractRule*,enforces head on avoidance, to avoid collision point.

`[Class]` *RuleOvertakeManevuer* - extends *AbstractRule*, forces overtaking UAS to follow divergence/convergence points.

`[Class]` *RulePostRulesOfAir* - extends *AbstractRule*, notifies the resolutions of active collision cases to UTM.

Miscellaneous: The additional modules of the framework supporting the basic functions, the modules are ordered by importance:

`[Module]` *Scenarios* - Implementation of all presented scenarios (tab. **??**) over UTM/Mission Control artifacts.

`[Module]` *Utilities* - common functionality package, logging, file exports, coordinate frame transformations, etc.

`[Module]` *Enumerations* - Aircraft categorizations, Airspace categorizations, Reach set types, Collision Case Resolution related, and Intruder related enumerations.

`[Module]` *Tests* - development test, functionality showcases, and scenarios prototypes (not just documented ones).

`[Module]` *Dijkstra* - graph representation of reach set implementation.

Bibliography

[1] Maria Cerna. Usage of maps obtained by lidar in uav navigation. Master thesis, Institute of Automotive Mechatronics, Faculty of Electrical Engineering and Information Technology, Slovak University of Technology, Ilkovicova 3, Bratislava. Slovak Republic, jun 2018.