A Smart Autoflight Control System Infrastructure

by

Stephan Heinemann

Dipl.-Inform., Technische Universität Berlin, Germany, 2004

Commercial Pilot, European Union Aviation Safety Agency, Germany, 2011

Flight Instructor, Transport Canada, 2015

Airline Transport Pilot, Transport Canada, 2020

A Dissertation Submitted in Partial Fulfillment of the
Requirements for the Degree of

Doctor of Philosophy

in the Department of Computer Science

© Stephan Heinemann, 2022
University of Victoria

All rights reserved. This dissertation may not be reproduced in whole or in part, by photocopying or other means, without the permission of the author.

We acknowledge and respect the lək̓ʷəŋən peoples on whose traditional territory the university stands and the Songhees, Esquimalt and W̱SÁNEĆ peoples whose historical relationships with the land continue to this day.

A Smart Autoflight Control System Infrastructure

by

Stephan Heinemann

Dipl.-Inform., Technische Universität Berlin, Germany, 2004

Commercial Pilot, European Union Aviation Safety Agency, Germany, 2011

Flight Instructor, Transport Canada, 2015

Airline Transport Pilot, Transport Canada, 2020

Supervisory Committee

Dr. Hausi A. Müller, Supervisor
(Department of Computer Science, University of Victoria)

Dr. Afzal Suleman, Supervisor
(Department of Mechanical Engineering, University of Victoria)

Dr. Alex Thomo, Departmental Member
(Department of Computer Science, University of Victoria)

Abstract

Connected aviation, the Internet of Flying Things and related emerging technologies, such as the System-Wide Information Management infrastructure of the FAA NextGen program, present numerous opportunities for the aviation sector. The ubiquity of aeronautical, flight, weather, aerodrome, and maintenance data accelerates the development of smarter software systems to cope with the ever increasing requirements of the industry sector. The increasing amount, frequency and variety of real-time data available to modern air transport and tactical systems, and their crews, creates exciting new challenges and research opportunities. We present an architectural approach toward the vision of increasingly self-separating and self-governed flight operations within the bigger picture of an evolving set of future Autonomous Flight Rules. The challenges in this field of research are manifold and include autonomic airborne trajectory optimization, data sharing, fusion and information derivation, the incorporation of and communication with rational actors—both human and machine—via a connected aviation infrastructure, to facilitate smarter decision making and support while generating economical, environmental and tactical advantages.

We developed a concept and prototype implementation of our Smart Autoflight Control System. The concept and implemented system follow the design principle of an Autonomic Element, consisting of an Autonomic Manager and its Managed Element, acting within an Autonomic Context. The Managed Element concept embraces an infrastructure featuring suitable models of manageable environments, airborne agents, planners, applicable operational cost and risk policies, and connections to the System-Wide Information Management cloud as well as to relevant rational actors, such as Air Traffic Control, Command and Control, Operations or Dispatch. The Autonomic Manager concept incorporates the extraction, that is, short-term sensing, of features from operational scenarios and the categorization of these scenarios according to their level of criticality and associated flight phase. The Autonomic Manager component, furthermore, continuously tunes, that is, actuates, manageable items of its Managed Element, such as environments and planners, and triggers competitions to assess their performance under the various extracted and dynamically changing features of their Autonomic Context. The performance reputations of the tuned manageable items are collected in a knowledge base and may serve as a long-term sensor. Both the managed items of the Managed Element as well the managing items of the Autonomic Manager

are extendable and may realize very different paradigms, including deterministic, non-deterministic, heuristically guided, and biologically inspired approaches.

We assessed the extensibility and maintainability of our Smart Autoflight Control System infrastructure by including manageable environments and planners of the Classical Grid Search, Probabilistic Roadmaps, and Rapidly-Exploring Random Trees families into its core component. Furthermore, we evaluated the viability of a simple heuristic and a more sophisticated Sequential Model-Based Algorithm Configuration Autonomic Manager to adaptively select and tune manageable planners of the supported families based on the extracted features from very simple to highly challenging scenarios. We were able to show that a self-adaptive approach, that heuristically tunes and selects the best performing planner following a performance competition, produces suitable flight trajectories within reasonable deliberation times. Additionally, we discovered options for improving our heuristic Autonomic Manager through a series of evaluation runs of the Sequential Model-Based Algorithm Configuration Autonomic Manager.

Our contributions answer how the manageable items, that is, environments and planners, of our Smart Autoflight Control System core component have to be modified in order to embed System-Wide Information Management data that feature both spatial and temporal aspects. We show how operational cost and risk policies help to assess environments differently and plan suitable flight trajectories accordingly. We identify and implement the necessary extensions and capabilities that have to be supported by manageable and managing items, respectively, to enable continuous feature extraction, adaptive tuning, performance competitions, and planner selection in dynamic flight scenarios.

Index terms— Autonomous Flight, Airborne Trajectory Optimization, Connected Aviation, System-Wide Information Management

Table of Contents

Supervisory Committee ii

Abstract iii

Table of Contents v

List of Tables xii

List of Figures xiv

List of Algorithms xx

Listings xxi

Acknowledgements xxiii

Dedication xxiv

I Introduction and Overview 1

1 Introduction 2

1.1 Motivation . . . 2
1.2 Statement of the Problem . . . 3
1.3 Purpose of the Study . . . 4
1.4 Research Questions . . . 7
1.5 Methods . . . 9
1.5.1 Research Design . . . 9
1.5.2 Limitations . . . 10
1.5.3 Evaluation . . . 10
1.6 Contributions . . . 10

1.7 Dissertation Outline 11

2 Background **12**
2.1 Planning and Decision Making 13
2.1.1 Artificial Intelligence Terminology 13
2.1.2 Motion Planning 14
2.1.3 Deterministic Heuristic-Based Planning 17
2.1.4 Probabilistic Sampling-Based Planning 29
2.1.5 Planning under Uncertainty 41
2.2 Autonomous Flight and Guidance 44
2.3 Self-Adaptive and Self-Managing Systems 49
2.4 Chapter Summary 53

3 State of the Art **54**
3.1 The SWIM Infrastructure 56
3.2 The NASA TASAR Project 58
3.2.1 Project History and Vision 60
3.2.2 Technical Aspects 61
3.2.3 Human Factors, Flight Trials and Operational Evaluation 64
3.2.4 Relationship between TASAR and the SAFCS 67
3.3 Automatic Selection and Tuning of Motion Planners 69
3.3.1 Sequential Model-Based Algorithm Configuration 70
3.3.2 Relevance to the SAFCS Autonomic Manager 73
3.4 Standards and Verification of Adaptive Systems 73
3.5 Chapter Summary 77

II Smart Autoflight Control System Infrastructure **79**

4 SAFCS — Overview **80**

5 SAFCS — Architecture **82**
5.1 SAFCS Components 82
5.2 SAFCS Core Architecture 83
5.2.1 Sessions, Scenarios, Factories and Specifications 84
5.2.2 Aircraft 88
5.2.3 Environments and Obstacles 90

5.2.4 Connections and Obstacle Reporting . . . 97
5.2.5 Planners . . . 100
5.3 SAFCS Management Architecture . . . 107
5.3.1 Managers . . . 109
5.3.2 Managed Planners . . . 110
5.3.3 Features and Tunings . . . 112
5.3.4 Performance and Knowledge Base . . . 113
5.4 SAFCS UI Architecture . . . 116
5.5 Chapter Summary . . . 119

6 SAFCS Core — Classical Grid Search 120
6.1 Cost Integration . . . 123
6.2 Basic Forward A* . . . 130
6.3 Anytime Repairing A* . . . 136
6.4 Anytime D* . . . 138
6.5 Online AD* . . . 153
6.6 Chapter Summary . . . 158

7 SAFCS Core — Rapidly-Exploring Random Trees 159
7.1 Cost Integration . . . 161
7.2 Basic RRT . . . 169
7.3 RRT* . . . 173
7.4 Heuristic RRT . . . 174
7.5 Dynamic RRT . . . 180
7.6 Anytime RRT . . . 186
7.7 Anytime Dynamic RRT . . . 194
7.8 Online ADRRT . . . 196
7.9 Chapter Summary . . . 201

8 SAFCS Core — Probabilistic Roadmaps 202

9 SAFCS — Autonomic Manager 203
9.1 Managed Planners . . . 203
9.2 Discriminating Features . . . 210
9.3 Tuning Parameters . . . 220
9.4 Knowledge Base . . . 223

9.5 Autonomic Managers . . . 226
9.6 Heuristic Autonomic Manager . . . 231
9.7 SMAC Autonomic Manager . . . 236
9.8 Chapter Summary . . . 244

10 SAFCS — User Interface 245
10.1 World . . . 246
10.2 Main Controls and Setup . . . 247
10.3 Time and Timer . . . 250
10.4 SWIM Data and Cost Threshold . . . 252
10.5 Scenarios . . . 253
10.6 Environment . . . 253
10.7 Plan . . . 254
10.8 Chapter Summary . . . 255

III Evaluation, Vehicle Integration and Experiments 256

11 Evaluation 257
11.1 Evaluation Criteria . . . 258
11.2 Evaluation Scenarios . . . 261
11.3 Cruise . . . 264
11.3.1 Cruise Scenario 1 . . . 265
11.3.2 Cruise Scenario 2 . . . 268
11.3.3 Cruise Scenario 3 . . . 271
11.3.4 Cruise Scenario 4 . . . 275
11.3.5 Cruise Scenario 5 . . . 278
11.4 Transition . . . 281
11.4.1 Transition Scenario 1 . . . 282
11.4.2 Transition Scenario 2 . . . 285
11.4.3 Transition Scenario 3 . . . 289
11.4.4 Transition Scenario 4 . . . 293
11.4.5 Transition Scenario 5 . . . 296
11.5 Terminal . . . 299
11.5.1 Terminal Scenario 1 . . . 300
11.5.2 Terminal Scenario 2 . . . 303

11.5.3 Terminal Scenario 3 . . . 306
11.5.4 Terminal Scenario 4 . . . 309
11.5.5 Terminal Scenario 5 . . . 312
11.6 Urgency . . . 315
11.6.1 Urgency Scenario 1 . . . 316
11.6.2 Urgency Scenario 2 . . . 319
11.6.3 Urgency Scenario 3 . . . 322
11.6.4 Urgency Scenario 4 . . . 325
11.6.5 Urgency Scenario 5 . . . 328
11.7 Emergency . . . 331
11.8 Interpretation . . . 331
11.9 Sequential Model-Based Algorithm Configuration . . . 333
11.10 Chapter Summary . . . 340

12 Vehicle Integration **341**
12.1 Hardware . . . 341
12.2 Software . . . 342

13 Experiments **346**
13.1 Flight Test Crew . . . 346
13.2 Flight Test Scenarios . . . 347
13.2.1 Setup and Limitations . . . 348
13.2.2 Datalink . . . 348
13.2.3 Take-Off, Landing, In-Flight Uplink and Returning . . . 349
13.2.4 Online Planning and Dynamic SWIM Obstacles . . . 350
13.3 Future Flight Tests . . . 352
13.3.1 Managed Planning and Dynamic SWIM Obstacles . . . 352
13.4 Lessons Learned . . . 352

IV Conclusions **355**

14 Summary **356**
14.1 Research Questions and Answers . . . 356
14.2 Challenges and Solutions . . . 361
14.3 Contributions and Significance . . . 364

14.4 Chapter Summary 366

15 Outlook and Future Work **367**

15.1 Outlook 367
15.2 Future Work 367
15.2.1 Environment Base Cost and Favorable Regions 368
15.2.2 Environment Tuning 368
15.2.3 Environment Elevation Models 369
15.2.4 Environment Air Data Intervals 369
15.2.5 Dynamic Aircraft Capabilities 370
15.2.6 Aircraft Safety Space and Intersections 370
15.2.7 Cost and Risk Policies 371
15.2.8 Planning Continuum: Shape, Resolution and Hierarchy 372
15.2.9 Planning Roadmap: IFR/VFR Air-Routes and Airways 372
15.2.10 Flexible Waypoint Slots 372
15.2.11 Significant Change Criteria and Policies 374
15.2.12 Adaptive Deflation 374
15.2.13 Priority Queue Pruning 375
15.2.14 Backups versus Planner Instances 375
15.2.15 Sampling Attempts and Sampling Sub-Space 376
15.2.16 Nearest Neighbors 376
15.2.17 In-Flight Alternates and Precautionary Landings 376
15.2.18 Core Planners 377
15.2.19 Features Horizon Model 378
15.2.20 Autonomic Managers 378
15.2.21 Aircraft Sensors and SWIM Publication 379
15.2.22 Distributed Multi-Agent SAFCS 379
15.2.23 Live and Simulated Connections 379
15.2.24 Physics Engine and Models 380
15.2.25 Statistics and Knowledge Base Inspection 380
15.3 Chapter Summary 380

References **382**

V Appendices **394**

A Glossary **395**

A.1 Acronyms 395

B Project Statistics **422**

B.1 Project Statistics 422

B.1.1 com.uvic-cfar.swim:worldwind:1.0.0-alpha4-SNAPSHOT 422

B.1.2 com.uvic-cfar.swim:worldwind-ui:1.0.0-alpha3-SNAPSHOT . . . 426

B.1.3 com.uvic-cfar.swim:worlwdind-jaxb:1.0.0-alpha1-SNAPSHOT . . 426

List of Tables

Table 1.1 Variables of the Proposed Research 6

Table 3.1 SWIM Data Information Exchange Models 57
Table 3.2 Applicable RTCA Standards 74
Table 3.3 DO-178C Item Development Assurance Level 75
Table 3.4 Statistical Techniques and Applications 75

Table 5.1 Operational Policies 104

Table 9.1 Difficulty 217
Table 9.2 Severity 217
Table 9.3 Flight Phase 218
Table 9.4 Tuning Parameters 222
Table 9.5 Performance Context 224

Table 11.1 Obstacle Cost Color Map 262
Table 11.2 Cruise 1 Feature Categories 266
Table 11.3 Cruise 1 Heuristic Tunings 266
Table 11.4 Cruise 2 Feature Categories 269
Table 11.5 Cruise 2 Heuristic Tunings 269
Table 11.6 Cruise 3 Feature Categories 273
Table 11.7 Cruise 3 Heuristic Tunings 273
Table 11.8 Cruise 4 Feature Categories 276
Table 11.9 Cruise 4 Heuristic Tunings 276
Table 11.10 Cruise 5 Feature Categories 279
Table 11.11 Cruise 5 Heuristic Tunings 279
Table 11.12 Transition 1 Feature Categories 283
Table 11.13 Transition 1 Heuristic Tunings 283
Table 11.14 Transition 2 Feature Categories 287
Table 11.15 Transition 2 Heuristic Tunings 287

Table 11.16 Transition 3 Feature Categories 291
Table 11.17 Transition 3 Heuristic Tunings 291
Table 11.18 Transition 4 Feature Categories 294
Table 11.19 Transition 4 Heuristic Tunings 294
Table 11.20 Transition 5 Feature Categories 297
Table 11.21 Transition 5 Heuristic Tunings 297
Table 11.22 Terminal 1 Feature Categories 301
Table 11.23 Terminal 1 Heuristic Tunings 301
Table 11.24 Terminal 2 Feature Categories 304
Table 11.25 Terminal 2 Heuristic Tunings 304
Table 11.26 Terminal 3 Feature Categories 307
Table 11.27 Terminal 3 Heuristic Tunings 307
Table 11.28 Terminal 4 Feature Categories 310
Table 11.29 Terminal 4 Heuristic Tunings 310
Table 11.30 Terminal 5 Feature Categories 313
Table 11.31 Terminal 5 Heuristic Tunings 313
Table 11.32 Urgency 1 Feature Categories 317
Table 11.33 Urgency 1 Heuristic Tunings 317
Table 11.34 Urgency 2 Feature Categories 320
Table 11.35 Urgency 2 Heuristic Tunings 320
Table 11.36 Urgency 3 Feature Categories 323
Table 11.37 Urgency 3 Heuristic Tunings 323
Table 11.38 Urgency 4 Feature Categories 326
Table 11.39 Urgency 4 Heuristic Tunings 326
Table 11.40 Urgency 5 Feature Categories 329
Table 11.41 Urgency 5 Heuristic Tunings 329
Table 11.42 Cruise MTP SMAC Tunings 336
Table 11.43 Transition MTP SMAC Tunings 336
Table 11.44 Terminal MTP SMAC Tunings 337
Table 11.45 Urgency MTP SMAC Tunings 337

List of Figures

Figure 1.1 Environment Information Layers 5

Figure 2.1 Field D* Interpolation 23
Figure 2.2 Multi-Resolution Field D* 25
Figure 2.3 Θ* String Pulling 27
Figure 2.4 AA* Geometric Ideas 28
Figure 2.5 Probabilistic Roadmap Examples 30
Figure 2.6 State-Time Probe 31
Figure 2.7 Rapidly-Exploring Random Tree Growth 34
Figure 2.8 Dynamic RRT Examples 36
Figure 2.9 MDP Example 41
Figure 2.10 POMDP Example 42
Figure 2.11 Evolutionary (Genetic) Path Planning 46
Figure 2.12 Adaptation Loop 50
Figure 2.13 Self-Adaptiveness 50
Figure 2.14 Techniques for Self-Adaptation 52

Figure 3.1 Smart Autoflight Control System Actors 54
Figure 3.2 Present Information Management 56
Figure 3.3 System-Wide Information Management 58
Figure 3.4 The TASAR Concept 59
Figure 3.5 The TAP Prototype 62
Figure 3.6 The TAP Operational Modes 63
Figure 3.7 The TAP UI Time Slider 66
Figure 3.8 SMAC: Fitting a Regression Tree 71
Figure 3.9 SMAC(PP): Fitting a Regression Tree 72

Figure 4.1 Smart Autoflight Control System UI 80
Figure 4.2 Smart Autoflight Control System Overview 81

Figure 5.1 SAFCS Core: Sessions, Scenarios, Factories and Specifications . . . 84
Figure 5.2 SAFCS Core: Specifications and Properties . . . 85
Figure 5.3 SAFCS UI: Specification Setup . . . 86
Figure 5.4 SAFCS UI: Scenarios . . . 87
Figure 5.5 SAFCS UI: Scenario Time Controller . . . 87
Figure 5.6 SAFCS Core: Aircraft . . . 89
Figure 5.7 SAFCS UI: Aircraft Symbol . . . 89
Figure 5.8 SAFCS Core: Environments . . . 90
Figure 5.9 SAFCS Core: Planning Grid . . . 91
Figure 5.10 SAFCS Core: Planning Continuum . . . 93
Figure 5.11 SAFCS Core: Obstacles . . . 93
Figure 5.12 SAFCS Core: Cost Intervals . . . 95
Figure 5.13 SAFCS UI: Grid Obstacles . . . 96
Figure 5.14 SAFCS UI: Continuum Obstacles . . . 97
Figure 5.15 SAFCS Core: Connections . . . 98
Figure 5.16 SAFCS Core: Obstacle Reporting . . . 99
Figure 5.17 SAFCS UI: Connections Setup . . . 100
Figure 5.18 SAFCS Core: Planners . . . 101
Figure 5.19 SAFCS Core: OPS Risk and Cost Polcies (1) . . . 102
Figure 5.20 SAFCS Core: OPS Risk and Cost Polcies (2) . . . 103
Figure 5.21 SAFCS Core: Online Planner . . . 105
Figure 5.22 SAFCS Core: Waypoints . . . 105
Figure 5.23 SAFCS UI: Datalink Tracking . . . 107
Figure 5.24 SAFCS Management: Autonomic Manager Interfaces . . . 108
Figure 5.25 SAFCS Management: Autonomic Managers . . . 109
Figure 5.26 SAFCS Management: Managed Planners . . . 111
Figure 5.27 SAFCS UI: Managed Planning Session . . . 111
Figure 5.28 SAFCS Management: Features . . . 112
Figure 5.29 SAFCS Management: Tunings . . . 113
Figure 5.30 SAFCS Management: Performance . . . 114
Figure 5.31 SAFCS Management: Knowledge Base . . . 115
Figure 5.32 SAFCS UI: Manager Setup . . . 116
Figure 5.33 SAFCS UI: Model-View-Presenter . . . 116
Figure 5.34 SAFCS UI: Packages . . . 117
Figure 5.35 SAFCS UI: World Package . . . 118

Figure 6.1 SAFCS Core: A* Family 122
Figure 6.2 SAFCS Core: A* Waypoints 123
Figure 6.3 SAFCS UI: Planning Grid Obstacles and Waypoints 124
Figure 6.4 SAFCS Core: Planning Grid Cost Interval Example 125
Figure 6.5 SAFCS Core: A* Waypoint Signatures 126
Figure 6.6 SAFCS UI: Forward A* 135
Figure 6.7 SAFCS UI: Θ* 136
Figure 6.8 SAFCS Core: AD* Class Signatures 139
Figure 6.9 SAFCS Core: AD* Single-Part Repair 144
Figure 6.10 SAFCS Core: AD* Multi-Part Repair 147
Figure 6.11 SAFCS Core: OAD* Class Signatures 153

Figure 7.1 SAFCS Core: RRT Family 160
Figure 7.2 SAFCS Core: RRT Waypoints 161
Figure 7.3 SAFCS Core: Edges and Cost Intervals 162
Figure 7.4 SAFCS UI: Planning Continuum Obstacles and Waypoints 163
Figure 7.5 SAFCS Core: Edges Cost Interval Example 164
Figure 7.6 SAFCS Core: RRT Waypoint Signatures 165
Figure 7.7 SAFCS Core: Basic RRT 169
Figure 7.8 SAFCS UI: RRT Sampling 170
Figure 7.9 SAFCS UI: RRT Connection Strategy 171
Figure 7.10 SAFCS Core: Heuristic RRT 175
Figure 7.11 SAFCS Core: Heuristic RRT Quality 176
Figure 7.12 SAFCS UI: Heuristic RRT 179
Figure 7.13 SAFCS UI: Heuristic RRT Planner Setup 180
Figure 7.14 SAFCS UI: DRRT Trimming 184
Figure 7.15 SAFCS Core: Anytime RRT 186
Figure 7.16 SAFCS UI: Anytime RRT 193
Figure 7.17 SAFCS Core: OADRRT Class Signatures 197

Figure 9.1 SAFCS AM: Managed Planners Signature 204
Figure 9.2 SAFCS AM: 4D Features 212
Figure 9.3 SAFCS AM: Features Horizon 213
Figure 9.4 SAFCS AM: Criticality 215
Figure 9.5 SAFCS UI: Features Instance 220
Figure 9.6 SAFCS AM: Performance Signature 223

Figure 9.7 SAFCS AM: Performance Plot . . . 224
Figure 9.8 SAFCS AM: Reputation Signature . . . 225
Figure 9.9 SAFCS AM: Abstract Autonomic Manager Signature . . . 227
Figure 9.10 SAFCS AM: Abstract Autonomic Manager Life Cycle . . . 227
Figure 9.11 SAFCS AM: Heuristic Autonomic Manager Tuning . . . 232
Figure 9.12 SAFCS AM: SMAC Autonomic Manager Tuning . . . 237

Figure 10.1 SAFCS UI: Views . . . 245
Figure 10.2 SAFCS UI: World . . . 247
Figure 10.3 SAFCS UI: Main Controls . . . 248
Figure 10.4 SAFCS UI: Activity . . . 250
Figure 10.5 SAFCS UI: Time and Timer . . . 251
Figure 10.6 SAFCS UI: SWIM Data and Cost Threshold . . . 252
Figure 10.7 SAFCS UI: Scenarios . . . 253
Figure 10.8 SAFCS UI: Environment . . . 253
Figure 10.9 SAFCS UI: Plan . . . 254

Figure 11.1 UVic CfAR Flight Test Area . . . 261
Figure 11.2 Managed Cruise Scenario . . . 264
Figure 11.3 Cruise 1 Scenario . . . 266
Figure 11.4 H-AM Quality and Performance: Cruise 1 . . . 267
Figure 11.5 Cruise 2 Scenario . . . 269
Figure 11.6 H-AM Quality and Performance: Cruise 2 . . . 270
Figure 11.7 Cruise 3 Scenario . . . 273
Figure 11.8 H-AM Quality and Performance: Cruise 3 . . . 274
Figure 11.9 Cruise 4 Scenario . . . 276
Figure 11.10 H-AM Quality and Performance: Cruise 4 . . . 277
Figure 11.11 Cruise 5 Scenario . . . 279
Figure 11.12 H-AM Quality and Performance: Cruise 5 . . . 280
Figure 11.13 Managed Transition Scenario . . . 281
Figure 11.14 Transition 1 Scenario . . . 283
Figure 11.15 H-AM Quality and Performance: Transition 1 . . . 284
Figure 11.16 Transition 2 Scenario . . . 287
Figure 11.17 H-AM Quality and Performance: Transition 2 . . . 288
Figure 11.18 Transition 3 Scenario . . . 291
Figure 11.19 H-AM Quality and Performance: Transition 3 . . . 292

Figure 11.20 Transition 4 Scenario . . . 294
Figure 11.21 H-AM Quality and Performance: Transition 4 . . . 295
Figure 11.22 Transition 5 Scenario . . . 297
Figure 11.23 H-AM Quality and Performance: Transition 5 . . . 298
Figure 11.24 Managed Terminal Scenario . . . 299
Figure 11.25 Terminal 1 Scenario . . . 301
Figure 11.26 H-AM Quality and Performance: Terminal 1 . . . 302
Figure 11.27 Terminal 2 Scenario . . . 304
Figure 11.28 H-AM Quality and Performance: Terminal 2 . . . 305
Figure 11.29 Terminal 3 Scenario . . . 307
Figure 11.30 H-AM Quality and Performance: Terminal 3 . . . 308
Figure 11.31 Terminal 4 Scenario . . . 310
Figure 11.32 H-AM Quality and Performance: Terminal 4 . . . 311
Figure 11.33 Terminal 5 Scenario . . . 313
Figure 11.34 H-AM Quality and Performance: Terminal 5 . . . 314
Figure 11.35 Managed Urgency Scenario . . . 315
Figure 11.36 Urgency 1 Scenario . . . 317
Figure 11.37 H-AM Quality and Performance: Urgency 1 . . . 318
Figure 11.38 Urgency 2 Scenario . . . 320
Figure 11.39 H-AM Quality and Performance: Urgency 2 . . . 321
Figure 11.40 Urgency 3 Scenario . . . 323
Figure 11.41 H-AM Quality and Performance: Urgency 3 . . . 324
Figure 11.42 Urgency 4 Scenario . . . 326
Figure 11.43 H-AM Quality and Performance: Urgency 4 . . . 327
Figure 11.44 Urgency 5 Scenario . . . 329
Figure 11.45 H-AM Quality and Performance: Urgency 5 . . . 330
Figure 11.46 SMAC-AM Quality and Performance: Terminal 3 . . . 339

Figure 12.1 Tarot UAV and Pixhawk 4 Autopilot . . . 341
Figure 12.2 UAS Illustration . . . 343
Figure 12.3 UAS Communication . . . 343
Figure 12.4 Mavlink Datalink . . . 344
Figure 12.5 Mavlink Datalink Setup . . . 345

Figure 13.1 Flight Test Crew . . . 347
Figure 13.2 Flight Scenario and Aircraft . . . 348

Figure 13.3 Basic Flight Tests 350
Figure 13.4 Online SWIM Flight 351
Figure 13.5 Augmented Reality 353

List of Algorithms

1 Basic Forward A* 18
2 Backward D* Lite 20
3 Backward AD* 21
4 Backward AD* Main 22
5 Field D* 24
6 Lazy Θ* 26
7 Probabilistic Roadmap 30
8 Rapidly-Exploring Random Tree and Extended RRT 35
9 Dynamic RRT 38
10 Anytime RRT 39
11 Anytime Dynamic RRT 40
12 Sequential Model-Based Optimization 70
13 AD* Key Function 141

Listings

6.1 SAFCS Core: A* Cost Computation . . . 127
6.2 SAFCS Core: Planning Grid Step Cost Computation . . . 129
6.3 SAFCS Core: A* Expansion . . . 131
6.4 SAFCS Core: A* Computation . . . 132
6.5 SAFCS Core: A* Priority Order . . . 134
6.6 SAFCS Core: AD* Consistency . . . 140
6.7 SAFCS Core: AD* Priority Order . . . 142
6.8 SAFCS Core: AD* Computation . . . 143
6.9 SAFCS Core: AD* Waypoint Repair . . . 145
6.10 SAFCS Core: AD* Elaboration . . . 146
6.11 SAFCS Core: AD* Repair . . . 149
6.12 SAFCS Core: AD* Repair — Affected Waypoints . . . 150
6.13 SAFCS Core: AD* Improvement . . . 152
6.14 SAFCS Core: OAD* Take-Off Conditions . . . 155
6.15 SAFCS Core: OAD* Progress Monitoring . . . 156
7.1 SAFCS Core: RRT Cost Computation . . . 166
7.2 SAFCS Core: Planning Continuum Step Cost Computation . . . 166
7.3 SAFCS Core: Edge Cost Computation . . . 168
7.4 SAFCS Core: Basic RRT Computation . . . 172
7.5 SAFCS Core: HRRT hRRT Sample Selection . . . 177
7.6 SAFCS Core: HRRT IkRRT Sample Selection . . . 178
7.7 SAFCS Core: HRRT BkRRT Sample Selection . . . 178
7.8 SAFCS Core: DRRT Elaboration . . . 182
7.9 SAFCS Core: DRRT Trimming . . . 183
7.10 SAFCS Core: DRRT Quality Update . . . 185
7.11 SAFCS Core: ARRT Elaboration . . . 187
7.12 SAFCS Core: ARRT Heuristics . . . 189

7.13 SAFCS Core: ARRT Cost Bias and Bound 190
7.14 SAFCS Core: ARRT Sampling 191
7.15 SAFCS Core: ARRT Extension 192
7.16 SAFCS Core: ADRRT Elaboration 195
7.17 SAFCS Core: ADRRT Repair 195
7.18 SAFCS Core: ADRRT Significant Change 196
7.19 SAFCS Core: OADRRT Rooting 198
7.20 SAFCS Core: OADRRT Elaboration 200
9.1 SAFCS AM: Invoking Managed Planners 205
9.2 SAFCS AM: Active and Standby Managed Planners 207
9.3 SAFCS AM: Joining Managed Planners 209
9.4 SAFCS AM: Features 211
9.5 SAFCS AM: Features Instance 219
9.6 SAFCS AM: Performance Evaluation 229
9.7 SAFCS AM: Tuning and Competition Joining 230
9.8 SAFCS AM: Obstacle-Free Heuristic Tuning 233
9.9 SAFCS AM: Obstacle-Cluttered Heuristic Tuning 235
9.10 SAFCS AM: MGP SMAC Parameter Configuration Space 238
9.11 SAFCS AM: MTP SMAC Parameter Configuration Space 238
9.12 SAFCS AM: SMAC Training 240
9.13 SAFCS AM: SMAC Tuning 243
11.1 SAFCS AM: MGP SMAC Parameter Configuration Space 334
11.2 SAFCS AM: MTP SMAC Parameter Configuration Space 334

Acknowledgements

This dissertation would not have been possible without the continuous support and enduring patience of many others along the way. First and foremost, I would like to thank my supervisors Dr. Hausi A. Müller and Dr. Afzal Suleman for their encouragement and valuable guidance over the past years. Their Rigi Research and Center for Aerospace Research labs at the University of Victoria and the Victoria International Airport provided me with the necessary equipment to accomplish my research goals and surrounded me with fellow computer scientists and mechanical engineers to discuss and reflect on many ideas. Many of these individuals have become good friends, although over time we may have spread out in all directions. In particular, I would like to thank Manuel and Henrique for their contributions to the core planners of our Smart Autoflight Control System as well as Charlie who implemented an early version of one of its datalink connections. I would also like to thank the remaining members of my committee, including Dr. Alex Thomo, Dr. Hugh Liu, and Dr. Yang Shi, for their time and effort, particularly considering the larger than originally anticipated size of this document. Furthermore, I would like to highlight and appreciate the continuous support of the administrative staff at the Department of Computer Science of the University of Victoria: Kath Milinazzo, Nancy Chan, Wendy Beggs, and Erin Robinson.

My family, who always endorsed me in my choices and way of life, deserves special credit. I am very grateful to have received their unconditional love and support in all my endeavors. Most importantly, I would like to thank my wife Stefanie for her love, patience and relaxing walks with Tobi. What primarily served as a welcome distraction and mental cool down as part of our daily routine, often led to new ideas and solutions to challenging problems. Thank you so much!

Dedication

To my grandmother Elisabeth. I am sure, you would have been very proud.

Part I

Introduction and Overview

Chapter 1

Introduction

1.1 Motivation

The aviation sector has constantly faced and coped with many challenges during its entire history. One of the primary goals has always been to ensure safety while exploring the opportunities of emerging technologies. As a result, trusted redundancies are maintained while shifting carefully toward new terrain. One of the most disruptive challenges ahead lies in the integration and exploitation of the ubiquitous data, which is being made available to aviation stakeholders via the Federal Aviation Administration (FAA) Next Generation Air Transportation System (NextGen) and System-Wide Information Management (SWIM) program. The SWIM program aims at substituting the present proprietary and incompatible data exchange between aviation stakeholders with an open service-oriented architecture. The standardized machine-readable data schemas of this architecture enable users to publish and subscribe to aviation data on demand. Aviation data comprises aeronautical, flight, weather, aerodrome, and maintenance messages featuring a higher frequency, diversity, and recency compared to everything available before. The exploitation of this big data is not manageable by human actors alone and requires automatic filtering, weighing, and fusing to synthesize useful information. The derived information may consecutively be used to support rational actors—both human and machine—to make better decisions.

Our evolving Smart Autoflight Control System (SAFCS) infrastructure presented in this dissertation aims to support operators and crews of both manned and unmanned aircraft in making better decisions regarding their flight trajectories (online, in a dynamically changing environment, and with limited deliberation time). Today's air crews have

already become system managers, and their role is going to shift further toward one of a mission supervisor acknowledging decision options and monitoring the progress of an airborne mission. At the same time, other human actors currently involved in Air Traffic Control (ATC), Command and Control, Operations or Dispatch (OPS) need to be kept in the loop and consulted in the decision-making process. ATC is responsible for granting or denying access to controlled airspace as well as allowing modifications to an established trajectory. Mission goals and the acceptable risk policies, on the other hand, need to be defined by OPS. Therefore, the SAFCS concept should accommodate the interaction with these actors, possibly using the existing communication infrastructure, such as or similar to the Controller-Pilot Data Link Communications (CPDLC) and Aircraft Communications Addressing and Reporting System (ACARS) protocols, to implement a paradigm of *communication as sensing*. Finally, the level of autonomy and the resulting human-machine interaction should ideally be situation-aware and avoid both overload and complacency being inflicted on human actors.

1.2 Statement of the Problem

Human performance and sensory limitations continue to impose significant challenges for airborne operations. Advanced flight control systems and the automation of cockpit crew duties in modern transport and military aircraft have addressed safer and more efficient operations. However, the full potential of automatic flight in future generations of aircraft is far from being leveraged. The increasing amount, frequency and variety of real-time data available from the aircraft and its environment is not fully exploited and lacks careful filtering and processing. Relevant data includes aircraft systems and capabilities (e.g., performance and failures), the air environment (e.g., airspace, weather and traffic), the ground environment (e.g., terrain, population and facilities), and operational aspects (e.g., human factors, company goals and policies). Depending on the current flight phase (e.g., take-off, departure, cruise, arrival, approach or landing) as well as the criticality of the situation (e.g., diversion or emergency), different degrees of automation and crew interaction are required to establish an appropriate stress level avoiding both overload and complacency. Hence, there is a huge potential for smarter Automatic Flight Control Systems (AFCSs) that continuously filter, fuse and process available sensor data in order to establish an adequate Situational Awareness (SA), take decisions autonomously, and deliberately suggest options for Pilot Decision Making (PDM).

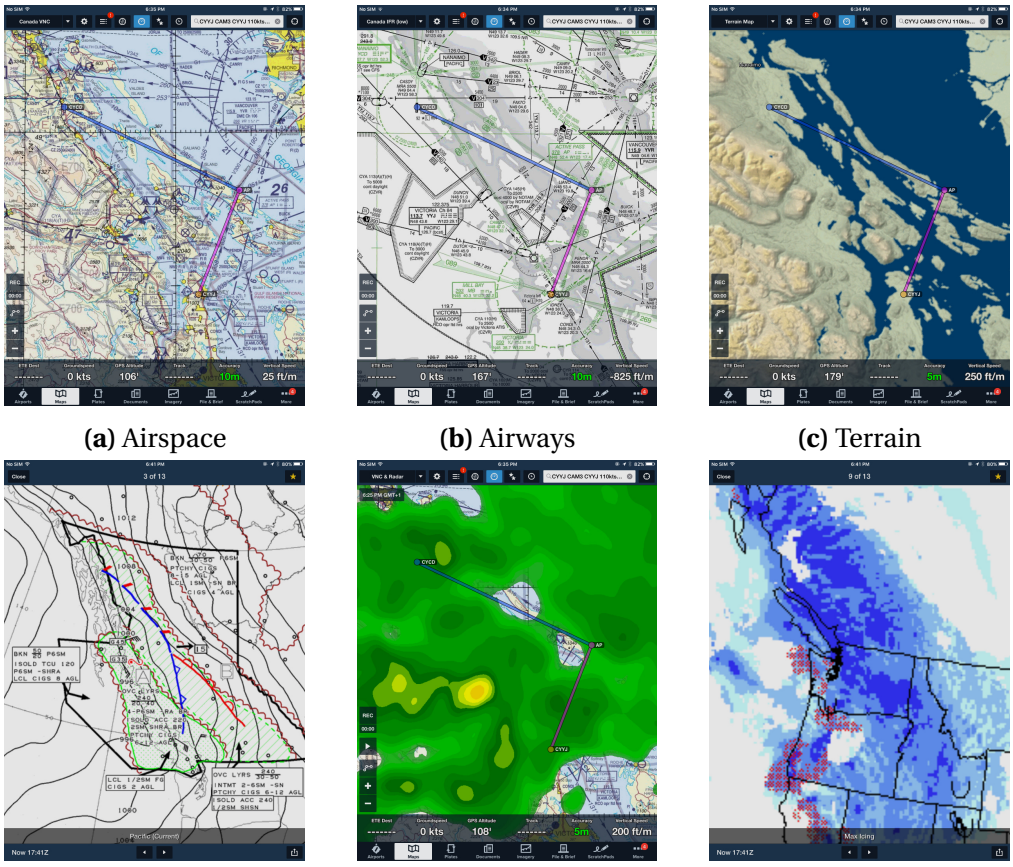
1.3 Purpose of the Study

Pilot Decision Making is constrained by the amount of data that can be taken into account and the time available to deduce reasonable decisions. This *bounded reactivity* and *rationality* also applies to AFCSs that aim at suggesting and performing decisions in order to achieve goals and act according to specified policies. Hence, the objective of our presented research is the identification and definition of suitable runtime models that fuse and scale available data, account for changing uncertainty, and are amenable to algorithms that can be interrupted at any given time. The performed computations enable autonomous actions and interactive decision support for online flight trajectory planning and aircraft systems configuration while balancing risk and reward. The degree of automation and interaction shall ideally depend on the assessment of the criticality of the situation.

As discussed by Heinemann, Müller and Suleman [HMS14], the relevant context required to establish an adequate SA for decision making consists of the aircraft *environment* (including operational constraints and higher level goals) and its *capabilities* (including its radius of action). Figure 1.1 shows some *Foreflight*[1] examples of different layers of information that apply to the same environment and would all have to be considered for decision support and smart online flight trajectory planning.

Airspaces can be considered as dynamic obstacles that change due to granted or denied clearances, if required. Existing protocols, such as CPDLC, allow for the standardized exchange of ATC messages and could be exploited by automatic systems. Airways represent an existing roadmap that is primarily used by Instrument Flight Rules (IFR) traffic and associated with standard rules regarding altitude assignments. Terrain awareness is necessary to ensure compliance with vertical clearance requirements from any ground obstacles. Weather significantly affects flight planning and many factors, such as temperature, pressure, humidity and the resulting obscuration, precipitation and wind, need to be taken into account. Many more detailed data, usually consisting of observations and predictions obtained *before* the flight,[2] include Aviation Routine Weather Reports (METARs), Terminal Aerodrome Forecasts (TAFs), Winds and Temperatures Aloft Forecasts (FDs), Pilot Reports (PIREPs), Airmen's Meteorological Informations (AIRMETs), Significant Meteorological Informations (SIGMETs) and Notice to Airmen (NOTAM) which all aim at establishing an *initial* SA. The applicable data formats

[1] https://www.foreflight.com/
[2] https://flightplanning.navcanada.ca



(a) Airspace **(b)** Airways **(c)** Terrain

(d) GFA **(e)** Radar **(f)** Icing

Figure 1.1: Environment Information Layers

are often standardized and can be processed electronically (e.g., in mobile apps). ICAO Meteorological Information Exchange Model (IWXXM) is one of these recently developed formats for the exchange of weather-related data.

Aircraft capabilities include the available fuel (or battery life in case of a small Unmanned Aerial Vehicle (UAV)), aerodynamic and powerplant configuration (e.g., flight for *range* or *endurance*), status of on-board sensors and systems (e.g., failures). Environment and capabilities limit the *radius of action* available for decision making. This radius of action applies to both space (e.g., distance) and (deliberation) time, and determines the criticality of the current situation.

Applying appropriately selected and tuned algorithms as well as adapted human machine interactions for decision making regarding trajectory planning options leads to a self-adaptive Smart Autoflight Control System which is the purpose of this study. The system has to fuse and weigh the available data, find options that optimize *risk* and *re-*

ward under given company policies or operational procedures (constraints), and finally implement solutions according to the available radius of action.

Table 1.1: Variables of the Proposed Research

Category	Variable	Description
Independent	Environment	The modelled environmental aspects, including airspace, traffic, weather, terrain, facilities, and operational procedures
	Capabilities	The modelled aircraft capabilities, including system aspects
Mediating	Algorithms	The algorithms that are available for selection and tuning
	Tuning Parameters	The tuning parameters of each algorithm
	Selection Policy	The scoring and selection policy for algorithms
Dependent	Computational Costs	The computational costs of a solution
	Solution Quality	The number and quality of solutions, including trajectory costs and heading/altitude changes in free space
	Interaction Level	The level of interaction with humans and rational agents

The *independent* variables of the study include all environment and capabilities factors that are modelled and affect the planning and decision making process. The applied algorithms, their tuning parameters, and the self-adaptive selection and tuning policy represent the *mediating* variables. The *dependent* and measurable variables are the required computational costs, the options and quality of computed solutions and, optionally, the interaction level with crew and ATC. These variables form the foundation of an *explanatory sequential mixed* research design with an emphasis on *quantitative* methods (Creswell [Cre13]), as further detailed in Section 1.5. The mentioned variables are summarized in Table 1.1.

The long-term goal of this research, beyond the scope of this dissertation, is to investigate and create systems that enable safer and more efficient air transport solutions. These systems should be able to adapt to the current situation with a more predictable performance than humans are capable of.

The short- and mid-term goals are to design, implement and evaluate the performance of a self-adaptive Smart Autoflight Control System that incorporates available SWIM data sets, generates flight trajectories and interacts with OPS and, potentially, crew and ATC (both simulated). Furthermore, algorithms and tuning parameters shall be identified which possess the best performance (in terms of dependent variables) in typical flight scenarios.

1.4 Research Questions

This section explains the main questions that drive the proposed research, including quantitative, qualitative and design questions to be answered.

> *RQ1.* Does a self-adaptive approach to decision making and trajectory planning improve the overall performance compared to a single (best on average) self-static approach using the quantifiable dependent variables mentioned in Section 1.3?

Self-adaptation will be realized in the form of selecting as well as tuning algorithms. Relevant tuning parameters are mentioned in Chapter 2 and may include changes in weights and strategies that affect the behavior of an algorithm. Optionally, adaptation can also embrace the situation-aware simulated interaction and communication with OPS, crews and ATC. If the radius of action and, hence, the deliberation time available is limited, then less interaction, fewer options or the direct implementation of a found solution may be required. This is in contrast with situations that allow for more interaction in order to achieve a more customized behavior and performance.

> *RQ2.* How do existing any-angle, anytime, dynamic, online algorithms have to be extended or modified to be included in a Smart Autoflight Control System?

This design question is a result of the initial literature survey on the topic. Many suitable approaches have been identified. However, some very promising algorithms may have to be modified to integrate them into a Smart Autoflight Control System. The applicable algorithm families shall be studied in detail and modifications or limitations, if any, shall be described.

> *RQ3.* Which algorithms and tuning parameters are suited best for particular flight situations?

This open-ended question is the *explanatory* qualitative extension to the first quantitative question. Different flight situations, such as departure, climb, cruise, arrival and approach, may, due to their inherent properties, favor a particular type of algorithm and tuning parameters.

RQ4. How can temporal SWIM data be integrated into a Smart Autoflight Control System to represent the associated costs in time while considering operational constraints?

The SWIM infrastructure enables the publication and subscription of aviation data with a temporality concept. Therefore, data sets usually feature both a space and time (duration) component. A Smart Autoflight Control System and its employed algorithms, therefore, need to be amenable to deal with associated costs in space and time. Furthermore, different air operators assess the same costs differently. For example, civil and military operations differ significantly in their risk assessment and tolerance. Therefore, details about cost integration and risk assessment need to be precisely defined.

RQ5. How could the existing communication infrastructure, such as CPDLC and ACARS, be integrated into a Smart Autoflight Control System design as an effective means to realize a paradigm of *communication as sensing* and allow a rational agent to interact with OPS and ATC to obtain policies, instructions or clearances and, hence, more options for automated planning?

CPDLC enables the exchange of standardized ATC messages that could be exploited by a Smart Autoflight Control System and, similar to ACARS communications with OPS, address the *human in the loop* concept of self-adaptive systems. Therefore, interaction is not limited to providing decision options to crew members but extends to actively seeking information (e.g., to confirm a belief state) from the operational management or clearances from air traffic controllers. As with air crew, the interaction level ideally needs to be adapted to human performance and limitations as well as the criticality of the current situation. Furthermore, obtaining clearances is associated with uncertainty and clear preferences which could be reflected in appropriate models, as discussed in Section 2.1.5. Our goal within the scope of this dissertation shall, however, be limited to be able to accommodate the existing infrastructure in the future and support the optional integration of plugins to simulate our *communication as sensing* paradigm for future research.

RQ6. What architectural design is suitable to support the dynamic extension and evolution of the Smart Autoflight Control System?

This qualitative design question aims at finding the most suitable architecture to support the registration (evolution) and tuning of available trajectory planning algorithms as well as the infrastructure to obtain real-time data required for decision making and support. A Monitor, Analyze, Plan, Execute - Knowledge (MAPE-K) architecture, as explained in Section 2.3, might be appropriate. Whether or not algorithms are deployed on-board or within an Autonomic Element (AE) in a cloud-based distributed infrastructure, is another important design decision to make; each option has its own advantages and disadvantages.

1.5 Methods

This section describes the applied design for the proposed research as well as its limitations.

1.5.1 Research Design

The presented research follows an *explanatory sequential mixed* design with a strong initial focus on *quantitative* methods (Creswell [Cre13]). The quantifiable variables of the study have been explained in Section 1.3. First, the developed SAFCS, that is, our implemented instrumentation, shall be evaluated in terms of quality and performance of the supported algorithms for selected representative scenarios. This initial step shall be followed by Software in the Loop (SIL) simulations with a selected flight model in order to prepare for the vehicle integration (Hardware in the Loop (HIL)) that eventually leads to actual flight tests. Finally, *experiments* shall be performed using the corresponding real aircraft to increase the confidence in the viability of our approach.

Each flight test shall cover a selected previously evaluated scenario suitable for the flight test area and instrumented Unmanned Aircraft System (UAS) (independent variables). Dynamic events will then be introduced to increase or relieve the stress on the system and to test its adaptability with respect to the employed algorithm selection and tuning. Using the supported self-adaptive modes (mediating variables), data will be collected that reflect the computational performance, the solution quality and, optionally, the simulated interaction level (dependent variables) with OPS and ATC.

For both initial evaluation and performed flight tests, the collected data shall be interpreted to give a rationale on why particular algorithms may have behaved better or

worse than others in given situations. The obtained results shall help to answer some of our research questions and may, in turn, be useful to improve our developed system.

1.5.2 Limitations

The limitations of the study are manifold. The instrumented UAS can certainly not cover the complexities involved in commuter or transport aircraft scenarios, although some systems could be simulated. Furthermore, the endurance of the UAV is limited to several minutes of flight time. Hence, dynamic changes in the environment (e.g., weather systems) have to be accelerated, and the use of real-time dynamic data can only be limited, if possible at all. However, real-time environmental data could potentially be collected in advance and replayed faster during the experiments. Consequently, data could be scaled in both space and time to account for the constrained flight test area and UAV endurance. This approach, in turn, affects available deliberation times. However, the mission progress (e.g., vehicle speed) as well as the frequency and impact of dynamic changes can be varied, and a more conservative approach might be advantageous. Another important limitation relates to the interaction with OPS, cockpit-crew, and ATC. These interactions could optionally be simulated by integrating corresponding SAFCS communication plugins and can, therefore, not be representative for a real OPS, cockpit-crew, or ATC station integration providing professional feedback.

1.5.3 Evaluation

The evaluation of the presented research is based on the data (dependent variables) collected for selected scenarios in simulations and during flight tests. The created scenarios attempt to mimic typical flight phases with the corresponding complexity and different requirements for adaptation. The interpretation of the collected data shall help assess the viability of our proposed system as well as improve its capabilities for planned extensions and future research.

1.6 Contributions

The main contribution of our research is a self-adaptive AFCS framework—the SAFCS—that manages suitable online, anytime and dynamic algorithms for airborne decision making support and trajectory planning. The models and algorithms shall adequately

establish the SA required to cope with a variety of situations, including different flight phases under defined operational constraints. A better understanding of the performance of certain algorithms and tuning parameters with respect to situations and flight phases shall be established.

Cockpit crews have already become system managers; they will eventually become mission supervisors interacting with humans and rational machine agents in an ultra-large scale and highly complex and dynamic environment of manned and unmanned vehicles, diverse operations and administrative organizations. Higher levels of automation in the aviation and transportation sector are required to increase growth and flexibility while remaining safe and competitive in future civil and defense applications. Our research constitutes a small step toward this vision.

1.7 Dissertation Outline

This dissertation is organized into parts and chapters. Part I provides the introduction to and relevant background for our contributions. The applicable terminology and existing research covering relevant models and algorithms are briefly reviewed in Chapter 2. Chapter 3 then highlights some of the most recent and influential research initiatives that are either closely related to our contributions in terms of visions and goals, or represent important building blocks for it. Part II represents the core of this dissertation and describes the architecture and functional aspects of our SAFCS. Whereas Chapters 4 and 5 provide a conceptual and architectural overview, the remaining Chapters 6-10 cover the detailed design and functional aspects of the main SAFCS components. Where applicable, references are provided to the associated software projects and contained artifacts that encourage the inspection and allow for an even deeper understanding of our proposed solutions, if desired. The evaluation, vehicle integration and experimental flight tests are covered in Part III of this dissertation. Chapters 11, 12, and 13 describe each activity, respectively, and discuss the obtained results. Finally, Part IV reviews our contributions, draws conclusions and provides an outlook for future research and planned extensions in Chapters 14 and 15, respectively. For convenience, an annex with acronym definitions and links as well as project statistics is made available in Part V.

Chapter 2

Background

This chapter provides a brief overview of some of the relevant literature for this dissertation. It is organized into three sections covering key topics that represent the fundamental building blocks required to accomplish the established research goals, as discussed in Sections 1.3 and 1.4.

Planning and Decision Making is one of the core pillars of Artificial Intelligence (AI) and Robotics. Our Smart Autoflight Control System presented in this dissertation is required to plan ahead, continually revise plans, and implement (or provide options for) rational decisions.

Autonomous Flight and Guidance is the application domain to which planning and decision making shall be applied. The domain has been approached by computer scientists researching relevant disciplines, such as AI and Robotics, as well as by control engineers implementing the paradigm of *optimal control*.

Self-Adaptive and Self-Managing Systems is a software engineering discipline that addresses the separation of concerns related to the adaptation and evolution of software systems at runtime from more traditional software engineering methodologies applied before. Systems designed accordingly are prepared to assess their own performance, efficiently deal with imperfect (uncertain) or dynamic information and evolve with changing requirements.

2.1 Planning and Decision Making

Planning and decision making is performed by an *agent* acting *rationally* within an *environment* in order to achieve *goals* with the highest performance possible. As one of the core AI disciplines it has been covered extensively in many books and research papers. Russel and Norvig [RN10] provide a comprehensive and practical introduction to AI in general. They focus on planning and decision making problems and algorithms in several chapters. LaValle [LaV06] devotes an entire book to planning algorithms and their theoretical properties. Furthermore, planning and decision theory embodies significant material from graph theory and computational geometry. De Berg, Cheong, Kreveld, Overmars [BCKO08], Devadoss and O'Rourke [DO11] cover computational geometry algorithms and their applications in great detail.

2.1.1 Artificial Intelligence Terminology

Before reviewing some of the most relevant and influential literature that has been published in the field over the past two decades, important terminology shall be established first. *Planning* is a task that we all perform every day. The process of planning addresses the foreseeable future and helps us cope with expected situations. Classic or *proactive* planning establishes a sequence of *actions* to perform and is closely related to the concept of *open-loop control*. *Reactive* planning, in contrast, evaluates the *outcome* of a selected and implemented action before proceeding with the next action and is, hence, related to *closed-loop* control. A *contingency plan* establishes a sequence of *conditional actions* before its implementation. A complete plan of actions that addresses all possible outcomes (a universal plan) is called a *policy*. In order to plan and decide for actions rationally, the current *state* in which the agent operates needs to be observed establishing an adequate SA required for planning and decision making. SA consists of relevant information about the environment the agent operates in as well as about the agent itself.

The interaction of a rational agent with the physical (real) world and other agents that reside within it realizes a Cyber-Physical System (CPS) and can be modelled in a number of different ways. Each modelling choice determines if and how the associated planning and decision making problem can be solved. The state, that is, the *environment* as well as the agent's *capabilities*, may be modelled as being *fully* or *partially* observable. A fully observable state assumes full knowledge and a certain pre-condition for any choice of action. Whatever the outcome of an action might be, in this model it can be observed with absolute certainty. In partially observable models, observations may

not always be correct; they may be incomplete or erroneous instead. As a consequence, the current state can only be estimated resulting in an SA based on these estimates and assumptions.

The actions that can be performed by an agent may be *known* or *unknown* in advance. If an agent has limited knowledge about its own capabilities in a certain environment, then not all actions might be known and have to be learned (or combined) while acting. The state-space in which an agent operates may be modelled to be *discrete* or *continuous* which includes both time and space. The outcome of an action may be assumed to be *deterministic* or *non-deterministic*, that is, the performance and result of an action is either completely known (a certain post-condition) or may lead the agent to a *random* or *probabilistically* determined state.

The environment and capabilities of an agent may be modelled to be *static* or *dynamic*. Obviously, the performance of any plan of actions is heavily influenced if the agent's environment and capabilities change dynamically while the plan is being executed. Actions may become inappropriate or unavailable to the agent. Both environment and capabilities determine the *radius of action* of an agent.

The environment may further be modelled to be *benign* or *adversarial*. In a benign environment an agent is not confronted with any intentional dynamics that aims at preventing the agent from achieving its goals. The opposite is true for the adversarial environment where one or multiple adversaries act against the agent.

A partially observable, unknown, continuous, non-deterministic, dynamic and adversarial domain intuitively implies a very high degree of *complexity*, and *uncertainty* which are both limiting factors for a rational agent. The scope of the projected research embraces both fully observable and partially observable components, known actions, discrete time and space, both deterministic and probabilistic actions, and a dynamic and benign environment. A rationale will be provided for each modelling choice whenever applicable.

2.1.2 Motion Planning

Motion planning is a planning approach for *mobile agents*, such as aerial vehicles. A mobile agent starts at an initial state and plans a *path* or *trajectory* to a goal state optimizing applicable performance criteria. As mentioned in Section 2.1.1, a state may possess an arbitrary number of *features* describing the agent's capabilities and its environment necessary for establishing the SA required for planning and decision making. While a path

is a *sequence* of actions that may or may not consider the time domain in some form, a trajectory considers the *kinodynamic* (differential) constraints of the mobile agent and, thus, implicitly time. Kinodynamic constraints apply, for example, to velocities and accelerations within the Degrees of Freedom (DoF) of the mobile agent.

Traditional path planning techniques for mobile agents are commonly based on performing optimizations in a *configuration space* that models the *physical space* sufficiently well for the problem to be tackled. Usually some form of *roadmap* is employed that the agent can follow executing its planned actions. A roadmap may be realized as a fix *grid* or *mesh*, a map that considers *static obstacles* in the environment, such as the *visibility graph*, *trapezoidal map* or *Voronoi diagram*, or else an arbitrary (possibly randomly generated) *collision-free* map. Different types of *graphs*, such as variants of *trees*, are employed to represent roadmaps.

The employed roadmap may be *exact* in terms of representing the available *free* configuration space, or it may be *approximate*. These concepts are closely related to the *resolution* of a roadmap, which is not necessarily constant but instead may employ *multiple resolutions* to account for more or less interesting regions of the configuration space. Algorithms that operate on roadmaps are, hence, often limited to their resolutions and can only be *resolution-optimal* in the best case.

An *optimal* planning algorithm always returns a best solution (there may be many), and a *complete* planner always returns a solution if one exists. Hence, an optimal planning algorithm is also complete by definition. Finally, a *sound* planner always enters and terminates within a goal region avoiding any obstacles. Thus, a sound planner successfully accounts for potential *uncertainties*.

Different model choices require different types of algorithms. The *computational complexity* and *optimality* of a planning algorithm are the prime concerns, in particular, when the mobile agent is acting in *real-time* within a real world environment. Increased complexity leads to increased *response times* and limited *deliberation time* to *sub-optimality* with respect to planning solutions. However, due to the high dimensionality of many real world problems, often a trade-off needs to be found between optimality and timeliness. In particular, in hard real-time and high-integrity domains, a late optimal solution is not acceptable and worse than a sub-optimal but timely solution.

Due to the above issues and based on the preferred modelling choices mentioned in Section 2.1.1, planning algorithms need to be identified that possess certain properties or can otherwise be extended. Algorithms ideally need to feature *dynamic* (*incre-*

mental, *repairing*), *anytime*, *online* (*interleaved*) and *any-angle* (obeying kinodynamic constraints or allowing for post-processing) properties.

A dynamic planner is able to cope efficiently with a dynamic state (agent environment and capabilities). Only invalid parts (or at least a reduced set) of previously computed solutions are disregarded and re-computed. The ability to repair inconsistencies dynamically instead of computing solutions from scratch is an important advantage when dealing with high dimensionality and limited deliberation time. Dynamic planners implicitly deal with *imperfect* or *incomplete* knowledge.

An anytime planner returns a first and feasible solution as quickly as possible. This solution is potentially highly sub-optimal, but sub-optimality can ideally be *bounded*. As long as deliberation limitations permit, the planner incrementally improves its solutions and ideally decreases the sub-optimality bound with each increment. Thus, an anytime planner resembles nicely the human concept of *bounded rationality*: The more deliberation time available, the better the solution obtained. Any computation of an algorithm is also subject to *bounded reactivity* which implies that instantaneous failure is always possible and can only be contained in the best case.

An online planner allows to interleave planning and execution. While the agent performs planned actions, the planner uses the next expected state as a starting point for a *revised* plan. This approach can be *agent-centered* if the available sensor range or deliberation time is limited. As a consequence *partial* plans or policies may result that are potentially generic failsafe-plans and prevent catastrophic failure.

When employing Classical Grid Search (CGS), the resulting path may be unnecessarily conservative and not implicitly consider any *non-holonomic* or kinodynamic constraints applicable to the mobile agent. Any-angle methods are based on *approximate cell decomposition*, such as grids, but compute paths that are not constrained to the planning grid (and adjacent grid nodes). These algorithms avoid the computation of a more expensive roadmap, such as a visibility graph or Voronoi diagram. Although efficient algorithms exist (e.g., sweep or beach line paradigm algorithms, such as Fortune's algorithm) to compute the latter in configuration spaces with lower dimensions, comparably efficient algorithms are unavailable for higher dimensions.

Single resolution grids do not necessarily imply *uniform* costs that are proportional to covered distances. In a *non-uniform* environment higher costs may apply to shorter distances. If the non-uniform cost environment is, furthermore, dynamic in nature, these costs may change frequently. Hence, qualifying grid-based algorithms need to be able to handle non-uniform cost environments.

LaValle, Branicky and Lindemann [LBL04] review existing motion planners based on different types of roadmaps. The paper establishes a connection between CGS and search based on sampling-based Probabilistic Roadmaps (PRMs). The authors discuss important criteria for the selection of sampling points that constitute a resulting roadmap, namely *discrepancy* and *dispersion*, and discuss solutions in between both PRM and CGS, such as Quasi-Random Roadmaps (QRMs), Lattice Roadmaps (LRMs) and Subsampled Grid Searches (SGSs). According to the authors, PRM-based planners are not as advantageous over CGS as often stated before, and methods such as CGS and Rapidly-Exploring Random Tree (RRT) (as a special variant of PRM discussed below) are more preferable.

Low computational complexity and optimal solutions are often conflicting requirements in high-dimensional search spaces. In order to avoid the enumeration of the entire search space, different techniques have been developed, such as employing algorithms that are deterministic and based on *heuristics* or rely on probabilistic *sampling* of the state space. The choice of the heuristic or sampling technique significantly affects the performance of the respective algorithm. The following sections survey some of the most important past and recent developments in this field.

2.1.3 Deterministic Heuristic-Based Planning

Ferguson, Likhachev and Stentz [FLS05] give an essential overview of *heuristic* path planning. The A* algorithm and the concept of considering both *known* and *estimated* costs to guide a search toward a solution, without necessarily examining the entire state space, is the key concept for many extensions that are being researched until today. Only the most promising states (or nodes) in a graph representing the configuration space are expanded. The employed *key* function $f(s) = g(s) + h(s)$ adds the currently computed cost $g(s)$ of being in state s to an estimated cost $h(s)$ from that state toward a goal state—the heuristic. The idea is often realized using a priority queue $OPEN$ implementing a priority order defined by this key function. The basic annotated forward (from start to goal) A* is shown in Algorithm 1 for reference.

A backward version of the algorithm proceeds with the expansions from a goal state toward the initial state instead and is particularly useful if the agent's (initial) state changes during planning, as with interleaved planning and execution (online planning). During the search *back-pointers* can store the currently best found path and return it once the goal state is about to be expanded.

Algorithm 1 Basic Forward A* [FLS05]

```
1:  procedure ComputeShortestPath
2:      while argmin_{s∈OPEN}(g(s) + h(s, s_goal)) ≠ s_goal do        ▷ as long as goal state is not expanded
3:          remove state s from the front of OPEN                     ▷ expand the next inconsistent state
4:          for all s' ∈ succ(s) do                                   ▷ update the current costs of all successors
5:              if g(s') > g(s) + c(s, s') then
6:                  g(s') ← g(s) + c(s, s')
7:                  insert s' into OPEN with value (g(s') + h(s', s_goal))
8:
9:  procedure Main
10:     for all s ∈ S do                                              ▷ initialize all states
11:         g(s) ← ∞
12:     g(s_start) = 0
13:     OPEN = ∅
14:     insert s_start into OPEN with value (g(s_start) + h(s_start, s_goal))
15:     ComputeShortestPath()
```

The performance of a heuristic algorithm, such as any A*-based search algorithm, depends significantly on the quality of the employed heuristic. A good heuristic *relaxes* the actual problem as little as possible to obtain a *lower bound* for the actual total cost of a state. A heuristic is said to be *admissible* if it does not overestimate the actual costs, or more formally (where c^* is the optimal cost function):

$$\text{h is admissible} \iff \forall s, s_{goal} \in S \bullet h(s) \leq c^*(s, s_{goal})$$

A* is guaranteed to be optimal if an admissible heuristic is employed. An example for a very simple heuristic often used in basic path planning is the Euclidean distance from a state to the goal state.

A heuristic can, furthermore, be *consistent*, that is, the actual cost to an arbitrary direct successor state is never overestimated, or more formally:

$$\text{h is consistent} \iff \forall s, s' \in S, s' \in succ(s) \bullet h(s) \leq c(s, s') + h(s')$$

A consistent heuristic ensures that each state is expanded at most once during a search, which further improves the algorithmic performance. However, employing *inadmissible* and, hence, inconsistent heuristics can be advantageous. An inadmissible heuristic may *focus* the search better toward a goal and find a sub-optimal solution faster. By *inflating* the heuristic on purpose with an inflation factor $\epsilon \geq 1$, the search may return an ϵ*-sub-optimal* solution, that is, a solution that is only within the factor ϵ of an optimal solution in terms of costs. The effects of assigning different weights to computed and estimated costs have already been discussed by Pearl [Pea84] and is referred to as Weighted A* (WA*).

Another important concept has been addressed by Koenig and Likhachev [KL02] in their Dynamic A* (D*) and D* Lite variants of A*. They formalize the concept of an *inconsistent* state which can be expanded and leads to a cost update during the expansion step. Inconsistency can manifest itself in a state being either *over-consistent* or *under-consistent*. Over-consistency implies that the currently computed cost of a state can be improved. Hence, all states in the *OPEN* queue of the basic A* can be considered to be over-consistent. An under-consistent state, in contrast, possesses a better cost value than the updated cost an expansion would yield. This situation occurs if costs change (increase) dynamically. D* and D* Lite repair these inconsistencies without disregarding

Algorithm 2 Backward D* Lite [FLS05]

1: **procedure** KEY(s) **return** $[\min(g(s), rhs(s)) + h(s_{start}, s); \min(g(s), rhs(s))]$
2:
3: **procedure** UPDATESTATE(s)
4: **if** s was not visited before **then**
5: $g(s) \leftarrow \infty$ ▷ initialize states implicitly
6: **if** $s \neq s_{goal}$ **then**
7: $rhs(s) \leftarrow \min_{s' \in succ(s)}(c(s, s') + g(s'))$ ▷ determine one step look-ahead costs
8: **if** $s \in OPEN$ **then**
9: remove s from $OPEN$
10: **if** $g(s) \neq rhs(s)$ **then** ▷ update inconsistent state
11: insert s into $OPEN$ with key(s)
12:
13: **procedure** COMPUTEORIMPROVEPATH
14: **while** $\min_{s \in OPEN}(key(s) < key(s_{start}))$ **or** $(rhs(s_{start}) \neq g(s_{start}))$ **do**
15: ▷ while expansion allows for improvement or start has become inconsistent
16: remove state s with the minimum key from $OPEN$ ▷ expand highest priority state
17: **if** g(s) > rhs(s) **then** ▷ over-consistent state
18: $g(s) \leftarrow rhs(s)$ ▷ update over-consistent state
19: **for all** $s' \in pred(s)$ **do** ▷ propagate over-consistency (expansion)
20: UpdateState(s')
21: **else** ▷ under-consistent state
22: $g(s) \leftarrow \infty$ ▷ update under-consistent state
23: **for all** $s' \in pred(s) \cup \{s\}$ **do** ▷ propagate under-consistency
24: UpdateState(s')
25:
26: **procedure** MAIN
27: $g(s_{start}) = rhs(s_{start}) = g(s_{goal}) \leftarrow \infty$
28: $rhs(s_{goal}) \leftarrow 0$ ▷ lazy initialization
29: $OPEN \leftarrow \emptyset$
30: insert s_{goal} into $OPEN$ with key(s_{goal}) ▷ proceed backward from goal to start
31: **loop**
32: ComputeorImprovePath()
33: wait for changes in edge costs
34: **for all** directed edges (u, v) with changed edge costs **do**
35: update the edge cost $c(u, v)$
36: UpdateState(u) ▷ propagate inconsistencies

the current solution completely. The algorithms are, therefore, able to cope efficiently with dynamic environments and, in particular, imperfect or partial knowledge. An annotated backward variant of D* Lite is shown in Algorithm 2.

The key in D* Lite realizes a lexicographical priority order for expansion. Its first component considers the sum of minimum computed and estimated cost as before. If the first component of the key is equal for two states, priority is assigned to the minimum computed cost only instead, or otherwise an arbitrary order is chosen.

These and other important concepts led to the developments described by Likhachev, Ferguson, Gordon, Stentz and Thrun [LFG+05, LFG+08]. By combining an incremental deflation of an initially inflated heuristic (WA*) with lifelong planning (Lifelong Planning A* (LPA*) Koenig, Likhachev and Furcy [KLF04]) and dynamic repairing techniques (D*, D* Lite and Anytime Repairing A* (ARA*) Likhachev, Gordon and Thrun [LGT03]), they created the first Anytime D* (AD*) variant of the algorithm. It is shown in the annotated Algorithm 3.

Algorithm 3 Backward AD* [LFG+05]

```
1:  procedure KEY(s)
2:      if g(s) > rhs(s) then                                   ▷ over-consistent state
3:          return [min(g(s), rhs(s)) + εh(s_start, s); min(g(s), rhs(s)))]   ▷ apply inflation factor
4:      else                                                    ▷ under-consistent state
5:          return [min(g(s), rhs(s)) + h(s_start, s); min(g(s), rhs(s)))]    ▷ ignore inflation factor
6:
7:  procedure UPDATESTATE(s)
8:      if s was not visited before then
9:          g(s) ← ∞                                            ▷ initialize states implicitly
10:     if s ≠ s_goal then
11:         rhs(s) ← min_{s'∈succ(s)} (c(s, s') + g(s'))        ▷ determine one step look-ahead costs
12:     if s ∈ OPEN then
13:         remove s from OPEN
14:     if g(s) ≠ rhs(s) then                                   ▷ update inconsistent state
15:         if s ∉ CLOSED then                                  ▷ expand each state at most once per iteration
16:             insert s into OPEN with key(s)                  ▷ process inconsistent state in this iteration
17:         else
18:             insert s into INCONS                            ▷ save inconsistent state for next iteration
19:
20: procedure COMPUTEORIMPROVEPATH
21:     while min_{s∈OPEN}(key(s) < key(s_start)) or (rhs(s_start) ≠ g(s_start)) do
22:                                   ▷ while expansion allows for improvement or start has become inconsistent
23:         remove state s with the minimum key from OPEN       ▷ expand highest priority state
24:         if g(s) > rhs(s) then                               ▷ over-consistent state
25:             g(s) ← rhs(s)                                   ▷ update over-consistent state
26:             CLOSED ← CLOSED ∪ {s}                           ▷ expand each state at most once per iteration
27:             for all s' ∈ pred(s) do                         ▷ propagate over-consistency (expansion)
28:                 UpdateState(s')
29:         else                                                ▷ under-consistent state
30:             g(s) ← ∞                                        ▷ update under-consistent state
31:             for all s' ∈ pred(s) ∪ {s} do                   ▷ propagate under-consistency
32:                 UpdateState(s')
```

Applying AD* backward from a goal to the current state creates an anytime, dynamic, online planning heuristic algorithm that can be applied to many demanding real world applications. The main procedure of the annotated online backward variant is shown in Algorithm 4.

Algorithm 4 Backward AD* Main [LFG+05]

```
1:  procedure MAIN
2:      g(s_start) = rhs(s_start) = g(s_goal) ← ∞
3:      rhs(s_goal) ← 0                                   ▷ lazy initialization
4:      ε ← ε_0                                           ▷ initial inflation factor
5:      OPEN = CLOSED = INCONS ← ∅
6:      insert s_goal into OPEN with key(s_goal)          ▷ proceed backward from goal to start
7:      ComputeorImprovePath()                            ▷ compute initial sub-optimal path
8:      publish current ε-sub-optimal solution
9:      loop
10:         if changes in edge costs are detected then
11:             for all directed edges (u, v) with changed edge costs do
12:                 update the edge cost c(u, v)
13:                 UpdateState(u)                        ▷ propagate inconsistencies
14:         if significant edge cost changes were observed then
15:             increase ε or replan from scratch         ▷ fast repair or disregard solutions
16:         else if ε > 1 then
17:             decrease ε                                ▷ deflate heuristic progressively
18:         move states from INCONS into OPEN             ▷ prepare inconsistent states for next iteration
19:         update the priorities for all s ∈ OPEN according to key(s)
20:         CLOSED ← ∅
21:         ComputeorImprovePath()
22:         publish current ε-sub-optimal solution
23:         if ε = 1 then                                 ▷ found optimal solution
24:             wait for changes in edge costs
```

The value of the initial inflation factor ϵ_0 and its decrement are the important parameters for AD*. These parameters have to be set according to the problem at hand, the deliberation time available, and the quality of the heuristic. Furthermore, whether or not a change is *significant* enough to justify re-inflating the heuristic to accelerate the repair, or otherwise discard all solutions and plan from scratch, also depends on the problem domain.

All algorithms of the A* family can be applied to a grid-based representation of the environment which is referred to as CGS. Since the applied resolution of the grid discretizes the problem, the solution of the presented algorithms can only be *resolution-optimal*. Furthermore, any non-holonomic or kinodynamic constraints that apply to the agent are not considered implicitly. These issues led to other extensions within the A* family, including *multi-resolution* and *any-angle* algorithms. Multi-resolution grids allow to represent the configuration space with more or less detail. Any-angle algorithms are based on CGS, but their solutions are not restricted to the planning grid. They implicitly implement a form of interleaved *smoothing* of the computed path. In 3D envi-

ronments with polyhedral obstacles, a visibility graph can be computed within the order of quadratic complexity but does not necessarily contain a resolution-independent optimal path. Computing an optimal path in a 3D environment is NP-hard and any-angle approaches operating on grids or meshes are the current state of the art. There are three main streams for any-angle approaches in the literature: planners that employ *interpolation*, *string-pulling* or *geometric* exploration. All approaches will be briefly discussed below.

Ferguson and Stentz [FS05a, FS05b] developed Field D*—an any-angle path planning algorithm operating on 2D planning grids. Field D* uses interpolation of grid node costs in order to estimate costs for arbitrary points on the edges of the grid. The resulting paths are, consequently, not restricted to the grid edges but can be any-angle. Their Field D* algorithm presented is based on D* Lite and is, thus, an incremental algorithm that repairs changing costs and is suitable for dynamically changing environments. Figure 2.1 shows the applied interpolation options. The geometric ideas are reflected in the annotated `ComputeCost` procedure of Algorithm 5.

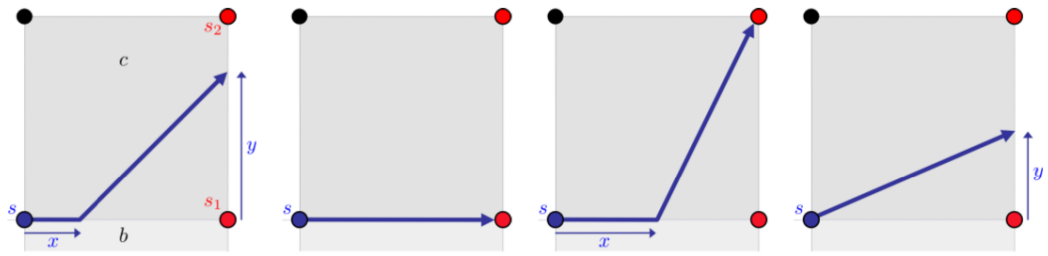


Figure 2.1: Field D* Interpolation [FS05a]

Field D* has subsequently been extended to 3D grids by Carsten, Ferguson and Stentz [CFS06]. Interpolation of the computed costs of two edge vertices to obtain the cost of an arbitrary point on their edge in 2D is extended to finding the corresponding point in a neighboring *face* of an expanded node. Thus, neighboring edges in $2D$ of an expanded node become neighboring faces in $3D$. The interpolation problem becomes more complex by adding an additional dimension (parameter) to optimize (minimize) for, and there is no closed solution. Since numerical methods are too costly, considering that the operation has to be repeated many times, the problem is reduced to finding the minimum edge points on a face border (2D interpolation) and computing the line intersection between the two line segments created by connecting the opposite points. The resulting intersection point represents the approximate solution, and its parameters can be used to calculate the approximate cost of the open form problem. The authors,

furthermore, show how directional transition costs can be incorporated to model kinematic constraints associated with mobile agents, such as aerial vehicles. Local or global scaling factors can be employed to integrate vehicle performance into voxel transition costs.

Algorithm 5 Field D* [FS05a]

procedure COMPUTECOST(s, s_a, s_b)
  if s_a is a diagonal neighbor of s **then** ▷ establish correct neighbor order
    $s_1 \leftarrow s_b$; $s_2 \leftarrow s_a$
  else
    $s_1 \leftarrow s_a$; $s_2 \leftarrow s_b$
  c is traversal cost of cell with corners s, s_1 and s_2 ▷ upper cell traversal costs
  b is traversal cost of cell with corners s, s_1 but not s_2 ▷ lower cell traversal costs
  if $\min(c, b) = \infty$ **then**
    $v_s \leftarrow \infty$ ▷ cost of non-traversable obstacle
  else if $g(s_1) \leq g(s_2)$ **then**
    $v_s \leftarrow \min(c, b) + g(s_1)$ ▷ minimum cost of lower edge path from s to s_1
  else ▷ diagonal neighbor is less costly than horizontal neighbor
    $f \leftarrow g(s_1) - g(s_2)$ ▷ edge cost difference between s_1 and s_2
    if $f \leq b$ **then** ▷ edge cost difference is less than or equal lower cell traversal cost
      if $c \leq f$ **then** ▷ $c \leq f \leq b$ (diagonal cost is best), edge cost difference is greater than or equal upper cell traversal cost
        $v_s \leftarrow c\sqrt{2} + g(s_2)$ ▷ proceed diagonally with resolution (edge length) 1
      else ▷ $c > f \leq b$ (cost of lower edge equals c), edge cost difference is smaller than upper cell traversal cost
        $y \leftarrow \min\left(\frac{f}{\sqrt{c^2 - f^2}}, 1\right)$ ▷ compute best interpolated vertical intersection point on edge $\overline{s_1 s_2}$ with resolution (edge length) 1
        $v_s \leftarrow c\sqrt{1 + y^2} + f(1 - y) + g(s_2)$ ▷ compute new cost from upper cell traversal, vertical edge portion and diagonal neighbor
    else ▷ edge cost difference is greater than lower cell traversal cost
      if $c \leq b$ **then** ▷ $c \leq b < f$ (diagonal cost is best), upper cell traversal cost is less than or equal lower cell traversal cost
        $v_s \leftarrow c\sqrt{2} + g(s_2)$ ▷ proceed diagonally with resolution (edge length) 1
      else ▷ $c > b < f$ (cost of lower edge equals b)
        $x \leftarrow 1 - \min(\frac{b}{\sqrt{c^2 - b^2}}, 1)$ ▷ compute best interpolated horizontal intersection point on edge $\overline{s s_1}$ with resolution (edge length) 1
        $v_s \leftarrow c\sqrt{1 + (1 - x)^2} + bx + g(s_2)$ ▷ compute new cost from upper cell traversal, horizontal edge portion and diagonal neighbor
  return v_s

Ferguson and Stentz [FS06b], furthermore, extended the Field D* algorithm to support multiple resolutions and, thus, be able to represent more or less interesting regions of the configuration space dynamically with the appropriate detail in *non-uniform grids*. The grid resolution of a planning environment may change over time. An initial planning grid my change with respect to the available data and its quality (precision). In particular, if on-board sensors are used in combination with agent-centered planning, additional data with higher precision could be obtained during the mission.

The formalism addresses mainly the issue of neighboring cells featuring different resolutions. Two options for interpolation are presented: One uses interpolation along the edge of the low-resolution cell, whereas the other uses the individual nodes of a high-resolution cell. If the ratio between edge lengths of two neighboring high-resolution

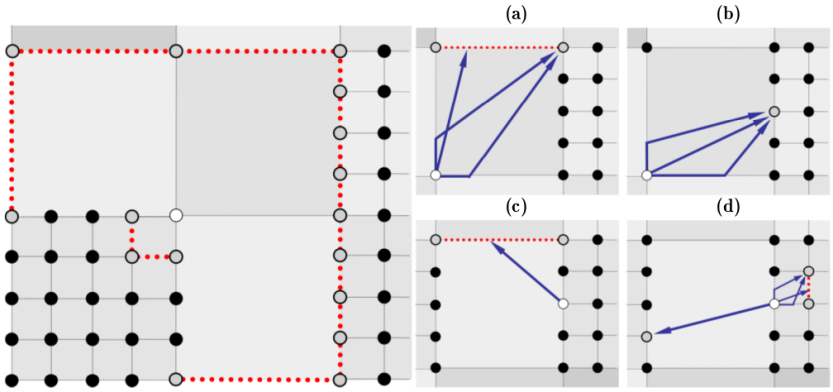


Figure 2.2: Multi-Resolution Field D* [FS06b]

and low-resolution cells is high, then planning via the high-resolution nodes may be more efficient (maintaining a sufficient approximation quality). Otherwise, if the ratio is lower, the interpolation method for high-resolution cell edges may provide a better approximation (adding some additional computation costs). The basic idea is depicted in Figure 2.2.

The second stream of any-angle algorithms applies some form of pulling strings. Based on CGS, these algorithms eliminate intermediate nodes within a computed path and, thus, connect non-neighboring nodes directly (which looks like pulling a string that connected the nodes indirectly before the operation).

Nash and Koenig [NK13] developed their Θ^* family of algorithms which are derived from A*. Planning is performed in a *regular grid*, such as a triangular, square or hexagonal in 2D, or cubic grid in 3D. All Θ^*-based algorithms interleave search with smoothing instead of relying on an explicit post-smoothing step only (A* with Post Smoothing (A*PS)). String pulling with Θ^* is shown in Figure 2.3.

Θ^* allows to connect the parent of the currently expanded node with all of its successors if they are visible from each other. Hence, it requires an implicit *visibility test* (to account for static obstacles) and the resulting paths do not need to follow the grid edges. The applied costs are direct Euclidean distances between the visible nodes and, thus, assume a uniform cost environment.

Nash, Koenig and Tovey [NKT10] address the issues of reducing the number of visibility checks and cost estimation in 3D with Lazy Θ^*. Lazy Θ^* reduces the number of visibility checks by assuming (*collision-free*) visibility between the parent (currently best predecessor) and the successors of an expanded node first, and instead performing visibility checks only for expanded vertices (correcting any visibility issues, if any). The

Algorithm 6 Lazy Θ^* [FS05a]

```
1:  procedure UpdateVertex(s, s')
2:      g_old ← g(s')
3:      ComputeCost(s, s')                                   ▷ compute updated costs
4:      if g(s') < g_old then                                ▷ improve costs
5:          if s' ∈ open then
6:              open.Remove(s')
7:          open.Insert(s', g(s') + h(s'))                   ▷ (re-)insert state for expansion
8:
9:  procedure SetVertex(s)
10:     if ¬ lineofsight(parent(s), s) then                  ▷ revert short-cuts to non-visible parent state
11:         parent(s) ← arg min_{s'∈nghbrvis(s)∩closed} (g(s') + c(s', s))   ▷ connect to best visible direct neighbor
12:         g(s) ← min_{s'∈nghbrvis(s)∩closed} (g(s') + c(s', s))            ▷ compute costs via best visible neighbor
13:
14: procedure ComputeCost(s, s')
15:     if g(parent(s)) + c(parent(s), s') < g(s') then      ▷ initially assume visibility
16:         parent(s') ← parent(s)                           ▷ connect parent and child of expanded state
17:         g(s') ← g(parent(s)) + c(parent(s), s')          ▷ compute direct costs from grandparent to child
18:
19: procedure Main
20:     open = closed ← ∅                                    ▷ lazy initialization
21:     g(s_start) ← 0
22:     parent(s_start) ← s_start
23:     open.Insert(s_start, g(s_start) + h(s_start))        ▷ expand forward from start to goal
24:     while open ≠ ∅ do                                    ▷ while states can be expanded
25:         s ← open.Pop()                                   ▷ expand highest priority state
26:         SetVertex(s)                                     ▷ check visibility only during expansion
27:         if s = s_goal then                               ▷ terminate if goal is expanded
28:             return path found
29:         closed ← closed ∪ {s}
30:         for all s' ∈ nghbrvis(s) do                      ▷ examine all visible neighbors
31:             if s' ∉ closed then                          ▷ assume consistent heuristic and expand only once
32:                 if s' ∉ open then                        ▷ implicit initialization
33:                     g(s') ← ∞
34:                     parent(s') ← nil
35:                 UpdateVertex(s, s')                      ▷ update costs
36:     return no path found
```

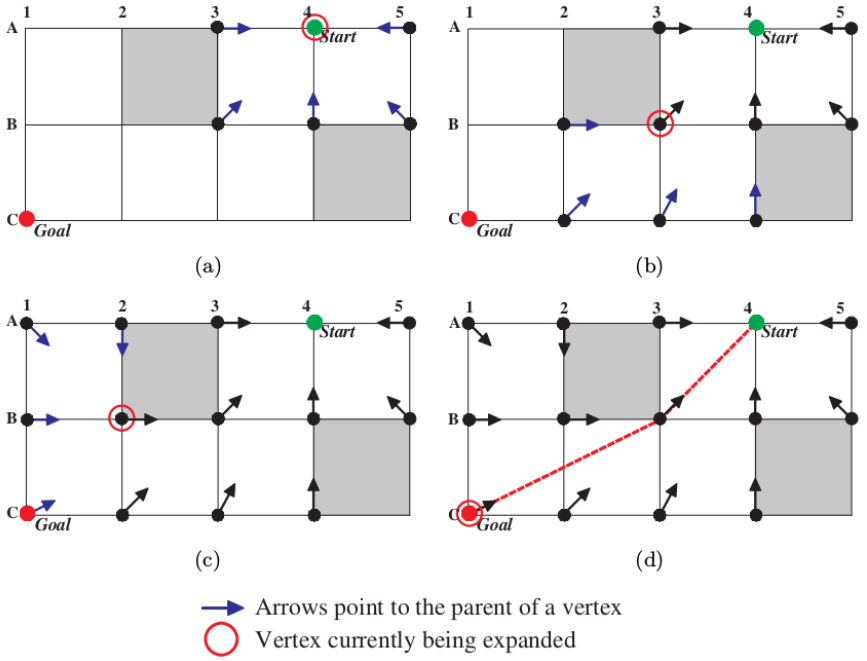


Figure 2.3: Θ* String Pulling [NK13]

approach trades off a significant number of visibility checks with a higher number of expanded nodes while improving the algorithmic performance. It is shown annotated in Algorithm 6.

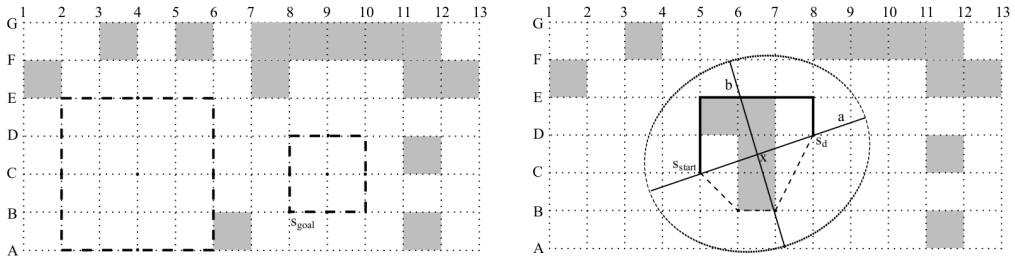
Further extensions have been developed in the Θ* family, addressing both bounded sub-optimal search using inflated heuristics and incremental (repairing) techniques. Weighted Lazy Θ* (Nash and Koenig [NK13]) works similar to WA* finding bounded sub-optimal solutions faster. Incremental Φ* (Nash, Koenig and Likhachev [NKL09]) can be applied in dynamic 2D grids or meshes to efficiently replan as information changes. There is, however, currently no extension to Incremental Φ* for 3D environments with anytime behavior available (Koenig and Uras private communication).

Another string-pulling based any-angle path planning algorithm is Block A* introduced by Yap, Burch, Holte and Schaeffer [YBHS11]. Block A* basically divides the planning grid into a coarser grid of blocks, expanding neighboring blocks instead of nodes based on their block key value. The block key value is the minimum of all block fringe node f-values. Fringe node g-values represent shortest paths within a block and are computed based on previous fringe vertices using a lookup table (database approach). Hence, block sizes and memory management matter for Block A*. Optimal paths are not

guaranteed and heading changes in the free configuration space are possible at fringe nodes.

After having introduced the most influential interpolation and string pulling any-angle heuristic path planning algorithms, this discussion shall conclude with two geometrically more sophisticated approaches.

Sislák, Volf and Pechoucek [SVP09a, ŠVP09b] have developed the any-angle heuristic path planning algorithm Accelerated A* (AA*). AA* modifies A* by reducing the branching factor of the search adaptively. The expansion of a node considers only four (a constant number) neighbors on the side-centers of a square with the expanded node at its center. The size of the square depends on the obstacles in the grid and is the maximum area obstacle-free square. Each of the four candidates is then connected, not necessarily to the expanded node, but to any visible, shortest distance node in the *CLOSED*-set that lies within an elliptic region uniquely defined by the start node and the considered candidate. Hence, the key components of the algorithm are the efficient calculation (lookup) of a maximum unblocked square with the expanded node at its center as well as the filtering of previously expanded nodes in *CLOSED* that are contained in the elliptic region. Cormen, Leiserson, Rivest and Stein [CLRS09] are cited by the authors for efficient hash-table solutions to these problems. The geometric ideas of AA* are depicted in Figure 2.4.



(a) AA* Square **(b)** AA* Ellipse

Figure 2.4: AA* Geometric Ideas [ŠVP09b]

The algorithm was originally intended to be used for free-flight navigation and obviously with the application in 3D in mind. An iterative version of AA* [KSPP10] is described by the same authors.

The most recent, first and only known optimal any-angle grid-based path planning algorithm to date has been published by Harabor and Grastien [HG13]. It is called Anya and has subsequently been evaluated by Uras and Koenig [UK15]. Anya identifies points on traversable adjacent intervals within the free configuration space to find the short-

est path between corners (resulting in the shortest path within the visibility graph) from start to goal. In *Anya* search nodes are represented by a root point and an interval. Initially, the root point is located outside the search space (2D), and the interval contains only the start location. Neighboring intervals are located on the same or immediately adjacent rows with respect to the root point. By identifying the point in a traversable interval (corner or intermediate point) which is closest to the intersection between the row containing that interval and the line segment connecting the root point and a target point, the optimal path is iteratively constructed. Neighboring intervals are computed online. The f function consists of the sum of the optimal costs from the start location to the root point, the Euclidean distance between the root point and the interval intersection point (taut path), and the heuristic from the interval intersection point to the goal location. Anya has not yet been extended to dynamic 3D environments with anytime behavior, if at all feasible (Koenig and Uras private communication). Only recently has Anya been applied to spherical navigation problems by Rospotniuk and Small [RS20].

2.1.4 Probabilistic Sampling-Based Planning

Sampling-based planning methods aim at breaking the curse of high dimensionality and intractable computational complexity by taking random or problem-biased samples of the configuration space in order to find a solution. The more samples are taken and the more problem-specific the applied sampling techniques are, the higher the probability of finding a near-optimal solution. The two most important (and interrelated) approaches that apply sampling to the search space are PRMs and RRTs. These approaches implicitly create problem-specific roadmaps, and their extensions address dynamic context changes (including imperfect or partial knowledge) as well as anytime online (interleaved) planning.

Kavraki, Svestka, Latombe and Overmars [KSLO96] introduce and evaluate the first planning algorithm based on PRM applicable for static environments. The original algorithm consists of an offline *learning phase* and an online *query phase*.

During the learning phase a PRM is created by connecting randomly selected (non-biased uniform distribution) configurations in the configuration space. Initial connections are then established between neighboring configurations within a maximum distance during the *construction step* employing a *local planner*. In more *difficult regions* where the local planner of the construction step has failed to connect configurations due to encountered obstacles, an *expansion step* is initiated. The expansion step performs

random walks with straight segments bouncing off any discovered obstacles. The maximum length of these *random bounce walks* is another parameter of the algorithm. In summary, the construction step creates connections using a uniform approach, whereas the expansion step invests more effort to equip the difficult regions of the configuration space with a denser non-uniform roadmap. Example PRMs are shown in Figure 2.5.

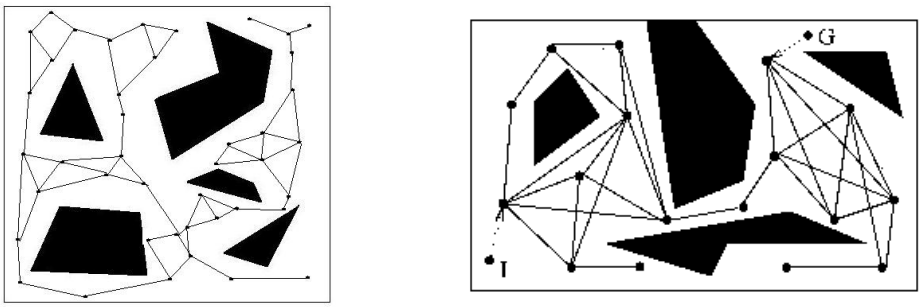


Figure 2.5: Probabilistic Roadmap Examples[1]

The stored collision-free roadmap graph contains the nodes and edges of each connected component. Edges that are created during the expansion step are stored with the associated paths that have been found during the random bounce walks. The algorithm can be customized using problem-specific local planners, distance functions and timing upper bounds for the different steps. Furthermore, a certain number of connected components can be kept or discarded depending on their resulting size (number of nodes and edges). The basic PRM construction is shown in the annotated Algorithm 7 where $\Delta : C_f \times C_f \rightarrow \{0,1\}$ indicates whether or not two samples of the free configuration space can be connected by the local planner.

Algorithm 7 Probabilistic Roadmap [KSLO96]

1: **procedure** CONSTRUCTPRM
2: $N \leftarrow \emptyset$ ▷ initialize nodes
3: $E \leftarrow \emptyset$ ▷ initialize edges
4: **loop**
5: $c \leftarrow$ a randomly chosen free configuration ▷ obtain next random sample
6: $N_c \leftarrow$ a set of candidate neighbors of c chosen from N ▷ filter candidate neighbors within a maximum distance
7: $N \leftarrow N \cup \{c\}$ ▷ add new sample to the node set
8: **for all** $n \in N_c$, in order of increasing $D(c,n)$ **do** ▷ examine candidate neighbors with increasing distance
9: **if** $\neg same_connected_component(c,n) \wedge \Delta(c,n)$ **then** ▷ both nodes can be connected by the local planner
10: $E \leftarrow E \cup \{(c,n)\}$ ▷ extend edges storing local planner path implicitly
11: update connected components

During a query phase (possibly multiple queries) a global planner, such as an A*-based algorithm, operates on the previously computed PRM after having connected

[1] http://www.kavrakilab.org

the query start and goal configurations to the roadmap (possibly using random bounce walks). Learning and query phases can be interleaved to obtain an online PRM-based planner.

Bohlin and Kavraki [BK00] modified the original PRM algorithm with lazy collision checking when connecting sampled configurations. Lazy PRM defers collision checking to the query phase of the algorithm. Hence, a path is computed based on a possibly non-collision-free roadmap first. Any colliding edges are removed from the PRM by the global planner instead, and a repairing re-construction step might be required.

Dynamic environments featuring both stationary and moving obstacles with known trajectories have been addressed by van den Berg and Overmars [vdBO05]. They present an A*-based algorithm to operate on a PRM avoiding collisions with moving obstacles.

A PRM is constructed in advance avoiding all stationary obstacles. In the query phase a feasible trajectory is found that is *near time-optimal* and avoids the dynamically moving obstacles. A query is divided into generating local trajectories (*probing*) and a global trajectory.

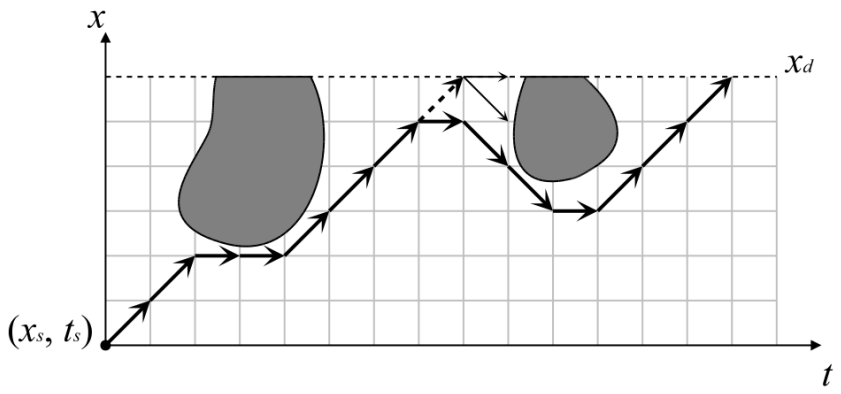


Figure 2.6: State-Time Probe [vdBO05]

A local trajectory is found by dividing a single roadmap arc into equal time steps and modelling the movement along the arc using an implicit *state-time* representation. An agent is allowed to move forward, stay put, or move backward using a constant speed for a single time step τ. Movements toward the destination node of an arc are always preferred and, consequently, expanded first. The result is a collision-free *state-time* trajectory along a PRM arc.

A global trajectory is found starting from the initial agent configuration by launching *probes* (local trajectory searches) along all incident arcs (edges). Each probe is associated with a *priority key* that indicates which probe to expand next in an A* fashion.

The result is a near time-optimal trajectory from the start to the goal configuration consisting of collision-free local trajectories. A single probe examining one PRM arc for a collision free trajectory is shown in Figure 2.6.

The time-step τ can be adjusted to trade-off solution quality for computation time. Hence, as τ approaches zero, the trajectory becomes time-optimal. Some optimization strategies are discussed that aim at deleting unsuccessful probes early in the process of finding local trajectories, including *interacting probes* that possess state-time *meeting points*.

Van den Berg, Ferguson and Kuffner [vdBFK06] focus even more on anytime behavior and aspects of dynamic elements in the environment. Dynamic elements can be modelled as completely unpredictable (random), predictable (probabilistic, uncertain), or known (deterministic). By always assuming unpredictable changes of dynamic elements of the environment, less efficient algorithms and less optimal solutions are more likely.

The authors (quote)

> "combine the efficient replanning ability of deterministic incremental planning algorithms with the efficient representations produced by probabilistic sampling approaches"

in an *anytime* algorithm accounting for limited deliberation time. The approach can be seen as a hybrid of probabilistic roadmap planning and deterministic, anytime, dynamic (repairing) planning. It consists of three steps:

- Construction of a static PRM
- Anytime planning of an initial trajectory in the state-time domain including known dynamics
- Incremental repairing of the current trajectory due to dynamic changes

The initially constructed PRM is an efficient representation of the traversable state space. Since it considers only static elements in the environment, it can be retained and used as a basis for all future planning efforts (it replaces the (multi-resolution-) grid in grid-based (any-angle) planning methods). The trajectories are planned and repaired incrementally using the AD* algorithm described under Section 2.1.3 using a heuristic that accounts for both time and space. The time domain and known kinodynamic constraints allow to realize the concept of minimum and maximum arrival times, that is, the

radius of action, and, hence, the pruning of the search graph. The space domain and its costs allow for guided optimal planning (based on the roadmap). As with AD* the inflation factor is reduced to progressively tighten the sub-optimality bound and improving the solution as deliberation limits permit. The heuristic of the key function is a weighted sum of path *time* and *costs*.

Another approach to anytime, dynamic PRM is named Flexible Anytime Dynamic PRM (FADPRM) and has been developed by Belghith, Kabanza, Hartman and Nkambou [BKHN06, BKH13]. FADPRMs combine traditional random sampling with the AD* approach and extend PRM with regions possessing different *degrees of desirability* (dd), a dynamic environment (changing edge costs and degrees of desirability leading to structural changes in the PRM) and an anytime incremental approach which starts with a fast but potentially highly sub-optimal solution. As planning iterations proceed, this solution is continuously improved. Optimality can, however, not be guaranteed.

The degree of desirability lies between [0..1], is encoded separately from the cost (e.g., Euclidean distance), and can be generated by merging available information (average of overlapping dd zones). The priority key for the selection of nodes to be expanded is a weighted (inflated) sum with inflation factor $0 \leq \beta \leq 1$ which allows to transition between fast exploration, as with standard PRM ($\beta = 0$), and guided search, as with A* ($\beta = 1$). The A* terms for cost (g) and heuristic (h) combine dd-values and costs by using another weight factor ($0 \leq \gamma \leq 1$) which allows to transition between preference for high degree of desirability ($\gamma = 0$) and least cost ($\gamma = 1$). The dd of a path is the average of all configuration dds in the path. The best path is defined as the one possessing the highest overall dd found.

Since AD* (or alternatively, Generalized Adaptive A* (GAA*) (Sun, Koenig and Yeoh [SKY08])) is used to extend PRMs, the algorithm can deal with inconsistent states (over-consistency leads to improvements, under-consistency leads to partial re-computation) in the case of dynamic environment changes and possesses anytime characteristics delivering progressively improving β-sub-optimal solutions. Implementing a Lazy PRM (LPRM), collision checking is only performed after a complete β-sub-optimal path has been returned. The search continues as long as collisions are present. According to the authors, optimality can, however, not be assured since considering degrees of desirability impedes the admissibility of the heuristic.

RRT is related to PRM but, in its simplest form, constructs a tree that is biased to develop from a start toward a goal configuration. LaValle and Kuffner [LKJ01] summa-

rize existing work and highlight research directions related to planning with RRTs. The growth of an example RRT is shown in Figure 2.7.

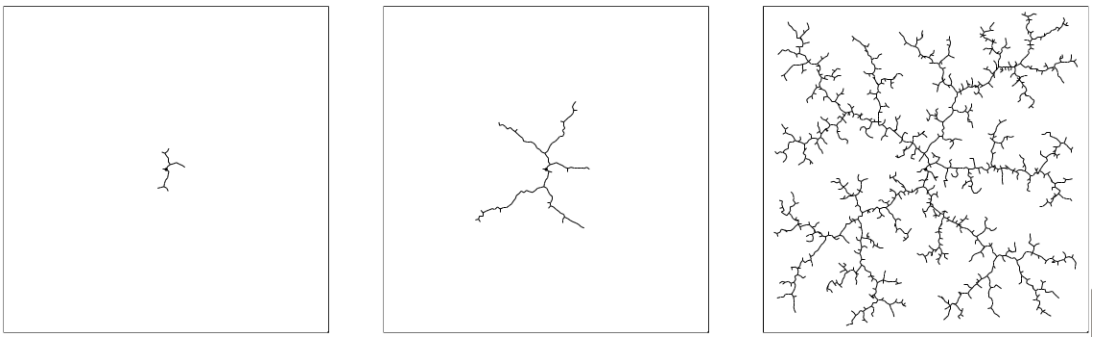


Figure 2.7: Rapidly-Exploring Random Tree Growth [LKJ01]

RRT may implicitly consider *non-holonomic*, *kinodynamic* and *kinematic closure* constraints and can be applied to problems involving many DoF. In its most basic form, and in order to find a path between an initial and a goal location, a tree is created containing the initial location first. This tree is iteratively extended using random samples that are taken from the free configuration space. The mentioned implicit constraints lead to a new node q_{new} to be appended to the tree that lies within a certain metric threshold to its nearest neighbor in the tree. Each new node and edge is implicitly checked for collisions. The RRT generation with an optional extension to use previously found waypoints as samples (Extended RRT (ERRT)) is shown in the annotated Algorithm 8.

The selected sampling and extension strategies significantly affect the performance of the algorithm. Sampling can, for example, focus on larger uncovered regions in the free configuration space using Voronoi regions.

Single RRT planners proceed from the initial location toward the goal location. The expansion strategy can be more or less aggressive toward the goal, potentially risking local minima. RRT-*GoalBias* (higher extension probabilities toward the goal) and RRT-*GoalZoom* (higher extension probabilities toward the goal region) can be employed to guide the extension more aggressively toward the goal. Another option would be to apply a non-uniform gradually increasing probability density function toward the goal location to guide the expansion. Furthermore, an extension iteration instead of a single extension for a single sample can be employed in an even more aggressive variant.

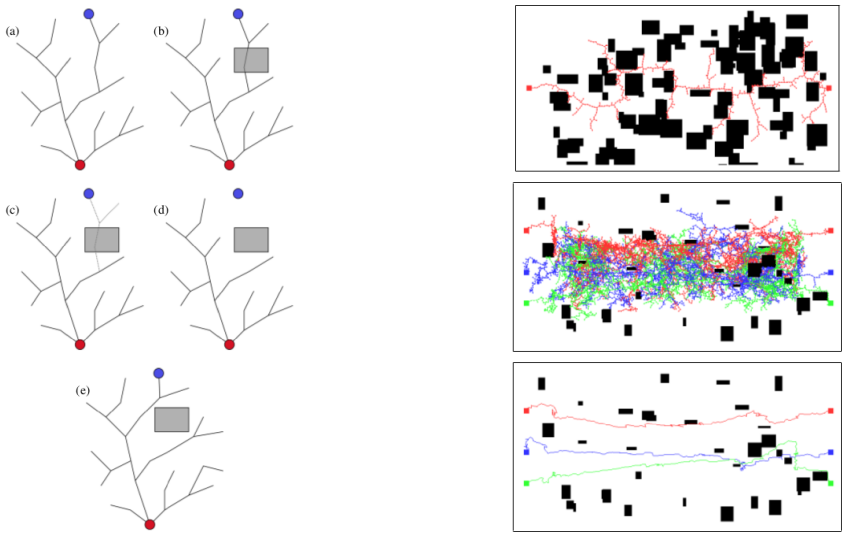
Bidirectional RRT (Kuffner and LaValle [JL00]) planners proceed from both the initial and the goal location constructing two trees that eventually have to be merged to one.

Algorithm 8 Rapidly-Exploring Random Tree and Extended RRT [FKS06]

```
1:  procedure InitRRT
2:      T.add(q_start)                                              ▷ add initial start node
3:
4:  procedure GrowRRT
5:      q_new ← q_start                                             ▷ determine initial distance to goal region
6:      while Distance(q_new, q_goal) > distance-threshold do       ▷ while tree has not grown into goal region
7:          q_target ← ChooseTarget()                               ▷ obtain a new sample to expand toward
8:          q_nearest ← NearestNeighbor(q_target)                   ▷ find the nearest neighbor in the tree
9:          q_new ← Extend(q_nearest, q_target)                     ▷ expand tree from nearest neighbor to feasible new node toward sample
10:         if q_new ≠ nil then
11:             T.add(q_new)                                        ▷ add feasible new node to tree
12:
13: procedure ChooseTarget
14:     p ← RandomReal([0.0, 1.0])                                  ▷ randomly select goal bias
15:     i ← RandomInt([1, num-waypoints])                           ▷ randomly select cached waypoint
16:     if p < goal-sampling-prob then                              ▷ grow tree toward goal
17:         return q_goal
18:     else if p < goal-sampling-prob + waypoint-prob then         ▷ grow tree toward cached waypoint
19:         return WaypointCache[i]
20:     else                                                        ▷ grow tree toward random sample
21:         return RandomNode()
```

The trees are expanded and merging attempts are performed in an alternating fashion. The option of iterative expansion or merging attempts leads to more or less aggressive variants of the algorithm where the latter are more appropriate for holonomic planners. These variants use the *Ext* and *Con* acronyms to label single or iterative extensions, respectively. RRT-*ExtExt*, RRT-*ConExt*, RRT-*ExtCon* and RRT-*ConCon* refer to the different variants where the first label specifies the extension and the second label the merging attempts, respectively.

Further extensions of the algorithm may even choose to extend multiple trees simultaneously up to a point where each sample will lead to a tree generation. The last approach is very similar to PRM where many smaller connected components are merged to generate a larger connected component to become the resulting probabilistic roadmap. The difference between RRT and PRM is the separation between the offline generation and online query phase for basic PRM, which restricts the approach to rather static environments. RRT, on the other hand, performs the entire computation for each query and, hence, is referred to as a *single query* method being more suitable for highly dynamic environments. Furthermore, RRT is intended to be strongly biased toward a goal region and, thus, rapidly explore the relevant free configuration space.



(a) DRRT Repair **(b)** RRT Examples

Figure 2.8: Dynamic RRT Examples [FKS06]

Ferguson, Kalra and Stentz [FKS06] developed Dynamic RRT (DRRT)—a repairing variant of RRT suitable to deal with dynamic environments of high dimensionality. DRRT grows random trees that are biased toward a goal (or start location with back-

ward RRT) quickly. Both, basic RRT and DRRT can be extended to allow for interleaved planning and execution. ERRT is such an extension that performs a backward planning from goal to the next agent location while the agent is moving on a previously planned arc of the tree. Hence, avoiding dynamic obstacles successfully depends on the lengths of the planned arcs and the obstacle frequency and velocity. ERRT reuses sampled waypoints of previous successful solutions from a waypoint cache with a certain probability. However, ERRT grows the entire tree from scratch given the next agent location. DRRT, in contrast, only invalidates parts of the tree and can bias re-growth toward these regions. During the re-growth phase ERRT can be employed to reuse previous samples. The approach does not retain any possibly valid sub-branches of invalid arcs. Instead, the entire sub-branch is trimmed resulting in a simpler algorithm that targets an agent-centered sensor model and follows the assumption that invalid branches have not yet grown very deep. According the authors, DRRT features an extremely fast response, even in high-dimensional search spaces outperforming ERRT by a factor of 5 on the average. Efficient employed algorithms include *kd-trees* to find the nearest neighbors in a tree for a newly established waypoint. A tree repair example and some generated trees in an environment containing obstacles are shown in Figure 2.8. The annotated DRRT has been implemented in Algorithm 9.

Ferguson and Stentz [FS06a], furthermore, developed Anytime RRT (ARRT)—an anytime variant of RRT suitable to deal with real-time domains and their associated deliberation limitations. While standard RRT does not consider non-uniform cost environments, it generates solutions very quickly. These solutions are potentially highly sub-optimal with respect to costs.

ARRT starts with an initial basic RRT that can, thus, be arbitrarily sub-optimal with respect to costs. However, it produces a first solution very fast if deliberation time is the limiting factor. While deliberation time allows, this first and highly sub-optimal solution is incrementally improved by progressively shifting from *least distance* to *least cost* expansion. To achieve this transition, more sophisticated *node sampling*, *node selection* and *node expansion* strategies are employed. Node sampling is restricted to nodes q_{target} that keep the sum of two heuristics $h(q_{start}, q_{target}) + h(q_{target}, q_{goal})$ below the upper cost bound C_S for a previously found solution. Node selection establishes a priority order with respect to the k nearest nodes to q_{target} in the tree that is biased toward expansion *distance* or *cost*. Initially the bias for distance d_b is high which results in standard RRT expansion behavior. With each iteration that aims at finding a new solution within an improved cost bound $(1-\epsilon)C_s$ (*improvement factor* ϵ_f), the distance bias is de-

Algorithm 9 Dynamic RRT [FKS06]

```
1:  procedure RegrowRRT
2:      TrimRRT()                                              ▷ trim invalid branch
3:      GrowRRT()                                              ▷ grow trimmed tree
4:
5:  procedure TrimRRT
6:      S ← ∅                                                  ▷ initialize node set
7:      i ← 1
8:      while i < T.size() do                                  ▷ check all parent nodes for validity
9:          q_i ← T.node(i)
10:         q_p ← parent(q_i)
11:         if q_p.flag = INVALID then                         ▷ trim entire invalid branch
12:             q_i.flag ← INVALID
13:         else if q_p.flag ≠ INVALID then                    ▷ maintain valid nodes
14:             S ← S ∪ {q_i}
15:         i ← i + 1
16:     T ← CreateTreeFromNodes(S)                             ▷ create trimmed tree
17:
18: procedure InvalidateNodes(obstacle)
19:     E ← FindAffectedEdges(obstacle)
20:     for all e ∈ E do                                       ▷ examine all colliding edges
21:         q_e ← ChildEndpointNode(e)                         ▷ invalidate all non-visible endpoints
22:         q_e.flag ← INVALID
23:
24: procedure Main
25:     q_start ← q_goal                                       ▷ perform backward planning from goal toward agent
26:     q_goal ← q_agent
27:     InitRRT()
28:     GrowRRT()                                              ▷ grow initial tree from goal to agent
29:     while q_goal ≠ q_start do                              ▷ continue to plan until the agent arrives at the goal
30:         q_goal ← parent(q_goal)                            ▷ plan from goal toward next agent waypoint
31:         Move to q_goal and check for new obstacles         ▷ move agent on current leg
32:         if any new obstacles are observed then
33:             for all new obstacle o do                      ▷ check all new obstacles for collisions
34:                 InvalidateNodes(o)                         ▷ invalidate branches of all invisible nodes
35:             if solution path contains an invalid node then ▷ only regrow tree if current plan is affected
36:                 RegrowRRT()
```

creased and the cost bias c_b is increased. The resulting solutions incrementally require more computation time but continuously improve the solution by the improvement factor as long as the available deliberation time allows. Node expansion can use either fast repetitive attempts to fulfil the cost constraint or, alternatively, use a large set of possible extensions and select the least costly. Again, these two approaches can be applied adaptively to the early or late phases of the algorithm, respectively.

Algorithm 10 Anytime RRT [FS06a]

```
1:  procedure REINITIALIZERTT(T)
2:      T.clearTree()
3:      T.add(q_start)
4:
5:  procedure GROWRRT(T)
6:      time ← 0                                              ▷ initialize timer for maximum tree computation time
7:      q_new ← q_start                                       ▷ determine initial distance to goal region
8:      while Distance(q_new, q_goal) > distance-threshold do ▷ while tree has not grown into goal region
9:          q_target ← ChooseTarget(T)                        ▷ obtain a new sample to expand toward
10:         if q_target ≠ nil then                            ▷ if a sample using the sampling strategy has been found
11:             q_new ← ExtendToTarget(q_target, T)           ▷ extend tree toward target sample
12:             if q_new ≠ nil then
13:                 T.add(q_new)                              ▷ add feasible new node to tree
14:         UpdateTime(time)                                  ▷ update current computation time
15:         if time > max-time-per-rrt then
16:             return nil                                    ▷ interrupt tree construction if maximum tree computation time has expired
17:     return T.c(q_start, q_new)
18:
19: procedure CHOOSETARGET(T)
20:     p ← RandomReal([0.0, 1.0])                            ▷ randomly select goal bias
21:     if p < goal-sampling-prob then                        ▷ grow tree toward goal
22:         return q_goal
23:     else
24:         q_target ← RandomConfiguration()
25:         attempts ← 0                                      ▷ initialize for maximum sampling strategy attempts
26:         while h(q_start, q_target) + h(q_target, q_goal) > T.C_s do ▷ while the sampling heuristic is not within the cost sub-optimality bounds
27:             q_target ← RandomConfiguration()              ▷ obtain a new random sample
28:             attempts ← attempts + 1                       ▷ update sampling attempts
29:             if attempts > max-sample-attempts then
30:                 return nil                                ▷ interrupt sample generation if maximum sampling attempts have been exceeded
31:         return q_target
32:
33: procedure SELCOST(T, q, q_target)
34:     return T.d_b * Distance(q, q_target) + T.c_b * T.c(q_start, q)   ▷ add distance bias times distance to cost bias times cost of extension
35:
36: procedure EXTENDTOTARGET(T, q_target)
37:     Q_near ← kNearestNeighbors(q_target, k, T)            ▷ obtain nearest neighbors in the tree
38:     while Q_near ≠ ∅ do
39:         remove q_tree with minimum SelCost(T, q_tree, q_target) from Q_near   ▷ establish priority order according to current distance and cost bias
40:         Q_ext ← GenerateExtensions(q_tree, q_target)      ▷ generate a set of possible extensions toward target (one possible strategy)
41:         q_new ← argmin_{q∈Q_ext} c(q_tree, q)             ▷ select the least cost extension
42:         T.c(q_start, q_new) ← T.c(q_start, q_tree) + c(q_tree, q_new)   ▷ update tree costs
43:         if T.c(q_start, q_new) + h(q_new, q_goal) ≤ T.C_s) then   ▷ computed and estimated costs are within the sub-optimality bound
44:             return q_new
45:     return nil
46:
47: procedure MAIN
48:     T.d_b ← 1                                             ▷ start with maximum distance bias for rapid exploration
49:     T.c_b ← 0                                             ▷ start with minimum cost bias for fast computation
50:     T.C_s ← ∞                                             ▷ initial costs (first sub-optimality bound)
51:     loop
52:         ReinitializeRRT(T)
53:         T.C_n ← GrowRRT(T)                                ▷ grow next tree and store its costs
54:         if T.C_n ≠ nil then
55:             PostCurrentSolution(T)
56:             T.C_s ← (1 − ε_f) T.C_n                       ▷ decrease cost sub-optimality bound
57:             T.d_b ← T.d_b − δ_d                           ▷ decrease distance bias (slower exploration)
58:             if T.d_b < 0 then                             ▷ distance bias lower bound
59:                 T.d_b ← 0
60:             T.c_b ← T.c_b + δ_c                           ▷ increase cost bias (better solutions)
61:             if T.c_b > 1 then                             ▷ cost bias upper bound
62:                 T.c_b ← 1
```

Ferguson and Stentz [FS07] have built on their previous work combining ARRT and DRRT to develop their algorithm Anytime Dynamic RRT (ADRRT). ADRRT aims at environments (and possibly capabilities) with incomplete or imperfect knowledge that may change frequently. Furthermore, challenging deliberation times require the use of anytime refinement, and complex high-dimensional search spaces are to be approached with probabilistic sampling. ARRT (with ERRT) interleaves planning and execution according to the available deliberation time. Each successive anytime solution guarantees an improvement by at least a user-defined improvement factor. If changes occur in the environment, then the current anytime solution is repaired using DRRT by trimming the inconsistent (non-collision-free) branches of the generated tree. ADRRT have been shown to outperform their predecessors in constraint exploration experiments with self-driving cars in partially-known environments. ADRRT is shown in the annotated Algorithm 11 (with procedure definitions of ARRT).

Algorithm 11 Anytime Dynamic RRT [FS07]

```
1:  procedure MAIN
2:      T.d_b ← 1                                  ▷ start with maximum distance bias for rapid exploration
3:      T.c_b ← 0                                  ▷ start with minimum cost bias for fast computation
4:      T.C_s ← ∞                                  ▷ initial costs (first sub-optimality bound)
5:      loop
6:          ReinitializeRRT(T)
7:          T.C_n ← GrowRRT(T)                     ▷ grow next tree and store its costs
8:          if T.C_n ≠ nil then
9:              R ← T
10:             PostCurrentSolution(R)
11:             R.C_s ← T.C_s                      ▷ copy cost sub-optimality bound
12:             R.d_b ← T.d_b                      ▷ copy distance bias
13:             R.c_b ← T.c_b                      ▷ copy cost bias
14:             T.C_s ← (1 − ε_f) T.C_n            ▷ decrease cost sub-optimality bound
15:             T.d_b ← T.d_b − δ_d                ▷ decrease distance bias (slower exploration)
16:             if T.d_b < 0 then                  ▷ distance bias lower bound
17:                 T.d_b ← 0
18:             T.c_b ← T.c_b + δ_c                ▷ increase cost bias (better solutions)
19:             if T.c_b > 1 then                  ▷ cost bias upper bound
20:                 T.c_b ← 1
21:         for all new obstacle o in configuration space do
22:             InvalidateNodes(R, o)
23:         if significant obstacle changes were observed then
24:             increase R.C_s, R.d_b, and T.d_b   ▷ increase sub-optimality bound and distance bias
25:             decrease R.c_b and T.c_b           ▷ decrease cost bias
26:         if solution path of R contains an invalid node then
27:             RegrowRRT(R)
28:             PostCurrentSolution(R)
29:             T.C_s ← (1 − ε_f) T.C_n            ▷ decrease cost sub-optimality bound
```

2.1.5 Planning under Uncertainty

Explicitly modelling *uncertainty*, in particular, if possible scenarios or events are more likely than others, has been successfully applied in many advanced domains. Thrun, Burgard and Fox [TBF05] provide a comprehensive overview of the field of probabilistic robotics.

As mentioned in Section 2.1.1, uncertainty can be modelled with respect to actions that an agent performs as well as to the state an agent observes. Uncertainty with respect to the outcome of performed actions can be modelled as *and-or trees* and dealt with finding appropriate contingency plans (Russell and Norvig [RN10]). Another option is to compute a policy that specifies an action to be performed for all possible states. Policies can be computed from Markov Decision Processes (MDPs) employing some form of Dynamic Programming (DP). Mausam and Kolobov [MK12] discuss available techniques for solving MDPs in great detail. An example MDP is shown in Figure 2.9.

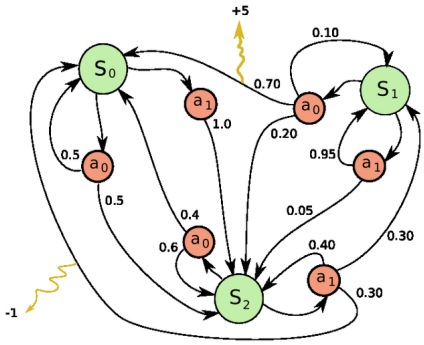


Figure 2.9: MDP Example[2]

The MDP framework assumes that only the outcome of actions is associated with uncertainty and that an agent's state can be determined with absolute certainty. In reality, however, mobile agents employ many sensors to observe and reason about their current state. If these observations are assumed to be imperfect and shall be modelled with their inherent uncertainty, the more sophisticated Partially Observable Markov Decision Process (POMDP) framework has to be used. Kaelbling, Littman and Cassandra [KLC95, KLC98] provide an overview of POMDP from an AI perspective. POMDPs generalize MDPs by introducing uncertainty with respect to observations. In this model, an agent's state is not known but *believed* in. By applying Bayesian inference, a probability

[2] https://en.wikipedia.org/wiki/Markov_decision_process

distribution over all states continuously evolves through gathering evidence via observations. The POMDP, thus, represents a so called belief MDP, that features uncertainty with respect to both states and actions. And example POMDP is shown in Figure 2.10.

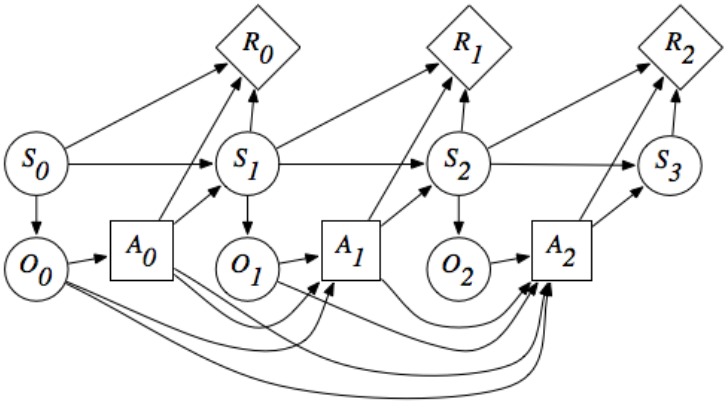


Figure 2.10: POMDP Example[3]

Solving MDPs online (interleaved) with anytime behavior requires techniques such as described by Barto, Bradtke and Singh [BBS95]. They developed Real-Time DP (RTDP) which uses asynchronous DP (Value Iteration (VI)) interleaved with the execution of the currently found best estimate for a policy. During the execution of an action according to that policy, several new backups may occur where the current state is always part of the subset of backup states. The selection of states to be backed up may strongly influence the convergence of the algorithm, and techniques such as priority-sweeping may apply a more focused state selection. However, in order to guarantee convergence of the policy, all states need eventually be backed up, which is achieved by performing several trials using different start states. Online and approximate algorithms for POMDPs are surveyed by Ross, Pineau, Paquet and Chaib [RPPC08].

A highly interesting approach to (motion) planning under uncertainty is described by Likhachev and Stentz [LS09]. They present the Probabilistic Planning with Clear Preferences (PPCP) algorithm which significantly reduces the computational complexity involved with planning in environments subject to uncertainties. Uncertainties are modelled with hidden variables where clear preferences regarding the value of these variables exist. This assumption applies for many real world planning problems. In aviation, for example, there are clear preferences with respect to weather, airspace clearances, traffic flow, available ground and space segment facilities, and aircraft performance.

[3] http://artint.info

The resulting model is a POMDP with its underlying belief MDP where each action can resolve at most one hidden variable. A resolved hidden variable can, however, affect the outcome of many actions. As mentioned before, there is a clear preference for each hidden variable.

Instead of solving the full belief MDP optimally, which is intractable and possesses an exponential growth depending on the number of hidden variables, several A*-like searches are performed based on assumed clear preferences for (a decreasing number of) unresolved hidden variables. Initially, all hidden variables are assumed to be unknown, assigned clear preferences and backward A*-planning is performed from goal to start (where start is a pivot state). Then, the underlying belief MDP is updated with the found optimal costs based on the assumptions made. A new pivot state is selected afterwards (with a reduced set of unresolved hidden variables) if its expected value is higher than the previously found lower bound. The selection of this pivot state can be based on the probability of this state or by backtracking to the first stochastic action in the updated belief MDP. Backward A*-planning is then continuously repeated with the respective new pivot state as long as costs are likely to increase.

The main difference to standard A* is the definition of the cost function which is based on MDP Q-values but incorporates clear preferences for unresolved hidden variables. Since PPCP is based on A*, many of the A* extensions, including anytime, incremental and any-angle variants, could adapt the approach to a specific domain. Some of the extensions are discussed in the paper. Mobile robot path-clearance and navigation in partially known terrain are the applications discussed as examples.

Hansen and Zilberstein [HZ01] combine heuristic search and DP to efficiently solve MDPs. Their approach avoids searching the entire state space. The heuristic search algorithm that serves as a starting point is A* for Acyclic Graphs (AO*), an A* for acyclic graphs, such as trees (e.g., and-or trees). Like A*, AO* expands states that are most promising with respect to already discovered costs and future expected costs based on an admissible heuristic. If actions have probabilistic outcomes, expanding the most probable states would be an alternative expansion policy. This *forward expansion step* is followed by a *backward induction step* (cost revision step). The latter updates the costs of all possible predecessors that can reach the expanded *non-terminal tip state* by applying the currently best solution (best *marked action* for each state) without updating all other states resulting in improved efficiency. This backward induction step can be seen as a single *Bellman backup* for DP in a graph without loops.

Hansen and Zilberstein then generalize the ideas of branch-and-bound forward expansion and DP backward propagation (cost revision) in acyclic (probabilistic) graphs to cyclic graphs, such as MDPs with indefinite horizon. Their Loop AO* (LAO*) algorithm replaces the single backward induction (cost revision) step with a full DP policy or value iteration to account for loops. Hence, to find the optimal solution, forward expansion guided by an admissible heuristic is interleaved with DP iterations to revise costs. However, only affected states (using marked actions) are being updated in contrast to computing a policy for every state (those that are not part of an optimal solution). LAO* can be extended like A* to incorporate consistent (probabilistic) heuristics and weighted (inflated) heuristics to improve efficiency and adjust a sub-optimality bound for the solution.

Ferguson, Stentz and Thrun also present an extension to AO*, called Propagating AO* (PAO*), for decision problems with hidden state. For their purpose, a static environment contains uncertainty with respect to the traversability of a finite number of so called *pinch points*. These points can turn out to be either an obstacle or traversable. There is a probability distribution assigned to each pinch point over these two options. As with AO*, an explicit solution graph is constructed linking the *faces* of the pinch points with arcs directly in between or passing through them. Forward expansion is guided by a heuristic as with AO*. Backward induction (cost revision) is modified to propagate updates not only to parent states but also to neighbors and children in order to distribute discovered information more quickly.

2.2 Autonomous Flight and Guidance

Planning and decision making in the context of autonomous flight and guidance has been surveyed by Goerzen, Kong and Mettler [GKM10]. The authors give a broad overview of problems, methods and algorithms that apply to motion planning for UAV. Their survey categorizes approaches to UAV motion planning into *path planning* without differential constraints and *trajectory planning* with differential constraints. While the first category is dominated by ideas originating in robotics and AI, the second category incorporates more control theory and fuses both streams.

Path and trajectory planning can go hand-in-hand or otherwise be realized in a *decoupled* fashion. Decoupled approaches include the *two-step* method where a path is generated first, which is then post-processed and smoothed to account for the nonholonomic and kinodynamic constraints of the vehicle. A practically very relevant and

clean architecture is the *hierarchical decoupled* planning and control method where a hierarchy of controllers guide the system. *Inner loops* realize low-level optimal control (e.g., pitch, roll and yaw) while *outer loops* keep higher-level targets within limits (e.g., heading, altitude and track). This hierarchy is very similar to the concepts of *autonomic manager* and *managed element* in autonomic computing, as discussed in Section 2.3. Consequently, higher levels of control that apply and adapt motion planning algorithms to generate navigation waypoints are very easy to integrate into such an architecture. However, any generated sequence of waypoints has to be at least kinodynamically feasible for lower-level controllers (or otherwise controllers may fail). Hence, any-angle, PRM and RRT algorithms, presented in Sections 2.1.3 and 2.1.4, should either incorporate some form of costs that reflect kinodynamic constraints appropriately or integrate them directly. Since RRTs have been developed with these constraints in mind, they have gained much attention in the robotics domain (e.g., by Hsu et al. [HKLR02], Jaillet et al. [JHvdB+11], Webb et al. [WvdB13]).

A second survey by Dadkhah and Mettler [DM12] gives an overview of the relevant literature for UAV motion planning under uncertainty. Following the organization of the previously mentioned survey, the authors consider approaches from both the robotics and AI, and the dynamical systems and controls communities. They address uncertainties related to vehicle dynamics, environment and environmental disturbances, and pose.

Both surveys agree on the high complexity and interdisciplinary character of the field concluding that specific problems continue to require specific solutions and extensive benchmarking. However, the potential value of a self-adaptive approach to motion planning algorithm selection and tuning is not explicitly mentioned anywhere. Another reference style book by Sebbane [Seb14] includes a more extensive discussion on the topics discussed in both surveys and, thus, may serve as a good starting point for more detailed research.

The existing literature on UAV motion planning applications is unmanageably vast and features several approaches, including, but not limited to, the ones presented in Sections 2.1.3, 2.1.4 and 2.1.5. Apart from using different combinations and optimizations of these approaches, other streams apply learning and evolutionary algorithms, such as described by Rathbun, Kragelund and Pongpunwattana [RDKP02]. Their concept of evolutionary path planning is shown in Figure 2.11.

Depending on the applicable planning environment and horizon as well as the required agility for the UAV, very different techniques have been used successfully. The

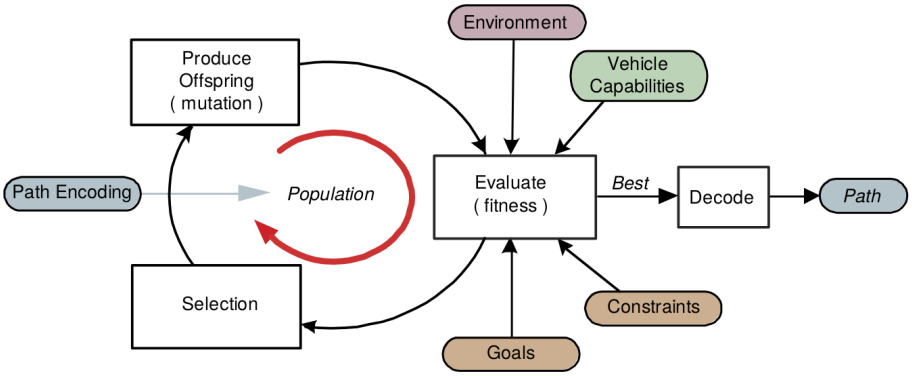


Figure 2.11: Evolutionary (Genetic) Path Planning [RDKP02]

following paragraphs summarize only a few of the most closely related publications that have been identified so far.

Wzorek et al. [WCR+06] implement the idea of helicopter *push-button missions*, letting the system itself plan and execute complex missions with as little effort as possible required from the ground operator, other than stating mission goals at a high-level of abstraction, and monitoring the execution of the ensuing mission.

Their architecture consists of three layers being *control*, *reactive* and *deliberative*. The hard real-time control kernel is required to perform tasks with hard deadlines. A reactive control part copes with already modelled changes of the controlled system. The non real-time deliberation part performs long-term decision-making. The controllers are realized using event-driven hierarchical state machines. Higher-level so called *task procedures* realize (distributed) reactive tasks.

The control kernel coordinates the real-time communication between all three on-board systems as well as with other cooperating robotic systems enabling multi-platform missions. The applied dynamic path planning algorithms are PRMs and RRTs. Furthermore, A* is used for plan optimization. Path planning requires a 3D world model to avoid collisions where a Geo-Information System (GIS) is used by the group, but Simultaneous Localization and Mapping (SLAM) using additional on-board sensors is proposed. The static model can then be updated by dynamic changes (e.g., no-fly zones). The generated PRM obeys runtime constraints, including altitude and airspace restrictions.

A path following mode is divided into a *trajectory generator* and a *path controller*. Dynamic path planning supports different strategies that are mainly aimed at collision

avoidance. The strategies, which are kept in a *strategy library*, allow for trade-offs with respect to path optimality and computational costs.

Wzorek and Doherty [WD06] also provide more details about their reconfigurable planning approach. For the PRM, an *OBB-Tree algorithm*[4] is used for collision checking and A* is used for graph search considering, for example, the shortest distance path and minimum fuel usage. The authors extend PRM with *multi-level planning* inserting cubic curves in order to allow smooth movements at higher speed and when hovering would be undesirable. PRM is, furthermore, extended with runtime constraint handling which is solved using A* to deal with altitude, airspace and vertical speed restrictions. The generated cubic splines are being followed by a *dynamic path following* controller.

New trajectory segments are requested online by the path following controller from the trajectory generator, and if a segment cannot be planned in time (including a breaking time), the helicopter UAV starts to hover. The addition of runtime constraints leads to the invocation of a *strategy selector service* for re-planning that considers the estimated time available and required for the selected re-planning strategy.

Wzorek, Kvarnström and Doherty [WKD10], furthermore, propose machine learning, and, in particular, Support Vector Machiness (SVMs) to select a strategy for dynamic replanning. The available strategies differ in complexity and time requirements. The available deliberation time depends on the distance to obstacles to be avoided as well as on the velocity of the UAV. Furthermore, the time and space required to change the trajectory or break (hover/orbit) have to be taken into account.

Wu, Campbell and Merz [WCM09] apply Multi-Criteria Decision Making (MCDM) to account for most of the applicable factors leading to dynamic path re-planning decisions in the typical Visual Flight Rules (VFR) airspaces E to G. They fuse multi-dimensional decision variables to a single value (reducing dimensions) and allow neighbors in a grid to be connected by vectors directly and, hence, not being restricted to the adjacent cells. This approach allows for finer track angles and can better compensate for wind effects. Gridded wind information maps are being used that are publicly available to aviation personnel. Digital airspace maps, encoding the relevant airspace structure with geometrical shapes, are used for trajectory planning avoiding the controlled airspaces A to D. Minimum altitudes and terrain elevations are taken into account using digital elevation maps. Preferred VFR cruising levels are considered as well as the potential risk to the population of the overflown area.

[4] https://en.wikipedia.org/wiki/Minimum_bounding_box

The authors look at the option of including filed flight plans and Automatic Dependent Surveillance-Broadcast (ADS-B) equipped agents to address lateral and vertical separation requirements and recommendations. They use a Gaussian distribution to model the probabilistic position error of conflicting agents. Critical weather and, in particular, storm cells that introduce additional uncertainty are treated and modelled similarly to conflicting traffic. The authors also consider the aircraft fuel consumption and its affecting factors, such as engine setup and flight altitude.

Algorithmically, A* with MCDM is used to derive a cost function for the remaining goal. Dynamic changes during the flight that lead to re-planning will invoke A* again deriving a new plan from scratch. To address the different weights of individual decision variables when deriving a cost function, Multi-Attribute Utility Theory (MAUT) is applied.

Wu, Campbell and Merz [WCM11], furthermore, extend their previous approach by introducing a *multi-step* A* algorithm for offline and online path planning that improves the performance of the pure vector-neighborhood A* by a factor of four. Optimality of their A* variant is proven, and the algorithm is applied to a multi-resolution $3D$ lattice structure. The lattice structure reduces the computational complexity by introducing different grid resolutions for planning. Larger distances between the lattice planes reduce the sample space and increase the track angle resolution but at the same time increase the number of possible successor states. The authors also consider the use of an anytime A* variant in their future work (a follow-up has not been published so far).

Garcia, Viguria and Ollero [GVO13] apply the path planning (and shaping) algorithms A*, A*PS, Θ^* and dynamic Lazy Θ^* to weather avoidance scenarios in a 2D and 3D grid. Dynamic Lazy Θ^* (LT*) seems to cope best with the state complexity and dynamic changes in weather during the flight. Replanning is based on the estimated position after the current planning episode. The estimated position depends on the execution time of the algorithm (and obviously on the velocity of the vehicle). The details for this position estimation are not given. Experiments have been performed with quadcopters in a lab environment with obstacles.

Filippis, Guglieri and Quagliotti [dFGQ12] evaluate Θ^* for path planning of a fixed wing UAV and confirm the advantages of Θ^* over A*. Their navigation problem consists of complex 3D maps containing tens of thousands of nodes and static obstacles to be avoided. Θ^* produces smoother paths that require significantly less heading changes and shorter distances. They, furthermore, post-process the resulting paths to account for the kinodynamic constraints of the UAV.

None of the literature examined so far, has employed different algorithms for different situations, such as flight phases, diversions or emergencies, possibly tuning parameters online. Furthermore, no approach explores the opportunities of datalink communication. Automatic communication with human actors, such as ATC, can be seen as a form of *sensing* the traversability of the environment. Existing protocols, such as CPDLC, could be exploited to automatically request clearances (e.g., for controlled airspaces), and assumptions (or clear preferences) could be established regarding the outcome of a communication (e.g., clearance confirmation or denial) depending on the current situation (e.g., reduced radius of action in an emergency).

2.3 Self-Adaptive and Self-Managing Systems

Autonomous systems observe their environment or *context* as well as their internal *self* in order to adapt their behavior and perform their duties in the best possible way. They are strongly related to self-adaptive systems and their software can be described following Laddaga [Lad06]:

> "Self-adaptive software evaluates its own behavior and changes behavior when the evaluation indicates that is is not accomplishing what the software is intended to do, or when better functionality or performance is possible."

Self-adaptive software *monitors* its internal *self-properties* and external *context*, *detects* and *evaluates* the consequences of internal and external changes—possibly predicting future observations—and *decides* on how to *act* adaptively in order to improve its own performance. This *adaptation loop* has been defined in slightly different variants, but its core idea and components are very similar and shown in Figure 2.12 as presented by Salehie and Tahvildari [ST09].

In autonomic computing, this loop is extended and referred to as MAPE-K loop where the *knowledge base* that collects data about the managed element and its environment plays an essential role for the successful autonomic management (Kephart and Chess [KC03]). Multiple layers of (possibly distributed) autonomic managers can address a hierarchy of adaptation issues while optimizing for higher-level goals.

Applying the self-adaptive feedback loop to a control problem may be confusing at first since the software itself performs its functions as part of a similar or even shared control loop. Hence, it is important to separate between the self-adaptive feedback loop on one level and the functional control loop in such systems on the other. The goal of the

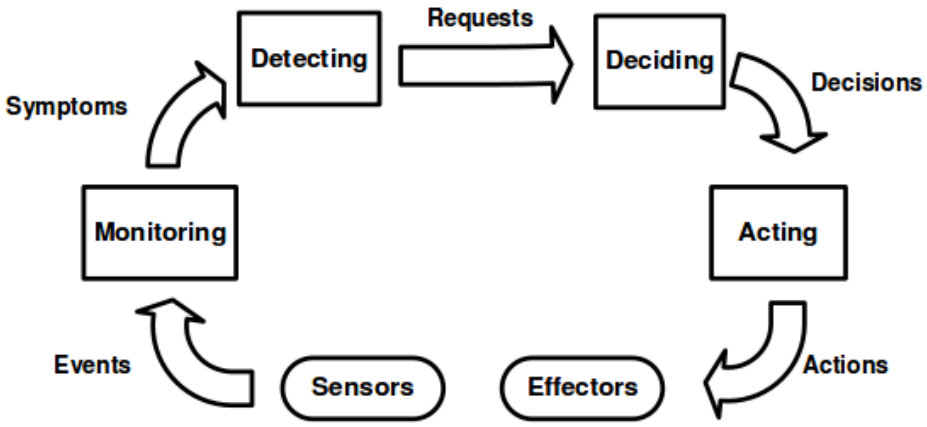


Figure 2.12: Adaptation Loop [ST09]

self-adaptive software engineering approach is to establish a better *separation of concerns*, that is, achieving the required functionality within a domain versus the dynamic adaptation required to perform well in a different (dynamically changing) domain (Calinescu, Ghezzi, Kwiatkowska and Mirandola [CGKM12]).

Self-adaptive software reasons about itself (its *self-properties*) in a domain (its *context*) and adapts its structure or behavior to achieve optimal performance. This *self-* and *context-awareness* form the basis of *self-adaptiveness* and the resulting hierarchy is depicted in Figure 2.13 following Salehie and Tahvildari [ST09].

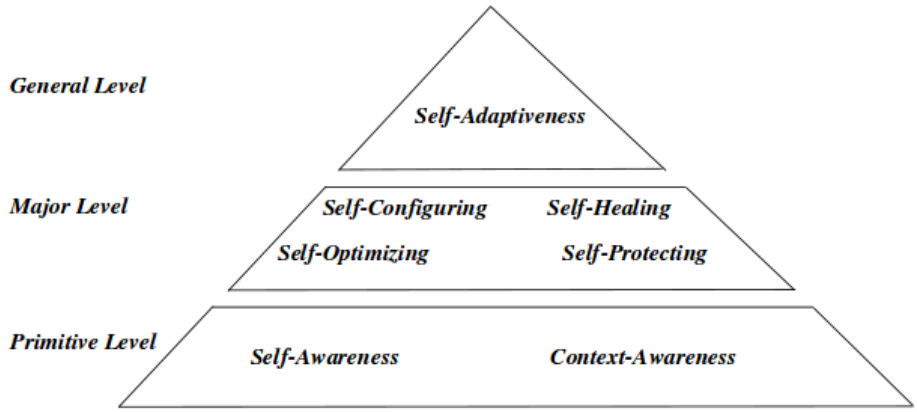


Figure 2.13: Self-Adaptiveness [ST09]

Adaptation happens dynamically during *runtime* and may be realized on different architectural abstraction levels, employing different techniques while focusing on different aspects and goals. Being self- and context-aware, systems may focus their adaptation efforts on any combination of

- Self-Configuration,
- Self-Optimization (Self-Tuning and Self-Adjusting),
- Self-Healing (Self-Diagnosing and Self-Repairing), and
- Self-Protecting.

A system may focus on re-configuring its structure by replacing a particular component in order to optimize its functionality in a dynamically changing environment but ignoring any aspects that involve self-healing or self-protecting. Another system may entirely focus on self-protection instead and remain self-static otherwise.

For example, aircraft systems may be configured for a flight using a direct route from a departure to a destination aerodrome employing the best engine *performance* parameters to minimize the flight time while trading in a higher fuel consumption. Bad weather, dense traffic or restricted airspace may force the aircraft to adapt its engine configuration accordingly to best *range* or best *endurance* following a different route instead. A new controller component could keep the aircraft within its currently required envelope. This component could even be further refined while in use, tuning individual parameters, verifying its integrity and simulating its behavior before deploying it while the system is kept operational. Possibly other aircraft in the area could publish their optimized controllers for the same environment, which leads to a *multi-agent* self-adaptive system. Although self-healing and self-protecting mechanisms are certainly relevant for this example, they do not necessarily need to be realized.

The different techniques that can be used to realize self-adaptation have been summarized by Oreizy, Medvidovic and Taylor [OMT98] and are shown in Figure 2.14.

Conditional expressions are the most basic and static form of adaptation where self-adaptation is mixed with functionality and often referred to as *moding*. The software may adapt to its self-properties and contextual information but is restricted to what modes have been specified before runtime. This approach does not address the separation of concerns and may not be flexible enough in order to cope with unexpected changes within the system itself or its context.

Online algorithms address the potential uncertainty of the self-properties and context of their software system. They continue to leverage knowledge in order to choose a more appropriate behavior in the future. These behavioral changes may not be deterministic but could be random or probabilistic. Assumptions are made based on the past that may or may not lead to the intended improvement in the future. User interfaces

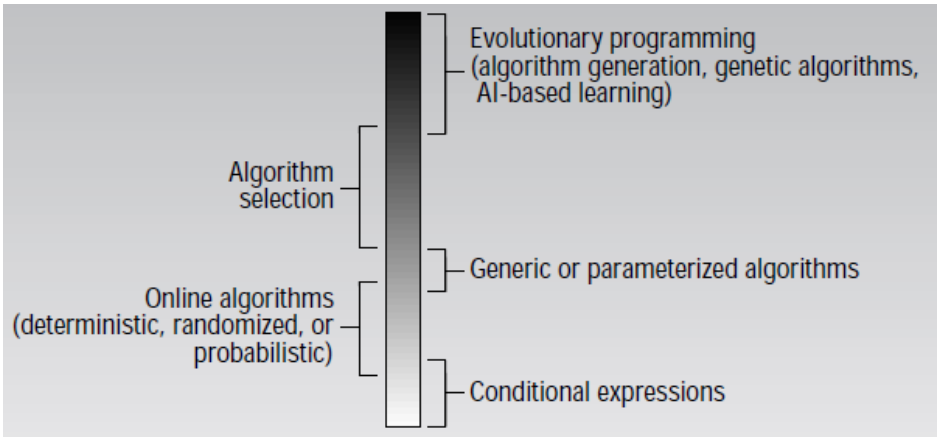


Figure 2.14: Techniques for Self-Adaptation [OMT98]

that adapt their layout to the preferences of their users are an example for this kind of adaptation.

Generic or parameterized algorithms can be employed to cope with and, hence, adapt to different kinds of data that has to be processed in a similar fashion. Container classes and their methods that are available in many higher level programming language libraries are an example for adaptation using generics not requiring any detailed knowledge of the concrete data type to be handled.

Algorithm selection allows for different and more appropriate algorithms to be employed for a particular problem. The set of available algorithms is fix and does not change during runtime and the selection of different algorithms may go hand in hand with the selection of different hardware. For example, different motion planning algorithms and their parameters may be adapted dynamically at runtime in order to achieve the required navigational performance.

Evolutionary programming represents the most flexible approach to self-adaptiveness and addresses the desired separation of concerns appropriately. With this approach, services or algorithms are not necessarily known or available a priori but instead could be made available and discovered during runtime whenever needed. Entire building blocks of an application may be replaced over time and its architectural structure may change as a result. Software components could be generated by the system (agent) itself or exchanged with other systems (agents) forming a distributed multi-agent environment. The decisions, cooperation and competition that lead to self-adaptation may involve artificial intelligence methods, such as learning, decision making and game theory (Shoham and Leyton-Brown [SLB09]). Relevant system goals

could be specified on a high level of abstraction, cover a longer period of time and be broken down by the participating agents.

2.4 Chapter Summary

In this chapter we briefly covered some of the most relevant streams of research directly applicable to certain components of our SAFCS. We surveyed some important ideas in the fields of *Planning and Decision Making*, *Autonomous Flight and Guidance*, and *Self-Adaptive and Self-Managing Systems*. In particular, we first reviewed the relavant AI terminology for autonomous trajectory planning and then discussed serveral algorithmic approaches. Most notably, we presented the three planning families A* Algorithms, Probabilistic Roadmaps, and Rapidly-Exploring Random Trees explaining their basic ideas as well as their more sophisticated *anytime*, *dynamic*, *any-angle*, and *online* extensions. We, furthermore, discussed some related practical applications and finally presented suitable design patterns that address the separation of concerns for self-adaptive and self-managing systems.

Chapter 3

State of the Art

Any airborne mission traditionally encompasses a thorough preparation phase. However, data that are available before the mission commences are not necessarily complete nor valid during the entire flight. Whatever plan is established before the flight is quite likely to be revised during its execution according to the actual situation. Today's air transport operations rely heavily on radio communication to exchange information during the flight. The principal actors involved in this communication and the resulting decision-making process are depicted in Figure 3.1.

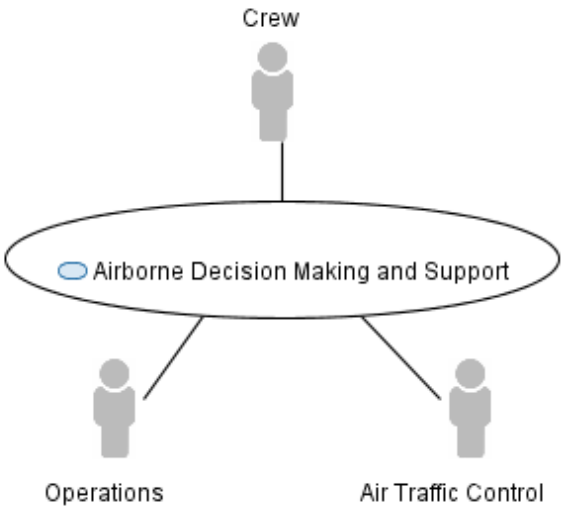


Figure 3.1: Smart Autoflight Control System Actors

The air crew has the final authority over any actions that are being implemented aboard the aircraft. Their actions are, however, constrained by business rules and Standard Operating Procedures (SOPs) of their operation. For example, a civil operation

follows very different operational constraints compared to a military one. The acceptable levels of risk and safety during the operation clearly depend on its type. During the mission, the ATC authorities are involved if the flight penetrates controlled airspace. For example, clearances may have to be obtained to ensure proper separation between and an efficient flow of airspace users. There are certainly other actors involved in the bigger picture, such as passengers, Flight Information Centers (FICs), Air Traffic Flow Management (ATFM), and Search and Rescue Centers (SARCs), but these can also be seen as resources of the depicted actors and are not directly involved in the airborne decision-making process.

The actions of human actors are always subject to *bounded rationality*[1] and *bounded reactivity*, as described by Simon [Sim82], Gigerenzer and Selten [GS02]. The first property essentially means that the quality of a human rational decision is bounded by the actor's performance, the information, and the deliberation time available for the decision-making process. The second property expresses that, even if some plan has already been established, an appropriate reaction might not be feasible in time; instantaneous failure is always a possibility. The same concepts can be assumed for any machine actor realizing higher levels of autonomy in this decision-making process. The machine actor's performance clearly depends on its employed hardware, software (e.g., algorithmic complexity), sensing and actuating capabilities, the completeness, correctness, and recency of the obtained data, and the deliberation time available to derive useful information and implement decisions. Ideally, fallback procedures are available to contain some instantaneous failures, similar to emergency checklists for air crews. Rationality has to be sacrificed to obtain satisfactory but not necessarily optimal solutions in many situations: a *satisficing*[2] strategy.

Considering the increasing amount, diversity, and frequency of available data for decision making, as described in Section 3.1, the rational and reactive limitations of human actors using radio and raw data communications only is rather obvious. To cope with the ever-increasing requirements of the aviation industry with respect to growth, efficiency, and safety, autonomous rational agents eventually have to derive, present, and implement decision-making options. Human actors will then choose, acknowledge, or simply monitor the outcome of a mission. The architecture described in this dissertation employs extensions to state-of-the-art algorithms to compute and support decisions for airborne trajectory planning. The final key set of algorithms acknowledges

[1] https://en.wikipedia.org/wiki/Bounded_rationality
[2] https://en.wikipedia.org/wiki/Satisficing

the limitations of bounded rationality and belongs to the class of *anytime* algorithms, as described by Zilberstein [Zil96]. They, furthermore, realize the requirements for *dynamic*, *lifelong*, and *online* planning, as detailed in Chapters 6, 7, and 8. In general, they are guided by heuristics to avoid a full state-space exploration, improve computed trajectories incrementally (ideally ensuring a bounded sub-optimality), and repair inconsistencies efficiently.

3.1 The SWIM Infrastructure

The data relevant for airborne decision making are manifold and encompass aeronautical, flight, weather, aerodrome, and maintenance data. Currently, these data are exchanged between aviation stakeholders employing proprietary and incompatible protocols, as illustrated in Figure 3.2. As a result, the availability and recency of data for interested parties are very limited due to the increased effort in sharing them.

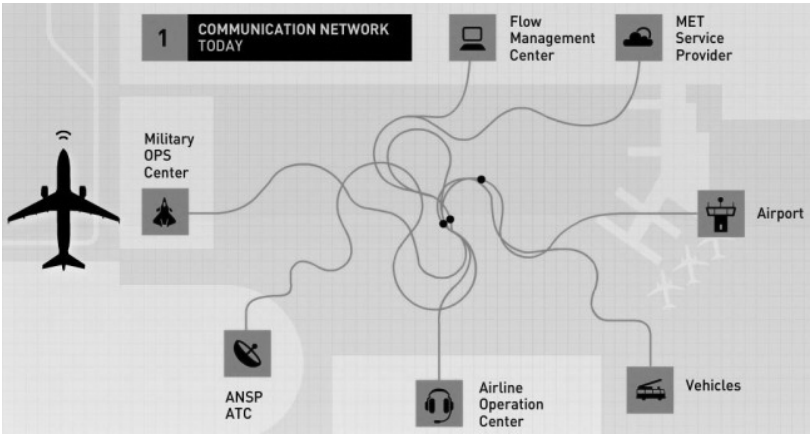


Figure 3.2: Present Information Management[3]

The FAA NextGen SWIM program and Eurocontrol actively drive the development of an open-standard, open-services infrastructure to publish and subscribe aviation data with the potential to substitute and obsolete existing distribution methods in the near future. XML-based schemas have been developed and are being continuously assessed and improved to describe the data mentioned before. These include the Aeronautical Information Exchange Model (AIXM),[4] the Flight Information Exchange Model (FIXM),[5]

[3] https://www.eurocontrol.int/information-management/swim-jumpstart
[4] https://www.aixm.aero/
[5] https://www.fixm.aero/

the International Civil Aviation Organization (ICAO) IWXXM and FAA Weather Information Exchange Model (WXXM),[6] the Aerodrome Mapping Exchange Model (AMXM),[7] and the Maintenance Management Information Exchange Model (MMIXM).[8]

Table 3.1: SWIM Data Information Exchange Models

AIXM	Aeronautical Information Exchange Model
FIXM	Flight Information Exchange Model
IWXXM / WXXM	Weather Information Exchange Model
AMXM	Aerodrome Information Exchange Model
MMIXM	Maintenance Management Information Exchange Model

For case of reference, these information exchange models are collected in Table 3.1. For example, the airspace structure, including active, inactive, or temporarily active airspaces, as well as available ground navigation facilities are part of the AIXM. Planned flights and trajectories, including very detailed data about the aircraft involved, are part of the FIXM. Fronts, clouds, precipitation, obscuration, or any significant weather and related threats, such as thunderstorms, icing, and turbulence, are part of the WXXM/IWXXM. The layout of an aerodrome, including its runways, taxiways, ramps, and gates, is part of the AMXM. Maintenance schedules and facilities are part of the MMIXM. The different topics of SWIM data can be published and subscribed through FAA servers using an implementation of the Jakarta Messaging[9] specification or via cloud-based platforms such as Cirium.[10]

The improved SWIM infrastructure provides common access to standardized data, as illustrated in Figure 3.3. It enables the development of intelligent systems by allowing the different stakeholders to subscribe, filter, fuse, and learn from applicable data sources. Examples of such intelligent systems include aviation weather observation and forecast systems such as the Integrated Terminal Weather System (ITWS),[12] air traffic flow and control systems supporting Initial 4D Trajectory (I4D)[13] operations,[14] airline and aerodrome smart planning and logistic systems, and flight crew decision-making

[6] https://www.wxxm.aero/
[7] https://www.amxm.aero/
[8] https://mmixm.aero/
[9] https://jakarta.ee/specifications/messaging/
[10] https://www.cirium.com
[11] https://www.eurocontrol.int/information-management/swim-jumpstart
[12] https://www.faa.gov/air_traffic/technology/itws/
[13] https://www.skybrary.aero/index.php/4D_Trajectory_Concept
[14] http://www.sesarju.eu/i4D

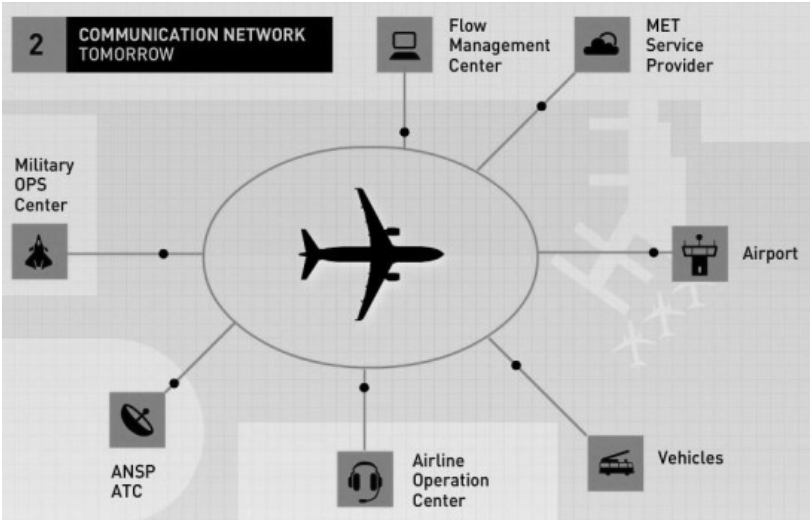


Figure 3.3: System-Wide Information Management[11]

and support systems such as the National Aeronautics and Space Administration (NASA) Traffic Aware Strategic Aircrew Requests (TASAR) (Wing et al. [Win19, WBB+19]) and our evolving SAFCS (Heinemann, Müller and Suleman [HMS18]). We have provided the SWIM community with Java Extensible Markup Language (XML) bindings[15] that allow for the convenient marshalling and unmarshalling of SWIM data supporting many of the available extensions. We use these bindings in our evolving SAFCS architecture presented in Part II of this dissertation.

3.2 The NASA TASAR Project

The NASA TASAR project (Wing et al. [BW12, Win19]) is conceptually closely related to our SAFCS and has been a successful milestone toward a broader understanding of the possibilities with respect to future autonomous flight within the evolving NextGen National Airspace System (NAS) and beyond. TASAR is a concept for cockpit-based trajectory management automation featuring advisory capability without necessarily performing safety-critical functions in its initial development stages. It can certainly be considered state-of-the-art regarding its vision, planning, implementation, and evaluation.

As its name implies, TASAR embraces three important concepts being

- Traffic Awareness and Separation,

[15] http://www.aixm.aero/page/open-source-projects

- Strategic Optimization and Decision-Support, and
- Aircrew Requests.

All three components have a common goal and a broader vision in mind that eventually is projected to lead to a higher airspace utilization, reduction of required resources and associated costs contributing to a lower overall environmental impact.

First, traffic awareness is an essential capability to enable a safe and efficient *self-separation*. Second, strategic optimization and decision-support relate to the possibilities of disrupting technological advances in the area of *connected aviation*. Finally, aircrew requests emphasizes the concept of *humans in the loop* with respect to the involved actors, such as cockpit crews, ATC and OPS.

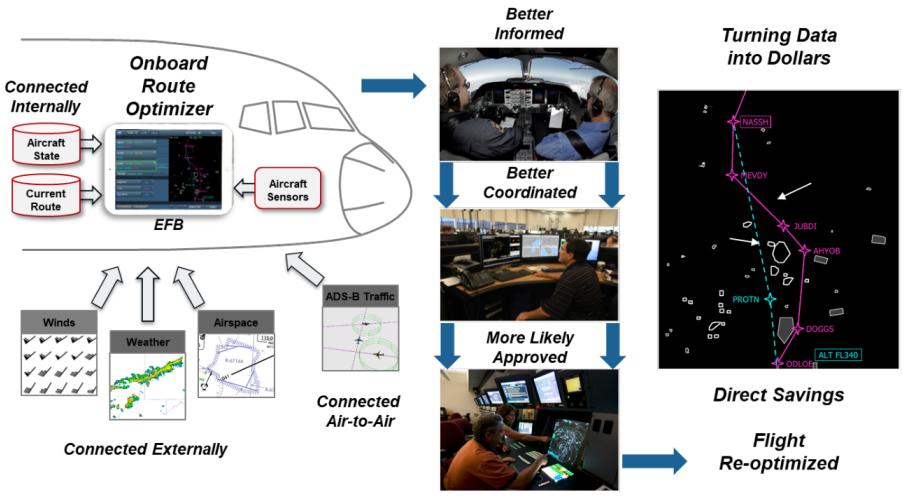


Figure 3.4: The TASAR Concept [Win19]

The TASAR concept is depicted in Figure 3.4. Its core component is an onboard route optimizer which is connected to both the onboard avionics systems as well as to external data sources. These connections enable the system to monitor and act on observed aircraft and navigational performance within a highly dynamic environment. By better informing cockpit crews about potential optimizations derived from these data sources, an improved integration with and higher acceptance of requested trajectory changes by ATC becomes more likely.

Where cockpit crews would otherwise not be aware of the operational implications of and potential optimizations for dynamic changes in their environment, TASAR works in the background collecting data and deriving useful information to keep pilots engaged and potentially avoiding complacency in longer enroute flight phases. Keeping

its human actors in the loop, TASAR continues to apply the traditional *request-check-approve* paradigm which allows cockpit crews to ignore suggested optimizations or else accept them and request approval from ATC.

3.2.1 Project History and Vision

Starting in the 1990s, the vision of and research for higher levels of *operational autonomy* for airspace users leading to increased *self-separation* and *self-governance* was the main driver for preceding programs and related technology demonstrators at NASA. Apart from the initial goal in the 1990s to early 2000s to triple the airspace utilization, a long term vision of a harmonious integration of autonomic and unmanned flight operations into the existing system and its established procedures became more apparent.

Traditionally, two sets of rules govern the operations in the skies and co-exists with each other: VFR operations require airspace users to maintain Visual Meteorological Conditions (VMC) and be responsible for their own trajectory planning and separation between each other. IFR operations, on the other hand, rely heavily on ATC, both while operating in VMC as well as in Instrument Meteorological Conditions (IMC). In controlled airspace, where radar surveillance is an option, ATC issues instructions and grants clearances from the point of departure to the final destination. Since the IFR system is historically based on ground-based navigation facilities, a route and procedural system has been developed that connects these navigational aids and accounts for the different requirements during different flight phases, including departure, enroute, arrival and approach. This system permits human ATC actors to supervise and control flow and separation of traffic in different sectors but leads to a concentration of traffic at the intersections of the enroute system.

With the advent of Global Navigation Satellite System (GNSS) navigation, datalink and ADS-B connectivity, a more flexible approach would technically be possible but is hard to achieve in practice due to the human limitations of the involved actors. The ATC system is still restricted by the cognitive capacity of human controllers to track and communicate with each aircraft in their sector. Consequently, only rather simple and obvious modifications to initially planned aircraft trajectories are often made that in some cases enable ATC to better handle the existing workload but are not necessarily beneficial for the aircraft operators themselves. Hence, GNSS, datalink, and ADS-B currently serve to increase the SA and facilitate human decision-making, but their full potential in terms of operational *self-separation* and *self-governance* is far from being fully exploited.

Wing and Cotton [WC11], therefore, proposed to extend the existing rule set by a third one addressing the current gap and pave the way toward future aircraft operations: the Autonomous Flight Rules (AFR).

The TASAR project attempts to push the boundaries of the current system one step closer to that final vision. The NASA TASAR team was able to answer many of the ensuing research questions over a 12-year period, including experiments that have verified the scalable-capacity characteristics of AFR by demonstrating safe separation at traffic densities of five times the current levels and above. Their simulations and flight tests focused on non-safety-critical elements of the future technology that would meet a current need and could be field-tested easily without disruptions of the existing SOPs in place. The technology would then serve as a basic infrastructure to overcome initial socio-technical obstacles and mature through operational use leading to new and potentially safety-critical applications.

Following simulations and initial flight trials, the operational evaluation of the TASAR concept was prepared with two engaged partner airlines, including Virgin America and Alaska Airlines (Wing, Burke, Ballard, Hendersen, Woodward [WBB+19]) representing a low-cost and network carrier, respectively. Following the merger of both partners, the collaboration continued with the final integration and evaluation on Boeing 737 – 900 transport category aircraft during scheduled revenue flights. The positive reception and results of this new technology helped to create the necessary initial momentum in the aviation community toward a future operational autonomy and demonstrated how data can be turned into dollars. The TASAR project concluded with its final prototype implementation in 2019.

3.2.2 Technical Aspects

The realization of the TASAR concept is the Traffic Aware Planner (TAP)—an implementation that is derived from a previous NASA project called Autonomous Operations Planner (AOP). TAP is an Electronic Flight Bag (EFB) application that integrates with the aircraft avionics system in a read-only fashion to enhance the cockpit-crew's SA by suggesting route optimizations in terms of time, fuel or a weighted combination of time and fuel. The avionics integration is achieved by means of standard Aeronautical Radio Incorporated (ARINC) or General Aviation Manufacturers Association (GAMA) protocols. Being a read-only system that requires human action and interaction to follow the suggested trajectory optimizations, it is considered non-safety critical and, therefore, low-

ers the certification requirements for field testing substantially. It is classified as a Type B EFB application that generally produces dynamic data with algorithms that must be tested for accuracy and reliability by the applicant. Furthermore, TAP is intended to be used outside of terminal airspace during the climb (above 10,000 feet for jet aircraft operators) and cruise portions of the flight, that is, during non-critical flight phases only.

The TAP prototype architecture is shown in Figure 3.5.

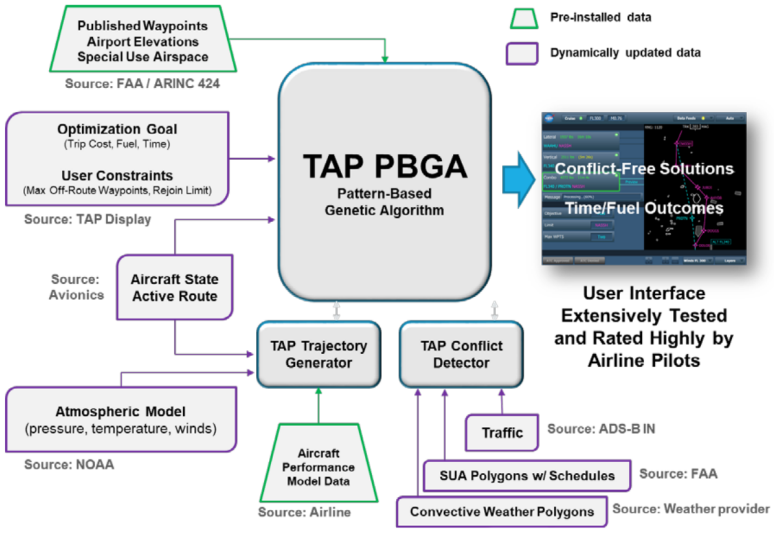


Figure 3.5: The TAP Prototype [Win19]

The principal TAP components are

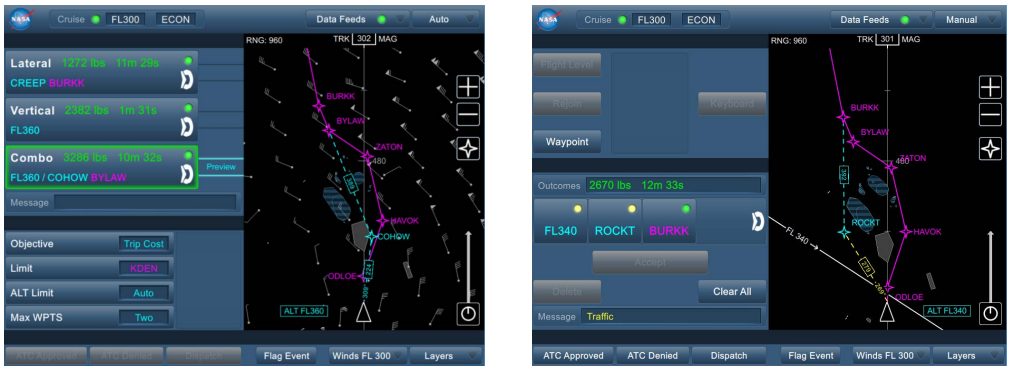
- the TAP Trajectory Generator,
- the TAP Conflict Detector, and
- the TAP Pattern-Based Genetic Algortihm (PBGA).

Where the TAP conflict detector and PBGA were developed for and inherited from the AOP, the TAP trajectory generator represents the addition to the newer TAP architecture. The three components connect to data sources supplying the system with operational constraints, aircraft capabilities, and the aircraft environment.

The conflict detection algorithm considers traffic, convectional weather and Special Use Airspaces (SUAs) and is designed to incorporate the movement intent of the ownship. The relevant data can be sourced from ADS-B but may very well be extended to leverage the evolving SWIM infrastructure explained in Section 3.1. The trajectory generation algorithm considers the performance of the aircraft within its environment and

generates possible trajectory candidates that are fed into the TAP PBGA. The patterns applied by the PBGA are preferred and feasible lateral and vertical maneuvers to generate the new offspring generations of trajectories which are assessed and selected by a fitness function to obtain an optimized solution, as detailed by Vivona, Karr and Roscoe [VKR06]. The suggested maneuvers are lateral, vertical or a combination thereof in order to provide the cockpit crew with more than just one option for improvement.

In summary, the TAP monitors, detects, and resolves conflicts while optimizing trajectories according to established operational constraints. It is, therefore, a key enabling technology toward AFR incorporating traffic surveillance, information connectivity, and trajectory-management automation. Once the application is active, it displays route optimization suggestions on the EFB User Interface (UI). As an optional, advisory-only decision support tool, TAP notifications and suggestions can be ignored or else assessed by the cockpit crew and subsequently be requested from ATC in order to improve operational efficiency during flight operations.



(a) TAP Auto Mode **(b)** TAP Manual Mode

Figure 3.6: The TAP Operational Modes [Win19]

As shown in Figure 3.6, TAP can, furthermore, be used in two different modes: automatic and manual. In addition to being notified about potential trajectory optimizations, the cockpit crew may actively request a performance assessment of a desired routing in manual mode. Regardless of the employed mode of operation, it is the cockpit crew that has the final authority and requests any trajectory changes from ATC using normal procedures and phraseology without any reference to TASAR or TAP. Consequently, TAP advisories have to be short enough and limited to a couple of waypoints to be processed by the cockpit crew and ATC in practice. From the ATC perspective, however, there is no difference between a request originating from an aircraft without versus one equipped with the TAP system.

3.2.3 Human Factors, Flight Trials and Operational Evaluation

The TASAR project followed a well-defined process to validate and improve its TAP with respect to the required functionality, features, user interaction, and finally operational fitness and benefit. Three dedicated and interleaved phases incrementally matured the system increasing the ultimately achieved Technology Readiness Level (TRL) to 7: system prototype demonstration in a space environment. These three phases included simulator evaluations, flight trials, and finally operational use during scheduled transcontinental revenue flights.

Having developed a socio-technical system introducing a significant technological and cultural change that shifts the role of cockpit crew members from mainly reactive to proactive trajectory managers, a careful human factors assessment was necessary and performed in full-flight simulators during the first phase. The evaluations began with a first set of sessions in 2013, continued with a second set in 2014, and followed an iterative and human-centered process to answer questions regarding pilot training, potential distractions and workload impact as well as integration with existing applications and SOPs in a Crew Resource Mangement (CRM) environment.

Two disjoint sets of airline pilots participated during the two studies with overall positive results: A Computer-Based Training (CBT) solution for TAP operation was developed and found superior to purely instructor-based training with respect to motivation, confidence and engagement. Pilot Monitoring (PM), ATC, and OPS interactions were simulated and flight-critical off-nominal events (fuel imbalance situations) were introduced to confirm a low and non-disruptive impact on cockpit scan pattern and workload. Both objective and subjective apparatus were employed to perform the experiments. Eye tracking system and electrocardiogram measurements (later discarded due to concerns regarding the metric validity) were supplemented with pilot surveys covering subjective perceptions of cognitive workload, functionality, usability and the resulting SA. The experiments in the simulators found no statistically significant differences between operations with and without TAP regarding response times, glance occupancy (cockpit scan pattern), and pilot workload while supporting high levels of SA. The TAP UI received high ratings of comprehension, usability, and usefulness for nearly all features, in particular, after a significant redesign following the first simulator sessions. Most importantly, the awareness of and interest in a potential business case gained momentum resulting in intensified partnerships that would later culminate in real-world operational flight evaluations.

The second evaluation phase was mainly concerned with the transition of TAP from a laboratory into a relevant environment. To achieve that objective, NASA performed three TASAR flight trials in 2013 and 2015, respectively. A Piaggio P180 Avanti[16] served as the aircraft platform to represent a potentially noisy and erroneous real-world system environment in terms of data communication. Challenges in this environment, furthermore, encompass ambient factors, such as light, temperature, and noise as well the pace and operational progress under real conditions. The necessary integration included the adaptation to the GAMA avionics protocol subset as well as the communication with relevant avionics components, such as the Flight Management System (FMS), the Air Data Computer (ADC), and the Flight Data Recorder (FDR). Apart from the verification of correct data interfaces and overall functionality, the flight trials again incorporated surveys covering subjective evaluations and assessed the fitness of the TAP for operational use.

The avionics integration to enable the flight trials turned out to be fairly complex due to the feature differences of the GAMA avionics protocol subset and the complete ARINC standard that is available in both the full-flight simulator and the transport category aircraft that would be used for the final operational tests. In order to succeed with the integration and solve discovered issues, it became necessary to develop a TAP data playback feature that could be used to replay the previously recorded avionics bus traffic. This feature turned out to be invaluable during the later integration steps.

As expected, some data-related problems that are inherent to a live avionics system environment persisted during the flight trials: Data dropouts, communication latency issues and unfiltered data noise in partiucular with respect to turbulence and vertical speeds received via ADS-B are some of the examples. Many improvements that resulted from the first trial, including an improved aircraft performance model, a new trajectory generation algorithm and a more stable PBGA as well as a modified TAP UI, found their way into the subsequent trials.

The second trial served as an important step to transition the system to its target platform, the Boeing 737 – 900, of which three would eventually be equipped by Alaska Airlines to participate in the operational evaluation. The Piaggio P180 Avanti avionics system interface was modified to resemble that of the B737 for that purpose. Other objectives focused on the processing capabilities of external data, such as winds, convective weather, and SUAs, from different sources based on non-SWIM formats. Furthermore, the TAP trajectory optimization performance, the acceptance by both cockpit crews and ATC, the UI usability as well as the effects on CRM were evaluated.

[16] https://en.wikipedia.org/wiki/Piaggio_P.180_Avanti

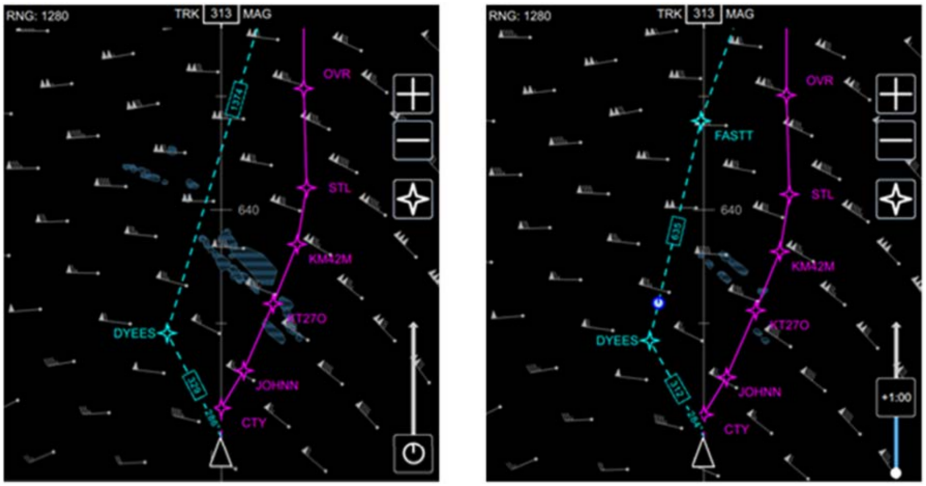


Figure 3.7: The TAP UI Time Slider [Win19]

The third and final trial focused primarily on how convective weather is embedded into the TAP system. Both options of a ground-based weather supplied via a ground data server as well as the translation of Airborne Weather Radar (AWR) images into weather polygons using a dedicated AWR data processor were studied. These studies led to further TAP UI improvements to increase SA by including a 4D weather representation and an associated time slider shown in Figure 3.7.

The operational flight tests that followed since 2017 with Alaska Airlines on three Boeing 737 – 900 likewise followed an iterative approach. First, ground tests at high taxi speeds were performed. Thereafter, TAP was integrated as a headless component that did not interfere with any operational aspects of the flight. This headless approach and the previously employed avionics bus TAP recording and playback allowed to analyze and verify proper system integration and functionality before introducing the UI into the operational environment. Finally, the TAP UI completed the setup in part being operated from the jump seat by a third crew member (NASA researchers and Alaska Airlines interns) as well from the cockpit seat by a designated group of Alaska Airlines tech-pilots. Although, the operational flight tests were not performed to the extent originally anticipated by NASA leaving only a limited number of flight hours for TAP operation by cockpit crew members (tech pilots), the TASAR project generated valuable insights and succeeded in generating momentum in the aviation industry and interest by other airlines to evaluate this new technology.

3.2.4 Relationship between TASAR and the SAFCS

It would be a long shot and foolish to even attempt to compare the TASAR project to our SAFCS on the organizational or administrative level. The TASAR project was developed over a long time span by a government-funded research agency with many industrial and academic partners, government agencies and airlines involved. The SAFCS, in contrast, is a purely academic project that neither has the goal to push the industry a step closer toward the future of aviation or create innovative real-world applications nor is it restricted by any established and accepted processes and procedures. However, on a purely conceptual and technical level parallels can very well be drawn and differences highlighted. The SAFCS may, therefore, serve as an academic tool to evaluate similar new concepts without any industry or governmental constraints.

First and foremost, both projects attempt to achieve the same goal on a conceptual and technical level, that is, increasing the level of operational autonomy of airborne missions in terms of a *self-managing* and *self-governing* approach by optimizing flight trajectories. However, in contrast to the TASAR project and its TAP architecture, our SAFCS is envisioned to reside in the cloud and not necessarily deployed aboard or connected with the various aircraft avionics systems. It can, therefore, be considered a much more aggressive paradigm shift as the aircraft progress is monitored remotely and the connection to the airborne agent is treated equivalently to all other connections that form the Autonomic Context (AC) of our system. As explained in greater detail in Part II, our SAFCS concept is designed as an AE interacting with its AC. Being a cloud-based architecture supporting higher levels of autonomy and decision-support in flight, there are no limitations with respect to communication with the evolving SWIM data sources or other resources available within the cloud. In fact, the SAFCS was understood as such a system and with the SWIM infrastructure in mind right from the beginning. Both projects, however, clearly identify the same rational actors to be included within the AC of their systems: cockpit crews, ATC, and OPS. The term *rational actor* is used intentionally here to highlight that machine actors will eventually perform tasks that are presently performed by human actors. As such, we propose ATC communications such as or similar to CPDLC to enable our rational machine actor to communicate with ATC actors in order to acquire clearances for the computed trajectory optimizations. Similarly, ACARS is an operationally established datalink protocol for communication with OPS. The communication between these actors and systems such as our SAFCS will, however, still be required in the future regardless of their concrete realization.

Similarly, the contextual elements airspaces, traffic, and weather are considered essential components of the dynamic environment in both projects. The vision of our SAFCS concept with a stronger initial focus on the disruptive impact of the evolving SWIM infrastructure does not stop here. Dynamic data about the aviation infrastructure, such as status of navigational aids (AIXM) and aerodromes (AMXM) as well as maintenance data (MMIXM), can be sourced to derive useful information for autonomous systems of the future and resemble the present practice of issuing NOTAMs and PIREPs for human actors.

With respect to trajectory optimization, the applied approaches of each project differ significantly, however. As already mentioned, the TAP architecture employs a PBGA to process generated trajectories of its trajectory generator and evaluates its fitness over multiple generations in terms of costs and potential conflicts found by its conflict detector. This approach is very similar to the one presented by Rathbun, Kragelund and Pongpunwattana [RDKP02] and was already mentioned in Section 2.2. The SAFCS concept, on the other hand, employs managed variants of *online*, *lifelong*, *dynamic*, and *anytime* algorithms that are members of different algorithmic planner families, such as CGS, RRTs, and PRMs. Conceptually, however, there is no limit to the supported algorithms in its core Managed Element (ME), and biologically-inspired algorithms could very well be included. Being designed as an AE in line with the MAPE-K pattern proposed as the autonomic vision by Kephart and Chess [KC03], our SAFCS features an Autonomic Manager (AM) that continuously monitors, evaluates, and acts on the computational and trajectory performance of its ME algorithms. As such, our SAFCS AM tunes and selects its managed planners according to the perceived *criticality* of the situation and uploads the most beneficial and timely trajectory via its agent datalink. Its planners operate on supported 4D planning environments and embedded 4D cost intervals, as further explained in Part II.

The computed trajectories and applied cost optimization themselves are also very different between both projects. The TAP optimizes for time, fuel, or a weighted combination of both referred to as the *cost index*. Presently, cockpit crews program the cost index into the FMS of their aircraft before a flight. It is an index that is determined by OPS and represents one of the operational constraints or preferences. It does not simply reflect the problems of time and fuel but also considers maintenance intervals and the number of possible revenue flights between those intervals. Maintenance and fuel costs are different in different parts of the world, which influences the resulting cost index on different routes. More precisely, the cost index is obtained by dividing aircraft operat-

ing costs by fuel costs. FMS performance data, such as the programmed cost index, are taken into account by the TAP. Furthermore, since the TAP has access to the aircraft FMS, it optimizes based on and rejoins a programmed route after the suggested optimizations. It is, therefore, directly applicable and beneficial within the current enroute system and embraces the established practices and workflows without disrupting them.

The SAFCS concept, in contrast, is much more simplistic and optimization applies only to AC costs that have been obtained from SWIM datasets and embedded into its environment models. Cost optimization for the SAFCS simply means minimizing costs in terms of environmental risk avoidance while observing operational cost and risk *policies*. Hence, the SAFCS optimization is not concerned with a cost index value that represents certain business goals but rather avoidance of costly environments, such as restricted airspaces, adverse weather, or unserviceable aviation infrastructure. How these costs are combined and what costs are acceptable to begin or complete a mission until its goal or else abandon the former is determined by the SAFCS cost and risk policies, respectively. These and other concerns are further discussed in Part II of this dissertation. Finally, the existing enroute system is entirely ignored by the current SAFCS prototype and no restrictions are imposed on generated trajectories whatsoever. However, the concept allows for the inclusion of environment models representing an enroute system (e.g., meshes in addition to grids) with cost interval embeddings and the application of CGS algorithms to find cost-optimal solutions.

3.3 Automatic Selection and Tuning of Motion Planners

The research in the field of automatic selection and tuning of motion planners has gained increased momentum in recent years. Hyperparameter optimization techniques have been successfully applied to find suitable configurations for general algorithms given a certain problem instance. The problem instance in case of motion planners is usually described in terms of the planner's environment and its features.

Burger, Bharatheesha, van Eert, and Babuska [BBvEB17] apply the Sequential Model-Based Algorithm Configuration (SMAC)—a more sophisticated instantiation of Sequential Model-Based Optimization (SMBO) using random forests—to sampling-based algorithms in order to find suitable configurations for their tuning parameters. Their evaluations compare the predicted and achieved performance of automatically tuned configurations for selected algorithms in different robotic manipulator scenarios of increasing difficulty.

Cano, Yang, Bodin, Nagarajan, and O'Boyle [CYB+18] compare different techniques to gain information about and predict the quality of configurations for sampling-based motion planning algorithms. To find promising configurations they apply random-sampling, Bayesian optimization, random forests, and AUC-Bandit and similarly compare the performance obtained with each approach in scenarios of increasing difficulty.

Moll, Chamzas, Kingston, and Kavraki [MCKK21] extend these ideas to predict both a suitable algorithm and an associated configuration for certain planning problems. They minimize loss functions capturing certain aspects of the algorithmic performance, such as planning speed, combined planning time and execution time (for online planners), and convergence to optimality. The set of algorithm candidates is contained in their Open Motion Planning Library[17].

3.3.1 Sequential Model-Based Algorithm Configuration

We consider SMAC as one of the highly relevant state-of-the-art approaches to tune the performance of our SAFCS core planning algorithms. Furthermore, SMAC is available as a Java artifact[18] and can, therefore, be readily integrated into the prototype of our proposed concept for evaluation.

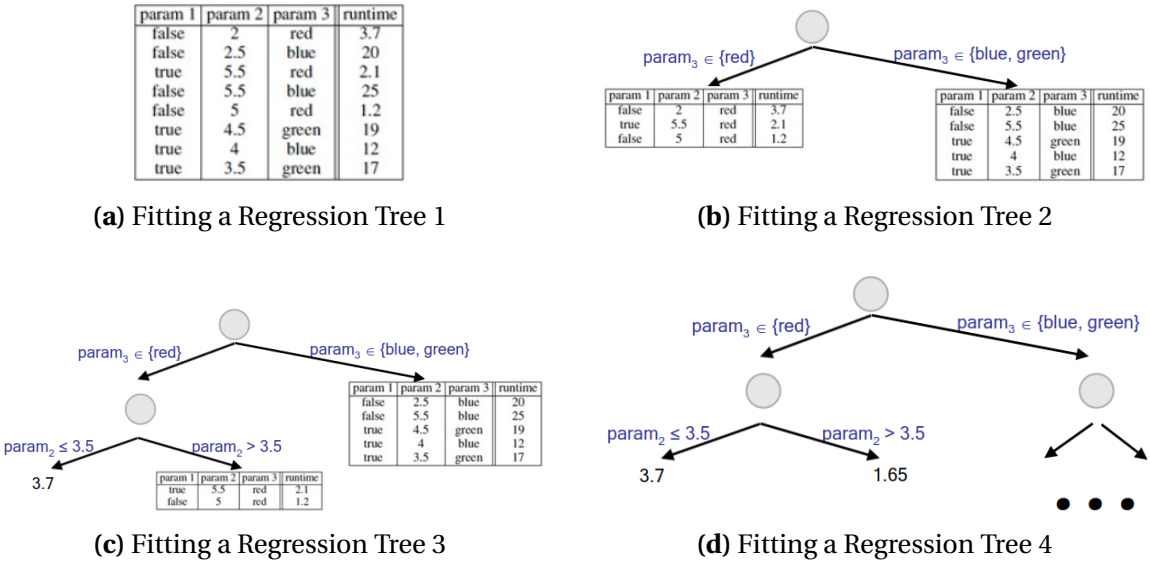
SMAC was first introduced by Hutter, Hoos, and Leyton-Brown [HHLB11] and represents an instantiation of the more generic SMBO. As such, SMAC follows the general idea shown in Algorithm 12.

Algorithm 12 Sequential Model-Based Optimization [HHLB11]

1: **procedure** TUNE(Algorithm A, Configuration Set Θ, Problem Set Π, Performance Metric m)
2: [Reputation R, Configuration θ_{inc}] ← Initialize(Θ, Π) ▷ obtain the initial reputation and default configuration
3: **repeat**
4: [Model M, Time t_{fit}] ← FitModel(R) ▷ fit the model according to the performance reputation
5: [Configuration List $\vec{\Theta}_{new}$, Time t_{sel}] ← SelectConfigurations(M, θ_{inc}, Θ) ▷ select a list of promising configurations
6: [R, θ_{inc}] ← Intensify($\vec{\Theta}_{new}$, θ_{inc}, M, R, $t_{fit}+t_{sel}$, Π, m) ▷ test configurations and collect empirical performance data
7: **until** time budget exhausted
8: **return** θ_{inc}

SMBO attempts to find a suitable (incumbent) configuration θ_{inc} according to a performance metric m as an instance of the configuration set Θ for an algorithm A to be applied to a problem set Π. A configuration $\theta_i \in \Theta$ comprises multiple (d) parameters $\theta_i = (\theta_{i,1}, \ldots, \theta_{i,d})$ that can be tuned to achieve the corresponding performance o_i collected as tuple (θ_i, o_i). Running multiple tests with promising configurations

[17] https://ompl.kavrakilab.org/
[18] http://www.cs.ubc.ca/labs/beta/Projects/SMAC/



(a) Fitting a Regression Tree 1

(b) Fitting a Regression Tree 2

(c) Fitting a Regression Tree 3

(d) Fitting a Regression Tree 4

Figure 3.8: SMAC: Fitting a Regression Tree

$\vec{\Theta}_{new}$ (line 6) until a time bound has been reached (line 7), the performance reputation R of tested configurations $\vec{\Theta}_{new}$ (consisting of configuration-performance tuples $R_{A,\pi} = \{(\theta_1, o_1), \ldots, (\theta_n, o_n)\}$ for an algorithm A applied to a problem instance π) and the associated model M, if any, are extended in the process (line 4) and may help to predict suitable configurations for similar problem instances in the future. SMBO may be model-free, that is, no model is actually being fitted by the collected performance reputation, or model-based, that is, a particular model represents the collected empirical performance data. The selection of promising configurations to test (line 5) can be random (e.g., in case of a model-free approach) or follow a particular modelled strategy.

SMAC instantiates the SMBO using a model-based approach and selection strategy. The model is a random forest of regression trees—one of the important ensemble machine learning data structures that can be used for classification and regression. Random forest is a form of supervised learning that applies bagging (sub-sampling with repetitions) to separate the collected reputation R for independent regression with higher variance in each tree and then combines a prediction according to the mean with lower overall variance. In the case of SMAC, a number of $B = 10$ independent regression trees M are constructed from the collected performance data R of multiple test runs of an algorithm A using a configuration $\theta \in \Theta$ to tackle a problem $\pi \in \Pi$. These regression trees contain the mean resulting performance values at their leafs. The nodes of the trees are split according to the determined configuration parameter thresholds (usually the minimum sum of squared residuals) until a leaf contains not more than $n_{min} = 10$

configuration-performance tuples (θ_i, o_i) to reduce overfitting. The splitting is limited to one parameter out of a certain subset of the d configuration parameters in each step, which for SMAC has a default cardinality of $\lceil d * p \rceil$ with $p = 5/6$. B, p, and n_{min}, therefore, represent the hyperparameters of SMAC itself. Training and evaluation of promising parameter configrations occurs interleaved (online). In this process, bad performing configurations, that is, worse than the presently incumbent one, are rejected quickly. However, promising configurations are tested extensively against the performance of the presently incumbent configuration with respect to all its previously assessed problem instances. Selecting promising configurations benefits from establishing a regression model that allows to predict the performance of a particular configuration. The selection of promising configurations applies local optimization on an *expected improvement* over the incumbent configuration and implicitly balances exploitation of those configurations with high predicted performance and exploration of those with high predicted uncertainty.

Figure 3.8 shows an excerpt from a presentation[19] by Hutter, Hoos, and Leyton-Brown [HHLB11] that demonstrates how an individual regression tree is fitted based on empirical performance data (in this case runtime) obtained for particular configuration parameter instances of $\theta \in \Theta$ with $\Theta = \mathbb{B} \times \mathbb{R} \times Color$ (without taking the hyperparameter n_{min} into account).

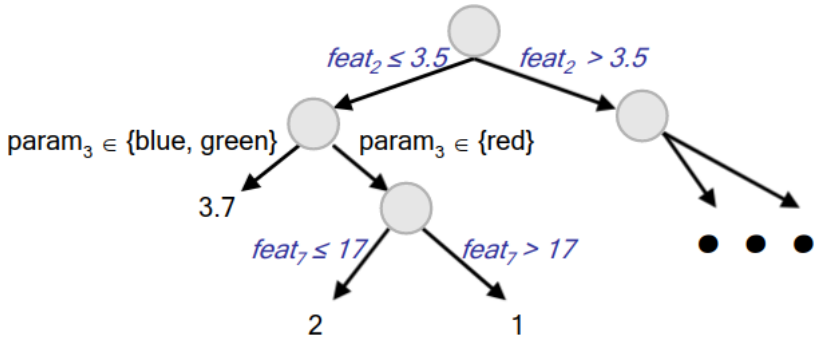


Figure 3.9: SMAC(PP): Fitting a Regression Tree

In addition to configuration parameter splitting, the regression trees can also be split based on features $x = (x_1, \ldots, x_n)$ of the problem instance π. Hence, R does not just store and fit M with the obtained empirical performance data of configurations (θ_i, o_i) but is extended by these features giving $([\theta_i, x_i], o_i)$. This extension is referred to as SMAC(PP) (for predicting performance across problem instances). An individual regression tree

[19] https://slideplayer.com/slide/5057779/

(contained in the referenced presentation) for combined parameter configuration and feature performance is shown in Figure 3.9.

3.3.2 Relevance to the SAFCS Autonomic Manager

SMAC is particularly interesting for the SAFCS and its Autonomic Manager component which is responsible for selecting and tuning managed motion planners of the SAFCS core Managed Element for a particular Autonomic Context, as discussed in Part II. The problem instances are, thus, different Autonomic Contexts comprising the planning environment, the agent capabilities within the environment, and operational goals and policies. As the authors of SMAC point out, there are domain-independent features to problem instances, such as their size and the algorithm's default configuration performance, and these would translate to the size of the environment, and quality in terms of trajectory costs as well as quantity in terms of computation time to obtain the trajectory under a default planner configuration.

There are, however, many more domain-specific features of interest that could be assigned to different classes of problem sets. Examples include the number of obstacles (e.g., weather, airspaces, facilities, traffic), the nearness of obstacles, the severeness of obstacles, the number of obstacles affecting a straight line between Points of Interest (POIs), the distance between POIs, the agent's altitude and radius of action, the operational policies and more. Classes with similar features could represent different flight phases, such as take-off, departure, cruise, approach, landing, and emergencies. These problem classes in general feature different levels of *criticality* determined by their features and, therefore, likely require dedicated tunings for their configuration parameters. The reputations of parameter configurations for planners within the problem sets of each class together with the associated fitted model may form the *knowledge base* of an SAFCS AM.

3.4 Standards and Verification of Adaptive Systems

Software engineering in the aviation domain follows rigorous processes that produce certifiable artifacts to ensure the required level of dependability and safety. The developed systems often contain high-integrity real-time software components that are required to satisfy a number of certification objectives. Furthermore, the satisfaction of these objectives may have to be shown by an independent entity in some cases. The

currently applicable standards for software engineering in this domain particularly include the Radio Technical Commission for Aeronautics (RTCA) standards listed in Table 3.2.

Table 3.2: Applicable RTCA Standards

Standard	Title
DO-178B/C	Software Considerations in Airborne Systems and Equipment Certification
DO-278A	Guidelines for Communication, Navigation, Surveillance, and Air Traffic Management (CNS/ATM) Systems Software Integrity Assurance
DO-332	Object-Oriented Technology and Related Techniques Supplement to DO-178C and DO-278A
A(M)C 20-193	Acceptable Means of Compliance: Use of Multi-Core Processors
DO-362	Command and Control (C2) Data Link Minimum Operational Performance Standards (MOPS) (Terrestrial)

Where DO-178B/C describe the considerations for certification of software deployed in an airborne system, DO-278A establishes similar objectives for ground based software systems. Representing the more recent standards, DO-178C and DO278A can be supplemented with DO-332 and A(M)C 20-193 to address the issues that arise with object-oriented technologies and multi-core (parallel) processing systems, respectively. Furthermore, DO-362 is concerned with the operation and datalink capabilities of UASs in designated airspaces.

An SAFCS, such as envisioned by our research, would have to respect and satisfy the applicable standards if deployed in an industrial context. More precisely, it would have to be classified according to an Item Development Assurance Level (IDAL), as shown in Table 3.3, and satisfy the associated objectives. Depending on the concrete realization of the SAFCS concept, an implementation could be classified as IDAL *D* if it is employed as a decision support system with human rational actors in the loop and similar in functionality to the TAP described in Section 3.2. If, however, the concept is realized to its full potential and at its highest level of autonomy, with both human and machine rational actors in the loop, then clearly an IDAL *A* classification would be appropriate.

Wilkinson, Lynch, Bharadwaj and Woodham [WLBW16], a team of NASA and Honeywell researchers, have investigated the feasibility of applying the mentioned standards

[20] https://www.rtca.org/standards/

Table 3.3: DO-178C Item Development Assurance Level[20]

Level	Failure	Description
A	Catastrophic	Failure may cause deaths, usually with loss of the aircraft.
B	Hazardous	Failure has a large negative impact on safety or performance, or reduces the ability of the crew to operate the aircraft due to physical distress or a higher workload, or causes serious or fatal injuries among the passengers.
C	Major	Failure significantly reduces the safety margin or significantly increases crew workload. May result in passenger discomfort (or even minor injuries).
D	Minor	Failure slightly reduces the safety margin or slightly increases crew workload. Examples might include causing passenger inconvenience or a routine flight plan change.
E	No Effect	Failure has no impact on safety, aircraft operation, or crew workload.

to adaptive systems in the aerospace domain. They acknowledge the potential of these systems and their associated paradigms and algorithms, particularly in the context of the FAA NextGen program, and highlight the challenges related to self-adaptive system certification under the mentioned standards that stem from these very paradigms and algorithms of the growing AI toolbox. Non-determinism as well as behavioral and structural modifications of the self-adaptive system during runtime clearly are difficult, if not impossible, to unite with the desired verification and validation of functional and temporal correctness of such a system.

Table 3.4: Statistical Techniques and Applications [WLBW16]

Role	Definition	Applications
Regression	Identifies mathematical relationships among variables	Adaptive Controls, Continuous Models, Parameter Estimation
Classification	Classifies input data into one of a number of discrete classes	Monitoring, Situational Awareness
Clustering	Identifies similarities between features of the data and groups similar data items into clusters with similar features	Monitoring, Situational Awareness, Data Post-Processing
Optimization	The selection of a “best” element from a set of available alternatives	Planning, Path/Trajectory Analysis

Self-adaptive systems may employ one or many of the techniques listed in Table 3.4. In fact, and, as described in Part II, the current SAFCS implementation employs non-deterministic *optimization* algorithms for managed trajectory planners, *clustering* to categorize similar extracted features from planning environments, and *regression* to autonomously manage planners without assuming any domain knowledge. Additionally, it is a concept that encourages a dramatic paradigm shift for the deployment of smart guidance systems in a data center, providing significantly increased resources in terms of processing power and memory, and low-latency access to the SWIM cloud. Being implemented as a collection of Java artifacts comprising an evolving pool of managed planners at its core component that can be assigned to different processor cores, it becomes apparent that standards DO-332, A(M)C 20-193, and DO-362 would equally apply.

As Wilkinson, Lynch, Bharadwaj and Woodham [WLBW16] point out, industrial certification of a system similar to our proposed SAFCS poses significant challenges in assuring safety by means of the available traditional safety assurance methods and procedures. Therefore, they propose to allocate additional effort on formal verification using formal methods and model-based design to satisfy system-level requirements that have been tailored to be verifiable by the applied methods. This approach would comprise (quote)

> "requirements expressed in formal or mathematical terms with well-defined syntax and semantics that are amenable to modern analysis methods capable of providing a high level of design assurance."

These requirements could, for example, describe valid value ranges of the system's output space that would be enforced by online monitoring components implementing a defined fallback behavior. The bounded values should ensure the system cannot diverge from safe operational margins and, therefore, prevent the (quote)

> "violation of assumptions or constraints, loss of stability or convergence, and excessive usage of computational resources."

To establish any assumptions or constraints, define conditions that lead to instability or divergence, or require excessive resources, in turn, requires significant domain knowledge. However, the lack of domain knowledge is precisely the reason to employ a self-adaptive system in the first place, and what can be considered rational behavior

in certain situations may not be known in advance, in particular, for arbitrary combinations of multiple parameter values in the real domain. Furthermore, the authors acknowledge that (quote)

> "in addition, to make the verification effort manageable, techniques analogous to equivalency classes that subdivide the learned state space and structural coverage analysis that measures the verification completeness of the learned state space [...] are needed to complete the approach. Currently, these techniques are not known."

and (quote)

> "One caution is that not all the necessary analysis and modeling tools are presently available and, therefore, further research is required before such an approach can be applied in a practical application."

With respect to real-time requirements, it should be noted that neither deadlines for bounded rationality nor expected deliberation times are necessarily known in advance. Our concept rather acknowledges the limitations of bounded rationality by employing anytime planners, such as the variants described in Chapter 2.1.2, that deliver sub-optimal feasible solutions that progressively improve on a best-effort basis.

We, therefore, conclude and acknowledge that the applicable standards for industrial aerospace applications are challenging, if not impossible, to be satisfied by any realization of our concept at this point in time. As mentioned in Section 3.2.4, we rather focus our efforts on an academic research infrastructure that is not constrained by industrial standards and practices. To achieve our research goals, an established agile software engineering methodology shall be applied instead.

3.5 Chapter Summary

In this chapter we discussed influential state-of-the-art developments and initiatives that are closely related to our research. We briefly described the open System-Wide Information Management infrastructure and its protocols and highlighted the potential benefits when deriving useful information from ubiquitously available aviation data. We studied the NASA TASAR project pursuing the goal of increasingly self-governing and self-managing flight operations in the context of a newly proposed rule set: the

Autonomous Flight Rules. Furthermore, we reviewed some recent approaches to hyper-parameter optimization that could be beneficial for the Managed Element of our SAFCS. Finally, we visited the challenges associated with state-of-the-art industrial standards in the context of self-adaptive systems for the aerospace domain.

Part II

Smart Autoflight Control System Infrastructure

Chapter 4

SAFCS — Overview

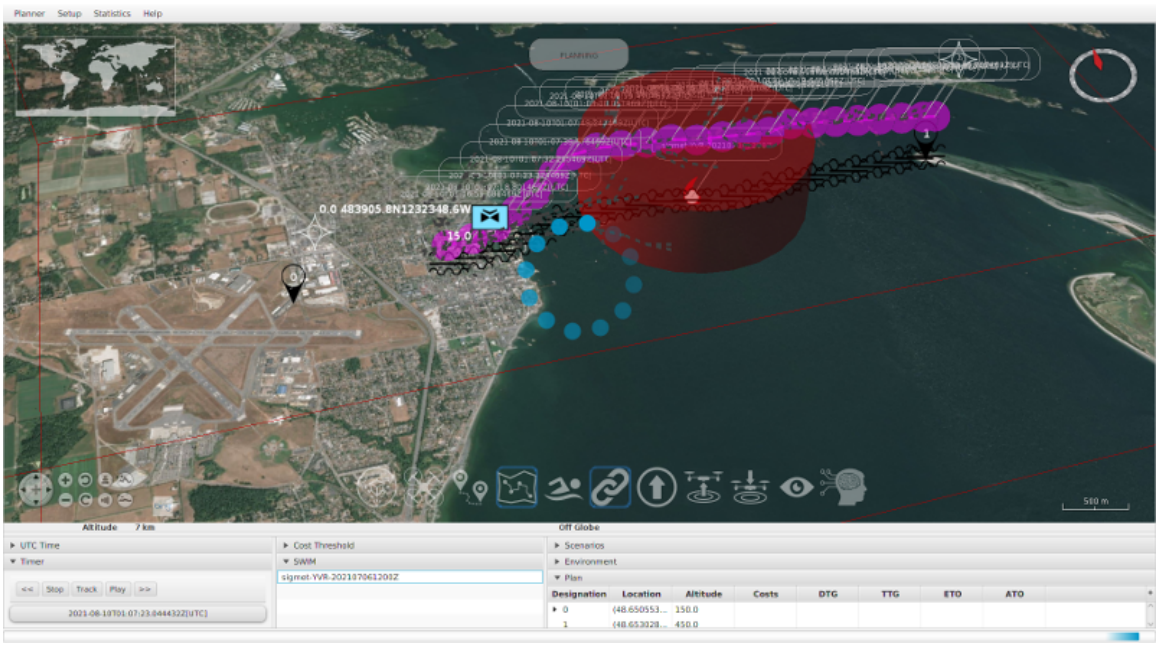


Figure 4.1: Smart Autoflight Control System UI

We designed and built the SAFCS core,[1] managing,[2] and UI[3] components as extensions to the NASA Worldwind[4] Application Programming Interface (API). NASA Worldwind Java[5] is an impressively powerful open-source Java development platform[6] for a

[1] https://github.com/stephanheinemann/worldwind/tree/develop
[2] https://github.com/stephanheinemann/worldwind/tree/develop
[3] https://github.com/stephanheinemann/worldwind-ui/tree/develop
[4] https://worldwind.arc.nasa.gov/
[5] https://worldwind.arc.nasa.gov/java/
[6] https://github.com/NASAWorldWind/WorldWindJava/

wide range of virtual globe applications. It features complex geometric computation and visualization functionalities and already supports renderable airspaces and standard symbology.

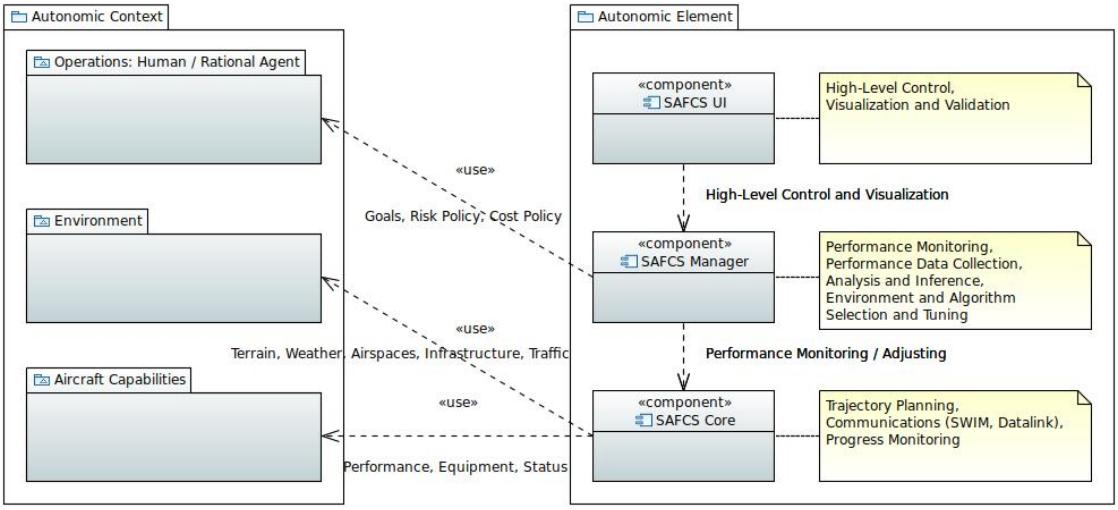


Figure 4.2: Smart Autoflight Control System Overview

We developed the SAFCS UI component shown in Figure 4.1 to control and visualize the activities of the SAFCS core as well as to validate its functionality and simulate some of the activities of the SAFCS manager. The SAFCS as an infrastructure integrates and extends many of the concepts presented in Chapters 2 and 3 and follows the conceptual design of a self-adpative Autonomic Element acting within its Autonomic Context, as illustrated in Figure 4.2. As such, it can be deployed in the cloud to obtain the relevant data and support airborne missions via established datalink connections.

The following chapters attempt to give an overview of the current SAFCS core, managing, and UI component's features and development options. For the sake of clarity, we only discuss selected packages and classes of our evolving infrastructure. The internals of individual classes, that is, their attributes and method signatures, are mostly abstracted. The discussion will hopefully provide an entry point to some of the SAFCS software packages discussed hereafter, and the reader is encouraged to follow the provided links for a more detailed inspection of the documented code. Selected SAFCS features are demonstrated in videos[7] and aim to provide an alternative starting point.

[7] http://www.rigiresearch.com/people/stephan-heinemann

Chapter 5

SAFCS — Architecture

5.1 SAFCS Components

This chapter gives an overview of the architectural and functional aspects of the SAFCS, as initially described by Heinemann, Müller and Suleman [HMS18]. Figure 4.2 depicts the three principal components of our SAFCS infrastructure enabling the required functionality as a self-adaptive Autonomic Element for online trajectory planning and optimization. These components are

- the SAFCS Core (Managed Element),
- the SAFCS Autonomic Manager, and
- the SAFCS User Interface.

The core component of the SAFCS provides the necessary infrastructure, such as environments, aircraft, planners, and connections, for trajectory planning of aircraft that are subject to bounded rationality within an uncertain dynamic environment. The SAFCS core, furthermore, provides interfaces to serve as the Managed Element for autonomic planning sessions that are controlled by an Autonomic Manager in line with the MAPE-K autonomic design pattern introduced by Kephart and Chess [KC03].

The SAFCS Autonomic Manager monitors and persists the long-term performance, or *reputation*, of the applied planning algorithms in situations of varying *criticality* and, based on the obtained *experience*, selects and tunes the most promising candidate to solve the planning problems at hand.

The SAFCS UI serves for high-level control, visualization, and validation by the human operator of the Autonomic Element. It plays an invaluable role to communicate

planning problems, actions, and resulting statistics with other researchers. It, furthermore, enables us to evaluate our SAFCS during experimental flight trials and serves as a Ground Control Station (GCS), as discussed further in Part III.

The connections of the SAFCS core ME and AM enable communications with the planned aircraft and the environment they operate in. Different types of connections cover different aspects of communicating with the SAFCS AC, as shown in Figure 4.2. As such, aircraft datalink connections are essential to initiate take-off and landing maneuvers as well as upload and subsequently revise established trajectories to the airborne agent. They permit access to the aircraft telemetry, including operational mode, equipment status, position and current executed mission. SWIM connections can be employed to publish and subscribe datasets related to weather, airspaces, traffic, and aviation infrastructure. These data are modelled as (traversable) obstacles, that is, spaces featuring off-nominal costs, within the SAFCS core ME and are assigned 4D cost intervals using an operationally defined cost map. ATC connections such as or similar to CPDLC could be leveraged to gain access to airspaces and, therewith, change their associated cost intervals. Finally, OPS connections such as or similar to ACARS would permit communications with operational control to define and potentially revise the applicable mission parameters, such as goals, risk and cost policies.

To facilitate practical research all SAFCS connection interfaces can be realized by simulated connections. For example, a simulated aircraft datalink connection enables us to validate the functionality of the SAFCS AE in the laboratory without having to establish a connection to an actual aircraft. Likewise, a simulated SWIM connection can be used to inject and embed temporal costs related to different SWIM topics as required.

5.2 SAFCS Core Architecture

The following sections highlight the architectural aspects of the SAFCS core as they relate to and form the infrastructure of the implementation of the currently supported planning families CGS, RRTs, and PRMs, particularly their online, lifelong, dynamic, and anytime key algorithms that are managed by the SAFCS AM component. For the sake of clarity and simplicity, the discussion is limited to class diagrams and the relations among classes without necessarily showing their package associations, signatures or any internals. Only as required for the understanding of certain architectural decisions will class signatures be exposed and internal aspects discussed. The shown class diagrams are excerpts only, incomplete and itentionally kept simple to permit an easier

access to the important concepts. The reader is, thus, encouraged to follow the provided links to the documented code in order to acquire a deeper understanding of the complex internals involved and the concrete solutions provided.

5.2.1 Sessions, Scenarios, Factories and Specifications

At the core of the SAFCS lies the concept of a planning `Session` which is depicted with closely related classes in Figure 5.1. A number of these `Sessions` are managed by a singleton `SessionManager`. This `SessionManager` instance needs to be obtained by any managing component, such as the SAFCS UI or the SAFCS AM. Any planning `Session`, in turn, has access to a number of `Registry` and `Factory` instances.

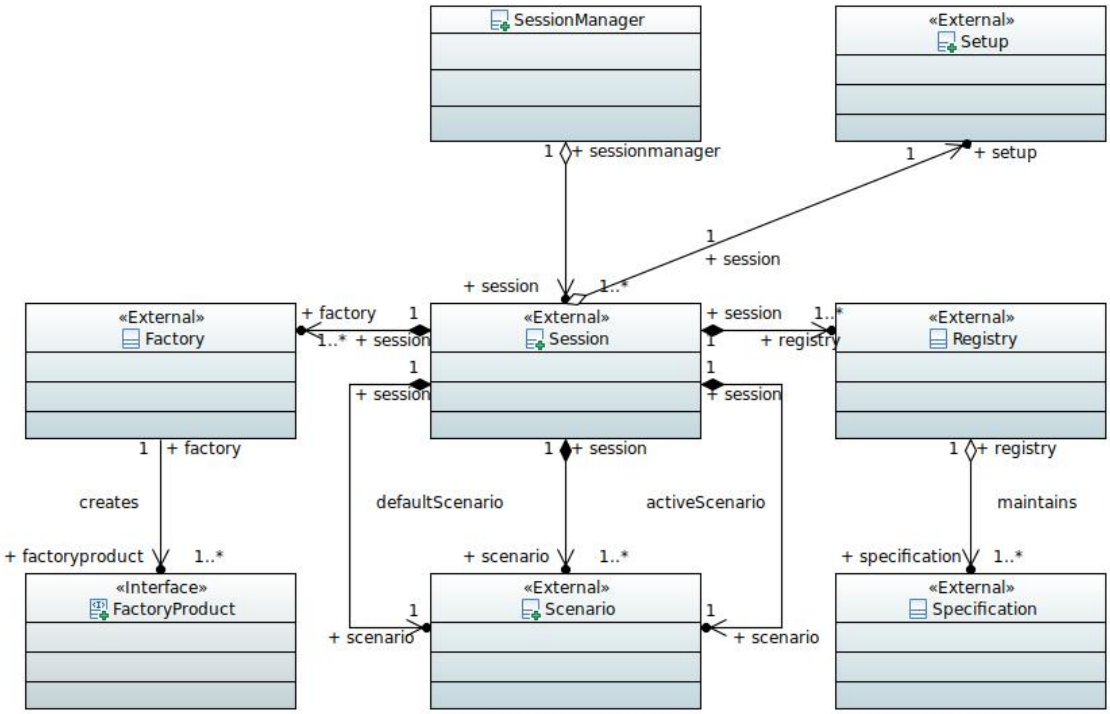


Figure 5.1: SAFCS Core: Sessions, Scenarios, Factories and Specifications

A `Registry` maintains a number of `Specifications` that describe `FactoryProducts`. A capable `Factory` can be passed such a `Specification` in order to produce a `FactoryProduct` that meets the `Specification` and its contained properties. The current `Setup` of a `Session` stores all presently active `Specifications`.

The `Properties` of a `Specification` represent its customizable part. There are abstract specifications for each producible item that are subsequently specialized and re-

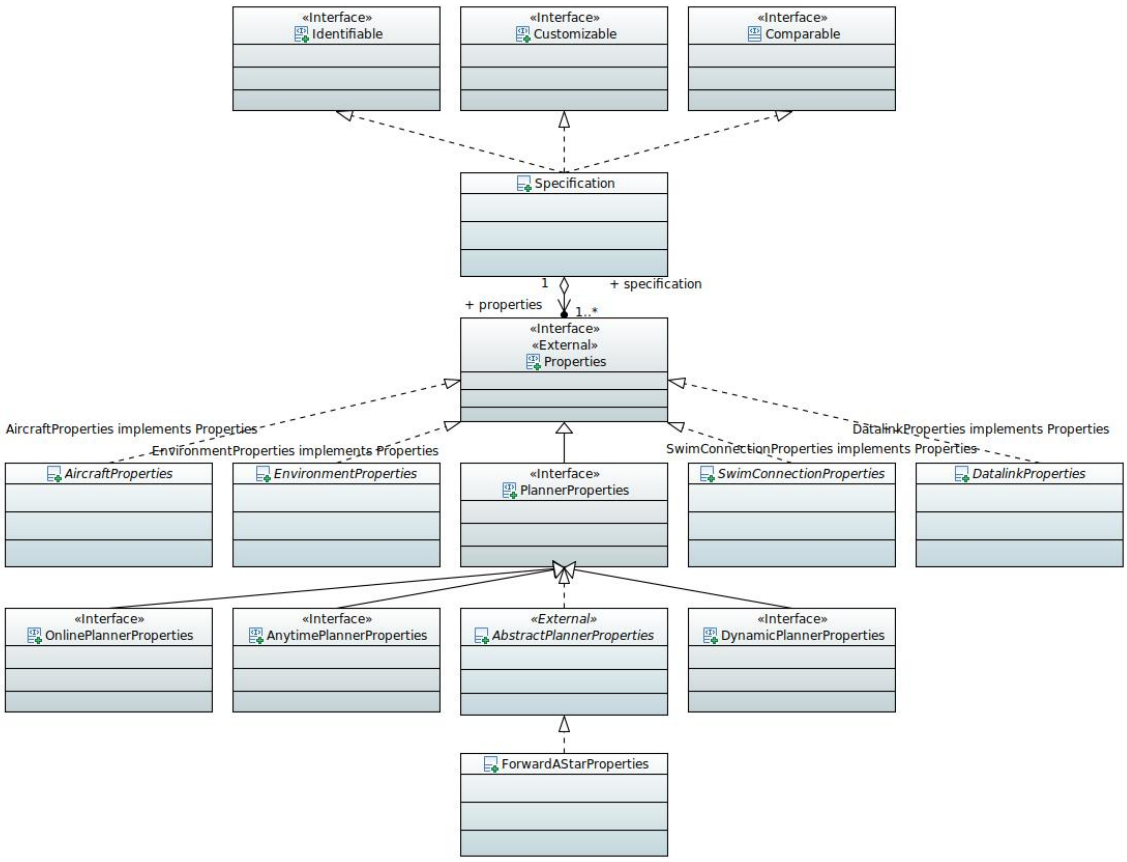


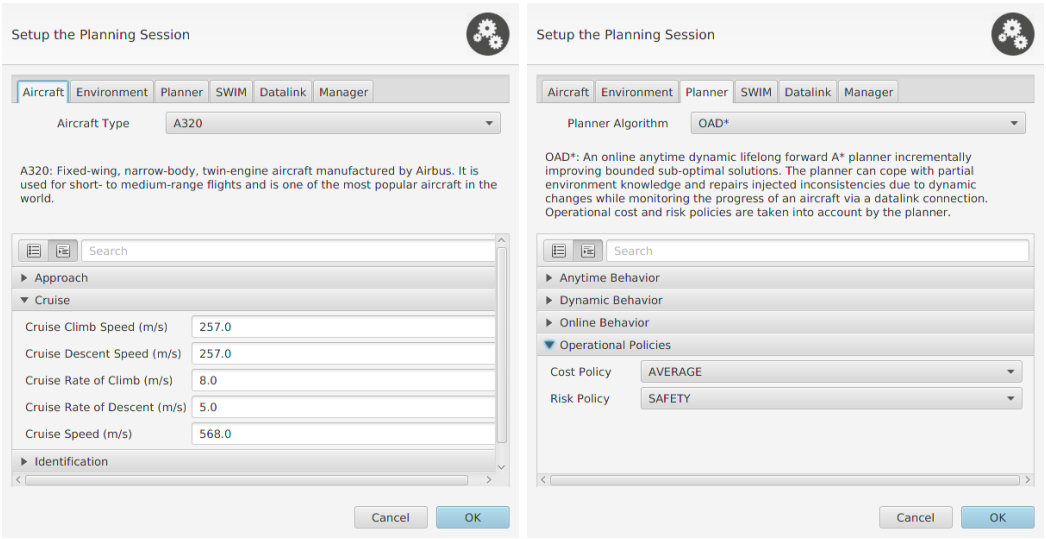
Figure 5.2: SAFCS Core: Specifications and Properties

fined. Figure 5.2 shows an excerpt of the `Properties` hierarchy leading to one of the concrete and most basic planner properties, namely the `ForwardAStarProperties`.

During the lifecycle of a planning `Session`, its `Setup` of active `Specifications` can be continuously modified. For example, an SAFCS AM owning a `Session` may continuously select and tune the employed planning environments and algorithms during its monitoring, analysis, planning, and execution cycles. The ability to adjust a planning `Scenario` using a pool of tunable supported environments and algorithms is one of the key features of the SAFCS core.

What may sound rather abstract at first is the required infrastructure for the main principle workflow when creating planners, planning environments, aircraft with their capabilities, connections to the aircraft, connections to ATC or OPS as well as to the SWIM AC. All these elements are products of their respective SAFCS factories and are created in accordance with their corresponding `Specification`.

A number of planning Scenarios can subsequently be equipped with these FactoryProducts in order to solve certain trajectory planning problems. For instance, one Scenario may be equipped with a quadrotor aircraft featuring certain capabilities and operating in a planning grid environment of a specified resolution in accordance with a trajectory that is continuously planned by an Online AD* (OAD*) planner following a tuned improvement strategy. The Scenario, furthermore, obtains dynamic changes to be embedded into the planning grid environment from a SWIM connection, and employs a datalink connection to communicate with the quadrotor aircraft. In this example, the quadrotor aircraft, the planning grid, the OAD* planner, the SWIM, and the datalink connections are FactoryProducts of capable factories that have been created in accordance with their customized Specifications.



(a) SAFCS UI: Aircraft Setup **(b)** SAFCS UI: Planner Setup

Figure 5.3: SAFCS UI: Specification Setup

Figure 5.3 shows two examples of how our SAFCS UI allows Specifications to be equipped with customized properties in case of a modelled aircraft and planning algorithm, respectively. These customized Specifications are then passed to capable factories in order to create the concrete FactoryProducts.

A Session can host a number of different planning Scenarios where only one can be the *active* Scenario at any given time. The active Scenario of a Session is the one that is currently being controlled by its owner, such as our SAFCS AM or UI component.

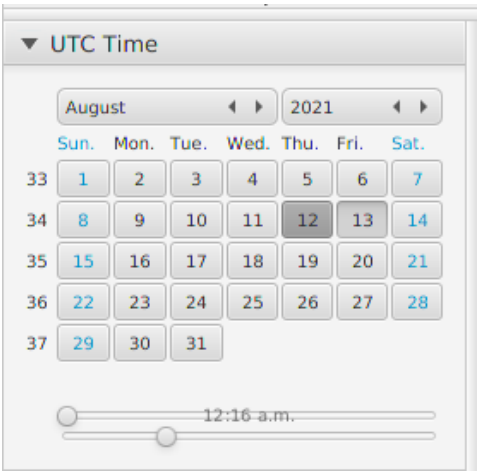
The planning Scenario plays a central role in the SAFCS core. It aggregates all aspects that are relevant for trajectory planning and decision support. The current time,

the aircraft and its environment featuring SWIM data embeddings, the planning algorithm, and the communication connections can be found here. Whenever modifications are made to an active `Scenario` (e.g., placing an aircraft, defining environment extensions, invoking a planning algorithm), the `Specifications` of the current `Setup` are applied.

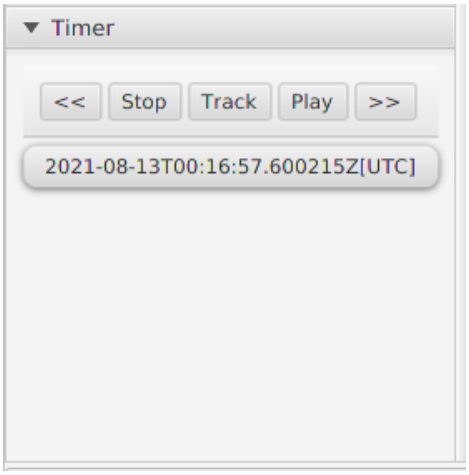


Figure 5.4: SAFCS UI: Scenarios

Figure 5.4 shows how our SAFCS UI displays available planning `Scenarios`. These identifiable `Scenarios` can be added, removed, activated, and supplied with all the elements mentioned above.



(a) SAFCS UI: Scenario Time



(b) SAFCS UI: Scenario Timer

Figure 5.5: SAFCS UI: Scenario Time Controller

A `Scenario`, furthermore, features a `TimeController` which can track real-time as well as reverse or fast-forward in time. Hence, the `TimeController` of a `Scenario` is a very useful tool to review and analyze past planning decisions or assess a computed trajectory in the future based on presently available data. Figure 5.5 shows how our SAFCS UI enables human operators to select a current `Scenario` time in Coordinated

Universal Time (UTC) and allows them to track the curent time as well as to play, stop, rewind or fast-forward a `Scenario` in time.

A `Session` and its `Scenarios` have only one controlling owner at a time. However, the same `Session` can be obtained by other entities to monitor any activities within them. For example, an SAFCS AM could be the controlling owner of a planning `Session` while an SAFCS UI component would monitor all activities of the former. More details about the SAFCS core `Session`, `Scenario`, `Factory`, `Specification`, and related classes can be found in the packages `session`[1] and `registries`[2] in our SAFCS core software repository.

5.2.2 Aircraft

The SAFCS domain-specific agents are `Aircraft`. As explained in Section 5.2.1, an `Aircraft` is created as a `FactoryProduct` of an `AircraftFactory` that features certain `Capabilities` according to an `Aircraft` `Specification`, as shown in Figure 5.6. Furthermore, all `Aircraft` specialize an `ObstacleSphere` that should be avoided by other traffic. The size of the `ObstacleSphere` depends on the `Capabilities` of the concrete `Aircraft` and should be larger for faster instances. Instead of a sphere, the more general ellipsoid could potentially be a better choice to model the different safety radii that stem from different longitudinal, lateral, and vertical speeds of an `Aircraft`. This option is revisited in Chapter 15.

Figure 5.6 shows only one of the possible class hierarchies leading to a concrete `Aircraft`—in this case the `A320`. Many more `Aircraft` hierarchies can be implemented as required. It should be noted that the `Capabilities` of an `Aircraft` are presently considered static and independent of the environment the `Aircraft` operates in. However, as further examined in Chapter 15, `Capabilities` may be considered dynamic due to failures, systems degradation or recovery, as well as dependent on the environment due to air data intervals constraining the performance of the `Aircraft`. Realizing dynamic `Capabilities` that depend on the environment would without question model the *criticality* of a situation more completely and serve as a valuable input to the SAFCS AM.

[1] https://github.com/stephanheinemann/worldwind/tree/develop/src/main/java/com/cfar/swim/worldwind/session

[2] https://github.com/stephanheinemann/worldwind/tree/develop/src/main/java/com/cfar/swim/worldwind/registries

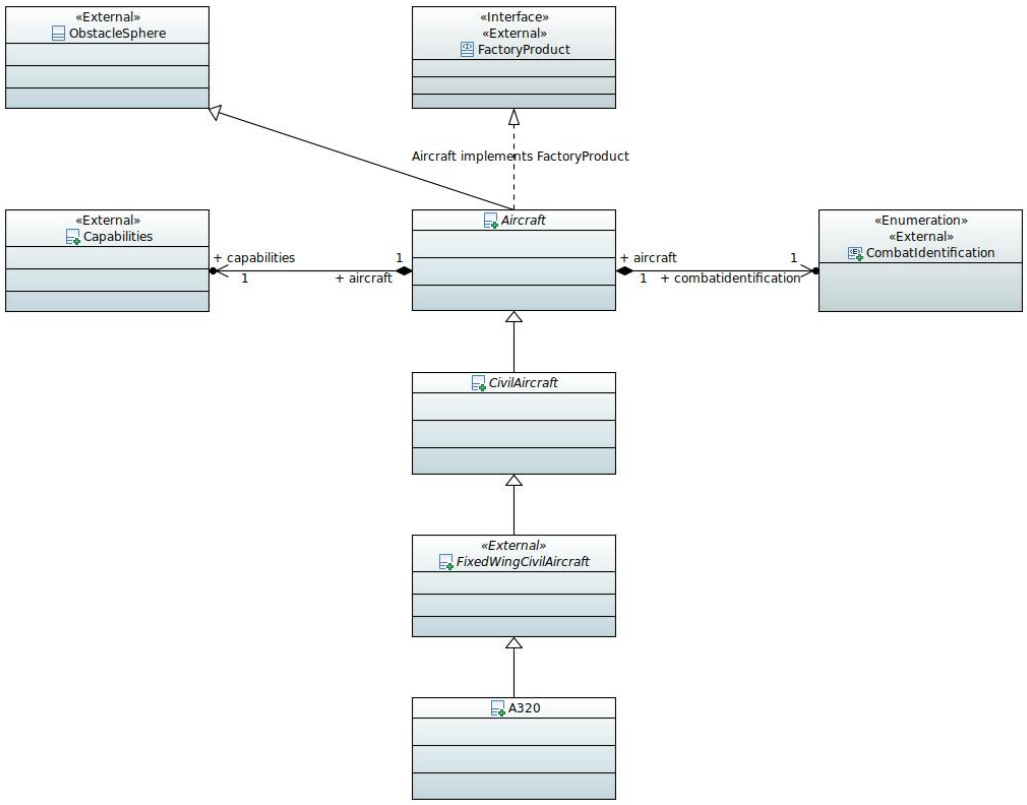


Figure 5.6: SAFCS Core: Aircraft



Figure 5.7: SAFCS UI: Aircraft Symbol

Finally, all `Aircraft` are identifiable via a standardized `CombatIdentifier` that implements the joint military symbology standard 2525[3] and can be depicted accordingly

[3] https://en.wikipedia.org/wiki/NATO_Joint_Military_Symbology

in the SAFCS UI. Figure 5.7 shows how our SAFCS UI displays the A320 according to its CombatIdentifier as a neutral and civilian target. More information about Aircraft, their Capabilities and related classes can be found in the aircraft[4] package in our SAFCS core software repository.

5.2.3 Environments and Obstacles

All Aircraft operate in an Environment and Figure 5.8 shows an excerpt of the Environment class hierarchy that provides contract interfaces for arbitrary concrete Environments. Without providing the details of each Environment contract interface, their main motivation shall be briefly reviewed.

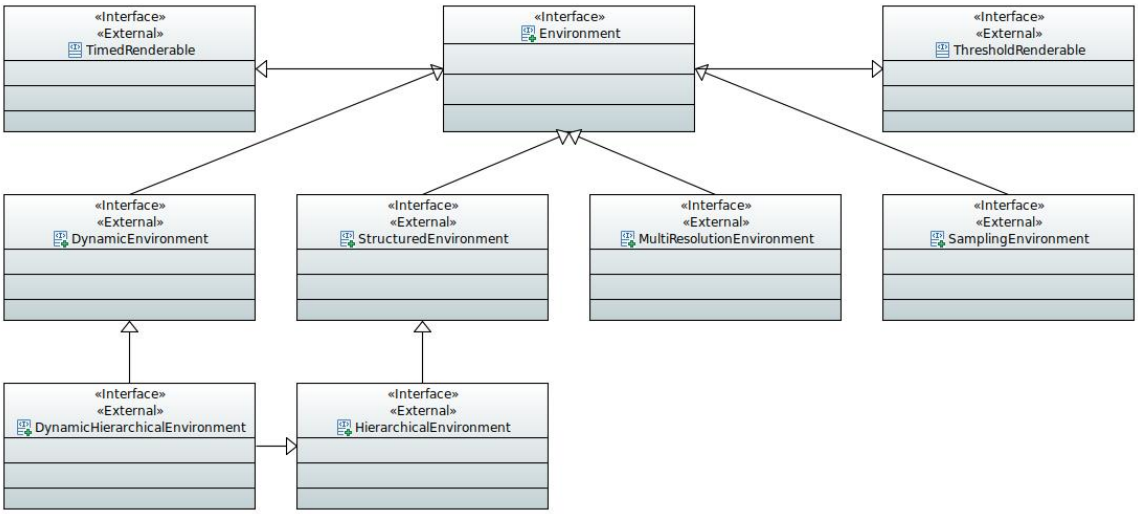


Figure 5.8: SAFCS Core: Environments

A DynamicEnvironment permits the dynamic addition and removal of Obstacles while an Aircraft instance is being planned within it. Hence, concrete DynamicEnvironments model the concept of partial or absent knowledge at the time the planning session is initiated, and any assumptions about costs that may be inflicted on an agent operating within it may be revised at a later point in time.

A StructuredEnvironment features a certain internal structure. This might be, for example, a regular, irregular, hierarchical or non-hierarchical structure. Regardless of the concrete internal structure, a StructuredEnvironment may change its structure while it is being employed in a planning session. Structural changes are usually the result

[4] https://github.com/stephanheinemann/worldwind/tree/develop/src/main/java/com/cfar/swim/worldwind/aircraft

of planning actions that add or remove components to the internal structure of such an `Environment`.

A `HierarchicalEnvironment` is a specialization of a `StructuredEnvironment` since any internal component hierarchy represents a structure. As containers (parents) and containees (children) may be added to a `HierarchicalEnvironment` dynamically, structural changes occur implicitly.

A `DynamicHierarchicalEnvironment` features aspects of both the `DynamicEnvironment` and the `HierachicalEnvironment`. Possessing an internal hierarchical structure, any dynamic `Obstacle` modifications have to be embedded and propagated properly within the components of the hierarchy. Since both its internal structure as well as the embedded costs may change dynamically, such an `Environment` is required to maintain its internal consistency regarding both aspects.

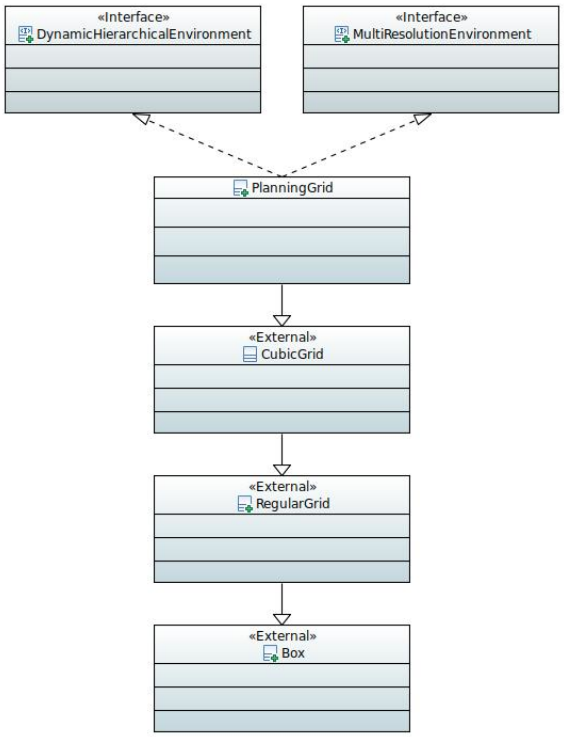


Figure 5.9: SAFCS Core: Planning Grid

A `MultiResolutionEnvironment` may consist of multiple resolutions that focus more or less on certain regions within it. A concrete `HierarchicalEnvironment`, for instance, could very well implement the concept of multiple resolutions where certain regions feature a deeper and more fine grained hierarchy. In general, however, any continuous `Environment` may be discretized by the application of multiple resolutions.

For example, all CGS planning algorithms, such as the members of the A* Algorithm (A*) family, operate on discretized models of their `Environment`. Consequently, grids or meshes featuring static or dynamic resolutions can be employed to serve as a planning ground for these algorithms. The SAFCS core supports multi-resolution `PlanningGrids` that are geometrically derived from the basic geometric classes `Box`, `RegularGrid`, and `CubicGrid`, as shown in Figure 5.9. A `PlanningGrid` is a concrete implementation of both the `DynamicHierarchicalEnvironment` and the `MultiResolutionEnvironment`. For multi-resolution grids each child again is a `PlanningGrid` with its (dynamically changing) resolution. Both the SAFCS UI and AM have access to and may change, that is, refine or coarsen the `Environment` resolution in certain regions as necessary to achieve improved planning performance in challenging conditions.

Regularity, in this case cubic regularity of a `PlanningGrid`, is both an advantageous and restrictive feature, and is discussed further in Chapter 6. Multi-resolution cubic shapes may intuitively not deliver the best planning ground for certain obstacle distributions which leads to the terrain of PRM planners—one of the other planning families supported by the SAFCS core. However, regularity turns out to be advantageous for dynamic repairs in terms of 4D (space and time) consistency.

A `SamplingEnvironment` permits to take samples from it, such as position samples out of a continuum. The methods by which these samples are taken and how they relate to each other with respect to certain metrics, such as distances, are specific to the concrete `SamplingEnvironment`.

For example, all sampling-based planning algorithms, such as members of the RRT and PRM families, operate in and sample vertices from within a continuum. The SAFCS `PlanningContinuum` shown in Figure 5.10 supports these families of algorithms. We derived a `PlanningContinuum` geometrically from a `Box`, but clearly other spatial extents are possible and could be supported. Realizing both a `SamplingEnvironment` and `StructuredEnvironment`, sampled vertices, that is, `Positions` as well as `Edges`, including more special `DirectedEdges`, can be dynamically added to or removed from a `PlanningContinuum`. Currently, the `PlanningContinuum` is modelled to be non-hierarchical. However, the endpoints of `Edges` could define a sub-continuum in various ways. Furthermore, the resolution of a `PlanningContinuum` is presently only used to determine collisions with high-resolution terrain of a certain magnitude but not for sampling. The different options for `PlanningContinuums` are, thus, revisited in Chapter 15.

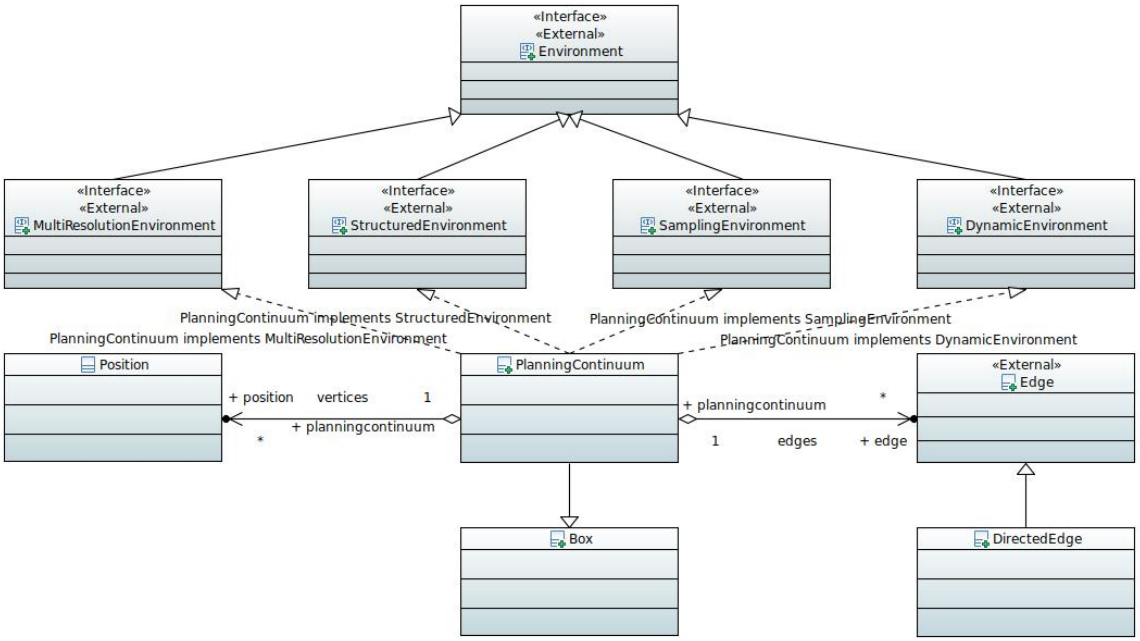


Figure 5.10: SAFCS Core: Planning Continuum

The `Environment` interface unifies all current and future supported planning environments. All concrete `Environment` instances shall have the ability to statically embed obstacles of different space and time extensions during a planning session. Hence, all `Environments` may embed concrete instances of the `Obstacle` interface.

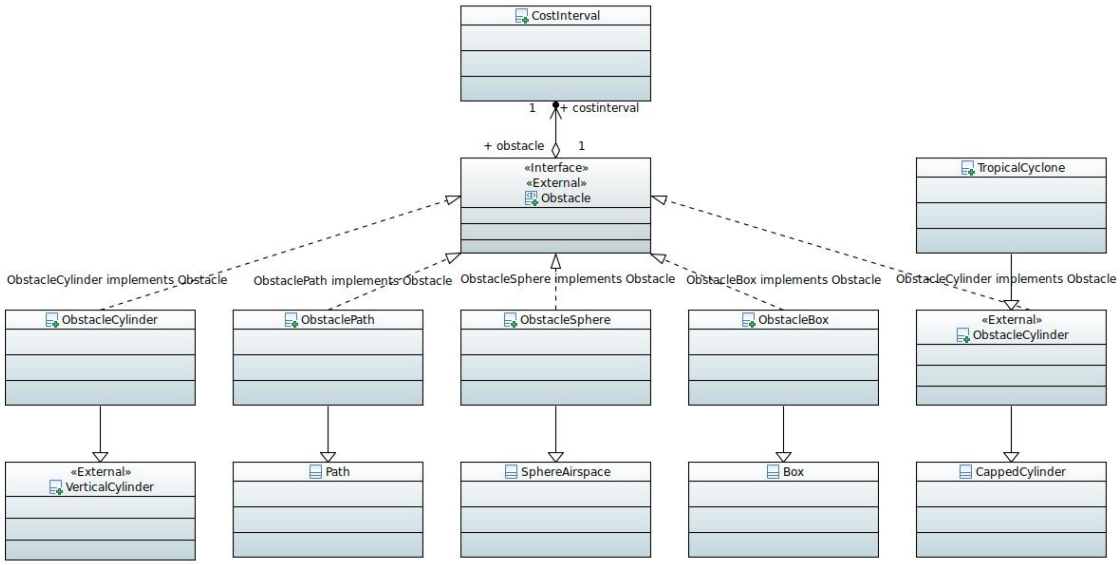


Figure 5.11: SAFCS Core: Obstacles

The SAFCS core Obstacle hierarchy is shown in Figure 5.11. Different concrete geometric shapes represent the space extension of the associated Obstacle cost. These basic geometric shapes are then extended by concrete domain-specific Obstacles with their respective properties. For example, a TropicalCyclone is an aviation weather domain-specific Obstacle that extends an ObstacleCylinder.

The costs associated with an Obstacle, furthermore, depend on time. As indicated before, the cost of an Obstacle may be relevant at one time and irrelevant at another. In general, an Obstacle space usually covers a time span (duration) featuring a specific start and end time. These can be forecast durations, such as for weather, traffic, airspace restrictions, or facility outages. This idea is embraced by the temporality concept of SWIM data and is further discussed in Section 5.2.4. Since all embedded obstacles represent temporality-enabled SWIM data that are defined in terms of validity durations, that is, a data item is valid from a start until an end time specified in UTC, all concrete Obstacles and Environments need to support the concept of costs within a time span—the CostInterval, as depicted in the class diagrams of Figures 5.11 and 5.12, respectively.

The concrete cost association has to be determined by standards of the operation and is crucially important and the basis for all planning and decision making algorithms. The SAFCS core employs a CostMap in order to associate a certain dataset with a fix cost value. The CostMap assigns Obstacle costs in the range from −100 to 100 and, therewith, acknowledges the fact that previously reported data may be cancelled by updated reports as well as that off-nominal conditions could actually be advantageous. For example, a new NOTAM may indicate the outage of an aerodrome or navigational facility until a cancelling NOTAM is issued. The first NOTAM would lead to the generation of an Obstacle without a finite expiration time, whereas the second NOTAM would generate an Obstacle possessing the inverted cost of the first one. Another example would be a region of favourable upper winds that would lower the existing nominal base costs within an Environment as compared to a situation without winds.

The SAFCS employs an efficient CostInterval tree implementing a binary tree of time intervals with their associated costs. Each voxel of a PlanningGrid and the Edges of a PlanningContinuum aggregate such a tree. When an Obstacle is embedded into or removed from an Environment, the aggregated CostInterval trees grow or shrink, respectively. As a balanced binary tree, this augmented tree is always height-balanced during any modification, such as insertion and removal of an interval, never possessing a depth difference of more than 1 between any of its branches. Each CostInterval oc-

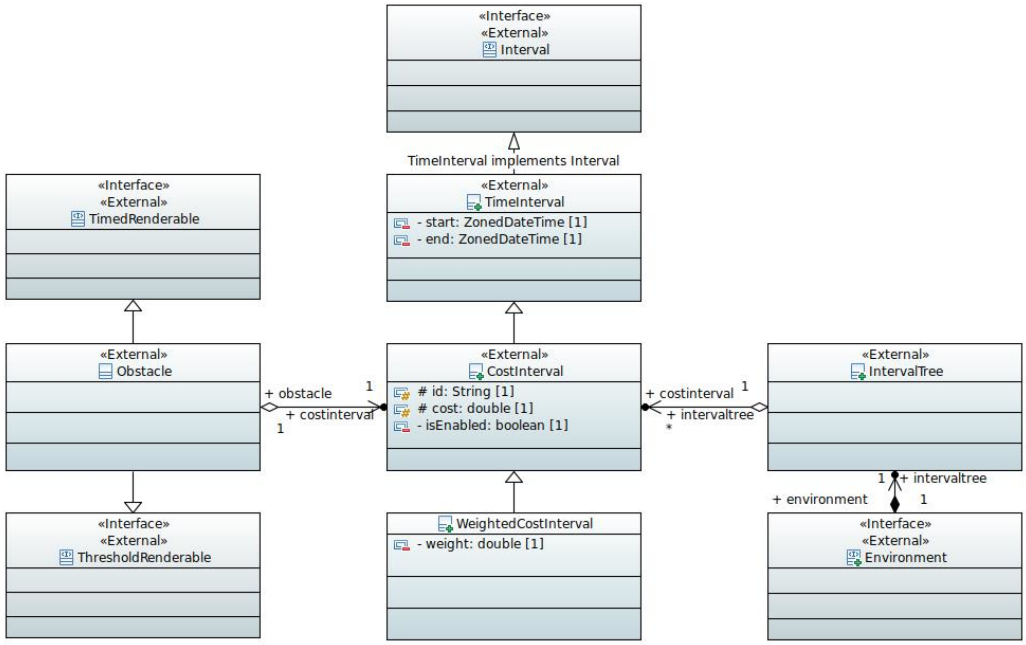


Figure 5.12: SAFCS Core: Cost Intervals

cupies one node in this tree, resulting in a space complexity of $\mathcal{O}(n)$ where n is the number of stored intervals. Querying the intersection intervals of a specified time instance or interval results in a time complexity of $\mathcal{O}(n+m)$ where n and m are the number of stored and intersecting intervals, respectively. Interval trees storing `CostIntervals` associated with SWIM data can, hence, be used for trajectory planning with any `Environment` and agent where passage time intervals can be estimated for discrete space portions. These discrete portions can be voxels, legs, sub-graphs, or any other hierarchical subcomponent or internal structure of the concrete `Environment`.

`CostIntervals` can further be refined into `WeightedCostIntervals`, as shown in Figure 5.12. `WeightedCostIntervals` assign a weight to the cost value of a `CostInterval`. This can be useful to differentiate between recent and older reports. For example, a weather forecast that was issued a week ahead of the `Environment` time should be weighted much less than a forecast that was issued only an hour before.

During the planned traversal of an `Environment` structure, such as a voxel or `Edge`, the time-relevant costs can, thus, be efficiently looked up recursively (for `HierarchicalEnvironments`) and incorporated into the total cost calculation. The functional aspects of this lookup and cost calculation are discussed in detail in Chapters 6, 7, and 8. In order to display certain costs not exceeding a threshold within a certain interval in our SAFCS UI, all `Environments` and `Obstacles` implement both the

ThresholdRenderable and TimedRenderable interfaces, as shown in Figures 5.8 and 5.12, respectively.

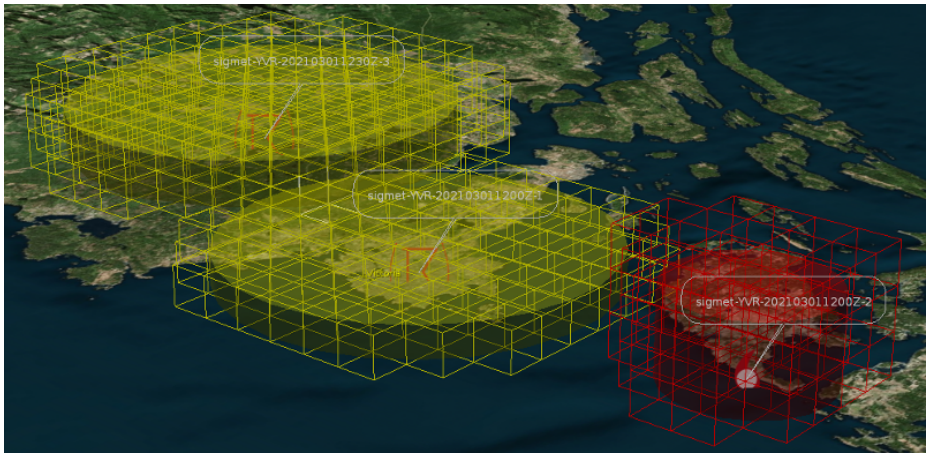


Figure 5.13: SAFCS UI: Grid Obstacles

Providing a more visual example, Figure 5.13 shows an SAFCS UI planning Scenario aggregating a single-resolution PlanningGrid with three embedded obstacles. It should be recalled that the embedded obstacles change the space they occupy in time. They may appear, disappear, move, become larger and smaller, and represent very different concerns to the aviation domain. Traversing Obstacles may be more or less costly, acceptable or not acceptable according to an operational cost and risk policy, as described in Chapters 6, 7, and 8. Recall as well that an Obstacle could represent adverse weather, restricted airspace, conflicting traffic, limited ground facilities, adverse terrain and others. It is the operational cost and risk policy that ultimately decides whether or not an Environment being affected by the CostIntervals of embedded Obstacles can be penetrated by a planning algorithm.

Another example is shown in Figure 5.14 and depicts how our SAFCS UI displays the CostIntervals of an embedded Obstacle for a PlanningContinuum in case of an RRT. Again, our SAFCS UI clearly permits to identify regions of higher costs within the internal structure, in this case the Edges, of a concrete planning Environment. More de-

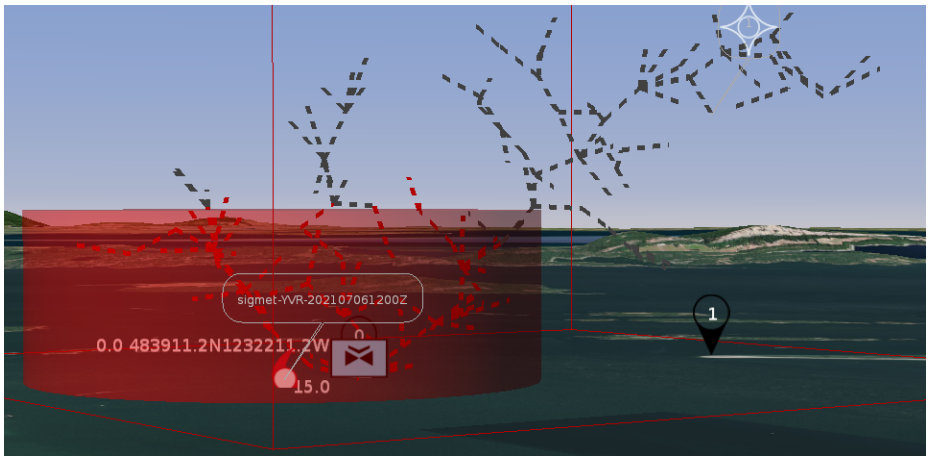


Figure 5.14: SAFCS UI: Continuum Obstacles

tails about `Environments`, `Obstacles` and related classes can be found in the packages `environments`,[5] `geom`,[6] and `render`[7] in our SAFCS core software repository.

5.2.4 Connections and Obstacle Reporting

The SAFCS is designed to be a self-adaptive autonomic system that requires to obtain updates from its AC in order to adapt and handle changes of the `Aircraft Capabilities`, the planning `Environment`, or even set goals and operational constraints, such as a modified cost and risk policy, accordingly. The `Connection` interface of the SAFCS satisfies precisely that requirement.

As explained in Section 5.2.1 and depicted in Figure 5.15, an SAFCS core `Connection` is a `FactoryProduct` of a capable connection factory. The SAFCS AC includes ATC, OPS, SWIM, and the aircraft capabilities. Therefore, the respective connections to these entities are designed into the SAFCS core: `AtcConnection`, `OpsConnection`, `SwimConnection`, and `Datalink`.

For the initial research scope of the SAFCS, connections to ATC and OPS are not intended to be implemented. The ATC communication would rely on protocols such as or similar to CPDLC and request necessary ATC clearances to be granted access to certain airspaces. Airspaces that require clearances which have not been obtained are equiva-

[5] https://github.com/stephanheinemann/worldwind/tree/develop/src/main/java/com/cfar/swim/worldwind/environments

[6] https://github.com/stephanheinemann/worldwind/tree/develop/src/main/java/com/cfar/swim/worldwind/geom

[7] https://github.com/stephanheinemann/worldwind/tree/develop/src/main/java/com/cfar/swim/worldwind/render

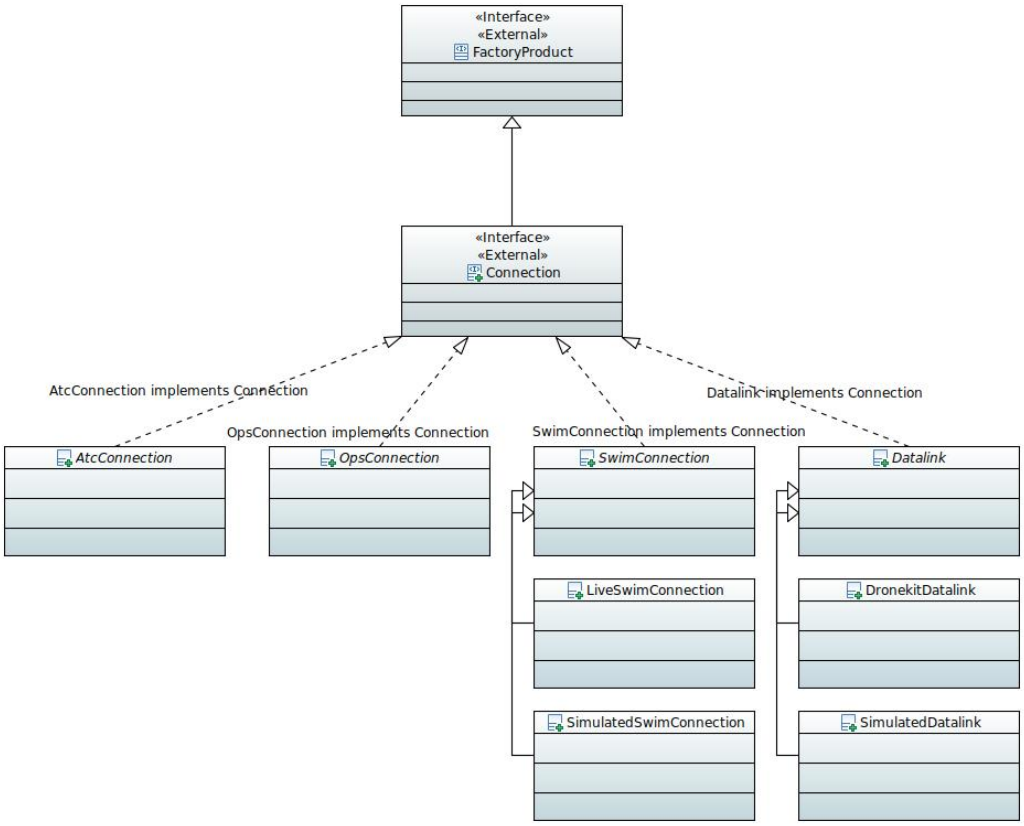


Figure 5.15: SAFCS Core: Connections

lent to high cost obstacles that may only be penetrated under certain higher operational risk policies (e.g., military activities). These risk and cost policies together with the initial goal may be updated via an `OpsConnection`. The `OpsConnection` would rely on protocols such as or similar to ACARS. Simulated variants of these connections may provide some valuable insights for adaptive planning at a later stage of the SAFCS research.

Currently, the SAFCS provides a `SimulatedSwimConnection` in order to consume and embed SWIM-related obstacles. Furthermore, `DronekitDatalink` and `SimulatedDatalink` connections allow to communicate with both real and simulated aircraft, respectively. All connection classes are intended to permit bi-directional communication. For example, air-traffic control clearance requests are sent to an `AtcConnection`, whereas granted clearances or instructions are received from the latter. Take-off or landing commands are sent (uplink) to a concrete `Datalink`, whereas telemetry data is received (downlink) from it. Aircraft sensors, such as weather radar, may pick up relevant obstacles to be published using a concrete `SwimConnection` while

receiving subscribed SWIM data. The last example resembles nicely the common practice of in-flight PIREPs occasionally issued by human actors.

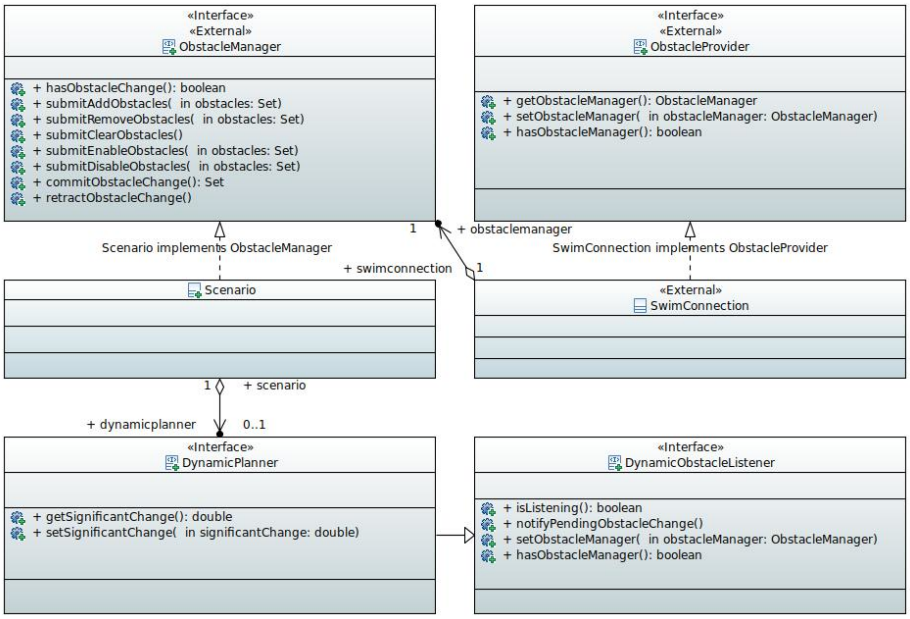


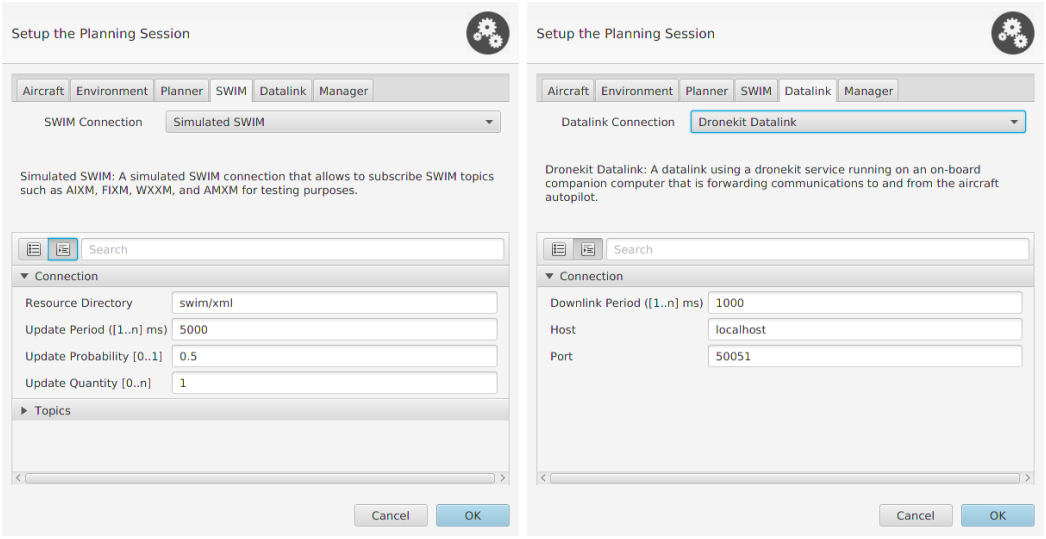
Figure 5.16: SAFCS Core: Obstacle Reporting

Introducing and embedding new `Obstacle` instances into the concrete planning `Environment`, while running an active planning scenario, requires the particular architectural approach shown in Figure 5.16. This class diagram exposes a few more signature details as they are more relevant for the key dynamic planning algorithms of each supported family.

As can be seen in Figure 5.16, any SAFCS `Connection` that provides `Obstacle` updates for the active planning `Scenario` implements the interface `ObstacleProvider`. An `ObstacleProvider`, such as the `SwimConnection`, always aggregates an `ObstacleManager`. It is the `ObstacleManger` that allows the `ObstacleProvider` to submit, commit, or retract `Obstacles` to or from an active planning `Scenario`.

The `ObstacleManager` serves as a facade to the planning `Scenario` allowing any `DynamicPlanner` to commit previously submitted obstacles into its `Environment` at particular elaboration steps during the trajectory planning computation. Therefore, each concrete `DynamicPlanner` is also a `DynamicOstacleListener` which is notified about pending obstacle changes by its `ObstacleManager`. Since `DynamicPlanners` usually do not terminate by themselves but instead keep incorporating contextual changes

and revising established trajectories as LifelongPlanners, they can be terminated and recycled explicitly.



(a) SAFCS UI: SWIM Setup **(b)** SAFCS UI: Datalink Setup

Figure 5.17: SAFCS UI: Connections Setup

Figure 5.17 shows how Connection Specifications are customized for a SwimConnection and a Datalink, respectively, using our SAFCS UI. More details about Connections and related classes can be found in the package connections[8] in our SAFCS core software repository.

5.2.5 Planners

The Planner class hierarchy is shown in Figure 5.18. More specialized Planners include the AnytimePlanner that incrementally improves bounded sub-optimal solutions, and the DynamicPlanner that is able to cope with previously unknown Obstacles and, therefore, addresses uncertainty while often being realized as a LifelongPlanner that requires explicit termination as dynamic changes may occur at any future time. Furthermore, an OnlinePlanner may ideally feature all of the above but, most importantly, is required to monitor the progress of and act on the data received from a Datalink source.

All Planners are created as FactoryProducts of a PlannerFactory that requires a Setup using a Specification describing the concrete Planner and its Properties.

[8] https://github.com/stephanheinemann/worldwind/tree/develop/src/main/java/com/cfar/swim/worldwind/connections

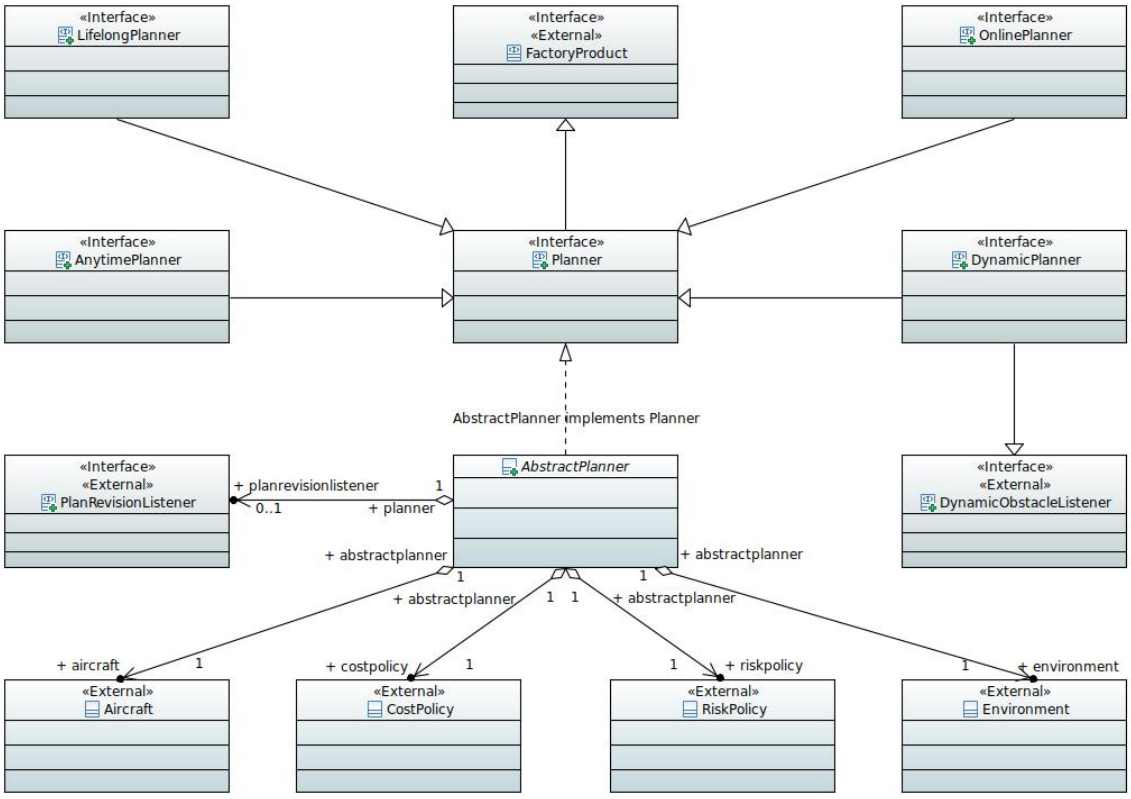


Figure 5.18: SAFCS Core: Planners

An `AbstractPlanner` abstracts some of the features shared by all concrete `Planner` instances. All `Planner` instances plan a trajectory for a particular `Aircraft` in a particular `Environment`, as explained in Sections 5.2.2 and 5.2.3, respectively. Furthermore, all concrete `Planners` operate according to a defined `CostPolicy` and `RiskPolicy`.

The operational `CostPolicy`[9] of a `Planner` is defined by OPS and expresses how optimistic or pessimistic the `Planner` assesses its `Environment`. For example, in case of a `PlanningGrid`, a step through the `Environment` may be affected by a number of voxels of possibly different resolutions and, thus, covered space. These voxels aggregate their own `CostInterval` trees and, consequently, may incur different costs for the same `TimeInterval`. The `Planner` may be configured to assume the minimum, maximum or average costs depending on the nature of the airborne mission or the intentions of an AM. Different cost policies, in turn, lead to different trajectories.

Figure 5.19 shows how each supported operational `CostPolicy`, that is, `MINIMUM`, `AVERAGE`, and `MAXIMUM`, results in a different cost assessment of same step taken by a

[9] https://github.com/stephanheinemann/worldwind/blob/develop/src/main/java/com/cfar/swim/worldwind/planning/CostPolicy.java

Planner through its Environment. In this example the Environment is affected by four distinct CostIntervals with the individual costs of 5, 20, 40, and 45, respectively. All these CostIntervals, in turn, are relevant for the Estimated Time Enroute (ETE) calculated by the Planner to transition the Environment between the Estimated Times Over (ETOs) at two dedicated waypoints 1 and 2, respectively. In order to consider the operational CostPolicy, the ETE is divided into sub-intervals, each representing the accumulated costs of all applicable CostIntervals. The individual accumulated sub-interval costs are then combined according to the CostPolicy. In the AVERAGE case, the accumulated cost of a sub-interval is multiplied by its sub-interval ETE normalized to the entire CostInterval ETE, that is, the corresponding fractional cost is determined first and then taken into account for the cost of the entire interval. For example, 0.2 is the cost that represents 4% of the corresponding sub-interval with a cost of 5.

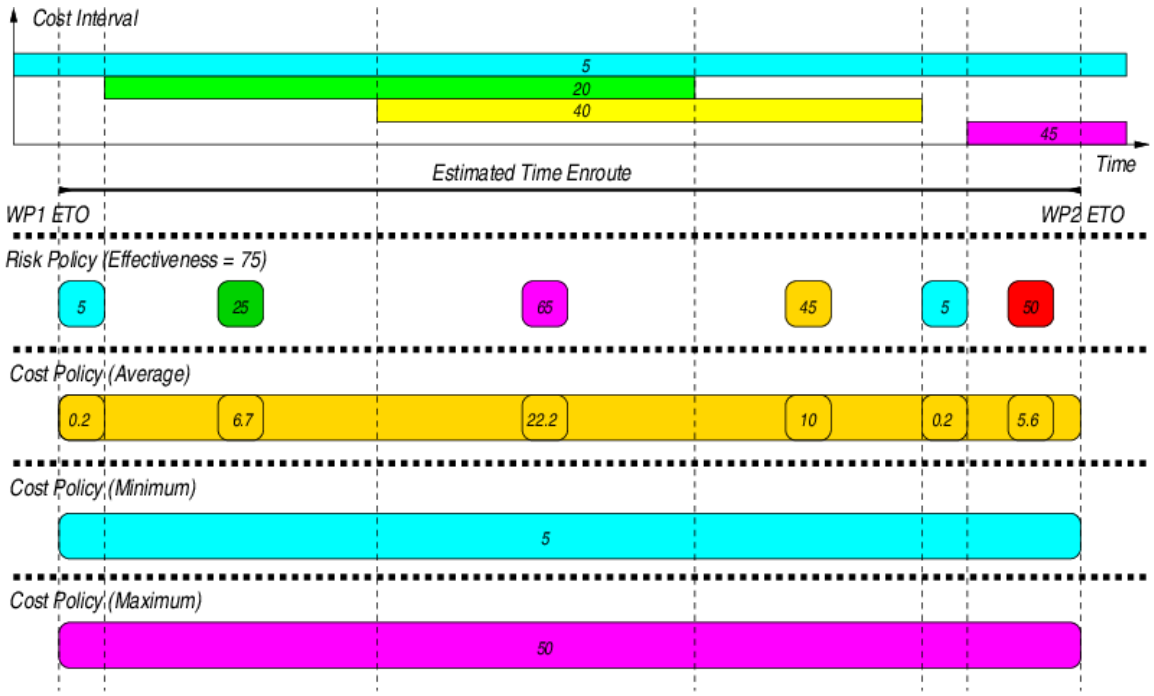


Figure 5.19: SAFCS Core: OPS Risk and Cost Polcies (1)

The operational RiskPolicy[10] of a Planner is also defined by OPS and determines how much local risk the Planner is willing to take. Consider the example of a far away but safe bridge to cross a dangerous creek. The direct passage through the creek may incur a very high cost for a short distance, whereas the bridge detour may result in a

[10] https://github.com/stephanheinemann/worldwind/blob/develop/src/main/java/com/cfar/swim/worldwind/planning/RiskPolicy.java

higher total cost for a long distance. Depending on the `RiskPolicy`, the `Planner` may be configured to accept the very high local cost to minimize the total cost of the computed trajectory. A more cautious policy may instruct the `Planner` to avoid the higher local risk instead. A `RiskPolicy` is usually established by OPS and flight crews as a general policy or personal minima, respectively. It describes what risk level is acceptable to complete a mission and achieve the planned goal or otherwise abandon it. This is a non-negotiable level and independent of any current situation. It is important to realize that the `RiskPolicy` does not increase simply because the `Environment` is subject to higher risk. In fact, such a behavior is considered unsafe and commonly referred to as *pilot get-homitis*. For example, if weather deteriorates at the destination, airspaces become restricted, or navigation facilities unserviceable beyond the acceptable risk level, the mission should not be started in the first place or the mission goal abandoned if already airborne.



Figure 5.20: SAFCS Core: OPS Risk and Cost Polcies (2)

The potential to abandon a mission, consequently, leads to the requirement of in-flight alternates or unplanned landings. It is important to, furthermore, emphasize that the `RiskPolicy` is not a risk-seeking policy and merely provides the option to accept a certain risk, that is local cost, in order to achieve the goal at the minimum global cost. However, the `RiskPolicy` has to be boosted in one case only: If an airborne rational

agent is unexpectedly and suddenly exposed to higher costs, it is required to boost its RiskPolicy to escape that threat. For example, the agent might suddenly be exposed to severe in-flight turbulence or icing. This region of higher cost has to be dealt with and planning has to continue to leave the hazardous region before resuming the established RiskPolicy. The SAFCS Planners support different RiskPolicy levels (cost thresholds) which need to be satisfied to consider or otherwise disregard a costly step. The available RiskPolicy levels with increasing risk are AVOIDANCE, PROBE, SAFETY, EFFECTIVENESS, and IGNORANCE. A RiskPolicy has to be satisfied, or otherwise the step costs are boosted to an unacceptable level. Tables 5.1 summarize the available operational policies. The RiskPolicy literals and thresholds are presently fix but could very well be adjustable by the operator, as discussed in Chapter 15.

Table 5.1: Operational Policies

(a) Risk Policies

RiskPolicy	Cost Threshold
AVOIDANCE	0
PROBE	25
SAFETY	50
EFFECTIVENESS	75
IGNORANCE	100

(b) Cost Policies

CostPolicy
MINIMUM
AVERAGE
MAXIMUM

Figure 5.20 shows how the situation depicted in Figure 5.19 is assessed differently by applying a more conservative operational RiskPolicy. In this example, the RiskPolicy SAFETY can not be satisfied by one of the aggregated sub-interval costs. Consequently, the final costs would exceed the acceptable level of risk and are boosted accordingly. In summary, both CostPolicy and RiskPolicy enable OPS to assess the *criticality* of a situation differently and according to their established or adjusted SOPs.

As viable trajectories are being found, improved or discarded, registered PlanRevisionListeners are notified. Examples of PlanRevisionListeners include the SAFCS UI in order to display the revision, or a Datalink to upload a revised trajectory to the aircraft guidance system.

OnlinePlanners, that feature a Datalink connection to the planned Aircraft also require dedicated Datalink Communication sequences, as shown in Figure 5.21.

As mentioned, OnlinePlanners are very similar to their generalizing offline counterparts in terms of planning functionality but in addition monitor the progress of their Datalink sources in space and time with respect to the established and continuously

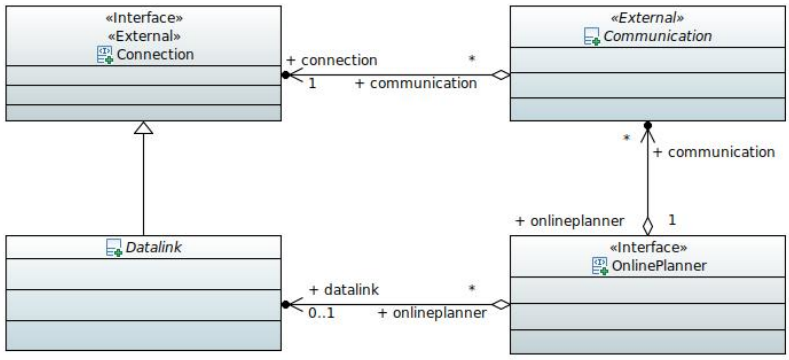


Figure 5.21: SAFCS Core: Online Planner

revised trajectories. The mentioned `Communication` sequences over `Datalink` enable the `Planner` to perform take-off and landing maneuvers as well as uploading and downloading mission parameters to and from the aircraft, respectively. In addition to revising and improving generated trajectories while the `Datalink` source is airborne, the `OnlinePlanner` recognizes off-track, that is, in space and time, as well as exceeded `RiskPolicy` situations and re-plans accordingly in order to establish compliant trajectories and escape potential threats. The possibility of an airborne `Datalink` source being directly exposed to unanticipated costs at its present position or at the planned destination, consequently, leads to the requirement of `RiskPolicy` boosting as well as in-flight alternates and unplanned (precautionary) landing `Communications`.

Each of the implemented `Planners` operates on dedicated `Waypoint` classes which again leverage generalization and specialization in order to reuse a significant amount of logic, as discussed in greater detail in Chapters 6, 7, and 8. These are extensions of the more basic `Position`, which is defined in terms of latitude and longitude of earth-geographic coordinates as well as elevation (altitude), with additional planning times and important attributes to support the logic of their associated concrete `Planner`.

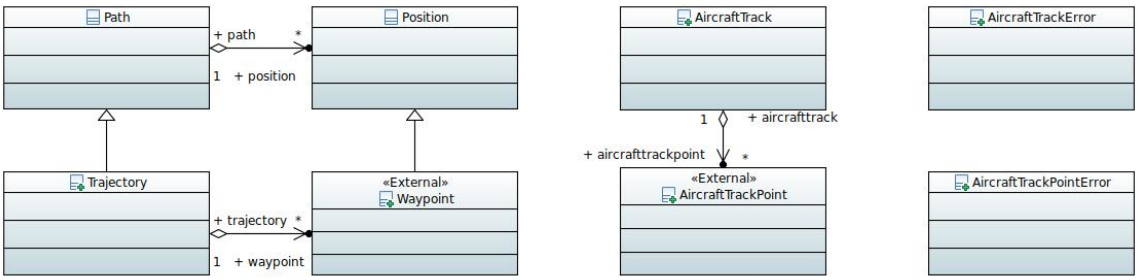


Figure 5.22: SAFCS Core: Waypoints

It should be emphasized that a Waypoint in practice does not equal another Waypoint simply because it occupies an identical Position, since it relates to both space and time. In practice a Position could be revisited several times. However, in order to limit the computational complexity associated with 4D Waypoints, the SAFCS implements a more restrictive policy. To this extent, Waypoints may not be revisited by SAFCS Planners for a single-part plan. Therefore, no single-part Trajectory may contain multiple Waypoints with equal Positions. Intuitively, this restriction prevents an agent from moving back and forth between Positions at different times in order to avoid costly Obstacles. However, this situation can be avoided altogether by modifying departure times or, more generally, introducing Waypoint slots. This interpretation is similar to the concept of in-flight orbit or holding procedures and is revisited in Chapter 15. The restriction does, however, not apply to multi-part plans with multiple POIs and, therefore, an agent may clearly return to its origin after having visited a POI using the reverse Trajectory.

As can be seen in Figure 5.22, concrete Waypoints form the basis of a planned Trajectory. It should be noted again that a Trajectory is defined in terms of both space and time and, consequently, consists of 4D Waypoints, whereas a more basic Path is defined in space only and, therefore, consists of Positions only.

Since a planned or desired Trajectory is not identical to the actual track flown by an aircraft after having uploaded a new or previously revised Trajectory via the Datalink, an AircraftTrack consisting of AircraftTrackPoints can be recorded by the SAFCS core as part of the downlink telemetry in order to display and evaluate the achieved navigational performance. The required performance is not achieved if operator-defined maximum AircraftTrackErrors or AircraftTrackPointErrors are being exceeded. These allowable errors are defined in terms of both space and time and can be different for different phases of the flight, such as take-off, enroute, and landing. The OnlinePlanner compares the actual with the maximum allowable track errors to determine whether or not an off-track situation has developed and initiates re-planning if necessary.

In order to visually compare the planned and achieved tracks, the SAFCS UI supports playing an established Trajectory while tracking AircraftTrackPoints from the downlink telemetry. Recall, that the SAFCS may employ either a SimulatedDatalink as well as a DronekitDatalink connection to a real-world source. Furthermore, the SAFCS UI allows to monitor the recorded AircraftTrackPoints from different perspectives, including a synthetic cockpit First Person View (FPV) even if there is no air-

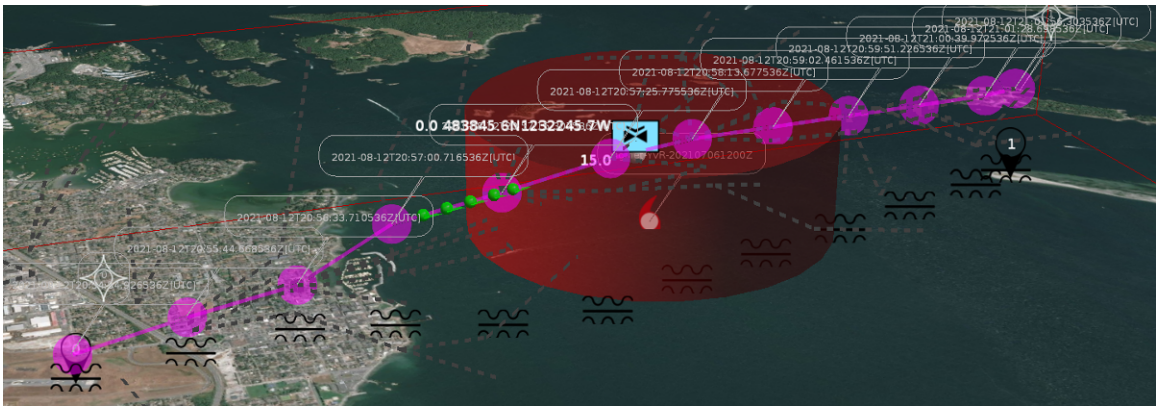


Figure 5.23: SAFCS UI: Datalink Tracking

borne video downlink available. Figure 5.23 shows the SAFCS UI during an online planning scenario. Several of the previsously discussed SAFCS core elements, such as POIs (0 and 1 on black markers), a planned `Trajectory` containing 4D `Waypoints` (magenta in the air, black over ground), the planned `Aircraft` position (black on blue) as well as the monitored `Datatlink` source `AircraftTrackPoints` (green), can be identified here. More details about `Planners` and related classes can be found in the packages `planners`,[11] `planning`,[12] and `tracks`[13] in our SAFCS core software repository.

Chapters 6, 7, and 8 cover the planning algorithm hierarchies of the currently supported planning families CGS, RRTs, and PRMs in greater detail and, in particular, elaborate on certain functional aspects and modifications.

5.3 SAFCS Management Architecture

The SAFCS management component hosts higher-level AI to select registered `ManagedPlanners`, that is, lower-level AI, of the SAFCS core component and improve their performance in the short as well as ideally in the long-term over many independent managed planning sessions. The higher-level AI of the SAFCS management component comprises registered `AutonomicManagers` featuring different capabilities (e.g., model-free or model-based). As such, the SAFCS management component is designed

[11] https://github.com/stephanheinemann/worldwind/tree/develop/src/main/java/com/cfar/swim/worldwind/planners

[12] https://github.com/stephanheinemann/worldwind/tree/develop/src/main/java/com/cfar/swim/worldwind/planning

[13] https://github.com/stephanheinemann/worldwind/tree/develop/src/main/java/com/cfar/swim/worldwind/tracks

to be extensible with respect to the concrete supported managers in a similar manner as the SAFCS core component is designed to be extensible with respect to its supported `Planners`.

Our SAFCS management component and its conceptual interfaces with the SAFCS core and UI components are shown in the component diagram of Figure 5.24. Recall and compare this component diagram with Figures 2.12, 2.13, and 4.2 as they are related and emphasize different perspectives of the same concept: An Autonomic Element hosts an Autonomic Manager that manages its Managed Element while being aware of its Autonomic Context. The AE proceeds according to higher level operational *goals* and *policies*.

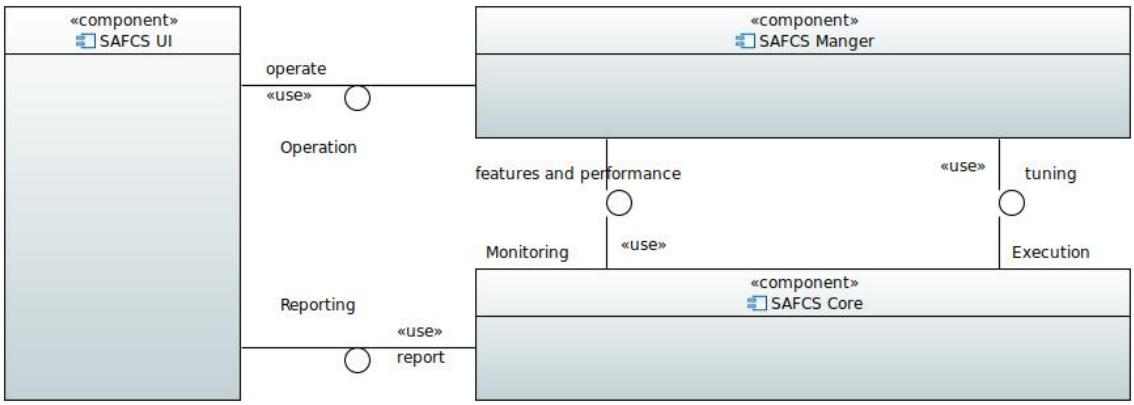


Figure 5.24: SAFCS Management: Autonomic Manager Interfaces

As shown in Figure 5.24, the SAFCS UI component interfaces the management component via a conceptual `Operation` interface. This interface allows OPS to specify the goals and policies of a managed planning session. More precisely, take-off, landing, intermediate, and alternate POIs with their respective altitudes and potential actions are specified here as goals. Furthermore, the discussed `CostPolicy` and `RiskPolicy` as well as operational tolerances of online `ManagedPlanners` are defined as policies. It should be explicitly emphasized that the `Operation` interface does *not* specify concrete `Planners` nor associated `Tunings` to achieve the desired operational goals within the set operational policies. The portfolio of `ManagedPlanners`, their selection and tuning is entirely opaque to OPS and covered autonomously by the AM.

The AM interfaces its ME, that is, the SAFCS core, via the two conceptual interfaces `Monitoring` and `Execution` and, hence, follows the interface design of a self-adaptive autonomic system shown in Figure 2.12. More precisely, the AM monitors discriminating `Features` of a managed `Scenario` as well as the resulting `Performance` of its

ManagedPlanners within the former. Both the architectural design and internals of Features and Performance are discussed in greater detail in Sections 5.3.3, 5.3.4 and 9.2, 9.4, respectively. Based on the observed Features, Performance, and possibly an internal model, a concrete Tuning is then derived by the AM for each of the supported ManagedPlanners. The derived Tunings ideally allow the ManagedPlanners to perform better within a Scenario of certain Features and are passed to the SAFCS core via the conceptual Execution interface. Recall from Section 5.2.4 that while the UI represents one of the OPS interfaces to the AE, the SAFCS core itself may be equipped to establish various Connections with it. These Connections may inject new information and, in turn, modify the discriminating Features of the active Scenario of the managed Session.

Finally, the progress of the AE in terms of selected ManagedPlanners, their Tunings and the initially computed or revised Trajectory is passed to the SAFCS UI via the conceptual Reporting interface.

5.3.1 Managers

The SAFCS AM component supports an extensible set of AutonomicManagers and the associated design is shown in Figure 5.25.

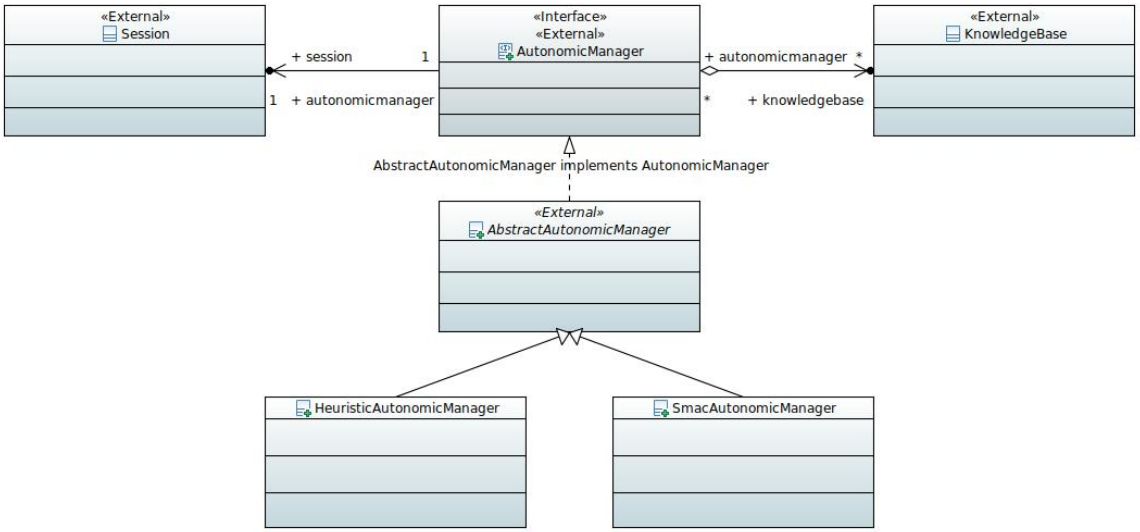


Figure 5.25: SAFCS Management: Autonomic Managers

An AutonomicManager of the SAFCS AM manages a planning Session of the SAFCS core. While doing so it collects its experience in a KnowledgeBase which can be per-

sisted and extended over separate and unrelated management sessions. Whether or not an AutonomicManager, furthermore, uses this KnowledgeBase to guide its selection and tuning activities, depends on its concrete realization. Currently, our SAFCS AM supports a HeuristicAutonomicManager and a SmacAutonomicManager. While the former employs a *model-free* approach to selection and tuning, the latter is *model-based* and internally uses a random forest of regression trees to learn suitable average Tunings based on extracted Features of the managed Scenario. Therefore, the HeuristicAutonomicManager establishes Tunings heuristically and based on extracted Features only. The SmacAutonomicManager, on the other hand, can be primed by the same heuristics but may be able to improve them during a training phase and use the gained experiences from its extended KnowledgeBase during the tuning process of an execution phase. Recall Section 3.3 for the basic ideas behind SMAC. Both the HeuristicAutonomicManager and the SmacAutonomicManager are discussed in detail in Chapter 9.

5.3.2 Managed Planners

An AutonomicManager selects and tunes registered ManagedPlanners to achieve the best possible performance for its managed planning Session. As shown in Figure 5.26, all ManagedPlanners extend the necessary interfaces to act as AnytimePlanner, DynamicPlanner, LifelongPlanner, and OnlinePlanner.

As such, a ManagedPlanner recognizes the limitations of bounded rationality, partial knowledge, and continuously revises a computed Trajectory online during a live airborne mission. Furthermore, it accepts ManagedGoals (e.g., relevant mission POIs) and can be queried about its current PlannerPerformance. ManagedPlanners are Callables that deliver a Trajectory and can, consequently, be invoked by a suitable ExecutorService. More precisely, the SAFCS AutonomicManager assigns each ManagedPlanner to a separate core processor in order to execute in parallel and obtain comparable PlannerPerformances.

Since the SAFCS core currently supports the three distinct Planner families CGS, RRTs, and PRMs, their anytime, dynamic, lifelong, and online key algorithms serve as the basis for the ManagedGridPlanner, ManagedTreePlanner, and ManagedRoadmapPlanner, respectively.

Figure 5.27 shows an active management Session in our SAFCS UI. The managed active Scenario is called *Default Scenario* in this example. In this case, the

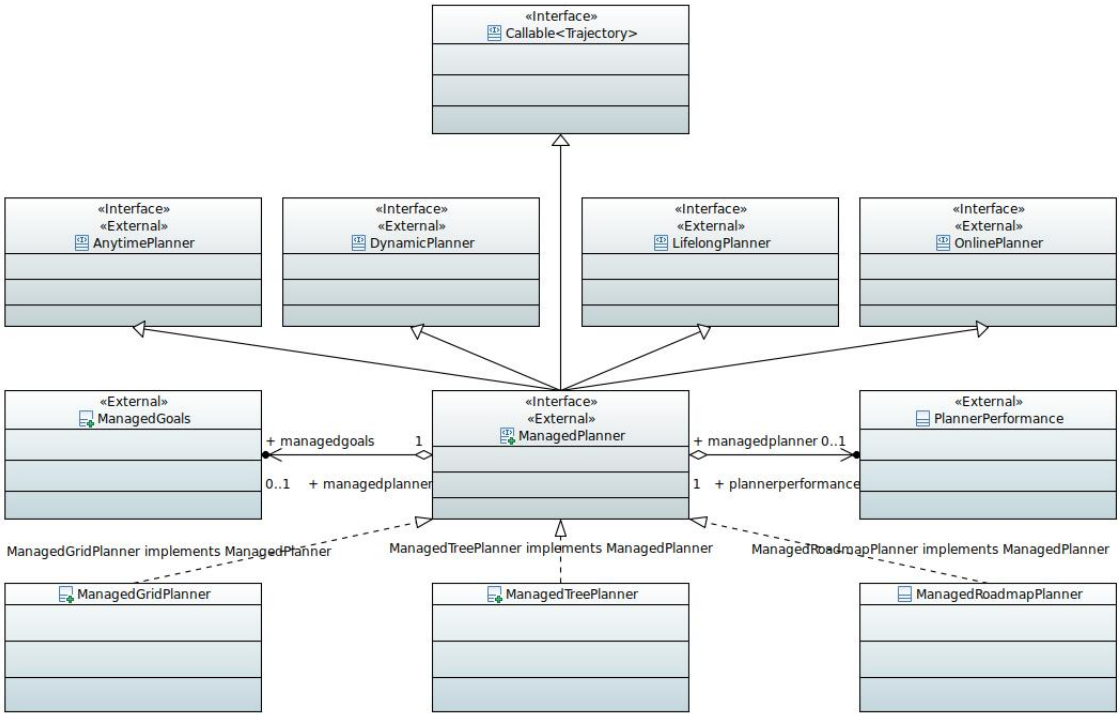


Figure 5.26: SAFCS Management: Managed Planners

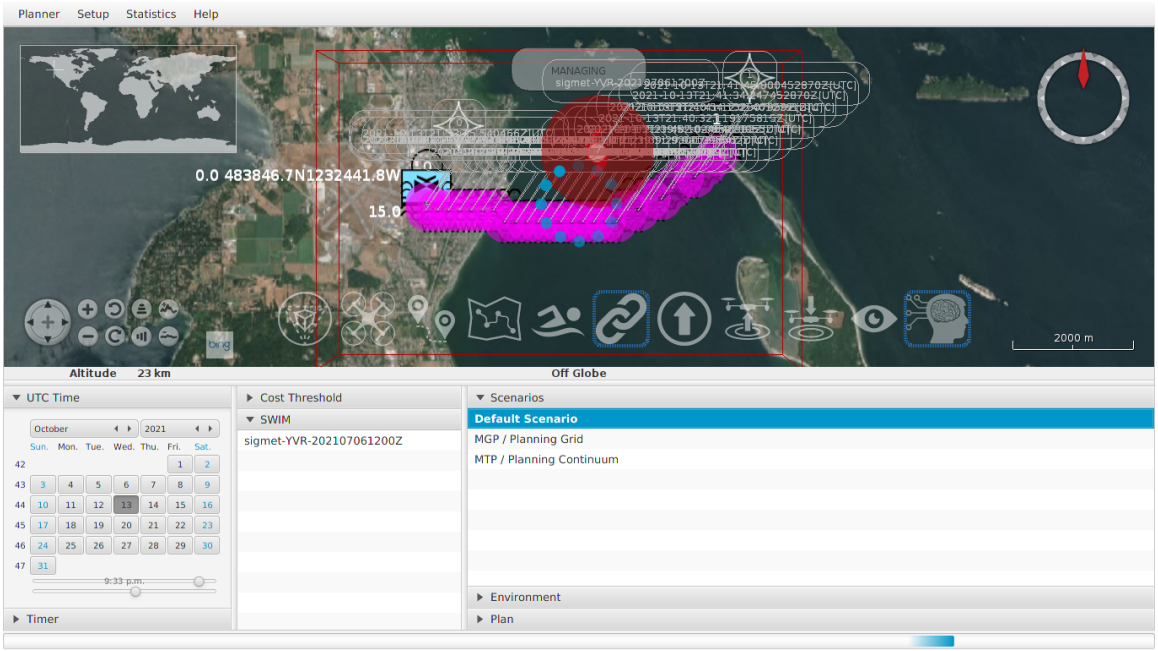


Figure 5.27: SAFCS UI: Managed Planning Session

`AutonomicManager` has created two evaluation `Scenarios` called *MGP / Planning Grid* and *MTP / Planning Continuum*, respectively. These evaluation `Scenarios` ex-

ecute their respective ManagedPlanners in parallel. The AutonomicManager tunes each of them to the best of its capabilities either heuristically and model-free in case of a HeuristicAutonomicManager or knowledge- and model-based in case of a SmacAutonomicManager. The competition results in the best computed Trajectory for the managed *Default Scenario*. It, furthermore, establishes the currently *active* and *standby* ManagedPlanners. While the active and currently best-performing ManagedPlanner continues to communicate via its Datalink Connection, its standby competitors remain available and are instructed by their AutonomicManager to rejoin the competition whenever the Features of the managed *Default Scenario* change.

5.3.3 Features and Tunings

The Features of a planning Scenario represent *sensors* of the SAFCS AM to capture its AC and become *context-aware* as required to achieve self-adaptiveness on a primitive level, as shown in Figure 2.13. Domain knowledge is required to select essential Features that affect the required Tuning of a ManagedPlanner. The captured Features are discussed in detail in Chapter 9, and examples include POI distances, obstacle distance, density, and severity among others.

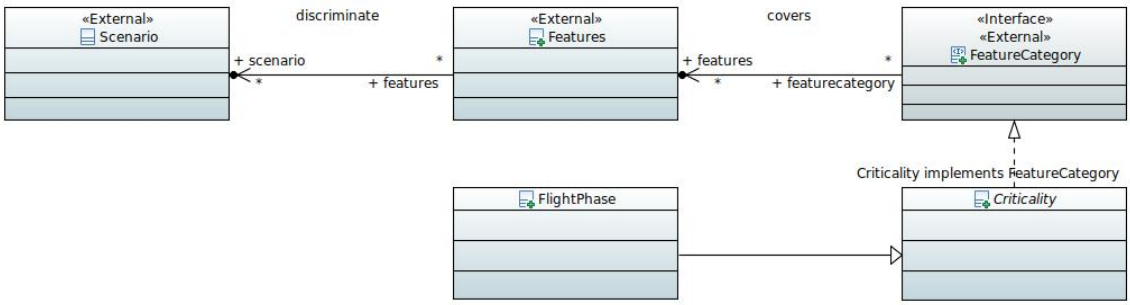


Figure 5.28: SAFCS Management: Features

The Features design is shown in Figure 5.28. Features effectively discriminate a managed planning Scenario and can be covered by a FeatureCategory. The FeatureCategory again employs domain knowledge to associate similar Features with each other. To achieve this, a FeatureCategory defines ranges or constraints on Features. For example, similar obstacle distances, densities and severities may be categorized to belong to the same Criticality, that is, threat level. Similarly, POI horizontal and vertical distances together with certain obstacle Features may be categorized to belong to the same FlightPhase. Both Features and FeatureCategory may

be extended or modified to be more or less discriminating. A FeatureCategory plays an important role with respect to model-based AutonomicManagers. A model-based AutonomicManager, such as the SmacAutonomicManager, may train on a number of Features belonging to the same FeatureCategory only, or otherwise perform training on a wide range of arbitrary Features. While the former will most likely result in dedicated Tunings for each FeatureCategory, the latter will provide a best on average Tuning with lesser performance only.

The selection and creation of suitable Tunings for ManagedPlanners represent the *actuators* of the SAFCS AM to effect its own performance as an AE in accordance with the self-adaptation loop shown in Figure 2.12. The SAFCS Tuning class hierarchy is shown in Figure 5.29.

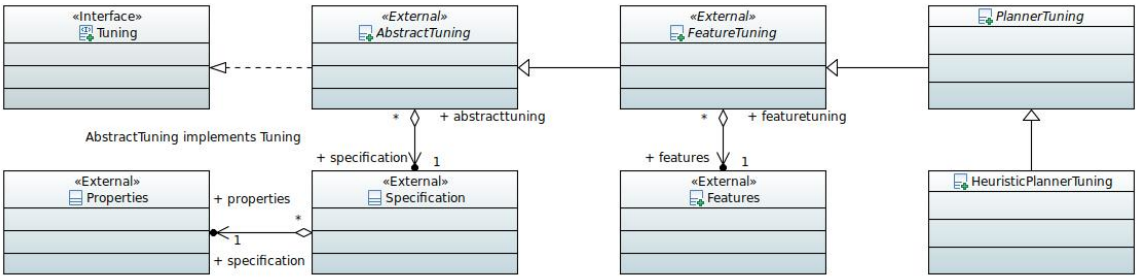


Figure 5.29: SAFCS Management: Tunings

In its most abstract form, that is, as AbstractTuning, it aggregates a Specification comprising Properties for an arbitrary FactoryProduct. As an abstract FeatureTuning, it, furthermore, aggreates Features to tune the Properties of its Specification for. Finally, the PlannerTuning abstracts exactly the same requirement for Planner Specifications. A HeuristicPlannerTuning is an example of a concrete PlannerTuning that applies heuristics to extracted Features to tune its PlannerProperties accordingly. Each concrete AutonomicManager is required to implement an associated concrete PlannerTuning. As such, the SmacAutonomicManager is associated with the SmacPlannerTuning being an extension of the HeuristicPlannerTuning.

5.3.4 Performance and Knowledge Base

The Performances of ManagedPlanners represent *sensors* of the SAFCS AM to capture itself and become *self-aware* as required to achieve self-adaptiveness on a

primitive level, as shown in Figure 2.13. As can be seen in Figure 5.30, the Performance of a ManagedPlanner is measured in terms of both Quality and Quantity. A TrajectoryQuality measures the cost of a computed Trajectory which encodes a weighted distance considering the applicable operational RiskPolicy and CostPolicy, as discussed in Section 5.2.5. A DurationQuantity measures the Duration required to compute the Trajectory. Since ManagedPlanners are AnytimePlanners that improve a (potentially multi-part) Trajectory incrementally, this Performance model is suitable to compare both earlier and later solutions of some TrajectoryQuality. Furthermore, it takes into consideration that shorter more costly solutions that satisfy the operational RiskPolicy may take longer to compute after longer less costly solutions have been evaluated.

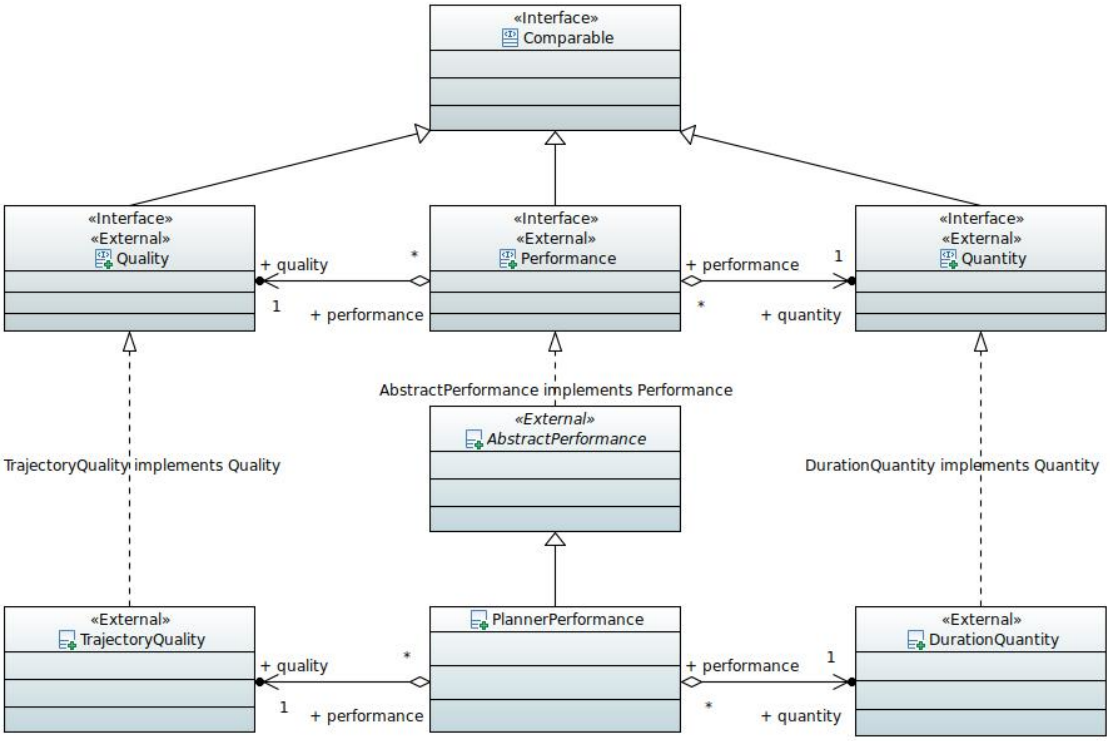


Figure 5.30: SAFCS Management: Performance

PlannerPerformances that have been measured for certain PlannerTunings, which have been derived based on extracted Features, are collected in a Reputation that is aggregated in the KnowledgeBase of the SAFCS AM component. The Reputation maps a Tuning, that is, a PlannerTuning in case of ManagedPlanners, to a collection of Performances, that is, PlannerPerformances in the same case.

Whereas the PlannerPerformance serves as an internal *short-term* sensor to achieve self-awareness, the Reputation contained in the KnowledgeBase is persisted and can serve the same purpose in the *long-term* and for independent and unrelated managed planning Sessions. The KnowledgeBase and associated classes are shown in Figure 5.31. It is loaded during the initialization phase of an AutonomicManager and persisted after its completion during its cleanup phase.

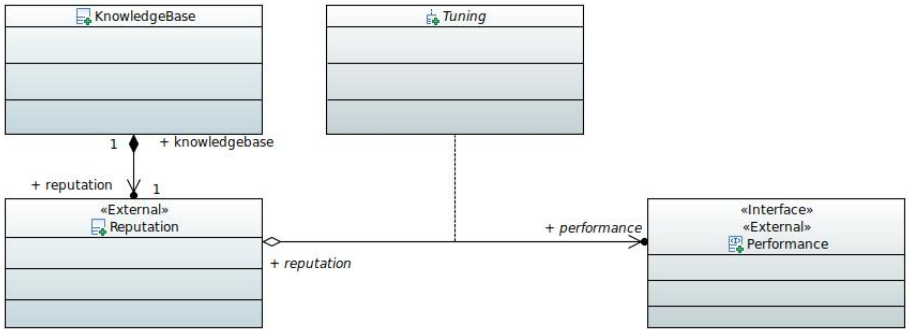


Figure 5.31: SAFCS Management: Knowledge Base

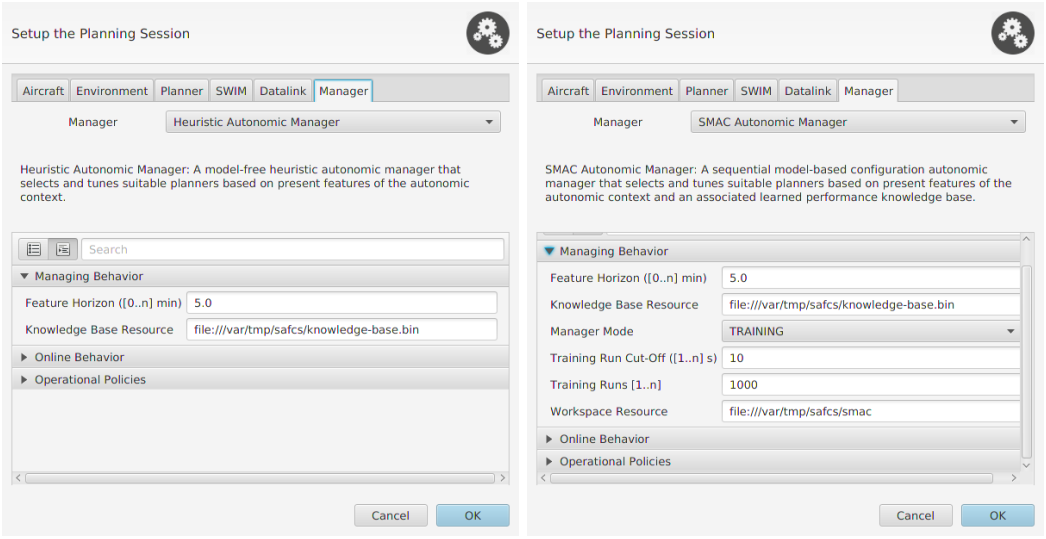
The KnowledgeBase can be extended in different ways depending on the concrete AutonomicManager. While managing a Session in the *executing* mode, it is extended implicitly. The KnowledgeBase can, however, also be extended in the *training* mode if the concrete AutonomicManager supports the latter. Figure 5.32 shows the setup of a managed planning Session.

The HeuristicAutonomicManager, being a model-free AutonomicManager, does not support training but only execution. The SmacAutonomicManager, on the other hand, evaluates candidate Tunings for a set of discriminating Features while in training mode and extends the KnowledgeBase for future use in the executing mode. This and related topics are discussed in greater detail in Chapter 9. More details about AutonomicManagers and related classes can be found in the packages managers,[14] managing,[15], and flight,[16] in our SAFCS software repository.

[14] https://github.com/stephanheinemann/worldwind/tree/develop/src/main/java/com/cfar/swim/worldwind/managers

[15] https://github.com/stephanheinemann/worldwind/tree/develop/src/main/java/com/cfar/swim/worldwind/managing

[16] https://github.com/stephanheinemann/worldwind/tree/develop/src/main/java/com/cfar/swim/worldwind/flight



(a) SAFCS UI: Heuristic Manager Setup **(b)** SAFCS UI: SMAC Manager Setup

Figure 5.32: SAFCS UI: Manager Setup

5.4 SAFCS UI Architecture

The SAFCS UI component is built on JavaFX[17] which facilitates a clean separation of concerns via *inversion of control* with *dependency injection* of UI elements. The UI elements themselves are entirely described in XML, more precisely in the EFF-ects eXtended Markup Language (FXML), created by the JavaFx *scene builder* application,[18] and instantiated by an `FXMLLoader`. Elements of the instantiated UI are, therefore, only loosely coupled to their use-case logic via identifiers used in the FXML. These identifiers reference the responsible controller as well as UI controls and associated action methods.

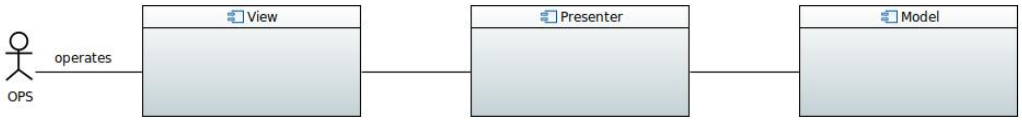


Figure 5.33: SAFCS UI: Model-View-Presenter

Our SAFCS UI, furthermore, applies *convention over configuration* using Adam Bien's Afterburner.fx[19] framework. Afterburner.fx is an opinionated Model-View-Presenter

[17] https://openjfx.io/
[18] https://www.oracle.com/java/technologies/javase/javafxscenebuilder-info.html
[19] http://afterburner.adam-bien.com/

(MVP) framework to create uniform UI components based on simple naming conventions. This uniformity, in turn, facilitates development and communication within and between development teams.

The MVP design pattern is shown in Figure 5.33. MVP has been further split into the two patterns MVP-Passive and MVP Supervising Presenter.[20] Afterburner.fx follows the MVP Supervising Presenter pattern. In this pattern, OPS operates UI elements of a `View` that invoke associated actions implemented in a `Presenter`. The `Presenter`, in turn, updates the internal business `Model`. While `View-Model` consistency can be established more directly via data-binding in simple cases, the `Presenter` otherwise separates the `View` from the associated `Model` and performs more involved tasks, such as data-conversion or service-invocation. In addition to dependency injection of UI elements, Afterburner.fx also supports the injection of Plain-Old Java Objects (POJOs) into the `Presenter`. These POJOs can, for example, represent `Models` or remote interfaces to them.

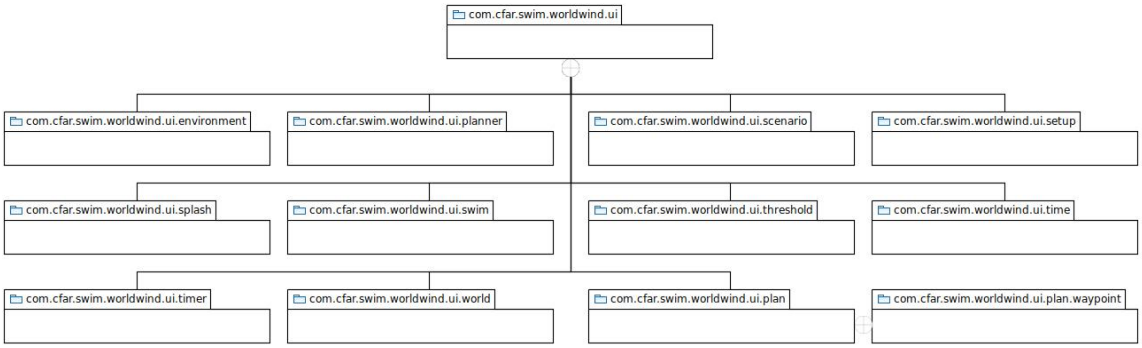


Figure 5.34: SAFCS UI: Packages

The SAFCS comprises several packages shown in Figure 5.34 that cover different UI use-cases. Each of these packages implements and follows the conventions of Afterburner.fx. More precisely, each of the SAFCS UI packages contains its own `View` and `Presenter`. With only a few exceptions, most of the associated `Models` are located in the SAFCS Core. As their names imply, the packages cover use-cases related to the operational control and monitoring of `planners` operating in geo-fenced `environments` of a `world` at a certain `time`, which is controllable using a `timer`. They allow for the definition of `setup` specifications to create associated `scenarios`, the inclusion of `swim` data filtered by cost `thresholds`, and the display of generated (multi-part) `plans` based on specified intermediate POI `waypoints`.

[20] `https://martinfowler.com/eaaDev/SupervisingPresenter.html`

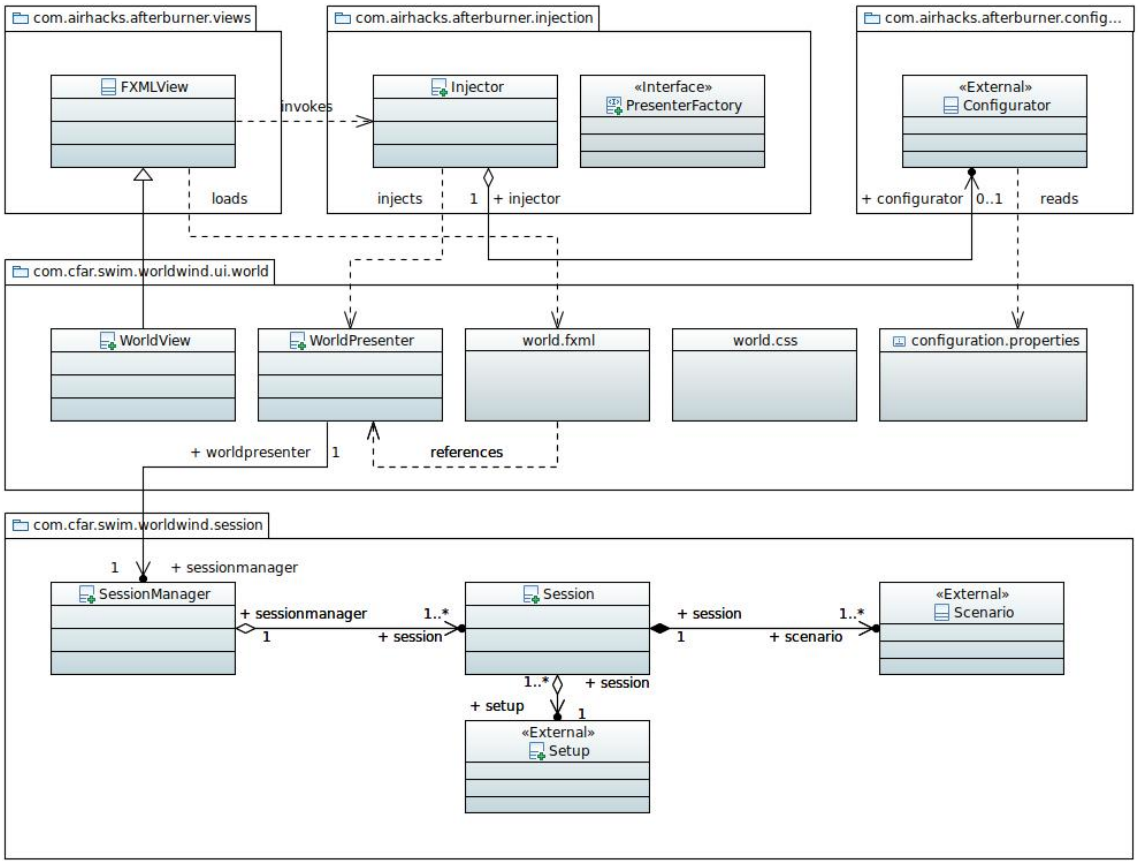


Figure 5.35: SAFCS UI: World Package

Figure 5.35 shows parts of the internal structure of the `world` package as an example representing the structure of all other packages contained in our SAFCS UI component. The diagram shows 3 levels of packages belonging to the Afterburner.fx framework, the SAFCS UI, and the SAFCS core, respectively. *Convention over configuration* applies to the used names and contents of the SAFCS UI package. Each UI package consists of a `<Topic>View`, `<Topic>Presenter`, `<topic>.xml` as well as the optional `<topic>.css` and `configuration.properties`. In case of the `world` package, a `world.xml` is created describing the actual `WorldView`. `WorldView` inherits from the Afterburner.fx `FXMLView` that contains all the required JavaFX functionality to load the `world.xml`, and instantiate the corresponding `WorldView` using an `FXMLLoader`.

`FXMLView`, furthermore, delegates the automatic instantiation of the `WorldPresenter`, which is referenced by the `world.xml`, and the automatic injection of both `WorldView` UI controls as well as any declared POJOs into it. To accomplish this instantiation and dependency injection, Afterburner.fx implements

its its own `Injector` for singleton injection. The `Injector`, furthermore, aggregates a `Configurator` that reads the optional `configuration.properties` and can, therefore, inject `String` properties more easily. Alternatively, the framework offers a `PresenterFactory` interface which can be realized, if necessary, to achieve a more custom instantiation and injection workflow.

Once the conventional classes have been instantiated and injected properly, the `WorldPresenter` mediates between the `WorldView` and the associated `Models` in the SAFCS core. In order to gain access to the SAFCS core `Models`, the `WorldPresenter` maintains a reference to the `SessionManager` explained in Section 5.2.1. An identified `Session` can be obtained from the `SessionManager` and used to select and modify the *active* `Scenario` of the `Session` using the defined `Setup` model of `Specifications`.

More details about the SAFCS UI component[21] and its packages can be found in our corresponding software repository.

5.5 Chapter Summary

In this chapter we presented an overview of and given a rationale for the architectural components of our SAFCS and briefly discussed the contained key packages and classes. In particular, we described the SAFCS core taking the role of the Managed Element within the SAFCS Autonomic Element being managed by the SAFCS Autonomic Manager. We, furthermore, introduced the SAFCS UI as one of the OPS interfaces to specify operational goals and policies. This chapter, therefore, provided the necessary prerequisites and context for the following chapters to expand on.

[21] https://github.com/stephanheinemann/worldwind-ui/tree/develop

Chapter 6

SAFCS Core — Classical Grid Search

This chapter gives an overview of the architectural and functional aspects regarding the implementation of the A* family of algorithms within the core component of the SAFCS.

The family of A* algorithms embraces the idea of informed search toward a goal while being guided by a heuristic. Whether or not this heuristic tends to be optimistic (*under-consistent*), realistic (*consistent*) or pessimistic (*over-consistent*) in comparison with the actual (uncertain) costs embedded in the environment, has a significant impact on planning performance and led to the various members of the A* family.

The most notable members of this family recognize certain real-world modeling assumptions, such as uncertainty and bounded rationality. Uncertainty may apply to both aspects of planning—the agent `Capabilities` themselves as well as the `Environment` the agent operates in. The D* branch of *dynamic* algorithms within the A* family, that has been mainly driven by the work of Koenig and Likhachev [KL02], addresses uncertain and dynamic changes within the `Environment` by repairing and revising generated plans efficiently enough to avoid planning from scratch.

Bounded rationality (Simon [Sim82], Gigerenzer and Selten [GS02]), on the other hand, has been addressed by the WA* and ARA* branches of algorithms. As shown by Likhachev, Gordon, and Thrun [LGT03], the incremental modification of the applied heuristic from inconsistency toward consistency leads to a faster sub-optimal expansion toward the planning goal while at the same time limiting the resulting sub-optimality of the solution at each step. If time-limited elaboration is a modelling assumption, then timely sub-optimal solutions may be more valuable than late optimal solutions. This concept is realized by the *anytime* algorithms of the A* family.

Combining both *dynamic* and *anytime* requirements led to the development of the AD* algorithm by Likhachev, Ferguson, Gordon, Stentz, and Thrun [LFG+05, LFG+08].

This algorithm represents one of the key algorithms within the research scope of the SAFCS as it features several tunable parameters that affect its performance. Together with tunable key algorithms of other planning approaches, such as RRTs and PRMs, it enables the SAFCS AM to adaptively select and tune its `ManagedPlanners` in order to achieve an improved planning `Performance` for different `FlightPhases` and `Scenarios` featuring their respective levels of dynamic change and bounded rationality. Defining and formalizing the `Criticality` of a situation and assigning the appropriately tuned algorithm to the problem at hand is going to be one of the contributions of the SAFCS AM.

Another notable A* branch of algorithms aims at alleviating the restrictions imposed by discretized `Environment` models, such as grids or meshes, which are required by members of the A* family in order to computationally cope with continuous space. Different approaches attempt to minimize the amount of heading changes and resulting legs, which are not necessarily the result of the actual but the modelled `Environment`. These approaches may, furthermore, address the agent `Capabilities` and result in smoother and feasible trajectories. The desired improvements are tackled by the *any-angle* algorithms of the A* family. The desired smoothing process can be realized implicitly or as an explicit (post-processing) step. Members of this branch include Θ* (T*) (Nash, Koenig, Likhachev [NKT10]), Incremental Φ* (IP*) (Nash, Koenig, Likhachev [NKL09]), Field D* (FD*) (Ferguson, Stentz [FS05b, CFS06]), Block A* (Block A*) (Yap, Burch, Holte, Schaeffer [YBHS11]), and AA* (Sislák, Volf, Pechoucek [SVP09a]).

A recent and very promising addition to the family of *any-angle* planning algorithms is Anya (Anya) (Harabor and Grastien [HG13]) and Spherical Anya (Rospotniuk and Small [RS20]). These optimal algorithms may turn out to be very interesting additions to the SAFCS core and have yet to be studied and evaluated carefully. In particular, Spherical Anya may avoid computationally expensive coordinate transformations when planning with reference to earth geographic coordinates.

The presently supported family of A* `Planners` is shown in Figure 6.1. The SAFCS core architecture leverages generalization and specialization to reuse a significant amount of logic shared between the implemented members of this family. The key algorithm with respect to the SAFCS AM component in terms of selection and tuning is OAD*, as realized in the `OADStarPlanner` class. However, its logic builds on the more basic A*, ARA*, and AD* `Planners` which are realized as `ForwardAStarPlanner`, `ARAStarPlanner`, and `ADStarPlanner`, respectively. The implicit any-angle algorithm

Θ* is a relatively simple derivative from the `ForwardAStarPlanner` and is implemented in the class `ThetaStarPlanner`. Currently, an explicit (post-processing) any-angle algorithm is not supported by the SAFCS and, hence, a dedicated interface, such as the `AnytimePlanner` and `DynamicPlanner`, is not required. Interesting options for other any-angle algorithms include the (multi-resolution) 3D FD* (Carsten, Ferguson, and Stentz [CFS06, FS06b]) and Anya (Harabor and Grastien [HG13], Rospotniuk and Small [RS20]) algorithms. However, these algorithms should ideally have tunable any-time and, most importantly, lifelong dynamic extensions to be suitable candidates for the SAFCS core.

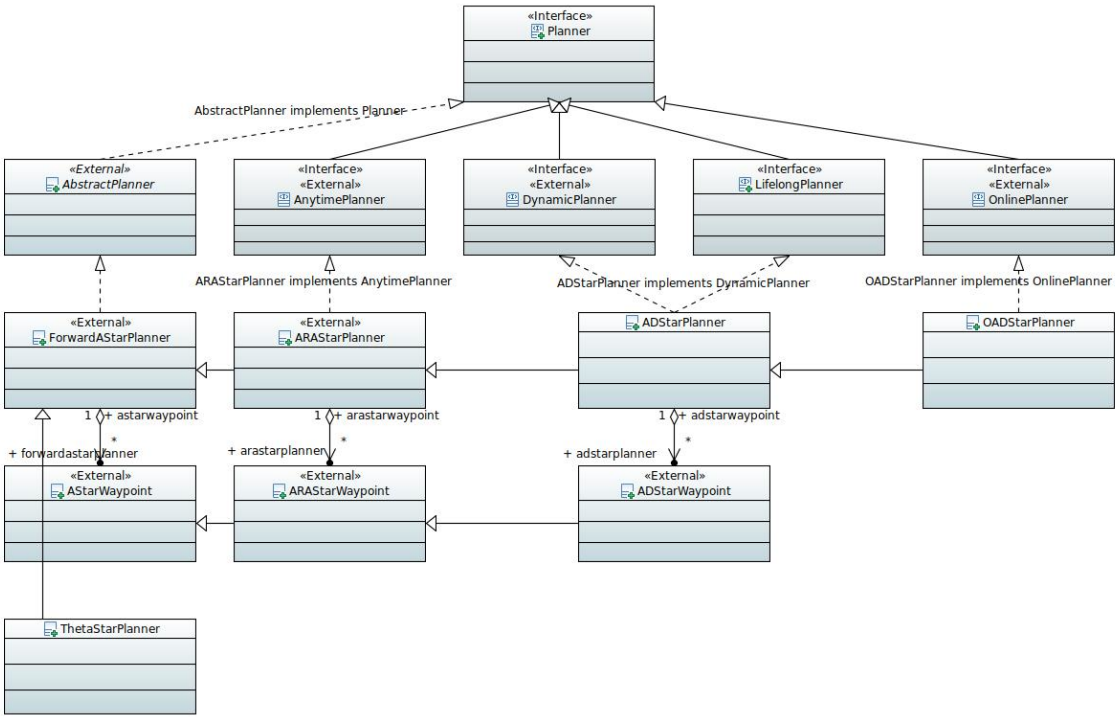


Figure 6.1: SAFCS Core: A* Family

The most sophisticated algorithm of this family within the SAFCS core is the `OADStarPlanner`. Being an `OnlinePlanner`, it features a `Datalink` connection to the planned `Aircraft` and dedicated `Datalink Communication` sequences, as shown in Figure 5.21. The `OADStarPlanner` is very similar to the `ADStarPlanner` in terms of planning functionality but in addition monitors the progress of its `Datalink` source in space and time with respect to the established and continuously revised trajectories, as explained in Section 5.2.5.

Each of the implemented algorithms of the A* family operates on dedicated `Waypoint` classes which again leverage generalization and specialization in order to reuse a significant amount of logic. The classes `AStarWaypoint`, `ARAStarWaypoint`, and `ADStarWaypoint` are extensions of the more basic `Position`, which is defined in terms of latitude and longitude of earth-geographic coordinates as well as elevation (altitude), with additional planning times and important attributes to support the logic of their associated concrete `Planner`, as explained in Section 5.2.5.

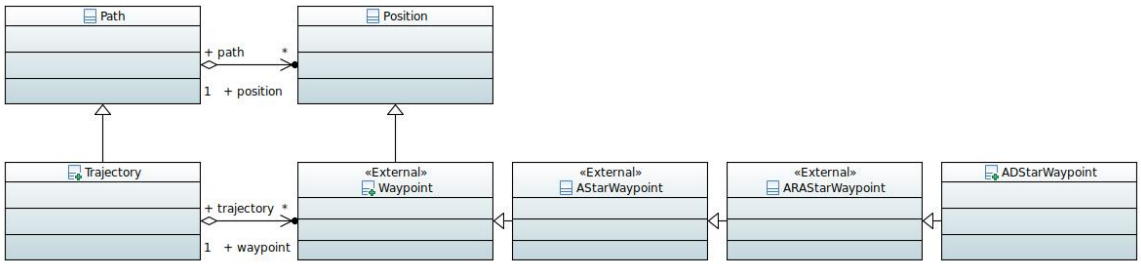


Figure 6.2: SAFCS Core: A* Waypoints

The following sections dive deeper into some details of the functional aspects of the SAFCS core as they relate to the implementation of the A* family. The discussion is limited to present mainly the differences from and extensions to the logic described in existing publications that have been surveyed in Chapter 2. Rationales are given as required to justify certain implementation decisions.

6.1 Cost Integration

A* planners apply the notion of known versus estimated costs in order to guide their search toward an operational goal. They operate on a discretized graph model (e.g., grids or meshes) of the environment where components of the graph (e.g., voxels) are associated with costs.

Section 5.2.3 already touched on the architectural structures integrated into the SAFCS to embed costs in both space and time into the planning `Environment`. Recall that costs in the aviation domain are associated with aeronautical-, weather-, flight- and aerodrome-related situations. These different concerns are covered by SWIM datasets and can be injected into the SAFCS core by means of a `SwimConnection` and an `Obstacle` reporting framework, as explained in Section 5.2.4.

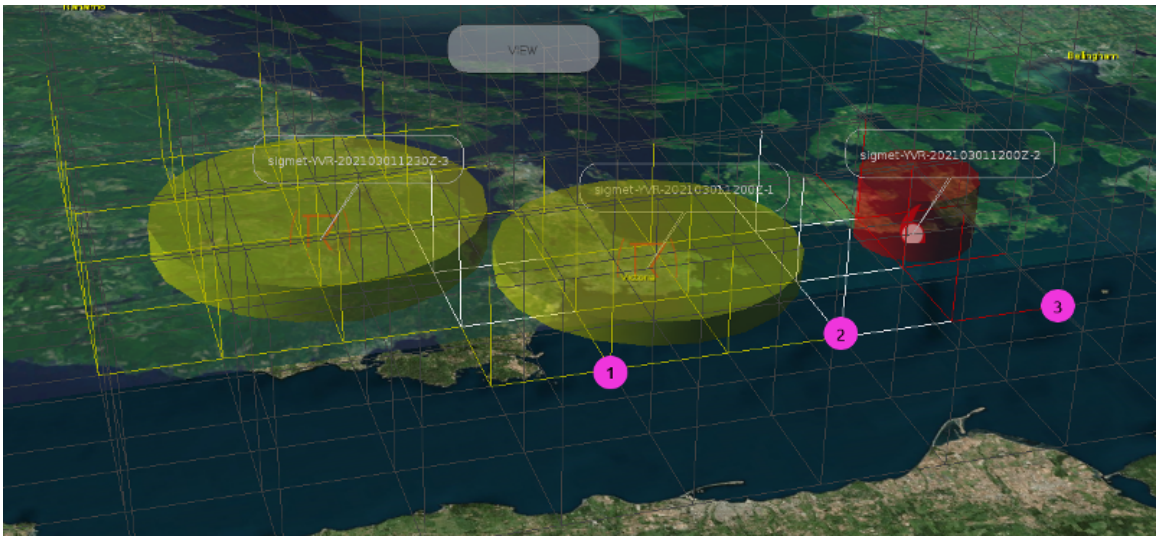


Figure 6.3: SAFCS UI: Planning Grid Obstacles and Waypoints

Figure 6.3 shows a `PlanningGrid` with three embedded SWIM `Obstacles`. The SAFCS UI readily displays different costs that have been assigned to a `CostInterval` according to a `CostMap` using different colors. For example, *yellow* corresponds to a cost of 40 and *red* to a cost of 60. Note that where and when both costs overlap in space and time, the resulting accumulated cost of more than 80 is displayed in *white*. Note, furthermore, that both the embedded `Obstacles` as well as the affected voxels are colored accordingly. As mentioned in Section 5.2.3, each voxel aggregates its own binary tree of `CostIntervals`. The present time of the `Environment` model (in this case the `PlanningGrid`) determines which are the presently active `CostIntervals`. The time of the `Environment` is used by `Planners` operating on it to lookup active `CostIntervals` for certain `Waypoint` ETOs. The ETOs to reach a `Waypoint`, in turn, are based on the `Aircraft Capabilities`, as described in Section 5.2.2. Hence, it should be emphasized again that the SAFCS core achieves tractable and efficient 4D planning in both space and time by integrating binary `CostInterval` trees representing `Obstacles` into 3D `Environment` models featuring a model time. In the depicted example, the western `Obstacle` moves southbound increasing in size, the center `Obstacle` eastbound decreasing in size, and the most costly (red-colored) eastern `Obstacle` is stationary in space and time.

Figure 6.4 shows what the `CostIntervals` for one of the affected adjacent voxels of the `Waypoints` 1, 2, and 3 depicted in Figure 6.3 may look like. Both the representative voxels adjacent to `Waypoints` 1 and 3 are affected by only one `CostInterval` during

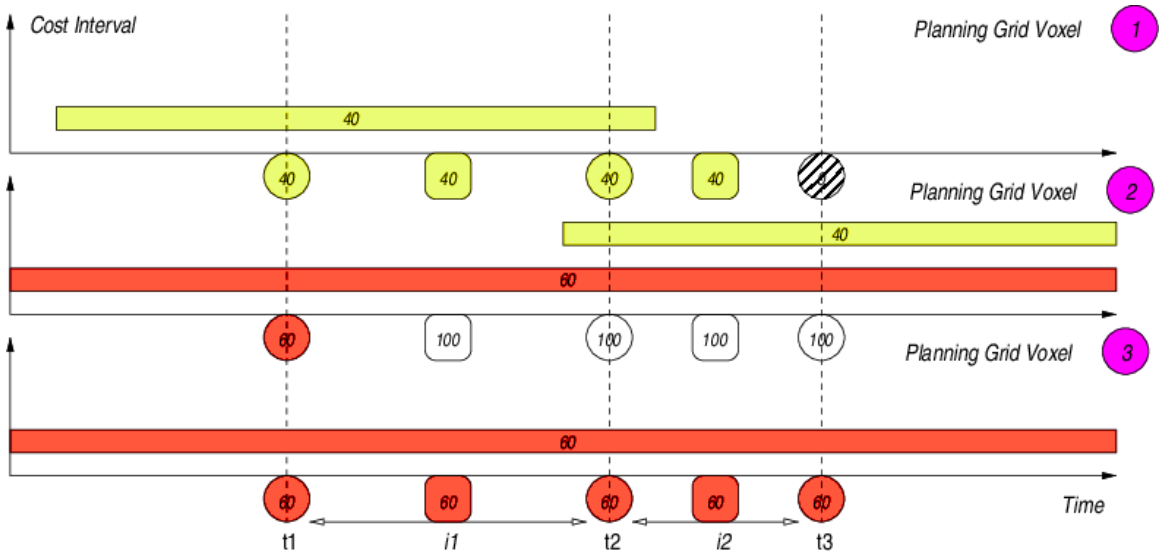


Figure 6.4: SAFCS Core: Planning Grid Cost Interval Example

the observation period with a cost of 40 and 60 each, respectively. `Waypoint` 2, on the other hand, is affected by both of these `CostIntervals` during the same observation period. Based on the displayed `CostIntervals` for all three voxels, one can deduce that the time of the `Environment` model is most likely close to the time instance $t2$ where the coloring of the voxels matches the resulting accumulated costs annotated in circles below the valid `CostIntervals`.

Figure 6.4 shows three such absolute time instances $t1$, $t2$, and $t3$. The rounded boxes in between the cost circles for these time instances may represent the intervals required for a planned voxel passage by an `Aircraft` with their lengths corresponding to the ETE required to enter and leave the particular voxels. Consequently, the most conservative cost assessment is the highest possible accumulated cost of all valid `CostIntervals` during the voxel passage. These are the basic cost determination steps that are implemented by all concrete SAFCS `Planners` operating on a `PlanningGrid`:

1. Determine the participating voxels for a step or leg
2. Determine the valid `CostIntervals` during the voxel passage
3. Ensure that local costs satisfy an operational `RiskPolicy`
4. Accumulate the resulting costs according to an operational `CostPolicy`

Since the cost calculation is elementary for the A* expansion step, the modifications implemented in the SAFCS core shall be explained in greater detail. First, a closer look

at the signatures of the `Waypoint` class hierarchy is necessary to understand how the SAFCS applies generalization and specialization to store relevant state information. Second, the SAFCS core cost computation process is examined more closely.

Figure 6.5 reveals the functional signatures of the `Waypoint` hierarchy previously shown in Figure 6.2. Recall that a `Waypoint` refines a basic `Position` with temporality attributes, such as `dtg`, `ttg`, `eto`, and `ato`, representing the Distance To Go, Time To Go, Estimated Time Over, and Actual Time Over, respectively. It, furthermore, adds a `designator`, precision `position`, and `cost`. Therefore, an SAFCS core `Waypoint` can, among other features, be identified, used in position calculations that are subject to numerical problems and require a certain precision, and are associated with an established cost value to reach them from an initial position. They can also be depicted in the SAFCS UI using their `depiction` that describes a symbol of the joint military symbology standard 2525, as explained in Section 5.2.2 for `Aircraft` instances.

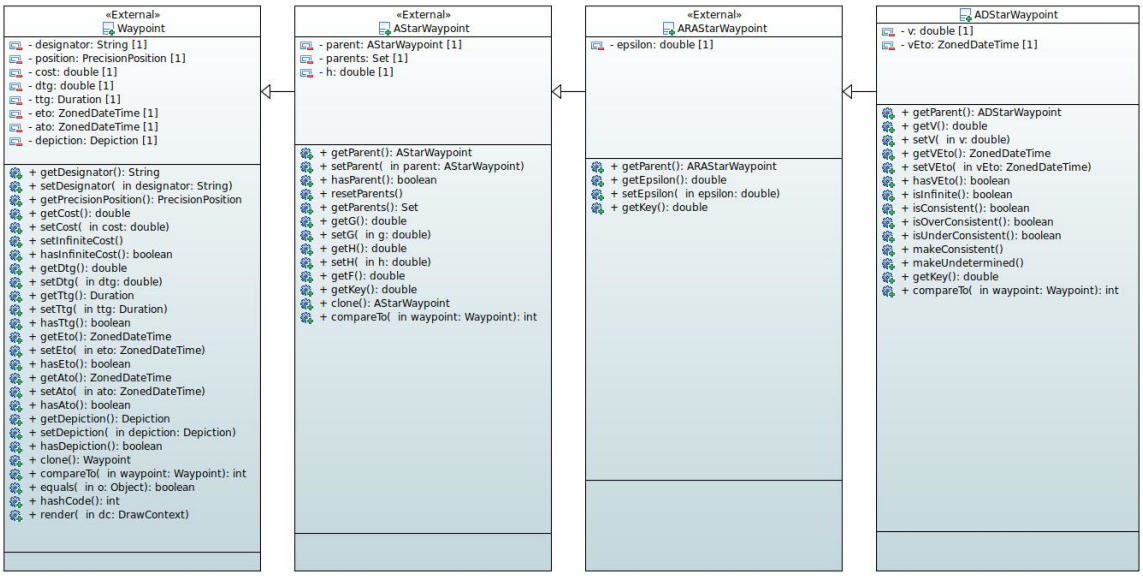


Figure 6.5: SAFCS Core: A* Waypoint Signatures

An `AStarWaypoint` refines a `Waypoint` with attributes required by `Planners` of the A* family. The heuristic `h` guides the expansion step of the algorithm toward the goal and is key for an informed search of an optimal `Trajectory` in space and time. Applying A* terminology, the `cost` attribute is commonly referred to as *g* value. Therefore, the `getG` and `setG` methods are provided to allow access to the `cost` attribute while maintaining this terminology.

```
1  protected void computeCost(AStarWaypoint source, AStarWaypoint target) {
2      Path leg = new Path(source, target);
3      Capabilities capabilities = this.getAircraft().getCapabilities();
4      Globe globe = this.getEnvironment().getGlobe();
5      ZonedDateTime end = capabilities.getEstimatedTime(leg, globe, source.getEto());
6
7      double cost = this.getEnvironment().getStepCost(
8              source, target,
9              source.getEto(), end,
10             this.getCostPolicy(), this.getRiskPolicy());
11
12     // consider expansion cost and break ties with travel time
13     boolean improvedCost = (source.getG() + cost) < target.getG();
14     boolean equalCost = (source.getG() + cost) == target.getG();
15     boolean improvedTime = (target.hasEto() && end.isBefore(target.getEto()));
16
17     // exceeded risk and parent needs to be known for dynamic repairs
18     boolean exceededRisk = (target.hasInfiniteCost())
19             && (Double.POSITIVE_INFINITY == cost);
20
21     if (improvedCost || (equalCost && improvedTime) || exceededRisk) {
22         target.setParent(source);
23         target.setG(source.getG() + cost);
24         target.setEto(end);
25     }
26 }
```

Listing 6.1: SAFCS Core: A* Cost Computation

As the neighborhood positions of an expandable `AStarWaypoint` are explored within a supported `Environment`, new `AStarWaypoints` are created and their attributes updated accordingly.

Listing 6.1 shows the concrete implementation of how costs are computed by the `computeCost` method of an A*-based `Planner`, when expanding from a `source` to a `target Waypoint` as the expansion is performed for an `Aircraft` within an `Environment`, following a specified `CostPolicy` and `RiskPolicy`.

After establishing a `Path` from `source` to `target` in line 2, the `Capabilities` of the `Aircraft` are obtained in line 3. These `Capabilities` determine the travel time from the `source` to the `target Waypoint`, resulting in an ETO (`end`) at the `target` based on the ETO (`start`) at the source in line 5.

The `cost` of this time interval is then determined within the supporting `Environment` in line 7. It is precisely within this method where embedded `CostIntervals` are looked up and accumulated according to the applicable `CostPolicy` and `RiskPolicy`.

An important modification to the original cost expansion can be seen in lines 13 to 25. Since time is not considered in the original A* algorithms but it is essential for the SAFCS core, the resulting ETOs of potential `targets` are taken into account to break ties

for equal costs. Furthermore, the original algorithms do not feature a `RiskPolicy` which represents the maximum acceptable single step cost (local cost) to achieve a goal. Costs in excess of the operational `RiskPolicy` are boosted to infinity.

The computed `cost` from `source` to `target` already includes a normalized distance and, therefore, ETE to some extend, but it may still be possible to reach a `target` at the same cost with reduced travel time (in particular, if air data is incorporated). It should be emphasized, however, that minimizing cost is implicitly treated with a higher priority than minimizing time during the expansion step. Thus, even if a `RiskPolicy` allows for the penetration of costly time-spaces within the `Environment`, it does not imply that a more risky approach is taken in favor of reduced ETEs but considered merely an option for higher flexibility. These and other options are further discussed in Chapter 15.

Listing 6.2 shows the details of the concrete step cost computation, that is invoked on line 7 of Listing 6.1, within a `PlanningGrid` as a supported planning `Environment` of the `ForwardAStarPlanner` and its derivatives. Within a `PlanningGrid`, one step corresponds to moving from one grid intersection or cell (voxel) to the next depending on the neighborhood relationship of the `PlanningGrid`. First, the participating cells for the step from `origin` to `destination` are determined in lines 9 and 10. If no cells are found that occupy the desired step, a prohibitive cost is assumed in lines 13 to 15. A normalized distance between the step `origin` and `destination` serves as the basis for the step cost computation in line 21. All participating cell costs are then determined by looking up their respective valid `CostIntervals` of their aggregated binary `CostInterval` trees (`segmentCell.getCost(start, end, costPolicy, riskPolicy)`) and multiplying these costs with the normalized distance in lines 24 to 33. An agent being exposed to a certain cost for a longer distance and, therefore, ETE, intuitively receives a higher cost penalty. If the unfactored cost penalty does not satisfy the specified `RiskPolicy`, then again a prohibitive cost is assumed. The applicable `RiskPolicy` needs to be established by the operational management similar to the `CostMap` discussed earlier, represents a certain acceptable cost threshold, and may potentially change during the adaptive airborne mission. For this purpose, the SAFCS core supports the risk thresholds `AVOIDANCE`, `PROBE`, `SAFETY`, `EFFECTIVENESS`, and `IGNORANCE`.

Finally, an operational `CostPolicy` determines whether relevant cell costs are averaged, minimized or maximized to reflect a realistic, optimistic (liberal) or pessimistic (conservative) view of the planning `Environment`, respectively. Note that the implementation of `getStepCost` method depends on the concrete `Environment` and may be very different for other supported instances.

```java
public double getStepCost( Position origin, Position destination,
        ZonedDateTime start, ZonedDateTime end,
        CostPolicy costPolicy, RiskPolicy riskPolicy) {

    double stepCost = 0d;

    // compute participating cells
    Set<? extends PlanningGrid> segmentCells = this.lookupCells(origin);
    segmentCells.retainAll(this.lookupCells(destination));

    // an invalid step results in infinite costs
    if (segmentCells.isEmpty()) {
        return Double.POSITIVE_INFINITY;
    }

    List<Double> costs = new ArrayList<Double>();

    // compute initial distance cost
    // explicit distance cost computation is required if neighboring
    // cells are of different size (different level in the hierarchy)
    double distance = this.getNormalizedDistance(origin, destination);

    // compute cost of each adjacent cell
    for (PlanningGrid segmentCell : segmentCells) {
        // obtain (weighted) cost of the cell
        double cellCost = segmentCell.getCost(start, end, costPolicy, riskPolicy);
        // boost cell cost if local risk is not acceptable
        if (riskPolicy.satisfies(cellCost - this.getBaseCost())) {
            costs.add(distance * cellCost);
        } else {
            costs.add(Double.POSITIVE_INFINITY);
        }
    }

    // apply cost policy for final cost
    switch (costPolicy) {
    case MINIMUM:
        stepCost = costs.stream()
            .mapToDouble(Double::doubleValue).min().getAsDouble();
        break;
    case MAXIMUM:
        stepCost = costs.stream()
            .mapToDouble(Double::doubleValue).max().getAsDouble();
        break;
    case AVERAGE:
        stepCost = costs.stream()
            .mapToDouble(Double::doubleValue).average().getAsDouble();
        break;
    }

    return stepCost;
}
```

Listing 6.2: SAFCS Core: Planning Grid Step Cost Computation

6.2 Basic Forward A*

The basic A* algorithm can proceed forward, that is, from the start toward the goal position or, alternatively, backward from the goal position toward the agent position. The latter approach is often referred to as *online* planning as it allows dynamic planners to repair an established trajectory toward a moving agent instead of planning from scratch starting at the present agent position.

Since the SAFCS incorporates time and plans trajectories based on an Estimated Time of Departure (ETD) (or ETO) of the start `Waypoint`, backward planning is not a feasible option. If planning were based on the Estimated Time of Arrival (ETA) (or ETO) of the goal `Waypoint`, backward planning would still not necessarily yield a trajectory to meet the moving airborne agent at its present 4D time and space `Waypoint` for dynamic repairs. Hence, the SAFCS currently only supports forward planners. A solution to this limitation would be the introduction of `Waypoint` slots and a proper extension to the definition of `Waypoint` consistency, as further discussed in Chapter 15.

Another related restriction is concerned with the theoretical possibility to revisit `Positions` in a single-part `Trajectory` at different ETOs and, therewith, via different `Waypoints`. The SAFCS core considers two `Waypoints` to be equal for a single-part `Trajectory` if their precision `Positions` are equal regardless of their ETOs. Without this limitation an airborne agent would be allowed to proceed back and forth between several visited `Positions` in the `Environment` without necessarily making any progress toward the goal. One could argue that this behavior adds flexibility to the agent when avoiding `Obstacles` in tight spaces. However, if the agent departed at a different time or remained at certain `Waypoints` for certain durations before proceeding, then this apparent issue would not be one. Furthermore, the resulting 4D state space explosion without such a limitation would clearly be intractable. Again, the necessary flexibility can be achieved by the introduction of `Waypoint` slots which would mimic the common practice of holding or loitering in case progress toward a goal is considered too risky according to the operational `RiskPolicy`. For example, a closed runway at the destination aerodrome would lead to arriving aircraft being established in a hold as long as their endurance (`Capabilities`) safely allow for, or otherwise cleared to an alternate destination.

In summary, the SAFCS A* implementation enforces progress toward the goal of a single-part `Trajectory` with sufficient flexibility to avoid `Obstacles` in space and time but prevents revisiting `Positions`. For a multi-part `Trajectory`, each part is considered

```
1  protected Set<? extends AStarWaypoint> expand(AStarWaypoint waypoint) {
2      Set<Position> neighbors = this.getEnvironment().getNeighbors(waypoint);
3
4      // if a start has no neighbors, then it is not a waypoint in the
5      // environment and its adjacent waypoints have to be determined for
6      // initial expansion
7      if (neighbors.isEmpty()) {
8          neighbors = this.getEnvironment().getAdjacentWaypointPositions(waypoint);
9      }
10
11     // create neighborhood of waypoints with precision positions
12     Set<AStarWaypoint> neighborhood =
13             neighbors.stream()
14             .map(n -> this.createWaypoint(n)).collect(Collectors.toSet());
15
16     // replace start and goal in neighborhood to avoid duplication
17     if (neighborhood.remove(this.getStart())) {
18         neighborhood.add(this.getStart());
19     }
20     if (neighborhood.remove(this.getGoal())) {
21         neighborhood.add(this.getGoal());
22     }
23
24     // expand start region position toward the start
25     if ((!waypoint.equals(this.getStart())) &&
26             this.isInStartRegion(waypoint.getPrecisionPosition())) {
27         neighborhood.add(this.getStart());
28     }
29
30     // expand a goal region position toward the goal
31     if ((!waypoint.equals(this.getGoal())) &&
32             this.isInGoalRegion(waypoint.getPrecisionPosition())) {
33         neighborhood.add(this.getGoal());
34     }
35
36     this.addExpanded(waypoint);
37
38     return neighborhood;
39 }
```

Listing 6.3: SAFCS Core: A* Expansion

independent with respect to the revisiting restriction. For example, a two-part trajectory from a start toward a goal POI and back is perfectly feasible and may take the reverse route revisiting `Positions` of the first part again in the second part.

Similar to the cost computation discussed in Section 6.1, the SAFCS core single-step expansion toward the neighborhood of a source `AStarWaypoint` is shown in Listing 6.3 and the same for all members of the A* family.

To expand a given `AStarWaypoint`, the supporting `Environment` is consulted for one-step neighbors in line 2. Recall that a supporting `Environment` can feature multiple resolutions and different neighborhood relationships depending on its concrete implementation and configuration. Hence, the set of feasible neighboring `Positions` in the consulted `Environment` depends on the structure and semantics of the latter.

```
protected void compute() {
    while (this.canExpand()) {
        AStarWaypoint source = this.pollExpandable();

        if (source.equals(this.getGoal())) {
            this.connectPlan(source);
            return;
        }

        Set<? extends AStarWaypoint> neighbors = this.expand(source);

        for (AStarWaypoint target : neighbors) {
            if (!this.isExpanded(target)) {
                this.updateWaypoint(source, target);
            }
        }
    }
}
```

Listing 6.4: SAFCS Core: A* Computation

Given the `Environment` is not singular (contains space), then if no direct neighbors can be found, the `Waypoint` to be expanded is not located at a valid discrete planning position (e.g., the intersections or voxel centers of a `PlanningGrid`) in the `Environment`. This is usually the case for start and goal `Waypoints` which can be located at arbitrary `Positions` within the `Environment`. For these special situations, the valid discrete adjacent neighbors in the `Environment` are determined in lines 7 to 9. Once all neighboring `Positions` are found, `Planner`-specific `Waypoints` are created in lines 12 to 14. Recall that each A* family member operates on its own dedicated `Waypoint` derivative, as shown in Figure 6.5. Hence, the method `createWaypoint(Position)` is implementation-specific but consistently avoids the duplication of visited `Positions` regardless of their ETOs, as explained before.

Lines 17 to 34 again simply ensure that the special `AStarWaypoints` start and goal are not duplicated during the expansion step, and that, furthermore, the start and goal regions, that is, their valid `Environment` neighborhood, are properly expanded. Each expanded `AStarWaypoint` is then placed into the appropriate set in line 36 before returning the expandable neighborhood in line 38. The set of expanded or expandable `AStarWaypoints` corresponds to the terminology of `closed` and `open` sets, respectively, in the classic literature on the topic.

The A* `compute` method in Listing 6.4 shows the well known basic forward A* algorithm assuming a consistent heuristic is being applied. The SAFCS core always applies the normalized Euclidean distance of the supporting `Environment` and, hence, guarantees consistency.

The algorithm in Listing 6.4 shall not be further explained since it has been sufficiently covered in the classic literature presented in Chapter 2, such as in [LFG+05] by Likhachev, Ferguson, Gordon, Stentz, and Thrun for the development of their AD* algorithm. What should be noted, however, is the use of dedicated methods to update `AStarWaypoints` and sets of them after the expansion and cost determination. These dedicated methods allow for efficient reuse or further specialization of logic in the derived `ForwardAStarPlanner` classes shown in Figure 6.1. Another modification, similar to the cost update adjustment described in Section 6.1, relates to the order of expandable `AStarWaypoints` in their associated priority queue. The method `pollExpandable` in line 3 removes the first expandable `AStarWaypoint` according to that order. For the basic A* algorithm this order is defined by its key function, where $g(s)$ and $h(s)$ are the known costs from the start and the heuristically assumed costs to the goal, respectively.

$$key(s) = f(s) = g(s) + h(s) \tag{6.1}$$

However, the SAFCS core needs to consider established ETOs in addition to computed costs in order to determine the necessary expansion priority. This requirement is realized using the algorithm-specific keys and breaking ties using the ETOs within the `compareTo` methods for all `Waypoints` of the class hierarchy shown in Figure 6.5.

Listing 6.5 shows how the SAFCS uses A* keys to establish the basis for the priority order and breaks ties with higher known costs (being closer to the goal) and lower ETOs (proceeding earlier to the goal), respectively.

Another added feature is the capability of all SAFCS core `Planners` to plan both single- and multi-part trajectories. A multi-part `Trajectory` comprises multiple independent plans (except for connecting ETOs) where each intermediate POI represents the start of the consecutive plan. Hence, for multi-part plans, a preceding part has to be fully established with a resulting goal ETO in order to plan the subsequent plan. This is, to some extend, a limitation if plans are incrementally generated or repaired while or after their generation, as for ARA* or AD*. This apparent drawback is discussed again in Sections 6.3 and 6.4 as well as Chapter 15, and once again could lead to the introduction of operationally constrained `Waypoint` slots (e.g., constrained by capabilities and established safety policies) for additional flexibility.

```
public double getF() {
    return this.getG() + this.getH();
}

public double getKey() {
    return this.getF();
}

public int compareTo(Waypoint waypoint) {
    int compareTo = 0;

    if (waypoint instanceof AStarWaypoint) {
        AStarWaypoint asw = (AStarWaypoint) waypoint;
        compareTo = Double.valueOf(this.getKey()).compareTo(asw.getKey());
        if (0 == compareTo) {
            // break ties in favor of higher G-values
            compareTo = Double.valueOf(asw.getG()).compareTo(this.getG());
            if ((0 == compareTo) && (this.hasETO())) {
                // break ties in favor of lower ETOs
                compareTo = this.getEto().compareTo(asw.getEto());
            }
        }
    } else {
        compareTo = super.compareTo(waypoint);
    }

    return compareTo;
}
```

Listing 6.5: SAFCS Core: A* Priority Order

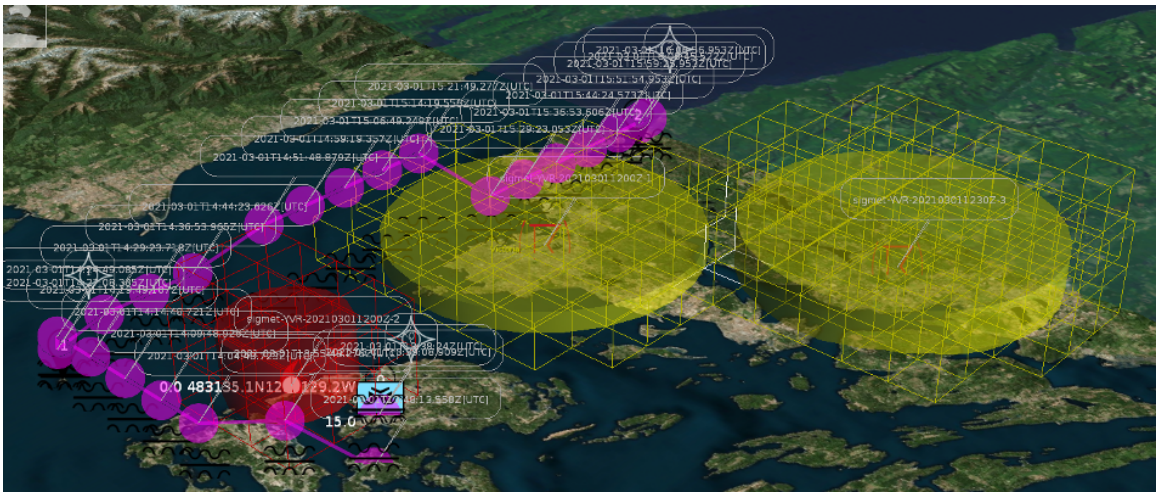


Figure 6.6: SAFCS UI: Forward A*

Figures 6.6 and 6.7 show how our SAFCS UI displays a computed `Trajectory` of `AStarWaypoints` for the A* and Θ* `Planners`, respectively. As can be seen in the Figures, Θ*, being an any-angle A* derivative, avoids unnecessary heading changes imposed by the `Environment` model to reach the same POIs and, thus, simplifies the generated `Trajectory` significantly.

More details about the SAFCS core `ForwardAStarPlanner`[1] and `ThetaStarPlanner`[2] can be found in our SAFCS software repository.

[1] https://github.com/stephanheinemann/worldwind/tree/develop/src/main/java/com/cfar/swim/worldwind/planners/cgs/astar

[2] https://github.com/stephanheinemann/worldwind/tree/develop/src/main/java/com/cfar/swim/worldwind/planners/cgs/thetastar

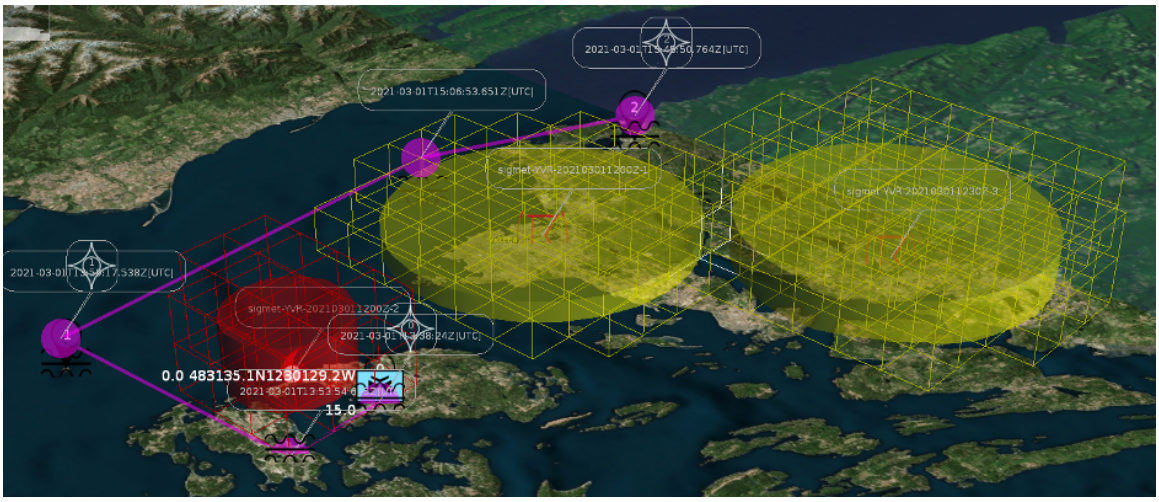


Figure 6.7: SAFCS UI: Θ^*

6.3 Anytime Repairing A*

Following the ideas of Likhachev, Ferguson, Gordon, Stentz, and Thrun [LFG+05], the next step toward the development of AD* is ARA*. Being an `AnytimePlanner`, ARA* incrementally improves trajectories with a known sub-optimality bound toward the desired optimality while acknowledging the limitations of *bounded rationality*. ARA* generally delivers these initial sub-optimal solutions faster and can, therefore, better deal with higher levels of `Criticality` and more aggressive `Environments`.

ARA* achieves this behavior by applying an inflation factor ϵ to the heuristic function in order to achieve a more aggressive expansion toward the goal. Hence, the ARA* key function to determine the priority order for expansion becomes

$$key(s) = g(s) + \epsilon h(s) \tag{6.2}$$

The attribute `epsilon` of `ARAStarWaypoint` extends the previously discussed `AStarWaypoint` attributes for that purpose, as shown in Figure 6.5. The modified key function, in turn, changes the behavior of the `compareTo` method shown in Listing 6.5. Hence, for ARA* the priority order is implicitly changed without overriding this method.

ARA* proceeds for one series of expansions for each value of ϵ at a time where any polled expandable `ARAStarWaypoint` is expanded at most once. After each expansion

series, the SAFCS core performs a backup of the internal `ARAStarPlanner` state before elaborating, then restoring the elaborated state and proceeding with the next series of expansions.

Elaboration is a step common to both the `ARAStarPlanner` and the `ADStarPlanner`, as depicted in Figure 6.8. For the `ARAStarPlanner`, elaboration simply means improving, that is, deflating the currently inflated factor $\epsilon \geq 1$ from a given minimum to a desired maximum quality. The initial inflation factor, the deflation amount, and the final inflation factor (e.g., $\epsilon = 1$ if optimality is required) are the tuning parameters of the algorithm.

Similar tuning parameters are shared by all `AnytimePlanners` and are inherited by the more sophisticated `ADStarPlanner`. They can be set by the respective methods exposed in the class signature shown in Figure 6.8, that is, `setMinimumQuality`, `setQualityImprovement`, and `setMaximumQuality`.

The SAFCS AM is designed to use these among other tuning parameters together with algorithm selection in order to achieve the best possible `Performance` for different levels of `Criticality`. This can potentially be combined with `Environment` tuning by adapting the resolution of the supporting `Environment` according to the present situation or `FlightPhase`.

The `ARAStarPlanner` allows for more improvement options, including an adaptive deflation according to the actually achieved sub-optimality bound as well as the pruning of the expandable priority queue by removing `ARAStarWaypoints` without any chance of being part of a higher-quality trajectory from an elaborated restored backup, as further discussed in Chapter 15.

Multi-part planning for ARA* would have two possible options: All consecutive parts could be planned out for the same inflation factor ϵ before an improvement (deflation) step is performed, beginning again at the first part. The second (and implemented) option is to fully improve one part before proceeding to the next one. This choice was made in favor of accelerated improvements during *online* planning, starting at a certain distance from the agent instead of proceeding from goal to start.

More details about the SAFCS core `ARAStarPlanner`[3] can be found in our SAFCS software repository.

[3] https://github.com/stephanheinemann/worldwind/tree/develop/src/main/java/com/cfar/swim/worldwind/planners/cgs/arastar

6.4 Anytime D*

AD* combines the incremental improvements of ARA* with the dynamic repairs of D* required when operating in unknown or partially-known (uncertain) environments. The D* algorithm was first introduced by Stentz [Ste93] and later improved as D* Lite (D* Lite) by Koenig and Likhachev [KL02].

As an important contribution they formalize the *consistency* of a state which becomes the main determinator for how the algorithm proceeds. States can be either *consistent* or *inconsistent*. For a consistent state the established costs reflect the presently discovered and possible costs to reach that state. Inconsistent states can, furthermore, be either *over-consistent* or *under-consistent*. An over-consistent state is one whose costs can still be part of an improved trajectory, whereas an under-consistent state possesses a cost value that is better than currently possible. For non-dynamic members of the A* family, such as A* and ARA*, all inconsistent states are over-consistent as expansions lead to cost improvements toward a more optimal solution. Under-consistency, on the other hand, indicates that costs in the `Environment` have increased, effectively invalidating previously found solutions. Hence, a `DynamicPlanner` is required to not just propagate over- but also under-consistency to repair an inconsistent solution efficiently without planning from scratch. D*Lite, being an extension of LPA* (Koenig, Likhachev, and Furcy [KLF04]), runs forever unless explicitly terminated, continuously integrating previously unknown costs and re-establishing consistency. These unknown costs can be both cost reductions and increases, which are covered by the explained over- and under-consistency definitions.

AD* integrates the consistency ideas of D* Lite and re-establishes consistency while operating under the current sub-optimality bound, that is, the inflation factor ϵ. The algorithm and its derivation are covered in great detail by Likhachev, Ferguson, Gordon, Stentz, and Thrun [LFG+05, LFG+08], and a backward version is explained in Algorithm 3 of our survey in Section 2.1.3 of Chapter 2. This Section, therefore, only describes the SAFCS core implementation and necessary modifications to and differences from the published original version.

The SAFCS core `ADStarWaypoint` extends the previously discussed `ARAStarWaypoint` attributes by adding the consistency-related attributes `v` and `vETO`, as shown in Figure 6.5. The attribute `v` is discussed in the original literature and describes a previously discovered cost during the expansion toward a neighboring (and subsequently, potentially expandable) `ADStarWaypoint`. During the expansion

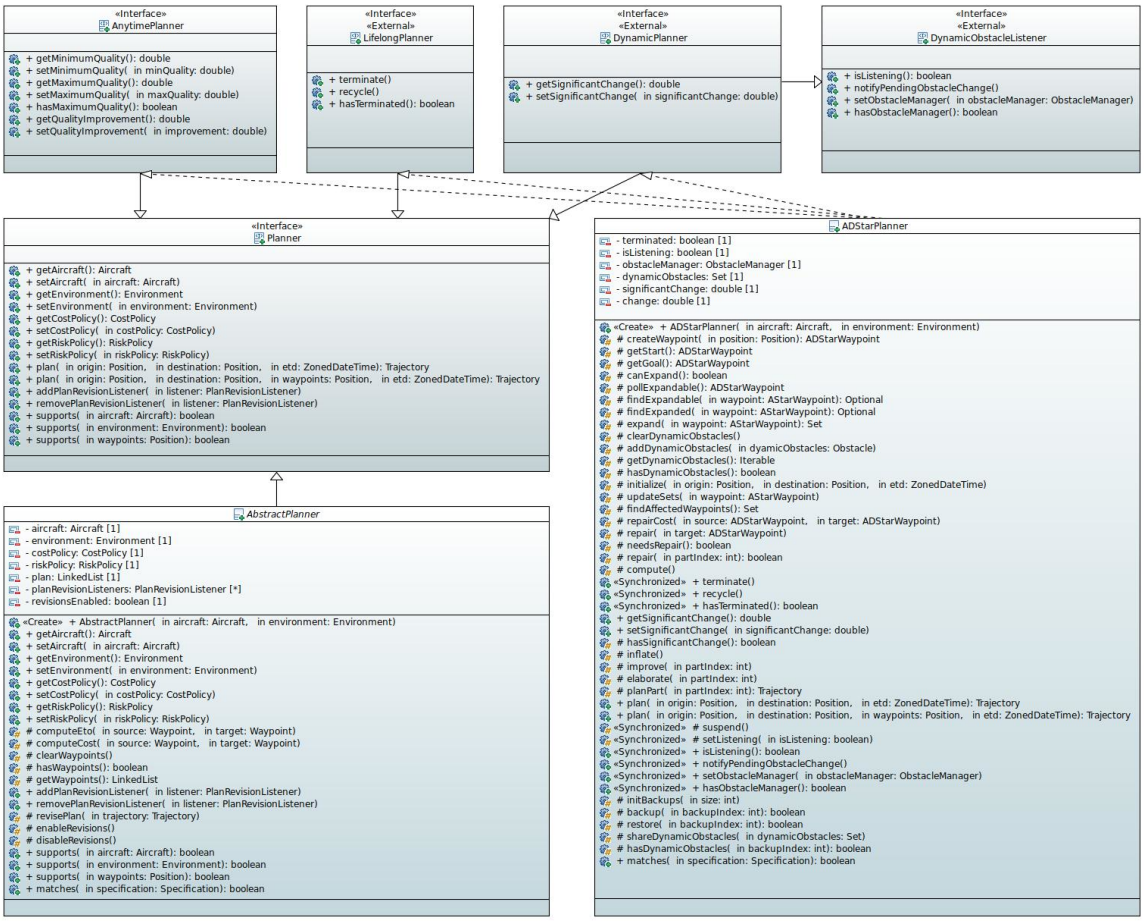


Figure 6.8: SAFCS Core: AD* Class Signatures

of an over-consistent `ADStarWaypoint`, consistency is established by assigning `v` the actual applicable `cost` value. The SAFCS core extends the concept of consistency to ETOs and, therefore, introduces the additional attribute `vETO`. Listing 6.6 shows the implementation of the associated `ADStarWaypoint` methods.

The SAFCS core requires that the previously discovered `vEto` equals the actual applicable `eto` in addition to the costs requirement of the original algorithm. This is a necessary extension on a 4D space and time planning ground. Establishing consistent costs only to reach a certain 3D `Position` at an arbitrary ETO is not a sufficient consistency condition since any dependent part of a `Trajectory` has been established beforehand in terms of both space and time.

The SAFCS core AD* consistency condition is arguably a very strong one and hard to achieve during repairs. Repairs usually lead to a different `Path` through the `Environment`, which most likely results in different ETOs and would lead to an incon-

```java
/**
 * Indicates whether or not this AD* waypoint is consistent, that is, if
 * its current estimated cost (g-value) equals the estimated cost of its
 * last expansion (v-value).
 *
 * @return true if this AD* waypoint is consistent, false otherwise
 */
public boolean isConsistent() {
    // consistency needs to consider ETOs: previous expansion ETOs required
    return (this.getG() == this.getV())
            && this.hasEto() && this.hasVEto()
            && this.getEto().equals(this.getVEto());
}

/**
 * Indicates whether or not this AD* waypoint is over-consistent, that is,
 * if its current estimated cost (g-value) is less than the estimated cost
 * of its last expansion (v-value). The expansion of an over-consistent
 * AD* waypoint may potentially lead to an improved solution (propagation
 * of over-consistency).
 *
 * @return true if this AD* waypoint is over-consistent, false otherwise
 */
public boolean isOverConsistent() {
    // consistency needs to consider ETOs: previous expansion ETOs required
    return (this.getG() < this.getV()) ||
            ((this.getG() == this.getV())
                    && this.hasEto() && this.hasVEto()
                    && this.getEto().isBefore(this.getVEto()));
}

/**
 * Indicates whether or not this AD* waypoint is under-consistent, that
 * is, if its current estimated cost (g-value) is greater than the
 * estimated cost of its last expansion (v-value). An under-consistent
 * state indicates an increase in costs and requires an invalidation of
 * dependent costs (propagation of under-consistency).
 *
 * @return true if this AD* waypoint is under-consistent, false otherwise
 */
public boolean isUnderConsistent() {
    // consistency needs to consider ETOs: previous expansion ETOs required
    return (this.getG() > this.getV()) ||
            ((this.getG() == this.getV())
                    && this.hasEto() && this.hasVEto()
                    && this.getEto().isAfter(this.getVEto()));
}

/**
 * Makes this AD* waypoint consistent.
 *
 * @see #isConsistent()
 */
public void makeConsistent() {
    // consistency needs to consider ETOs: previous expansion ETOs required
    this.setV(this.getG());
    this.setVEto(this.getEto());
}

/**
 * Makes this AD* waypoint undetermined.
 */
public void makeUndetermined() {
    // consistency needs to consider ETO: previous expansion ETOs required
    this.setV(Double.POSITIVE_INFINITY);
    this.setVEto(null);
}
```

Listing 6.6: SAFCS Core: AD* Consistency

sistency propagation that, under many circumstances (e.g., necessary repairs close to the start, or in earlier parts of a multi-part `Trajectory`), would not be advantageous compared to the performance achievable by simply planning from scratch. However, consider that for a regular (cubic) `PlanningGrid` different paths may very well lead to equal ETOs more often than in the general case and still justify the repair.

Another more flexible option again leads to the introduction of operationally constrained `Waypoint` slots. Using slots, consistency could be defined based on acceptable ETO tolerances that would force an agent to adapt its cruise speed or, otherwise, loiter or hold before proceeding. Under-consistency in terms of late ETOs would require increasing the aircraft cruise speed. Over-consistency in terms of early ETOs would require decreasing the aircraft cruise speed or entering a loitering or holding procedure. These options are discussed again Chapter 15 and not known to be covered in any literature on the topic of CGS.

In addition to the introduction of consistency definitions in order to handle dynamic cost changes within the supported SAFCS `Environment`, the key function that prioritizes the expansion toward the goal has to be addressed and modified accordingly. The original priority-order key function is shown and commented in Algorithm 13. Other keys are possible for improved performance in certain domains as long as important key properties are maintained, as explained in the original literature.

Algorithm 13 AD* Key Function [LFG+08]

```
1: procedure KEY(s)
2:     if v(s) ≥ g(s) then                          ▷ if not under-consistent (consistent or over-consistent)
3:         return [g(s) + εh(s); g(s)]   ▷ apply the established costs and inflated heuristic as key and break ties in favor of lower established costs
4:     else                                          ▷ if under-consistent
5:         return [v(s) + h(s); v(s)] ▷ apply the inconsistent costs and non-inflated heuristic as key and break ties in favor of lower inconsistent costs
```

Analyzing the original key function, it is important to note that no inflation factor is applied to the heuristic of under-consistent states, which are generally prioritized for expansions as actual costs have increased. Therefore, under-consistency is propagated in favor of over-consistency, which results in invalidating under-consistent parts of the explored `Environment` by invoking `makeUndertermined` before proceeding with the necessary repairs and re-establishing consistency by invoking the method `makeConsistent` on the affected `ADStarWaypoints` shown in Listing 6.6.

The SAFCS core again needs to incorporate the notion of time and, therefore, integrates ETOs into the priority-order key function of `ADStarWaypoint`. Listing 6.7 shows the necessary modifications. The `getKey` function shown in lines 1 to 11 follows the

```
1  public double getKey() {
2      double key = Double.POSITIVE_INFINITY;
3
4      if (!this.isUnderConsistent()) {
5          key = super.getKey();
6      } else {
7          key = this.getV() + this.getH();
8      }
9
10     return key;
11 }
12
13 public int compareTo(Waypoint waypoint) {
14     int compareTo = 0;
15
16     if (waypoint instanceof ADStarWaypoint) {
17         ADStarWaypoint adsw = (ADStarWaypoint) waypoint;
18         compareTo = Double.valueOf(this.getKey()).compareTo(adsw.getKey());
19         if (0 == compareTo) {
20             if (!this.isUnderConsistent()) {
21                 // break ties in favor of higher G-values
22                 compareTo = Double.valueOf(adsw.getG()).compareTo(this.getG());
23             } else {
24                 // break ties in favor of higher V-values
25                 compareTo = Double.valueOf(adsw.getV()).compareTo(this.getV());
26             }
27
28             if ((0 == compareTo) && (this.hasEto())) {
29                 // break ties in favor of lower ETOs
30                 compareTo = this.getEto().compareTo(adsw.getEto());
31             }
32         }
33     } else {
34         compareTo = super.compareTo(waypoint);
35     }
36
37     return compareTo;
38 }
```

Listing 6.7: SAFCS Core: AD* Priority Order

original priority order. However, the method `compareTo` of `ADStarWaypoint` modifies the existing and includes the new ETO-based tie breaker.

The modified consistency definitions and priority order used for a series of expansions under the same sub-optimality bound, that is, the present inflation factor ϵ, are used in the AD* `compute` method shown in Listing 6.8. After expanding `ADStarWaypoints` according to the described priority order and expansion policy in lines 4 and 5, over- and under-consistencies are propagated in order to improve or repair an existing plan, respectively. Improvement between a series of expansions is identical to ARA* described in Section 6.3, part of the elaboration step, and consists of a controlled deflation of the inflation factor ϵ. In addition to improving over-consistent `ADStarWaypoints`, the `ADStarPlanner` initiates necessary repairs for under-consistent `ADStarWaypoints` during the elaboration in between expansion series and then prop-

```
1
2  protected void compute() {
3      while (this.canExpand()) {
4          ADStarWaypoint source = this.pollExpandable();
5          Set<? extends ADStarWaypoint> neighbors = this.expand(source);
6
7          // source is always inconsistent as per expandable invariant
8          if (source.isOverConsistent()) {
9              // establish consistency
10             source.makeConsistent();
11             // propagate over-consistency
12             for (ADStarWaypoint target : neighbors) {
13                 this.updateWaypoint(source, target);
14             }
15         } else {
16             // eliminate under-consistency
17             source.makeUndetermined();
18             // expanding implicitly adds the expanded waypoint to the expanded set
19             this.removeExpanded(source);
20             this.updateSets(source);
21             // propagate under-consistency
22             for (ADStarWaypoint target : neighbors) {
23                 // find targets that depend on under-consistent source
24                 if (target.hasParent() && target.getParent().equals(source)) {
25                     this.repair(target);
26                     this.updateSets(target);
27                 }
28             }
29         }
30     }
31
32     this.connectPlan(this.getGoal());
33 }
```

Listing 6.8: SAFCS Core: AD* Computation

agates latter by invalidating and repairing under-consistencies during the subsequent computations in lines 17 and 25, respectively.

The repairing idea of under-consistent `ADStarWaypoints` within one part of a generated multi-part `Trajectory` is shown in Figure 6.9. The `repair` method attempts to re-establish consistency by replacing the `Trajectory` to dependent targets with the best alternative found so far. However, these could become invalid as well in subsequent iterations and, therefore, the process continues until all under-consistencies that prevent an alternative `Trajectory` are removed. This process ideally leads to a consistent `ADStarWaypoint` where both the previous and repaired `Trajectory` meet again. In Figure 6.9 this `ADStarWaypoint` is WG. Whether or not re-establishing consistency in this manner is more efficient than planning from scratch, depends on the impact of the `Obstacle` and the regularity of the planning ground. If the new `Obstacle` is located closer to the start or the supporting `Environment` is irregular, then most likely an under-consistency propagation followed by an over-consistency propagation, in a pos-

sibly large already established state space, all the way to the goal might be the result. This is intuitively more costly than planning from scratch. However, if the impact of the `Obstacle` is less *significant* and the supporting `Environment` is regular, such as for the `PlanningGrid`, then the hard consistency condition of re-establishing both previous costs and ETOs might be affordable and more beneficial.

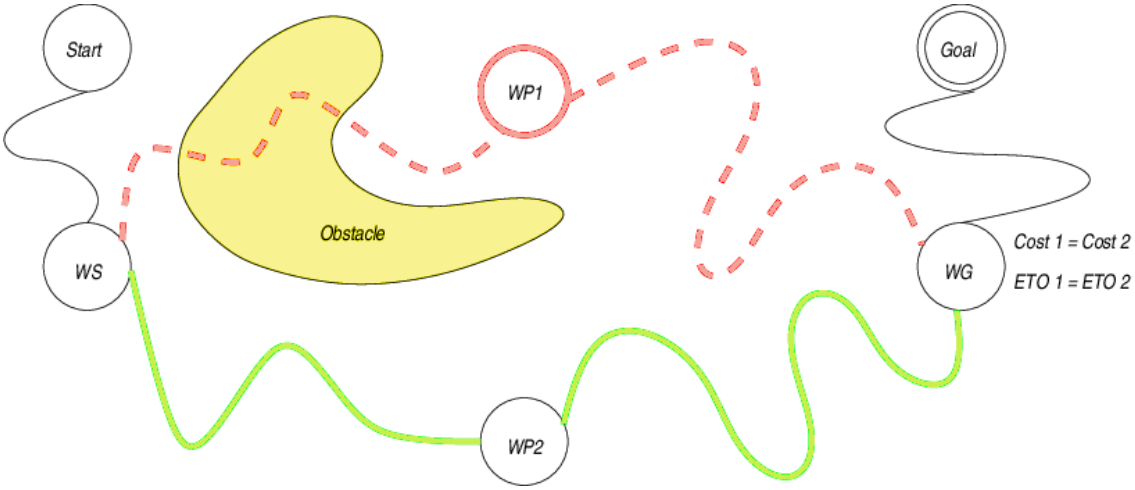


Figure 6.9: SAFCS Core: AD* Single-Part Repair

The `repair` and `repairCost` methods of the `ADStarPlanner` are shown in Listing 6.9. The `repair` method in lines 1 to 7 first invalidates the previously found costs to a dependent target and then connects the target to the best source alternative found so far. The `repairCost` method, that examines the available sources for that purpose, is listed in lines 9 to 30. It is very similar to the basic `computeCost` method already explained in Section 6.1 and shown in Listing 6.1. However, it calculates costs based on already expanded `ADStarWaypoints`, that is, their `v` attributes, since only these can be part of an existing alternative `Trajectory`. This cost and the associated ETO repair is performed in lines 21 to 29.

As mentioned before, the impact of injected inconsistencies may not always justify a repair if re-establishing the harder consistency condition of Listing 6.6 effectively re-computes the entire solution at a higher computational cost than planning the same from scratch. Learning and tuning what an impactful under-consistency constitutes, that is, defining and adapting what is referred to as a *significant change* in the original literature, in order to justify either a dynamic controlled re-inflation of ϵ or planning entirely from scratch, is another highly interesting opportunity for the SAFCS AM.

```
protected void repair(ADStarWaypoint target) {
    target.setG(Double.POSITIVE_INFINITY);

    for (AStarWaypoint source : target.getParents()) {
        this.repairCost((ADStarWaypoint) source, target);
    }
}

protected void repairCost(ADStarWaypoint source, ADStarWaypoint target) {
    Path leg = new Path(source, target);
    Capabilities capabilities = this.getAircraft().getCapabilities();
    Globe globe = this.getEnvironment().getGlobe();
    ZonedDateTime end = capabilities.getEstimatedTime(leg, globe, source.getEto());

    double cost = this.getEnvironment().getStepCost(
            source, target,
            source.getEto(), end,
            this.getCostPolicy(), this.getRiskPolicy());

    // consider expansion cost and break ties with travel time
    boolean improvedCost = (source.getV() + cost) < target.getG();
    boolean equalCost = (source.getV() + cost) == target.getG();
    boolean improvedTime = (target.hasEto() && end.isBefore(target.getEto()));

    if (improvedCost || (equalCost && improvedTime)) {
        target.setParent(source);
        target.setG(source.getV() + cost);
        target.setEto(end);
    }
}
```

Listing 6.9: SAFCS Core: AD* Waypoint Repair

```
1  protected void elaborate(int partIndex) {
2      // proceed to next part only if fully deflated and not in need of repair
3      while ((!this.hasMaximumQuality() || this.needsRepair())
4          && !this.hasTerminated()) {
5          if (!this.repair(partIndex)) break;
6          this.improve(partIndex);
7          this.updateDynamicObstacles();
8      }
9      // always backup at least once for potential repair later
10     this.backup(partIndex);
11 }
12
13 public synchronized boolean hasMaximumQuality() {
14     return this.isDeflated();
15 }
16
17 protected synchronized boolean isDeflated() {
18     return (this.inflation == this.finalInflation);
19 }
20
21 protected boolean needsRepair() {
22     return this.hasDynamicObstacles();
23 }
24
25 protected boolean hasDynamicObstacles() {
26     return !this.dynamicObstacles.isEmpty();
27 }
28
29 protected void updateDynamicObstacles() {
30     if (this.hasObstacleManager() && this.getObstacleManager().hasObstacleChange()) {
31         Set<Obstacle> dynamicObstacles =
32             this.getObstacleManager().commitObstacleChange();
33         this.addDynamicObstacles(dynamicObstacles);
34         this.shareDynamicObstacles(dynamicObstacles);
35     }
36 }
```

Listing 6.10: SAFCS Core: AD* Elaboration

The methods related to the elaboration phase of an SAFCS `ADStarPlanner` for one part of a multi-part `Trajectory` are shown in Listing 6.10. As already mentioned, this elaboration is initiated by the `ADStarPlanner` in between a series of expansions achieving the current sub-optimality bound but also after having fully established and revised a `Trajectory` for the desired maximum optimality upon the injection of dynamic changes via the `DynamicObstacleListener` interface explained in Section 5.2.4 and shown for the SAFCS `ADStarPlanner` in Figure 6.8.

Elaboration continues as long as the inflation factor has not been fully deflated or if dynamic repairs are necessary given that the planner has not been terminated explicitly by the SAFCS UI or AM components. This logic is part of the method `elaborate` and shown in lines 1 to 11 of Listing 6.10. It includes the generation of backups which can be re-used for future elaborations in line 10. Whether or not a backup or entirely new `ADStarPlanner` instance for each elaboration would be the better choice, can be con-

sidered and shall be mentioned again in Chapter 15. This consideration applies to both the `ARAStarPlanner` as well as the `ADStarPlanner`, since they both feature an elaboration phase.

An elaboration repair is performed whenever new dynamic `Obstacles` have been submitted by an `ObstacleProvider`, such as a `SwimConnection`, to an `ObstacleManager` that is being listened to for any dynamic `Obstacle` changes by the `ADStarPlanner`, which is a `DynamicObstacleListener` like all `DynamicPlanners`. Recall the `Obstacle` reporting framework shown in Figure 5.16 for this discussion. Such a repair requirement is determined by the method `needsRepair` in lines 21 to 23 of Listing 6.10 that determines the existence of newly committed dynamic `Obstacles`.

Depending on when the elaboration repair occurs, all previously computed parts of a multi-part `Trajectory` have to be examined for inconsistencies. Again, these inconsistencies may be new over-consistencies, that is, `Obstacle` removals, that could lead to cost improvements, or under-consistencies, that is, `Obstacle` additions, that could lead to cost deteriorations of established `Trajectory` parts featuring a sub-optimality bounded by the current inflation factor ϵ.

The general idea of an elaboration repair of a multi-part `Trajectory` is shown in Figure 6.10. A multi-part `Trajectory` comprises a number of POIs. These POIs need to be visited in sequence to complete the airborne mission. Therefore, multiple `Waypoint`- and cost-independent single-part trajectories are computed that interconnect at common ETOs.

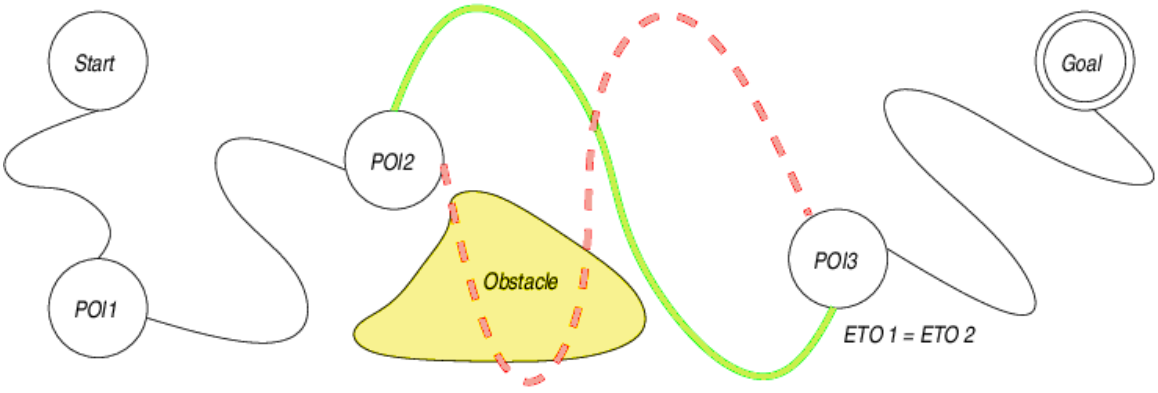


Figure 6.10: SAFCS Core: AD* Multi-Part Repair

All previously computed `Trajectory` parts have to be examined and repaired as necessary. However, the repair of a single-part `Trajectory` may require subsequent parts to be re-computed from scratch if the resulting goal ETO changes. Hence, a dy-

namic change in an early part of a multi-part `Trajectory` within a non-regular planning `Environment` again is more likely to be impactful enough to justify planning from scratch. As already mentioned, the introduction of operationally constrained `Waypoint` slots could alleviate this and related limitations, and is discussed in Chapter 15.

The elaboration repair itself is covered by the `repair` method shown in Listing 6.11. This method proceeds recursively repairing previously backed up parts first in lines 8 to 20 before considering repairs for the current part in the remainder of the method. Three cases have to be considered when proceeding with the repairs of the current part within a multi-part plan: In the first case (lines 22 to 27), the previous part may not have been successfully repaired or may have exceeded the applicable operational `RiskPolicy`. In this case, the resulting cost at the connecting POI is infinite and, therefore, prohibits the calculation of any consecutive part, including the current one. In the second case (lines 28 to 51), a repair of the previous part was not necessary or has been successful resulting in an ETO-consistent connecting POI state, that is, its ETO was not affected by the repair. This case is the most desirable one and allows for the current part to be repaired instead of being planned from scratch. The last case (lines 52 to 59) covers the more likely situation of an inconsistent connecting POI. Since its ETO has changed in this case, all subsequent parts, including the current one, have to be planned from scratch.

If a solution for the previous part has been found (regardless of the connecting POI consistency), then all previous parts have been fully established to their maximum desired quality, and the repairs are performed using the same maximum quality initially (line 11). However, depending on the impact of the change, the inflation factor ϵ may be inflated subsequently again in favor of faster sub-optimal repairs initially.

As explained, the second case is the only one allowing for a potentially more efficient repair and proceeds by first propagating a potential cost change at the connecting POI (lines 30 to 37) followed by connecting the consecutive parts (lines 40 to 42) and then repairing the current part (lines 45 to 51). First, all `ADStarWaypoints` that are affected by the dynamically injected `Obstacles` are filtered and collected in line 45. This procedure is shown in greater detail in Listing 6.12. It essentially determines all intersections in both space and time between steps or legs to target `ADStarWaypoints` and the dynamically added or removed `Obstacles`. To achieve this, first geometric intersections in space are found within the supported `Environment` and subsequently filtered for intersecting `TimeIntervals` (lines 4 to 14). Then, since the connecting `ADStarWaypoint`

```
protected boolean repair(int partIndex) {
    boolean repaired = true;

    if (this.needsRepair()) {
        ADStarWaypoint partStart = (ADStarWaypoint) this.getStart().clone();

        // repair previous parts before current part
        if (this.hasDynamicObstacles(partIndex - 1)) {
            this.backup(partIndex);
            double partInflation = this.getInflation();
            this.setInflation(this.getMaximumQuality());
            this.restore(partIndex -1);
            this.planPart(partIndex - 1);
            partStart.setCost(this.getGoal().getCost());
            partStart.setEto(this.getGoal().getEto());
            partStart.setParent(this.getGoal().getParent());
            this.backup(partIndex - 1);
            this.setInflation(partInflation);
            this.restore(partIndex);
        }

        if (partStart.hasInfiniteCost()) {
            // no previous part without exceeded risk policy
            this.clearDynamicObstacles();
            this.clearWaypoints();
            this.getGoal().setInfiniteCost();
            repaired = false;
        } else if (this.getStart().getEto().equals(partStart.getEto())) {
            // propagate potential cost change
            double deltaCost = partStart.getCost() - this.getStart().getCost();
            if (0d != deltaCost) {
                for (AStarWaypoint waypoint : this.visited) {
                    ADStarWaypoint adswp = (ADStarWaypoint) waypoint;
                    adswp.setG(adswp.getG() + deltaCost);
                    adswp.setV(adswp.getV() + deltaCost);
                }
            }

            // connect current to previous part
            if (partStart.hasParent()) {
                this.getStart().setParent(partStart.getParent());
            }

            // repair affected waypoints for current part
            for (ADStarWaypoint target : this.findAffectedWaypoints()) {
                if (!this.getStart().equals(target)) {
                    this.repair(target);
                    this.updateSets(target);
                }
            }
            this.clearDynamicObstacles();
        } else {
            // plan current part from scratch if start ETO has changed
            this.initialize(this.getStart(), this.getGoal(), partStart.getEto());
            if (partStart.hasParent()) {
                this.getStart().setParent(partStart.getParent());
            }
            this.backups.get(partIndex).clear();
            super.planPart(partIndex);
        }
    }

    return repaired;
}
```

Listing 6.11: SAFCS Core: AD* Repair

```
protected Set<ADStarWaypoint> findAffectedWaypoints() {
    Set<ADStarWaypoint> affectedWaypoints = new HashSet<>();

    for (Obstacle obstacle : this.getDynamicObstacles()) {
        affectedWaypoints.addAll(
                ((DynamicEnvironment) this.getEnvironment())
                .getAffectedWaypointPositions(obstacle).stream()
                .map(position -> this.createWaypoint(position))
                .filter(waypoint -> waypoint.hasEto() && waypoint.hasParent())
                .filter(waypoint -> obstacle.getCostInterval().intersects(
                        new TimeInterval(
                                waypoint.getParent().getEto(),
                                waypoint.getEto())))
                .collect(Collectors.toSet()));
    }

    if (affectedWaypoints.containsAll(this.getGoalRegion().stream()
            .map(p -> this.createWaypoint(p))
            .collect(Collectors.toSet()))) {
        affectedWaypoints.add(this.getGoal());
    }

    return affectedWaypoints;
}
```

Listing 6.12: SAFCS Core: AD* Repair — Affected Waypoints

representing the POI between the previous and current parts is ETO-consistent (line 28), the current part is repaired accordingly (lines 46 to 49).

The implementation of the improvement strategy that controls this inflation and deflation of ϵ is shown in the `improve` and related methods in Listing 6.13 and is the second step of elaboration, as shown in Listing 6.10. Before completing the initiated repair in the `compute` method by propagating inconsistencies and re-establishing consistency accordingly, the improvement step controls the appropriate quality by either inflating (line 8) or deflating (line 11) ϵ.

What constitutes a *significant change* to justify an inflation of ϵ is determined in the SAFCS core by comparing a threshold change that describes the maximum acceptable percentage of inconsistent `ADStarWaypoints` in the `Trajectory` that is to be repaired. This is merely one of many possibilities and not necessarily the best one under all circumstances, but it provides a quantitative measure for the inconsistency of an established `Trajectory`. Other options are discussed in Chapter 15 and include more qualitative measures, such as the proximity of the change to the start `Waypoint`. The SAFCS AM is designed to have access to the `significantChange` threshold attribute of the `ADStarPlanner` in order to tune the algorithm for improved performance.

More details about the SAFCS core `ADStarPlanner`[4] can be found in our SAFCS software repository.

[4] https://github.com/stephanheinemann/worldwind/blob/develop/src/main/java/com/cfar/swim/worldwind/planners/cgs/adstar/

```
protected void improve(int partIndex) {
    this.backup(partIndex);

    synchronized(this) {
        // determine significant change
        if (this.hasSignificantChange()) {
            // inflate to lower quality or re-plan from scratch
            this.inflate();
        } else if (!this.isDeflated()) {
            // deflate to higher quality and continue improving
            this.deflate();
        }
    }

    this.restore(partIndex);

    this.compute();
    this.revisePlan(this.createTrajectory());
}

protected synchronized boolean hasSignificantChange() {
    boolean hasSignificantChange = false;
    // TODO: examine different determination policies including
    // (1) consistent, under-consistent, over-consistent waypoints
    // (2) inconsistencies close to start versus close to goal (ETOs)
    // (3) even if the current trajectory is not directly affected,
    //     the surrounding environment could allow for improvements

    // the amount of waypoints in the current plan
    int wc = this.getWaypoints().size();
    // the amount of inconsistent waypoints in the current plan
    long ic = this.getWaypoints().stream()
        .filter(waypoint -> !((ADStarWaypoint) waypoint).isConsistent())
        .count();
    // the ratio of inconsistent to all waypoints in the current plan
    if (0 != wc) {
        this.change = ((double) ic) / wc;
        hasSignificantChange = (this.change >= this.significantChange);
    }

    return hasSignificantChange;
}

protected synchronized void inflate() {
    // TODO: consider different significant change handling strategies
    // increase epsilon or re-plan from scratch
    double qualityRange = this.getMinimumQuality() - this.getMaximumQuality();
    double qualityAdjustment = this.getMaximumQuality()
        + (this.change * qualityRange);

    if (this.getInflation() < qualityAdjustment) {
        this.setInflation(qualityAdjustment);
    }
}

protected synchronized void deflate() {
    if (this.finalInflation < (this.inflation - this.deflationAmount)) {
        this.inflation -= this.deflationAmount;
    } else {
        this.inflation = this.finalInflation;
    }
}
```

Listing 6.13: SAFCS Core: AD* Improvement

6.5 Online AD*

The final and most sophisticated extension within the A* family is the OAD* algorithm. As mentioned in Section 5.2.5 and shown in Figure 5.21, an `OnlinePlanner`, such as the `OADStarPlanner`, has access to a `Datalink` source and associated `Communication` sequences in order to monitor and affect the progress of a planned `Aircraft` presently executing a live mission.

In addition to dynamically revising and improving trajectories, as realized by the more general `ADStarPlanner`, an `OADStarPlanner` is required to initiate take-off and landing sequences as well as to recognize off-track or unanticipated threat situations of its airborne `Datalink` source. Take-off and (planned or unplanned) landing sequences are `Datalink` `Communications` and aggregated by the `OADStarPlanner`, as shown in Figure 6.11, and initiated upon request of the `Planner` if the `Datalink` source is within a prescribed take-off or landing zone and time window, respectively.

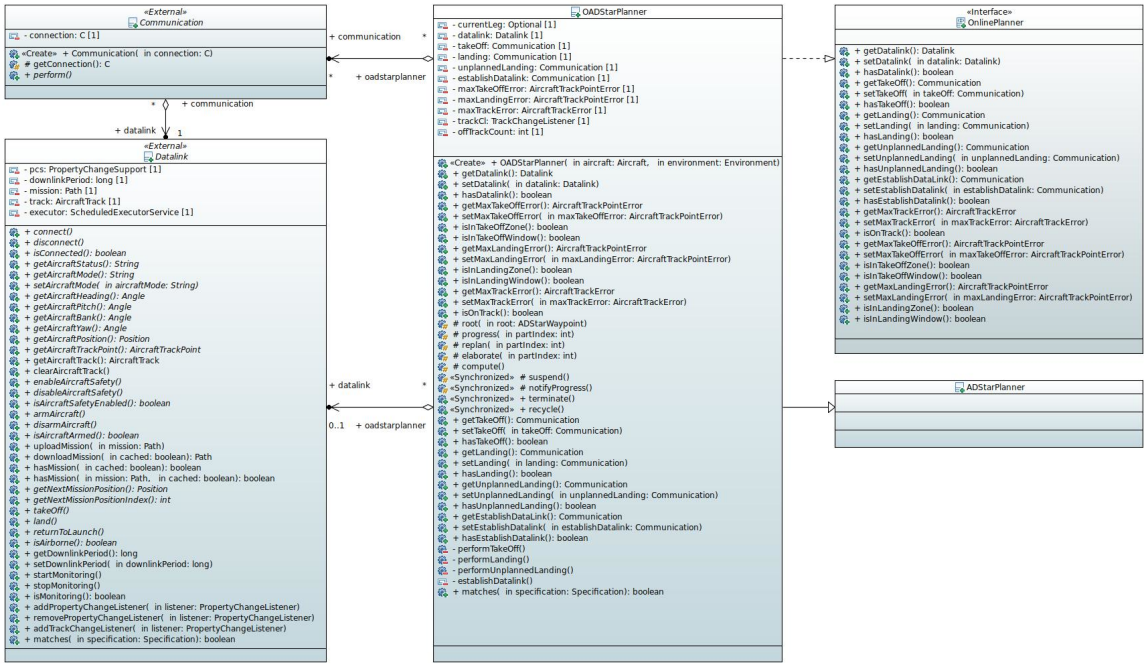


Figure 6.11: SAFCS Core: OAD* Class Signatures

Listing 6.14 details the conditions to determine whether or not a connected `Datalink` source can take-off within an operationally constrained take-off zone and time window, respectively. This procedure is very similar to the landing scenario and shall not be repeated for the equivalent case. It should be obvious that the `Datalink` source is unlikely to be found at the exact position and time an `OnlinePlanner` expects

it to be. This is due to different aspects, including, but not limited to, navigational sensor errors and the resulting aircraft performance. Hence, the operator has to define appropriate maximum `AircraftTrackPointErrors` for take-off and landing in order to determine whether or not the observed error for the last `AircraftTrackPoint` is within limits. An `AircraftTrackPointError` comprises both spatial and temporal components and, therefore, serves to verify both take-off zone and time window requirements.

Once airborne, the aircraft progress is monitored and off-track situations are determined according to an operator-defined maximum `AircraftTrackError`. As with `AircraftTrackPointErrors`, this error is defined in terms of both maximum spatial and temporal tolerances from the planned and, therefore, expected track. Consequently, the progress of a `Datalink` source is continuously monitored and re-planned if an off-track situation occurs. It should be emphasized that the `OADStarPlanner` initiates an elaboration not just after the injection of dynamic changes into its `PlanningGrid`, as discussed in Section 6.4, but also after recognizing an off-track situation in order to re-plan a `Trajectory` and re-establish the Required Navigation Performance (RNP) in terms of space and time. Progressing the `Aircraft` is also part of the elaboration itself (line 21) and shown in Listing 6.15. Progressing essentially means modifying the established plan by removing already visited `Waypoints`. During an off-track situation, the last received `AircraftTrackPoint` is used as the basis to determine the new ETO of the next regularly planned `ADStarWaypoint` by attempting a direct flight path to it.

Another important aspect of `OnlinePlanners` is the requirement to deal with unanticipated high costs due to dynamic changes in the supported `Environment`. If the accumulated costs at the present `Datalink` source position in flight suddenly and unexpectedly exceed the operational `RiskPolicy`, then this `RiskPolicy` has to be temporarily boosted in order to escape the perceived threat. Giving up planning for an airborne agent is not an option. However, ideally the operator needs to be informed about any unanticipated changes to the `RiskPolicy`. On the other hand, boosting the `RiskPolicy` should not be applied simply because the `Environment` gets more saturated with higher cost `Obstacles`, including, but not limited to, the destination position. This behavior would correspond to a *pilot get-homitis* and is undesirable as it unnecessarily raises the risk beyond the limit that had previously been established as acceptable to achieve the mission goal. In these cases, proceeding toward in-flight alternates or performing unplanned precautionary landing `Communications` via the `Datalink` are the favorable actions to be taken instead. In-flight alternates and unplanned precautionary landings are not yet supported and further discussed in Chapter 15.

```
public boolean isInTakeOffZone() {
    boolean isInTakeOffZone = false;

    if (this.hasDatalink() && this.getDatalink().isConnected()
            && this.getDatalink().isMonitoring()) {
        AircraftTrack track = this.getDatalink().getAircraftTrack();
        AircraftTrackPoint last = track.getLastTrackPoint();
        Position start = this.getStart().getPrecisionPosition().getOriginal();

        // horizontal error
        Position spos = new Position(start, 0d);
        Position lpos = new Position(last.getPosition(), 0d);
        double dhor = this.getEnvironment().getGreatCircleDistance(spos, lpos);

        // vertical error
        double dver = Math.abs(start.getElevation()
                - last.getPosition().getElevation());

        // position error
        Angle dlat = start.getLatitude()
                .angularDistanceTo(last.getPosition().getLatitude());
        Angle dlon = start.getLongitude()
                .angularDistanceTo(last.getPosition().getLongitude());

        isInTakeOffZone = (this.getMaxTakeOffError().getHorizontalError() >= dhor)
            && (this.getMaxTakeOffError().getVerticalError() >= dver)
            && (this.getMaxTakeOffError().getPositionError().getElevation() >= dver)
            && (0 >= dlat.compareTo(this.getMaxTakeOffError().getPositionError()
                .getLatitude()))
            && (0 >= dlon.compareTo(this.getMaxTakeOffError().getPositionError()
                .getLongitude()));
    }

    return isInTakeOffZone;
}

public boolean isInTakeOffWindow() {
    boolean isInTakeOffWindow = false;

    if (this.hasDatalink() && this.getDatalink().isConnected()
            && this.getDatalink().isMonitoring()
            && this.getStart().hasEto()) {
        AircraftTrack track = this.getDatalink().getAircraftTrack();
        AircraftTrackPoint last = track.getLastTrackPoint();

        // calculate take-off window with take-off error margin
        ZonedDateTime takeOffTime = this.getStart().getEto();
        TimeInterval takeOffWindow = new TimeInterval(
                takeOffTime.minus(this.getMaxTakeOffError()
                        .getTimingError().getSeconds(), ChronoUnit.SECONDS),
                takeOffTime.plus(this.getMaxTakeOffError()
                        .getTimingError().getSeconds(), ChronoUnit.SECONDS));

        isInTakeOffWindow = takeOffWindow.contains(last.getAto());
    }

    return isInTakeOffWindow;
}
```

Listing 6.14: SAFCS Core: OAD* Take-Off Conditions

```
protected void elaborate(int partIndex) {
    // limit deliberation for active part
    if (partIndex == this.getActivePart()) {
        this.setDeliberationStart(ZonedDateTime.now());
        this.setDeliberationEnd(
            this.getDeliberationStart().plus(this.getDeliberation()));
    } else {
        this.setDeliberationEnd(null);
    }

    if (!this.isOnTrack()) {
        this.offTrackCount = 0;
        this.replan(partIndex);
    }

    // proceed to next part only if fully improved and not in need of repair
    while (this.canDeliberate() && !this.hasTerminated()
            && (!this.hasMaximumQuality() || this.needsRepair())) {
        this.repair(partIndex);
        this.improve(partIndex);
        this.progress(partIndex);
        this.updateDynamicObstacles();
    }
    // backup after elaboration
    this.backup(partIndex);
}
```

Listing 6.15: SAFCS Core: OAD* Progress Monitoring

More details about progress monitoring, off-track determination, supported communication sequences, and other aspects of the SAFCS core `OADStarPlanner`[5] can be found in our SAFCS software repository.

[5] https://github.com/stephanheinemann/worldwind/tree/develop/src/main/java/com/cfar/swim/worldwind/planners/cgs/oadstar

6.6 Chapter Summary

In this chapter we studied some implementation details addressing important modifications and extensions to the supported members of the A* family of algorithms. The presented modifications and extensions relate to the integration of `CostIntervals` for SWIM `Obstacles` into the supported planning `Environments` as well to the temporal aspects affecting the consistency definitions of `Waypoints` in 4D single- and multi-part trajectories. We, furthermore, identified problematic areas and options for improvement that shall be addressed in future research of our CGS core algorithms.

Chapter 7

SAFCS Core — Rapidly-Exploring Random Trees

This chapter gives an overview of the architectural and functional aspects regarding the implementation of the RRT family of algorithms within the core component of the SAFCS.

As surveyed in Chapter 2, the family of RRT algorithms embraces the idea of a rapidly and biased growing tree from a start toward a goal configuration. The idea of RRTs was first introduced by LaValle [Lav98] and can, as further detailed by LaValle and Kuffner [LK99], implicitly consider non-holonomic kinodynamic constraints of an agent operating within a high-dimensional search space. The original, most basic, algorithm takes random samples from a continuous environment, sufficiently biased toward a goal, and connects steps toward them to a growing tree. The consecutively added branches of the tree are checked against collisions with obstacles within the environment and may implicitly consider the capabilities of an agent. Performing an uninformed exploration of the environment, the original basic RRT does not incorporate costs or any cost estimates—it rather simply generates a collision-free and feasible path from the start to the goal configuration.

The original idea was subsequently refined and modified producing many different variants of more informed RRT algorithms. The most notable extensions incorporate the notion of costs and optimality for Optimal RRT (RRT*) (Karman and Frazzoli [KF10]), apply heuristics to guide the search for Heuristic RRT (HRRT) (Urmson and Simmons [US03]), are capable of coping with partial knowledge about a dynamically changing environment for DRRT (Ferguson, Kalra and Stentz [FKS06]), acknowledge the

issue of bounded rationality under limited deliberation time for ARRT (Ferguson and Stentz [FS06a]), and combine those features for ADRRT (van den Berg, Ferguson and Kuffner [vdBFK06], Ferguson and Stentz [FS07]) to create algorithms that are capable of dealing better with many real world problems.

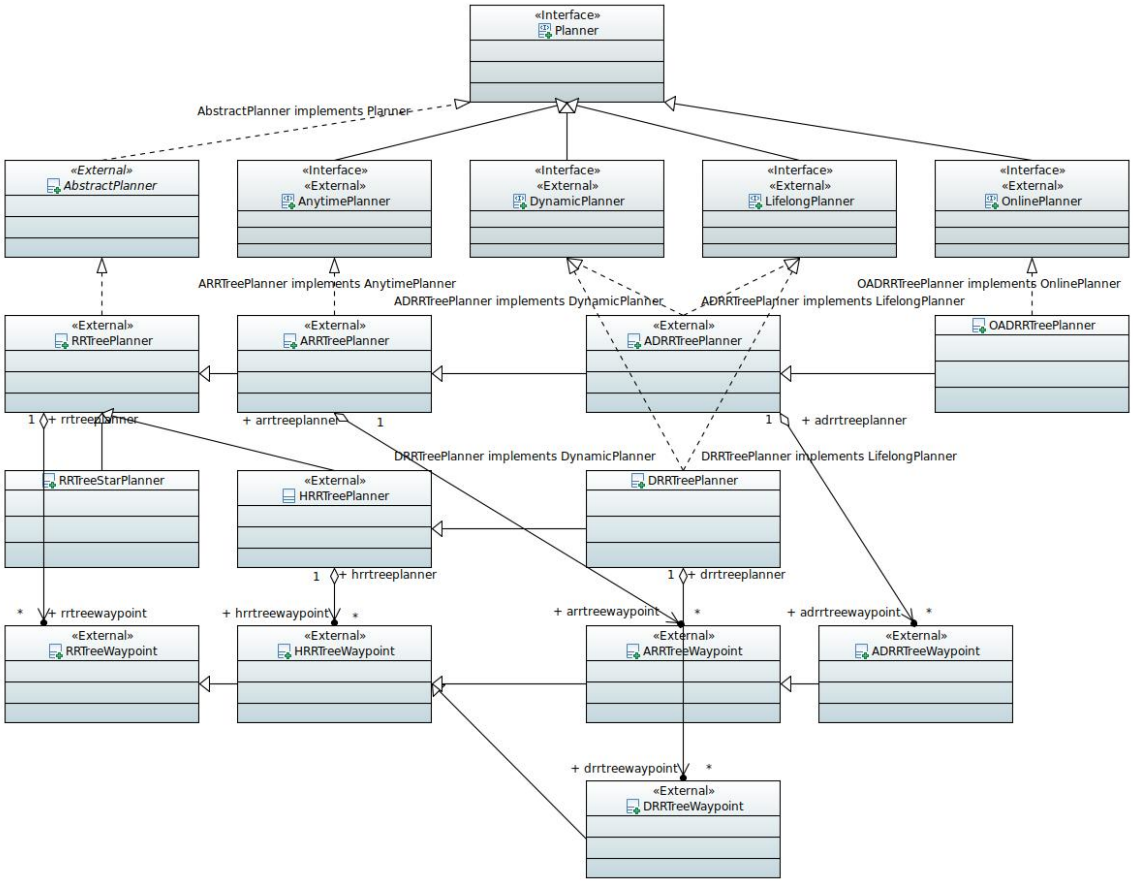


Figure 7.1: SAFCS Core: RRT Family

Figure 7.1 shows the hierarchy of `RRTreePlanners` that are currently supported by the SAFCS core. As with all supported planning families, this hierarchy is far from being exhaustive and covers only a small portion of the existing research in this area. It can, however, be easily extended for the integration, modification and evaluation of existing and newly developed algorithms that are of particular interest for our domain and possible candidates to serve as tunable `ManagedPlanners` of the SAFCS AM. These algorithms ideally incorporate the concepts of anytime and lifelong dynamic planning to be suitable candidates. The SAFCS core architecture again leverages generalization and specialization to reuse a significant amount of logic shared between the implemented members of this family. The key algorithm with re-

spect to the SAFCS AM component in terms of selection and tuning is Online ADRRT (OADRRT), as realized in the `OADRRTreePlanner` class. However, its logic builds on the more basic RRT, ARRT, and ADRRT `Planners` which are realized as `RRTreePlanner`, `ARRTreePlanner`, and `ADRRTreePlanner`, respectively. Other `RRTreePlanners`, such as the `RRTreeStarPlanner` and `HRRTreePlanner`, have been integrated into and evaluated within the SAFCS core but found to be less suitable to form the basis of manageable `Planners` for reasons given in the following sections.

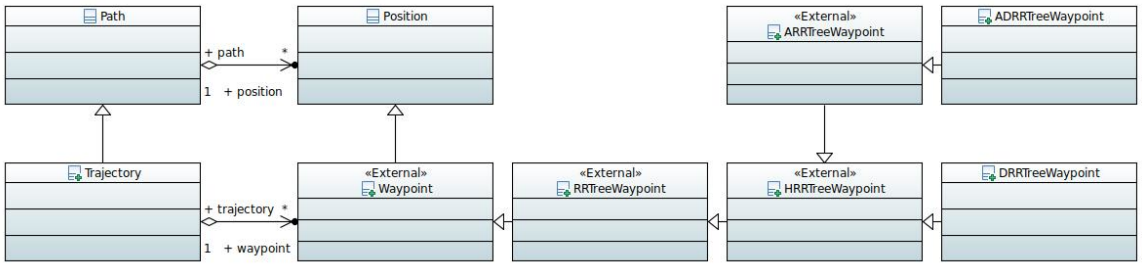


Figure 7.2: SAFCS Core: RRT Waypoints

The supported `RRTreePlanners` operate on the dedicated hierarchy of `RRTreeWaypoints` shown in Figure 7.2. Recall that all SAFCS core `Waypoints` include the concept of both space and time to form a 4D `Trajectory`. Each individual class in the `RRTreeWaypoint` hierarchy only adds properties as required by its corresponding more specialized `RRTreePlanner`.

Similar to the integration of A* algorithms into the SAFCS studied in Chapter 6, the integration of RRTs requires several modifications particularly with respect to the embedding of `CostIntervals` associated with temporal SWIM data as well as the consideration for an operational `RiskPolicy` and `CostPolicy`. The following sections, therefore, focus primarily on the particularities of our SAFCS core implementation and the differences between the ideas surveyed in Chapter 2 and presented in the original literature.

7.1 Cost Integration

RRT planners may or may not apply the notion of known versus estimated costs in order to guide their search toward an operational goal. They operate on a continuous model of the environment that is sampled in a particular fashion to create its internal structure which comprises the branches of a Rapidly-Exploring Random Tree or, more generally,

vertices and Edges. In case of RRTs, the Edges are more special DirectedEdges as they represent branches grown from their root within the tree to the tip of the branch. It is these Edges representing the branches of the tree that are associated with costs. Whereas more basic RRT variants ignore these embedded costs, more advanced probabilistically optimizing or heuristic algorithms consider them while sampling new vertices and extending branches from the existing tree.

Section 5.2.3 already described the architectural structures integrated into the SAFCS core to embed costs in both space and time into the planning Environment. Recall that costs in the aviation domain are associated with aeronautical-, weather-, flight- and aerodrome-related situations. These different concerns are covered by SWIM datasets and can be injected into the SAFCS core by means of a SwimConnection and an Obstacle reporting framework, as explained in Section 5.2.4.

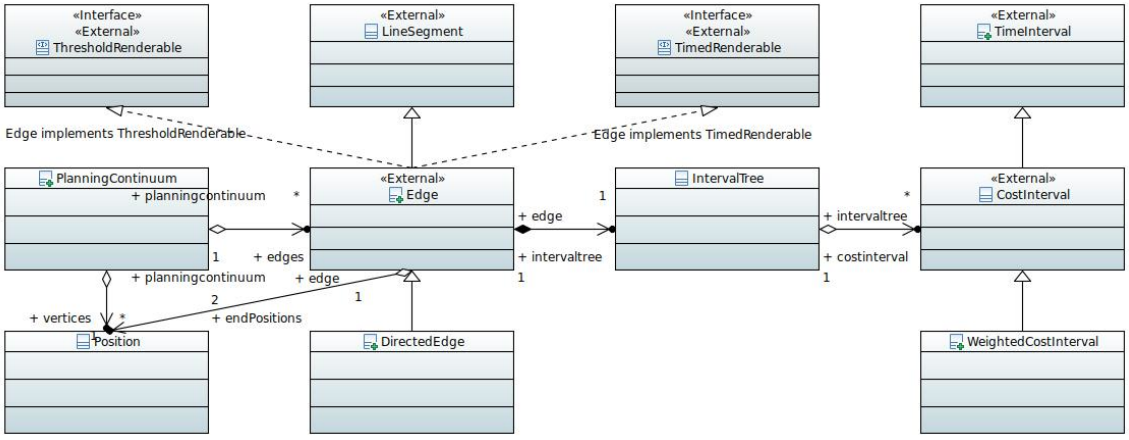


Figure 7.3: SAFCS Core: Edges and Cost Intervals

The architectural design to integrate costs into the Edges of a PlanningContinuum is shown in Figure 7.3. Recall from Section 5.2.3 that these Edges represent the dynamically changing internal structure of a PlanningContinuum realizing a StructuredEnvironment. An Edge, as a specialized LineSegment, furthermore, realizes the interfaces ThresholdRenderable and TimedRenderable. As such, its costs can be readily displayed in our SAFCS UI according to a certain cost threshold and Environment model time, respectively. The similarities and differences with respect to the internal structure and their cost integration between the PlanningGrid presented in Section 6.1 and the PlanningContinuum described here should be emphasized again. Whereas the internal structure of a PlanningGrid is a potentially empty cubic hierarchical structure that can subsequently be refined by a compatible Planner,

the internal structure of a PlanningContinuum is initially empty and subsequently created as the Planner operates on it. As opposed to the voxels of a PlanningGrid, the Edges of a PlanningContinuum are currently not considered Environments themselves, but an extension could potentially allow that aspect to be realized and evolve the PlanningContinuum into a HierarchicalEnvironment, as further discussed in Chapter 15.

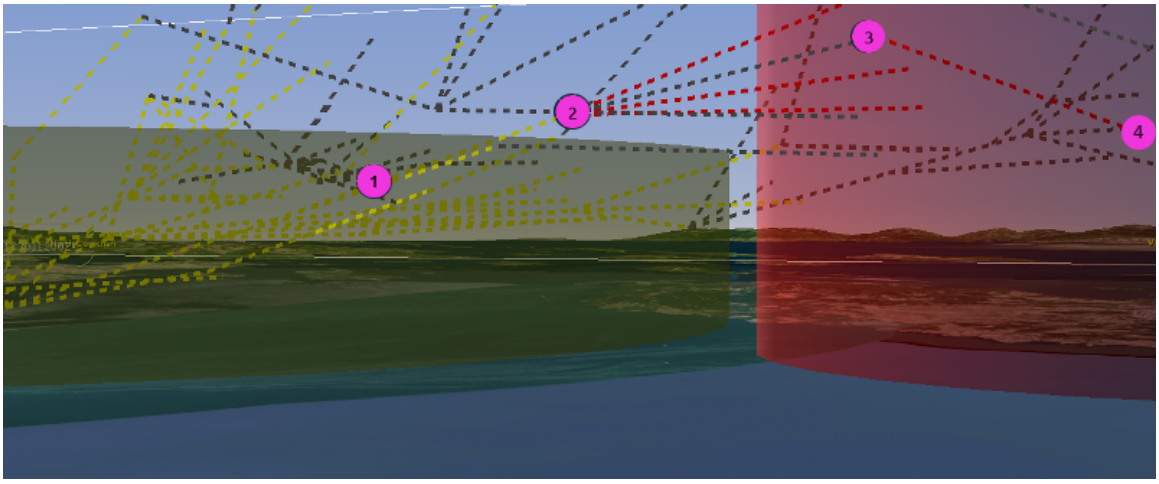


Figure 7.4: SAFCS UI: Planning Continuum Obstacles and Waypoints

Figure 7.4 shows a PlanningContinuum with two embedded SWIM Obstacles. The SAFCS UI readily displays different costs that have been assigned to a CostInterval of an Obstacle according to a CostMap using different colors. In the example, *yellow* corresponds to a cost of 40 and *red* to a cost of 60. As the Rapidly-Exploring Random Tree grows into the Obstacle extents, its Edges embed the associated CostInterval. Whether or not an Edge is affected by the cost of the embedded CostInterval, consequently, depends on the time of the Environment model. Thus, if an Edge intersects an Obstacle in space and its CostInterval in time, it is affected by the associated cost. Where multiple costs overlap in space and time, the resulting accumulated cost is computed according to the applicable operational RiskPolicy and CostPolicy, and displayed accordingly.

As mentioned in Section 5.2.3 and shown in Figure 7.3, each Edge aggregates its own binary tree of CostIntervals. The CostIntervals for the Environment model shown in Figure 7.4 are examined more closely in Figure 7.5. In the depicted example, the western (yellow-colored) Obstacle moves eastbound is decreasing in size and dissipating, and the most costly (red-colored) eastern Obstacle is stationary in space and time.

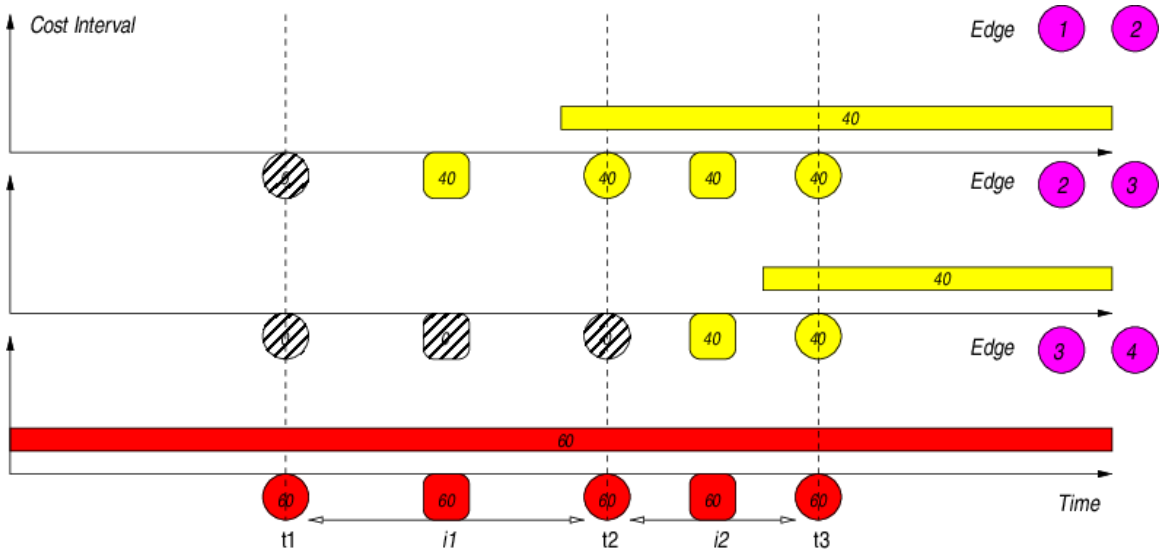


Figure 7.5: SAFCS Core: Edges Cost Interval Example

The diagram shown in Figure 7.5 depicts the `CostIntervals` of the three `Edges` $\overline{12}$, $\overline{23}$, and $\overline{34}$, respectively. It, furthermore, shows the 3 different time instances $t1$, $t2$, and $t3$. As can be deduced from the depiction of the `Environment` model in Figure 7.4, its time instance may likely correspond to $t2$ where the three `Edges` $\overline{12}$, $\overline{23}$, and $\overline{34}$ are affected by an individual cost of 40, 0, and 60, respectively. As the western `Obstacle` moves eastbound, `Edge` $\overline{23}$ is eventually affected by its higher cost of 40 at time instance $t3$. Since the eastern `Obstacle` is stationary in both space and time, the intersecting `Edge` $\overline{34}$ is, consequently, affected by its cost of 60 during the entire observation period. The rounded boxes in between the cost circles for the time instances may represent the intervals required for a planned `Edge` passage by an `Aircraft` with their lengths corresponding to the ETE required to transit the particular `Edge`. Consequently, the most conservative cost assessment is the highest possible accumulated cost of all valid `CostIntervals` during the `Edge` passage.

The `Edges` of a `PlanningContinuum` are based on vertices represented by `Positions`, as shown in Figure 7.3. It is these vertices which are sampled by concrete RRT `Planners` and are represented by dedicated `RRTWaypoints` of the `Waypoint` hierarchy shown in Figure 7.6. This hierarchy exposes the functional signatures of the `RRTWaypoints` previously abstracted in Figure 7.2. Recall that any `Waypoint` features an `eto` property which is assigned by RRT `Planners` during the tree extension according to `Aircraft Capabilities` based on an ETD at the root of the RRT. Hence, the `Edge` passage `TimeInterval` representing the corresponding ETE can be used by RRT `Planners` to ef-

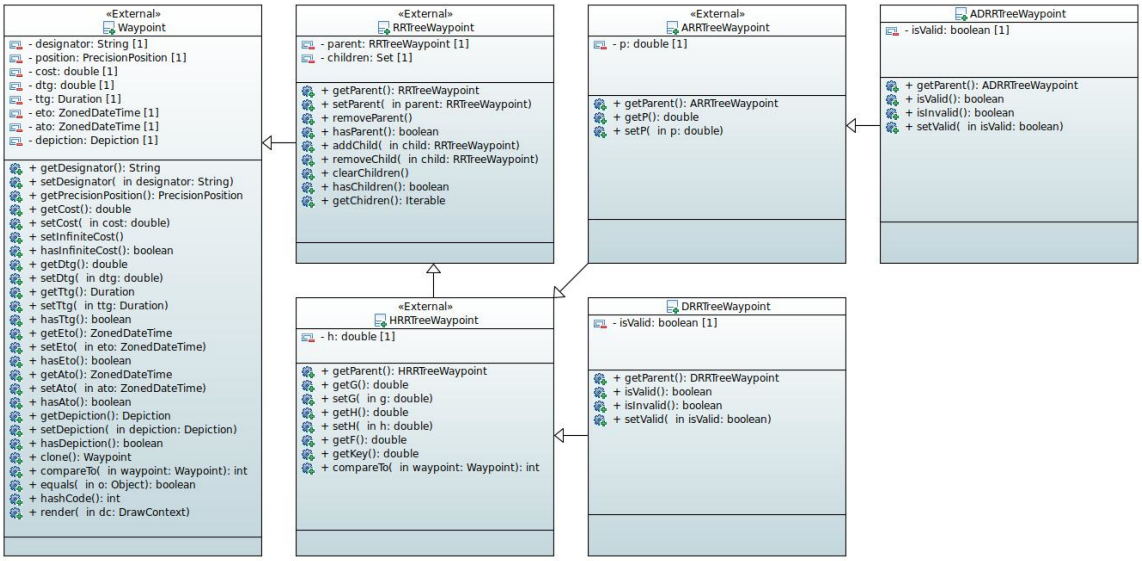


Figure 7.6: SAFCS Core: RRT Waypoint Signatures

ficiently lookup the relevant intersecting `CostIntervals` in the `IntervalTree` and calculate the resulting costs, as described in Sections 5.2.3 and 5.2.5. The result of this cost calculation is stored in the `cost` attribute of a `Waypoint` and represents the accumulated costs from the point of departure at the ETD to reach that `Waypoint` at its ETO. Clearly, the `cost` of a `DirectedEdge` target cannot be lower than that of its source `Waypoint`. Notice that regardless of the concrete underlying `Waypoint` hierarchy, the cost integration and determination at the `Waypoint` level is equivalent for RRT and the CGS algorithms explained in Chapter 6. Both concrete hierarchies are derived from the same `Waypoint` base class and only add attributes as required by their compatible `Planners`.

These are the basic cost determination steps that are implemented by all concrete RRT `Planners` operating on a `PlanningContinuum`:

1. Determine the participating `Edges` for a step or leg
2. Determine the valid `CostIntervals` during the `Edge` passage
3. Ensure that local costs satisfy an operational `RiskPolicy`
4. Accumulate the resulting costs according to an operational `CostPolicy`

Listing 7.1 reveals the basic cost computation performed by all concrete RRT `Planners`. Once a new vertex, that is, target `Waypoint`, has been sampled from the `PlanningContinuum`, the ETO to reach that target is estimated based on its source `eto`

```
1  protected void computeEto(Waypoint source, Waypoint target) {
2      Path leg = new Path(source, target);
3      Capabilities capabilities = this.getAircraft().getCapabilities();
4      Globe globe = this.getEnvironment().getGlobe();
5      target.setEto(capabilities.getEstimatedTime(leg, globe, source.getEto()));
6  }
7
8  protected void computeCost(Waypoint source, Waypoint target) {
9      double cost = source.getCost();
10     cost += this.getEnvironment().getStepCost(
11             source, target,
12             source.getEto(), target.getEto(),
13             this.getCostPolicy(), this.getRiskPolicy());
14     target.setCost(cost);
15 }
```

Listing 7.1: SAFCS Core: RRT Cost Computation

```
1  public double getStepCost(
2          Position origin, Position destination,
3          ZonedDateTime start, ZonedDateTime end,
4          CostPolicy costPolicy, RiskPolicy riskPolicy) {
5
6      double stepCost = Double.POSITIVE_INFINITY;
7      Optional<Edge> edge = this.findEdge(origin, destination);
8
9      if (edge.isPresent()) {
10         double distance = this.getNormalizedDistance(origin, destination);
11         double cost = edge.get().getCost(start, end, costPolicy, riskPolicy);
12         stepCost = distance * cost;
13     } else {
14         throw new IllegalStateException("no edge containing both positions");
15     }
16
17     return stepCost;
18 }
```

Listing 7.2: SAFCS Core: Planning Continuum Step Cost Computation

attribute and the `Aircraft Capabilities` within the `Environment`. This computation is performed by the `computeEto` method in lines 1 to 6. If the `Aircraft Capabilities` allow for the transition along an `Edge` from the source to the target `Waypoint` with an established target ETO, the resulting cost is computed by the `computeCost` method listed in lines 8 to 15.

The concrete `Environment`, the `PlanningContinuum` in this case, implements the `getStepCost` method which is invoked in line 10 of Listing 7.1 and shown in Listing 7.2. The internal structure of the `PlanningContinuum` has to contain the queried `Edge`, which is always created by the RRT `Planner` once the target ETO has been successfully established. As mentioned in Section 6.1 for Listing 6.2, the implementation of `getStepCost` depends on the concrete `Environment`. Since there are no neighboring voxels (cells) as in `PlanningGrids`, the operational `CostPolicy` and `RiskPolicy` is only

incorporated in the `getCost` method of the `Edge` (line 11). It can be argued that a capped cylinder with the queried `Edge` as its main axis and a radius that corresponds to the `Aircraft` `ObstacleSphere` shown in Figure 5.6 should be employed for spatial intersection tests instead. This more conservative approach is revisited in Chapter 15. The `getCost` method is, however, the same for CGS and RRTs, as discussed in Section 5.2.5. Similar to CGS `Planners`, a normalized distance between the step origin and destination serves as the basis for the step cost computation in line 12. An agent being exposed to a certain cost for a longer distance and, therefore, ETE intuitively receives a higher cost penalty.

Listing 7.3 shows how these ideas are implemented depending on whether or not a `CostPolicy` and `RiskPolicy` are provided. To calculate the effect that `Obstacles` and their `CostIntervals` have on internal structures of the concrete `StructuredEnvironment` for the purpose of showing them in our SAFCS UI, operational policies are considered irrelevant. The method `getCost` shown in lines 1 to 14, thus, simply assumes the maximum cost an internal structure, such as an `Edge`, is affected by and does not boost any discovered costs. However, if a `Planner` is involved, the costs are assessed differently and according to the specified operational policies, as shown in lines 16 to 68. Both methods obtain the sub-cost intervals first, as explained in Section 5.2.5, and then proceed with the necessary cost adjustments. Our SAFCS UI could arguably display a concrete `Environment` according to an operational policy in order to provide the same view regarding costs to the operator as taken into account by the `Planner`. This option is further discussed in Chapter 15.

```java
public double getCost(ZonedDateTime start, ZonedDateTime end) {
    double cost = 0d;

    // obtain sub-cost intervals
    Set<CostInterval> subCostIntervals =
            this.getSubCostIntervals(start, end);

    // assume maximum cost
    cost = subCostIntervals.stream()
            .map(c -> c.getCost()).mapToDouble(Double::doubleValue)
            .max().getAsDouble();

    return cost;
}

public double getCost(ZonedDateTime start, ZonedDateTime end,
            CostPolicy costPolicy, RiskPolicy riskPolicy) {
    double cost = 0d;

    // obtain sub-cost intervals
    Set<CostInterval> subCostIntervals =
            this.getSubCostIntervals(start, end);

    // adhere to operational risk policy
    subCostIntervals.stream()
        .forEach(c -> {
            if (!riskPolicy.satisfies(
                    c.getCost() - this.getEnvironment().getBaseCost()))
                c.setCost(Double.POSITIVE_INFINITY);
        });

    // apply operational cost policy
    switch (costPolicy) {
    case MINIMUM:
        cost = subCostIntervals.stream()
            .map(c -> c.getCost()).mapToDouble(Double::doubleValue)
            .min().getAsDouble();
        break;
    case MAXIMUM:
        cost = subCostIntervals.stream()
            .map(c -> c.getCost()).mapToDouble(Double::doubleValue)
            .max().getAsDouble();
        break;
    case AVERAGE:
        if (1 == subCostIntervals.stream()
            .map(c -> c.getCost()).mapToDouble(Double::doubleValue)
            .distinct().count()) {
            // avoid numerical issues in case of equal costs
            cost = subCostIntervals.stream().findFirst().get().getCost();
        } else {
            Duration duration = Duration.between(start, end);
            double intervalSeconds = duration.getSeconds()
                    + (duration.getNano() * 1E-9d);
            double subIntervalSeconds;
            for (CostInterval subCostInterval : subCostIntervals) {
                duration = Duration.between(subCostInterval.getLower(),
                        subCostInterval.getUpper());
                subIntervalSeconds = duration.getSeconds()
                        + (duration.getNano() * 1E-9d);
                cost += subCostInterval.getCost()
                        * subIntervalSeconds / intervalSeconds;
            }
        }
        break;
    }

    return cost;
}
```

Listing 7.3: SAFCS Core: Edge Cost Computation

7.2 Basic RRT

As shown in Figure 7.1, all concrete more sophisticated RRT `Planners` are derived from the basic `RRTreePlanner` and inherit a significant amount of logic and related properties. Therefore, these common aspects need to be examined more closely to eventually approach the discussion of the key OADRRT algorithm as one of the managed candidates of our SAFCS AM.

An SAFCS core `RRTreePlanner` samples vertices, that is, `Positions`, from its `PlanningContinuum` according to a configurable `Sampling` approach, `Extension` technique, and growth `Strategy`, as shown in Figure 7.7. The `Sampling` approach determines how samples (vertices) that are taken from the `PlanningContinuum` are distributed. The currently supported approaches are `UNIFORM`, `GAUSSIAN`, and `ELLIPSOIDAL`. Whereas `UNIFORM` and `GAUSSIAN` potentially sample `Positions` from the entire `PlanningContinuum`, `ELLIPSOIDAL` sampling is much more focused on the area of interest in between the start and goal POIs specified by the operator. It essentially treats both POIs as the foci points of an ellipsoidal space that features certain standard parameters and intersects with the `PlanningContinuum`.

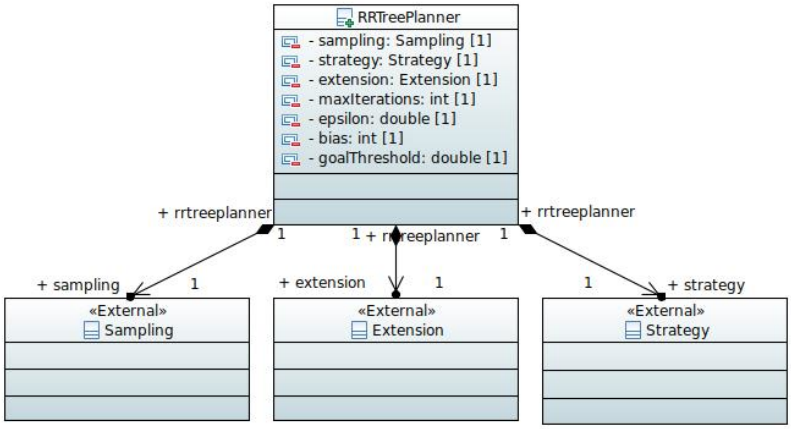
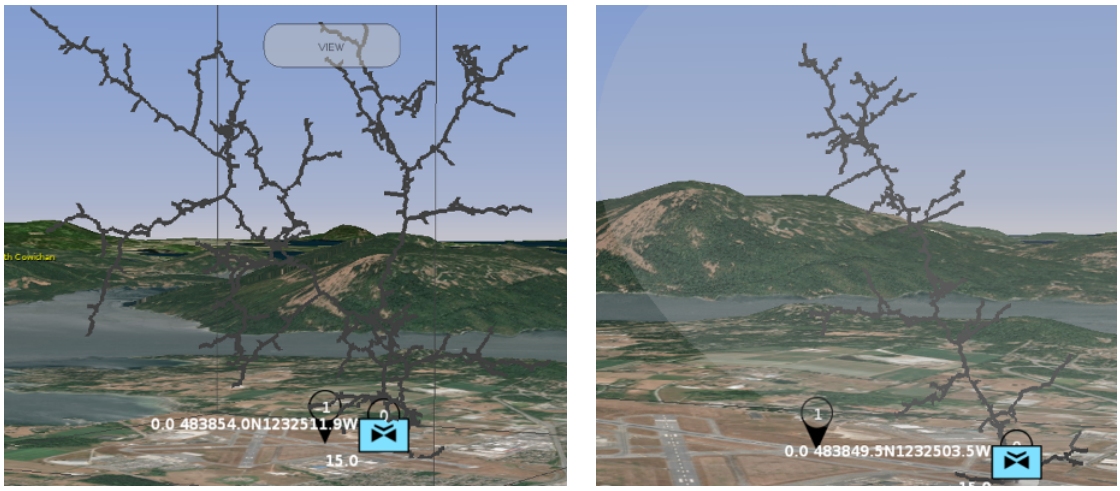


Figure 7.7: SAFCS Core: Basic RRT

Figure 7.8 shows a visual comparison between the `UNIFORM` and `ELLIPSOIDAL` `Sampling` approaches using our SAFCS UI. While sampling, the SAFCS UI highlights the sampling space in order to provide an improved feedback. This sampling space is partially visible in Figure 7.8b and shows the boundary of the associated ellipsoid. In both situations the start and goal POIs are laterally fairly close but vertically well separated. Hence, a biased climbing growth can be seen in both depictions. The more

focused growth of the ELLIPSOIDAL Sampling can still be clearly identified. A certain goal bias is always required for RRTs in order to reach the goal POI eventually, but values in the range of 1 to 5% of the taken samples are usually sufficient. In fact, the growth depicted in Figure 7.8 is biased toward the goal POI with only 1%. The corresponding attribute bias shown in Figure 7.7 can be configured for all derived RRTreePlanners and is an important tuning parameter for the SAFCS AM.



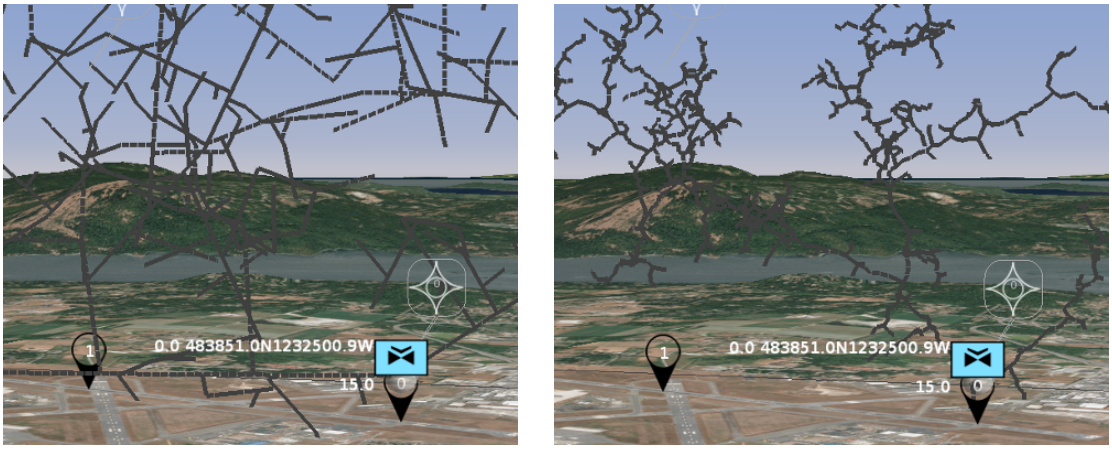
(a) SAFCS UI: RRT Uniform Sampling **(b)** SAFCS UI: RRT Ellipsoidal Sampling

Figure 7.8: SAFCS UI: RRT Sampling

While GAUSSIAN and, in particular, UNIFORM Sampling explore the unexplored search space more evenly, the ELLIPSOIDAL approach extends the tree more aggressively toward the goal POI. Clearly, ELLIPSOIDAL Sampling generally finds solutions more quickly, in particular, if the PlanningContinuum covers a large space compared to a much smaller region of interest between two POIs. However, what may seem advantageous at first sight, can become an issue with an Obstacle-saturated PlanningContinuum, in particular, for more conservative operational RiskPolicy and CostPolicy settings. In cases of close-by Obstacles, that is, in situations featuring increased levels of Criticality, an ELLIPSOIDAL approach may neither be able to penetrate the space ahead nor find a detour around the problematic region. Thus, tuning the Sampling approach for particular situations of different Criticality is an interesting opportunity for the SAFCS AM.

The growth Strategy of an RRTreePlanner is another important tuning parameter as it defines how the RRT grows toward newly sampled vertices. The two available growth strategies are CONNECT and EXTEND, and a visual comparison using our SAFCS UI

is shown in Figure 7.9. Whereas CONNECT attempts to directly connect a new sample with the existing RRT, the EXTEND growth Strategy extends from the existing RRT toward the new sample for a certain maximum distance only. This maximum distance corresponds to the attribute epsilon of RRTreePlanners, as shown in Figure 7.7. Intuitively, larger values of epsilon lead to a behavior that resembles the CONNECT Strategy more closely.



(a) SAFCS UI: RRT Connect Strategy **(b)** SAFCS UI: RRT Extend Strategy

Figure 7.9: SAFCS UI: RRT Connection Strategy

As can be seen from Figure 7.9, RRTs which are grown using CONNECT appear more clunky and do not explore the PlanningContinuum as gracefully as the EXTEND Strategy. They may, therefore, not be able to find very efficient solutions in a more challenging Environment from within a denser Obstacle field. However, their generated solutions contain fewer Waypoints and, hence, fewer heading changes.

Finally, the Extension technique is an important parameter to limit the growth of branches and, hence, the creation of Edges in the PlanningContinuum to those that are kinodynamically feasible considering the Aircraft Capabilities. Since samples are taken probabilistically, heading and altitude changes may be too aggressive and beyond the Capabilities of the planned Aircraft. In order to cope with this issue, the SAFCS core supports the two Extension options LINEAR and FEASIBLE. Whereas the LINEAR technique does not consider any Aircraft Capabilities, a FEASIBLE Extension leads to the creation of kinodynamically feasible Edges only and discards other samples. Currently, the SAFCS core supports only static and fairly basic Capabilities without considering the various air data affecting air density and the resulting Aircraft performance. However, more realistic models could be included to better approximate kinodynamic limitations, as discussed in Chapter 15.

```
protected boolean compute() {
    if (this.getStart().equals(this.getGoal())) {
        this.connectPlan(this.getStart());
        return true;
    }

    this.createSamplingShape();

    int iteration = 0;
    while ((!this.isInGoalRegion()) && (this.getMaxIterations() > iteration)) {
        // sample a new waypoint
        RRTreeWaypoint sample = this.sampleBiased();

        // connect to or extend toward sample according to strategy
        Status status = Status.TRAPPED;
        switch (this.getStrategy()) {
        case CONNECT:
            status = this.connectRRT(sample);
            break;
        case EXTEND:
        default:
            status = this.extendRRT(sample);
        }

        // check goal region
        if ((Status.TRAPPED != status) && this.isInGoalRegion()) {
            if (this.getNewestWaypoint().equals(this.getGoal())) {
                this.setGoal(this.getNewestWaypoint());
            } else {
                this.createExtension(this.getNewestWaypoint(), this.getGoal());
            }
            this.connectPlan();
        }

        iteration++;
    }

    this.disposeSamplingShape();

    if (!this.isInGoalRegion()) {
        Logging.logger().info("no trajectory found after "
            + this.getMaxIterations() + " iterations");
    }

    return this.isInGoalRegion();
}
```

Listing 7.4: SAFCS Core: Basic RRT Computation

Listing 7.4 shows the general computation of an RRT as performed by the most basic RRTreePlanner. Any RRTreePlanner attempts to find a solution for up to a specified maximum number of iterations. The attribute maxIterations shown in Figure 7.7 mainly influences the runtime of the algorithm, and it is important to realize its relationship with the epsilon attribute while growing an RRT using the EXTEND Strategy. The smaller the value of epsilon, the higher is the number of required extensions in order to eventually reach the goal POI. Hence, if epsilon is chosen too small with respect to the value of the maxIterations attribute, the RRTreePlanner will not find a solution. Consequently, the tuning SAFCS AM must ensure viable settings that can actually lead to a solution. Lines 9 to 36 of Listing 7.4 contain the essential steps to compute a Trajectory of RRTreeWaypoints from a start to a goal POI. As long as the goal region is not reached and the number of iterations does not exceed the value of maxIterations (line 10), the algorithm samples a new vertex according to the Sampling technique (line 12), extends the RRT employing a growth Strategy (lines 15 to 23), and tests if the goal region has been reached to connect the established plan (lines 26 to 33). Since sampling the goal POI precisely is unlikely (except for the biased goal sample itself) in a PlanningContinuum, the attribute goalThreshold shown in Figure 7.7 holds the maximum acceptable distance to consider a sample within a sphere around the goal. It is important to note that the final sample might be arbitrarily close to the actual goal and not permit a kinodynamically feasible solution to the goal in terms of Aircraft Capabilities.

More details about the SAFCS core RRTreePlanner[1] and related classes[2] can be found in our SAFCS software repository.

7.3 RRT*

It turns out that the RRT*, a probabilistically optimal variant of RRT, cannot be be trivially adapted and included into the SAFCS core. The algorithm assumes that greedily favoring local minima always leads to a global minimum, which is not the case in a 4D PlanningContinuum. However, the issues with respect to the RRTreeStarPlanner, shown as part of the SAFCS core RRT family in Figure 7.1, shall be briefly discussed.

[1] https://github.com/stephanheinemann/worldwind/tree/develop/src/main/java/com/cfar/swim/worldwind/planners/rrt/brrt

[2] https://github.com/stephanheinemann/worldwind/tree/develop/src/main/java/com/cfar/swim/worldwind/planners/rrt

RRT* achieves probabilistic optimality in a 3D environment using a relatively simple approach. It first connects new samples to the best vertex in terms of cost within a limited neighborhood of nearest neighbors of the established RRT. It then attempts to reach the other vertices within the same neighborhood from the new sample in order to potentially improve their costs. For each neighbor that receives improved costs in this manner, its entire sub-tree is updated propagating the costs accordingly. The second step is referred to as *re-wiring* the tree and assumes the improved costs at a local source vertex always lead to improved costs at its dependent vertices regardless of time.

However, as explained in Section 7.1, the ETOs of `RRTreeWaypoints` determine the `TimeInterval` in which an enclosed `Edge` is exposed to intersecting `CostIntervals`. Hence, propagating a local cost minima at a different ETO does not necessarily lead to a cost improvement. In fact, there is no guarantee for increased overall optimality using this approach in a 4D space-time `PlanningContinuum`. Even if propagation continues only if the entire subtree can be improved, it does not guarantee improved trajectories for consecutive samples at future ETOs. A subtree improvement may lead to a global regression based on different ETOs. Similarly, a subtree regression may lead to a global improvement based on different ETOs. It does not appear to be feasible to solve this dilemma without frequent and costly rollbacks.

It should be pointed out that other algorithms in the SAFCS core RRT family also apply local in addition to global heuristics when selecting or connecting to new samples. Growing an RRT in time may, therefore, always result in non-optimal global costs. However, none of the other algorithms claim to be optimal and the applied heuristics merely help to guide them to an ideally more successful outcome.

More details about the SAFCS core `RRTreeStarPlanner`[3] can be found in our SAFCS software repository.

7.4 Heuristic RRT

Uninformed variants of RRT are unlikely to find solutions in an `Obstacle`-saturated `PlanningContinuum` in a single shot, in particular, if these `Obstacles` exceed the operational `RiskPolicy`. There are two available approaches to mitigate this issue. The first approach is presented here and applies heuristics and an informed selection of new samples. The second approach performs multiple runs instead of just one and only re-

[3] https://github.com/stephanheinemann/worldwind/tree/develop/src/main/java/com/cfar/swim/worldwind/planners/rrt/rrtstar

tains solutions that satisfy a tightening solution quality; it is further discussed in Section 7.6.

The informed variant of RRT presented here is referred to as HRRT and was developed by Urmson and Simmons [US03]. As shown in Figure 7.10, our `HRRTreePlanner` aggregates a `HRRTreeAlgorithm` in its attribute `algorithm` as well as a `HRRTreeQuality` in its attribute `quality`.

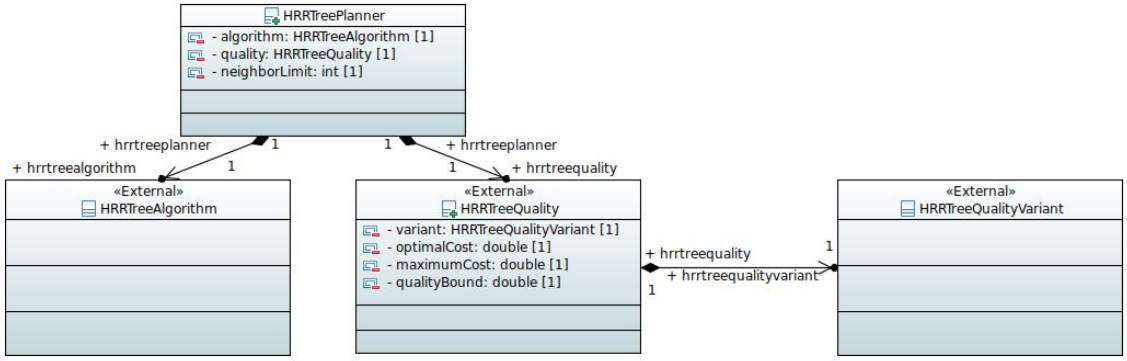


Figure 7.10: SAFCS Core: Heuristic RRT

The original literature defines the quality $q(s)$ of a vertex s in the grown RRT (adapted to our established notation) as shown in Equation 7.1.

$$q(s) = 1 - \frac{f(s) - h(s_{start})}{m(\mathcal{T}) - h(s_{start})} \tag{7.1}$$

, where $f(s) = g(s) + h(s) \wedge m(\mathcal{T}) = \max\{f(s) | s \in \mathcal{T}\}$

Recall that $f(s)$ is similarly used as the priority key in the A* family of algorithms and represents the sum of the established cost $g(s)$ from the start to the vertex s and the heuristic $h(s)$—usually a consistent heuristic, such as the normalized distance—from the vertex s to the goal. The maximum $f(s)$ of all vertices s in the current RRT $\mathcal{T}$ is referred to as $m(\mathcal{T})$ in the adapted Equation 7.1. Hence, the fraction expresses the relationship of the estimated total cost of a vertex s compared to the best possible cost $h(s_{start})$ given a current maximum estimated total cost of $m(\mathcal{T})$ in the RRT. For example, if $f(s) = h(s_{start})$, that is, the best possible solution for s has been found, then $q(s) = 1$. If on the other extreme $f(s) = m(\mathcal{T})$, that is, s represents the worst currently found solution, then $q(s) = 0$. Hence, the quality of a vertex in the current RRT may

range from 0 (lowest quality) to 1 (highest quality). Extending a higher quality vertex intuitively may lead to improved overall solutions but may also mislead the algorithm into local minima if exclusively applied. Therefore, the SAFCS core limits the maximum quality by a set `qualityBound` attribute (referred to as *probability floor* in the original literature) tuning parameter in order to avoid an overly pronounced exploitation and in favor of some level of exploration. We have, furthermore, performed experiments (Rosa, Heinemann, Suleman [RHS19]) with the modified quality function shown in Equation 7.2 to cope with discovered deficiencies of the original when dealing with abnormally high cost regions, and a modified application of the `qualityBound` to fine tune the ratio of exploration versus exploitation of the `PlanningContinuum`.

$$q(s) = 1 - \exp\left(-\frac{h(s_{start})}{f(s) - h(s_{start})}\right) \tag{7.2}$$

$$\text{, where } f(s) = g(s) + h(s) \wedge g(s) > 0$$

The original linear relationship unfortunately leads to the same quality assessment of different lower cost estimates if the current worst estimate is abnormally high. The improved quality function mitigates this issue by eliminating the current worst cost estimate and has been included as one of the available SAFCS `HRRTreeQualityVariants` shown in Figure 7.10. The exponent of Equation 7.2 intuitively represents the inverted cost overhead due to the accumulated detour from the heuristic optimum—it increases with increasing $f(s)$ approaching the upper limit of 0. Figure 7.11 shows how the quality of a vertex s decreases with increasing cost overhead $f(s) - h(s_{start})$.

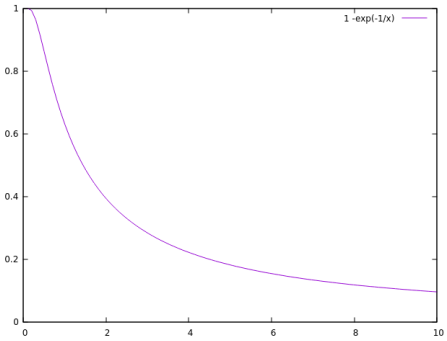


Figure 7.11: SAFCS Core: Heuristic RRT Quality

```
1  protected HRRTreeWaypoint selectWaypoint() {
2      HRRTreeWaypoint sample = null, nearest = null;
3
4      Random random = new Random();
5      double quality = 0d, threshold = 1d;
6
7      while (threshold > quality) {
8          sample = (HRRTreeWaypoint) super.sampleBiased();
9          nearest = (HRRTreeWaypoint) this.getPlanningContinuum()
10                 .findNearest(sample, 1).iterator().next();
11         quality = this.getQuality().get(nearest);
12         threshold = random.nextDouble();
13     }
14
15     this.setNearestWaypoint(nearest);
16     return sample;
17 }
```

Listing 7.5: SAFCS Core: HRRT hRRT Sample Selection

Regardless of the concrete applied quality assessment function, the `HRRTreePlanner` has three distinct selection `HRRTreeAlgorithms` at its disposal to determine whether or not the existing RRT can be extended toward a newly sampled vertex under a given quality constraint. These algorithms are referred to as *hRRT*, *IkRRT*, and *BkRRT*, respectively.

Listing 7.5 shows the most basic *hRRT* sample selection algorithm. It continues to take goal-biased samples from the `PlanningContinuum` (line 8) until the quality of the nearest vertex in the current RRT (lines 9 to 11) satisfies a random threshold (lines 7 and 12). The random threshold has a similar function as the quality bound (probability floor) discussed before and allows for lower quality neighbors to be included, although generally favoring higher quality samples.

Listing 7.6 shows the more sophisticated *IkRRT* sample selection algorithm. This algorithm establishes a quality priority order (line 3) of a limited number of neighbor vertices (lines 11 and 12) to each newly sampled vertex (line 9). It subsequently considers each neighbor in quality order to become the extension point for the new sample in the existing RRT (lines 14 to 21). Again, a random threshold (lines 16 and 17) is used to allow for lower-quality neighbors to be included but generally favors candidates possessing a higher quality. The process continues until the first neighbor vertex satisfying the random quality threshold has been found. The attribute `neighborLimit` shown in Figure 7.10, representing the maximum number of neighbors in the RRT to consider as extension point for new samples, is an important tuning parameter of the `HRRTreePlanner` applying *IkRRT* as it affects the structure of the growing RRT.

```
protected HRRTreeWaypoint selectWaypointIk() {
    HRRTreeWaypoint sample = null;
    PriorityQueue<HRRTreeWaypoint> neighbors = new PriorityQueue<HRRTreeWaypoint>
        (Comparator.comparingDouble(this.quality));

    Random random = new Random();

    while (true) {
        sample = (HRRTreeWaypoint) super.sampleBiased();
        neighbors.clear();
        neighbors.addAll((Set<HRRTreeWaypoint>) this.getPlanningContinuum()
                .findNearest(sample, this.getNeighborLimit()));

        for (HRRTreeWaypoint neighbor : neighbors) {
            double quality = this.getQuality().get(neighbor);
            double threshold = random.nextDouble();
            if (threshold < quality) {
                this.setNearestWaypoint(neighbor);
                return sample;
            }
        }
    }
}
```

Listing 7.6: SAFCS Core: HRRT IkRRT Sample Selection

```
protected HRRTreeWaypoint selectWaypointBk() {
    HRRTreeWaypoint sample = null, nearest = null;
    PriorityQueue<HRRTreeWaypoint> neighbors = new PriorityQueue<HRRTreeWaypoint>
        (Comparator.comparingDouble(this.quality));

    Random random = new Random();
    double quality = 0d, threshold = 1d;

    while (threshold > quality) {
        sample = (HRRTreeWaypoint) super.sampleBiased();
        neighbors.clear();
        neighbors.addAll((Set<HRRTreeWaypoint>) this.getPlanningContinuum()
                .findNearest(sample, this.getNeighborLimit()));
        nearest = neighbors.poll();
        quality = this.getQuality().get(nearest);
        threshold = random.nextDouble();
    }

    this.setNearestWaypoint(nearest);
    return sample;
}
```

Listing 7.7: SAFCS Core: HRRT BkRRT Sample Selection

The third sample selection algorithm *BkRRT* is shown in Listing 7.7. Like *IkRRT*, it establishes a quality priority order (lines 3 and 4) on a maximum number of neighbors being potential candidates for an extension (lines 12 and 13) toward a new sample (line 10). However, unlike *IkRRT*, the *BkRRT* selection algorithm only considers the *best* possible neighbor (lines 14 and 15) to satisfy a random quality threshold (lines 9 and 16). If the random quality threshold cannot be satisfied, then an entirely new sample and neighborhood is assessed in the same manner. Again, the maximum number of neighbors to consider as extension candidates is an important tuning parameter for the `HRRTreePlanner` applying *BkRRT*.

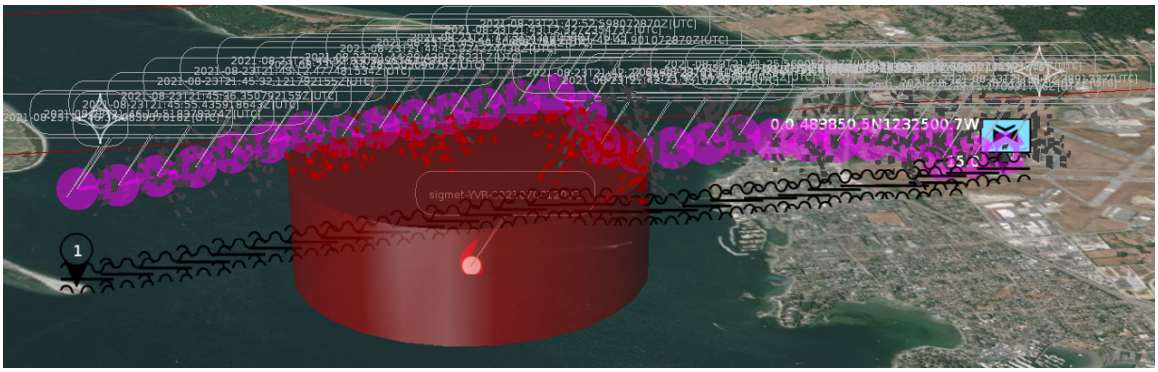


Figure 7.12: SAFCS UI: Heuristic RRT

Figure 7.12 shows the result of a single-shot HRRT and demonstrates how the `HRRTreePlanner` may find a solution while adhering to an operational `RiskPolicy` that prevents the `Aircraft` from penetrating a risky `Obstacle`. Notice, in particular, how certain costly branches (dashed red line segments) reaching lower-quality expansion candidates (neighbors) of the RRT have been abandoned in favor of higher-quality ones.

The settings to compute this particular `Trajectory` are shown in Figure 7.13 and apply an *IkRRT* sample selection algorithm with a `neighborLimit` of 20 and a `qualityBound` of 0.5. Furthermore, the enhanced quality assessment function of Equation 7.2 is used in this example.

More details about the SAFCS core `HRRTreePlanner`[4] and related classes can be found in our SAFCS software repository.

[4] https://github.com/stephanheinemann/worldwind/tree/develop/src/main/java/com/cfar/swim/worldwind/planners/rrt/hrrt

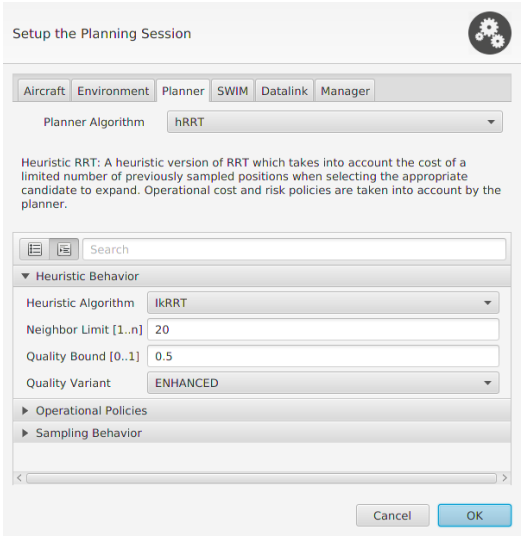


Figure 7.13: SAFCS UI: Heuristic RRT Planner Setup

7.5 Dynamic RRT

The SAFCS core supports a dynamic variant of the `HRRTreePlanner` presented in Section 7.4. As can be seen from Figure 7.1, a `DRRTreePlanner` is derived from the `HRRTreePlanner` operating on its own dedicated class of `DRRTreeWaypoints`. It should be noted that both the `HRRTreePlanner` and `DRRTreePlanner` are not part of the hierarchy leading to the managed key algorithm `OADRRTreePlanner`. However, since it applies the same approach to an informed search as the more general `HRRTreePlanner`, it could be extended with an online variant that produces dynamic informed one-shot results without relying on multiple shots with a tightening quality, such as realized by the `ARRTreePlanner` discussed in Section 7.6. This extension option is revisited in Chapter 15.

The implementation of the `DRRTreePlanner` is fairly similar to the `ADStarPlanner` presented in Section 6.4 in terms of the required elaboration step and explicit termination. As can be seen in Figure 7.1, the `DRRTreePlanner` realizes a `LifelongPlanner` that continues to wait for any dynamic changes and revises its computed `Trajectory` accordingly until explicitly terminated by an operator. However, unlike the `ADStarPlanner`, the `DRRTreePlanner` does not apply any concept of `Waypoint` consistency and associated under- or over-consistency propagation. A `DRRTreePlanner` instead simply trims invalid parts of its RRT.

As a realization of the `DynamicPlanner`, the `DRRTreePlanner` is also a `DynamicObstacleListener` in line with the `Obstacle` reporting framework intro-

duced in Section 5.2.3 and shown in Figure 5.16. Whenever Obstacles are added to or removed from its PlanningContinuum, its ObstacleManager (being the facade of its Scenario) notifies the DRRTreePlanner to initiate an elaboration step.

The elaboration and associated repair of a DRRTreePlanner are shown in Listing 7.8. As mentioned, the elaboration repair is similar to the one shown in Listing 6.11 for the ADStarPlanner and proceeds recursively beginning with the first part of an established multi-part plan (lines 7 to 16). Again, three distinct cases have to be considered. In the first case (lines 17 to 22), no solution could be found for the immediately preceding part. Consequently, there cannot be a solution for the current or any subsequent parts. This situation could arise due to a challenging Environment saturated with many Obstacles, possibly exceeding the DRRTreePlanner's operational RiskPolicy. It could also be the result of an unfortunate combination of settings for the tuning parameters of the DRRTreePlanner. Lastly, since probabilities are always involved in the growth of RRTs, there cannot be any guarantee with respect to the quality of sampled vertices and the resulting Trajectory. In the second case (lines 23 to 45), the previous part was either not affected by the repair or, following a repair, its goal ETO remained unchanged. Intuitively, meeting the goal ETO precisely, following the dynamic repair (trimming and re-growing) of an RRT, is highly unlikely. Due to the probabilistic nature of the algorithm and the changing distances between samples, meeting the goal ETO is even less likely than within a regular PlanningGrid featuring equal distances between Waypoints. Hence, this case often applies to single-part plans or the first part in a multi-part plan being affected by dynamic Obstacles. The latter usually leads to a planning from scratch cascading to all subsequent parts, which is shown in lines (46 to 53). In summary, the requirement in terms of ETO-consistency for connecting POIs in the repair of multi-part plans is the same for all DynamicPlanners in the SAFCS core and shown in Figure 6.10.

In the most desirable cases, which are single-part repairs or late part repairs of multi-part plans, the existing RRTs are trimmed by removing invalid branches, as shown in Listing 7.9 and invoked in line 37 of Listing 7.8. Trimming an RRT in the SAFCS core means identifying intersections between the Edges and dynamically added or removed Obstacles in both space (line 26) and time (lines 27 to 30). Each intersecting Edge (branch) of the RRT needs to be trimmed, leading to the removal of all its children in the tree (lines 32 and 1 to 21). All removed children vertices, that is, DRRTreeWaypoints, are, furthermore, tagged as invalid (line 8) using their attribute isValid shown in Figure 7.6. The tagging allows the DRRTreePlanner to validate the currently established plan in

```
protected boolean repair(int partIndex) {
    boolean repaired = true;

    if (this.needsRepair()) {
        DRRTreeWaypoint partStart = (DRRTreeWaypoint) this.getStart().clone();
        // repair previous parts before current part
        if (this.hasDynamicObstacles(partIndex - 1)) {
            this.backup(partIndex);
            this.restore(partIndex -1);
            this.planPart(partIndex - 1);
            partStart.setCost(this.getGoal().getCost());
            partStart.setEto(this.getGoal().getEto());
            partStart.setParent(this.getGoal().getParent());
            this.backup(partIndex - 1);
            this.restore(partIndex);
        }
        if (partStart.hasInfiniteCost()) {
            // no previous part without exceeded risk policy
            this.clearDynamicObstacles();
            this.clearWaypoints();
            this.getGoal().setInfiniteCost();
            repaired = false;
        } else if (this.getStart().getEto().equals(partStart.getEto())) {
            // connect current to previous part
            if (partStart.hasParent()) {
                this.getStart().setParent(partStart.getParent());
                // propagate potential cost change
                double deltaCost = partStart.getCost() - this.getStart().getCost();
                if (0d != deltaCost) {
                    for (Position vertex : this.getPlanningContinuum().getVertices())
                    {   DRRTreeWaypoint waypoint = (DRRTreeWaypoint) vertex;
                        waypoint.setCost(waypoint.getCost() + deltaCost);
                    }
                }
            }
            // trim and re-grow tree if plan is affected
            this.trim();
            this.clearDynamicObstacles();
            if (!this.hasValidPlan()) {
                this.getGoal().setInfiniteCost();
                if (!this.compute() || this.getGoal().hasInfiniteCost()) {
                    // no trajectory or exceeded risk policy
                    this.clearWaypoints();
                }
            }
        } else {
            // plan current part from scratch if start ETO has changed
            this.initialize(this.getStart(), this.getGoal(), partStart.getEto());
            if (partStart.hasParent()) {
                this.getStart().setParent(partStart.getParent());
                this.getStart().setCost(partStart.getCost());
            }
            super.planPart(partIndex);
        }
    }
    return repaired;
}

protected void elaborate(int partIndex) {
    // proceed to next part only if not in need of repair
    while (this.needsRepair() && !this.hasTerminated()) {
        if (!this.repair(partIndex)) break;
    }
    // backup after elaboration
    this.backup(partIndex);
}
```

Listing 7.8: SAFCS Core: DRRT Elaboration

```
protected void trim(DRRTreeWaypoint root) {
    if (this.getStart().equals(root)) {
        // purge the entire tree
        this.clearExpendables();
        this.getStart().clearChildren();
        this.getPlanningContinuum().addVertex(this.getStart());
    } else {
        root.setValid(false);

        // trim all sub-branches of the invalid vertex
        if (root.hasChildren()) {
            for (RRTreeWaypoint child : root.getChidren()) {
                this.trim((DRRTreeWaypoint) child);
            }
            root.clearChildren();
        }
        // remove the invalid vertex
        root.removeParent();
        this.getPlanningContinuum().removeVertex(root);
    }
}

protected void trim() {
    // trim all affected tree branches
    for (Obstacle obstacle : this.getDynamicObstacles()) {
        this.getPlanningContinuum().findAffectedEdges(obstacle).stream()
        .filter(edge -> obstacle.getCostInterval()
                .intersects(new TimeInterval(
                    ((DRRTreeWaypoint) edge.getFirstPosition()).getEto(),
                    ((DRRTreeWaypoint) edge.getSecondPosition()).getEto())))
        .map(edge -> (DRRTreeWaypoint) edge.getSecondPosition())
        .forEachOrdered(waypoint -> this.trim(waypoint));
    }

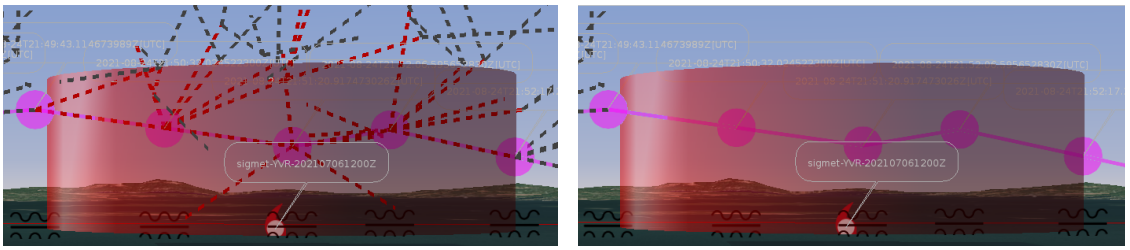
    // handle a significant change
    if (this.hasSignificantChange()) {
        this.handleSignificantChange();
    }

    this.updateQuality();
    this.setNewestWaypoint(this.getStart());
}
```

Listing 7.9: SAFCS Core: DRRT Trimming

line 39 of Listing 7.8 and to determine if re-growing the RRT needs to be performed (line 41). As mentioned before, there is, however, no guarantee that a repair will succeed.

Note that the presented approach only identifies the affected parts of the current RRT and only initiates a re-grow if the established plan becomes invalid. Thus, if `Obstacles` are added or removed from parts of the established plan, then an elaboration repair ideally results in a revised `Trajectory`. However, if only areas adjacent to the established plan are affected, then the `DRRTreePlanner` does not revise the established plan. Thus, if `Obstacles` are being removed from within the `PlanningContinuum`, the potential for an improved `Trajectory` is not necessarily examined and would theoretically require to plan from scratch. This deficiency is, however, not a factor for the `ADRRTreePlanner` performing multiple shots and one of its advantages, as described in Section 7.7.



(a) SAFCS UI: DRRT Before Trimming **(b)** SAFCS UI: DRRT After Trimming

Figure 7.14: SAFCS UI: DRRT Trimming

Figure 7.14 shows how our SAFCS UI displays the DRRT trimming step. In this example, the planned `Aircraft` follows a `Trajectory` from left to right. With the injection of a dynamic `Obstacle` all intersecting `Edges` become invalid as their `IntervalTrees` of `CostIntervals` have to be modified and the resulting costs change. Consequently, all dependent `Edges` and their invalid `DRRTreeWaypoints` have to be removed from the `PlanningContinuum`. The trimmed RRT is subsequently re-grown from its remaining `Edges` and valid `DRRTreeWaypoints` on the left.

Recall that the possibility of significant changes to the RRT, that generally do not benefit from trimming and re-growing the RRT compared to purging it and planning from scratch, exists for all `DynamicPlanners`, as discussed for AD* in Section 6.4. Figure 5.16 exposes the functional signature of a `DynamicPlanner` for that purpose. Hence, if a significant change to the RRT occurs, it can be handled accordingly (lines 36 to 38). As discussed in Section 6.4, there are different quantitative and qualitative indicators as well as strategies to handle significant changes. Currently, the indicator is purely quan-

```
protected void updateQuality(DRRTreeWaypoint root) {
    if (root.hasChildren()) {
        for (RRTreeWaypoint child : root.getChidren()) {
            this.updateQuality((DRRTreeWaypoint) child);
        }
    } else {
        this.getQuality().setMaximumCost(Math.max(
            this.getQuality().getMaximumCost(),
            root.getF()));
    }
}

protected void updateQuality() {
    this.getQuality().setMaximumCost(this.getStart().getF());
    this.updateQuality(this.getStart());
}
```

Listing 7.10: SAFCS Core: DRRT Quality Update

titative and computes the overall ratio of invalid `DRRTreeWaypoints` in the established plan. If this ratio exceeds a tunable threshold, the RRT is purged. This approach corresponds and is equivalent to the current approach for AD* in Listing 6.13. Clearly, a qualitative approach considering the location of invalid `DRRTreeWaypoints` in the computed `Trajectory` is another option revisited in Chapter 15.

Since the `DRRTreePlanner` is derived from the `HRRTreePlanner`, as shown in Figure 7.1, and employs the same informed search and quality assessment options discussed in Section 7.4, the quality of the trimmed RRT needs to be updated with respect to the current maximum total estimated cost $m(\mathcal{T})$ in the tree before re-growing it. Recall the original quality assessment function shown in Equation 7.1 for this discussion. Therefore, the method `updateQuality` is invoked on line 40 of Listing 7.9 and detailed in Listing 7.10. The quality update proceeds from the root of the trimmed RRT (line 14) to its leafs (lines 15 and 2 to 5) and re-computes the maximum total estimated cost $m(\mathcal{T})$ (line 7 to 9). After having trimmed the previously invalid parts of the RRT, and updated its quality accordingly, the `DRRTreePlanner` can proceed with re-growing it toward the goal POI (line 41 of Listing 7.8).

More details about the SAFCS core `DRRTreePlanner`[5] can be found in our SAFCS software repository.

[5] https://github.com/stephanheinemann/worldwind/tree/develop/src/main/java/com/cfar/swim/worldwind/planners/rrt/drrt

7.6 Anytime RRT

The SAFCS core ARRTreePlanner realizes an important step toward the managed RRT key algorithm OADRRT. Being an AnytimePlanner, as shown in Figure 7.1, it can produce improving solutions featuring a bounded decreasing sub-optimality. Sub-optimal solutions are generally available within a shorter deliberation time and, hence, this extension is suitable to deal with changing levels of Criticality within the PlanningContinuum. The ARRTreePlanner is able to revise a sub-optimal Trajectory and deliver timely solutions as long as the available deliberation time allows. The ideas behind ARRT are studied in detail in Chapter 2. This section, therefore, primarily emphasizes implementation details and, in particular, necessary modifications with respect to the SAFCS operational RiskPolicy. It, furthermore, highlights the differences in the applied heuristics compared to the HRRTreePlanner.

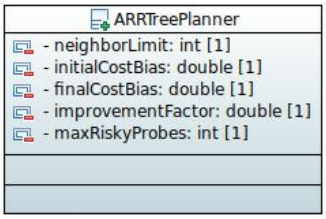


Figure 7.15: SAFCS Core: Anytime RRT

Similar to the ARAStarPlanner discussed in Section 6.3, the ARRTreePlanner features an elaboration phase that incrementally tightens the sub-optimality bound to find improved Trajectory solutions that adhere to an operationally defined RiskPolicy. Recall the different RiskPolicy thresholds and their application during the cost estimation within an Environment, as studied in Section 5.2.5. Listing 7.11 shows the elaboration of an ARRTreePlanner.

Elaboration generally continues and, hence, a revised Trajectory is computed until a maximum tuned quality has been achieved (lines 27 and 28). However, the introduction of an operational RiskPolicy or, more generally, the possibility of an infeasible Trajectory needs to be incorporated. The unextended ARRTreePlanner is not a DyanmicPlanner and, therefore, the Obstacles within its PlanningContinuum can be considered static. Consequently, the elaboration has to conclude at some point that no feasible solution can be found. For example, if the planned Aircraft was located on an aerodrome affected by adverse weather or having an icy closed runway, then obviously any initial extension of the RRT could violate a certain operational RiskPolicy

```
protected void improve(int partIndex) {
    if (!this.hasMaximumQuality()) {
        this.backup(partIndex);
        this.getPlanningContinuum().clearVertices();
        this.getPlanningContinuum().addVertex(this.getStart());
        this.setNewestWaypoint(this.getStart());
        this.getStart().clearChildren();
        this.getGoal().removeParent();

        if (this.compute() && !this.getGoal().hasInfiniteCost()) {
            // improved trajectory without exceeding risk policy
            this.revisePlan(this.createTrajectory());
            this.updateCostBound();
        } else {
            // no trajectory or exceeded risk policy
            this.restore(partIndex);
        }

        this.updateBiases();
    }
}

protected void elaborate(int partIndex) {
    // do not elaborate an exceeded risk policy solution beyond limit
    int riskyProbes = (this.getGoal().hasInfiniteCost()) ? 1 : 0;

    while (!this.hasMaximumQuality() && (this.getMaxRiskyProbes() > riskyProbes)) {
        this.improve(partIndex);
        if (this.getGoal().hasInfiniteCost()) {
            riskyProbes++;
        }
    }
}
```

Listing 7.11: SAFCS Core: ARRT Elaboration

and never reach the defined maximum quality. Hence, a maximum number of risky probes (lines 25, 27, and 29 to 31) needs to limit the number of failed attempts. Figure 7.15 shows some of the important tunable attributes of the `ARRTreePlanner` with `maxRiskyProbes` being one of them.

The `ARRTreePlanner` performs multiple shots growing an RRT subject to an incrementally tightening sub-optimality bound (lines 1 to 21). Each shot commences at the root (lines 3 to 8), that is, the start POI, and only considers the achieved quality of previous solutions (line 13). As studied in Chapter 2, a weighted cost selector and a composite heuristic shown in Equations 7.3 and 7.4, respectively, together guide the search toward improved solutions.

$$c(s,t) = b_d\,h(s,t) + b_c\,h(s,s_{goal}) \qquad (7.3)$$

$$\text{with } 1 = b_d + b_c$$

Recall that the weighted cost selector is applied to selectively connect a suitable source s (neighbor) in the RRT to a newly sampled target t. The lowest weighted cost $c(s,t)$ combination of s and t is more likely to be a good candidate for a new `Edge` leading to an improved `Trajectory`. The two weights of the cost selector are a distance bias b_d and a cost bias b_c with $1 = b_d + b_c$. Usually, the `ARRTreePlanner` favors shorter distances to the existing vertices (neighbors) in the RRT initially and subsequently shifts the weight toward less costly `Edges`. This approach is very similar to the inflation factor ϵ discussed for ARA* in Section 6.3. The initial and final cost bias are represented by the corresponding `ARRTreePlanner` attributes `initialCostBias` and `finalCostBias`, respectively, and shown in Figure 7.15. Furthermore, the attribute `neighborLimit` tunes the maximum number of neighbor vertices in the RRT to consider for the cost selector application given a newly taken sample.

$$p(s) = h(s_{start},s) + h(s,s_{goal}) \qquad (7.4)$$

```
1  protected double computeHeuristic(
2          ARRTreeWaypoint source, ARRTreeWaypoint target) {
3      return this.getEnvironment().getNormalizedDistance(source, target);
4  }
5
6  protected double computeCompositeHeuristic(ARRTreeWaypoint waypoint) {
7      return this.computeHeuristic(this.getStart(), waypoint)
8      + this.computeHeuristic(waypoint, this.getGoal());
9  }
10
11 protected double computeSelectionCost(
12     ARRTreeWaypoint source, ARRTreeWaypoint target) {
13     return this.getDistanceBias() * this.computeHeuristic(source, target)
14             + this.getCostBias() * source.getCost();
15 }
16
17 protected class CostSelector implements ToDoubleFunction<ARRTreeWaypoint> {
18
19     ARRTreeWaypoint target;
20
21     public CostSelector(ARRTreeWaypoint target) {
22         this.target = target;
23     }
24
25     public double applyAsDouble(ARRTreeWaypoint source) {
26         return computeSelectionCost(source, this.target);
27     }
28
29 }
```

Listing 7.12: SAFCS Core: ARRT Heuristics

The composite heuristic of Equation 7.4 represents the best possible cost of a computed `Trajectory` including the `ARRTreeWaypoint` s. It serves to assess if a new sample s can improve the currently achieved quality of a solution. Each time an improved solution is found, the sub-optimality cost bound is tightened (line 13 of Listing 7.11). New samples with a composite heuristic value that exceed the currently established cost bound can, therefore, be readily discarded.

Listing 7.12 shows the methods related to the applied `ARRTreePlanner` heuristics and cost selection. The `CostSelector` (lines 17 to 29) can readily be used as the basis for a weighted cost `Comparator` in a `PriorityQueue` of suitable neighbors during the extension of the RRT.

Shifting the cost bias and tightening the cost bound after consecutive attempts to improve a solution is shown in Listing 7.13. The updated cost bound is simply derived from the quality of the currently best `Trajectory` subtracted by a tunable quality improvement (lines 24 and 25). Recall that the minimum and maximum quality as well as the desired quality improvement are common tunable properties of all `AnytimePlanners`. In case of the `ARRTreePlanner`, the minimum and maximum quality correspond to the

```
protected void updateBiases() {
    // delta cost bias
    double dq = this.getMaximumQuality() - this.getMinimumQuality();
    // optimal cost fraction
    double fc = this.getStart().getH() / this.getCostBound();

    // adjust cost bias to current cost bound
    double minCostBias = fc * dq;
    if (this.getCostBias() < minCostBias) {
        this.setCostBias(minCostBias);
    } else {
        double costBias = this.getCostBias() + this.getStep();
        costBias = (costBias > this.getMaximumQuality()) ?
            this.getMaximumQuality() : costBias;
        costBias = (costBias < this.getMinimumQuality()) ?
            this.getMinimumQuality() : costBias;
        this.setCostBias(costBias);
    }

    this.setDistanceBias(1d - this.getCostBias());
}

protected void updateCostBound() {
    double costBound = (1d - this.getQualityImprovement())
            * this.getGoal().getCost();

    if (this.getStart().getF() > costBound) {
        this.setCostBound(this.getStart().getF());
    } else {
        this.setCostBound(costBound);
    }
}
```

Listing 7.13: SAFCS Core: ARRT Cost Bias and Bound

```
protected Optional<ARRTreeWaypoint> sampleTarget() {
    Optional<ARRTreeWaypoint> target = Optional.empty();

    ARRTreeWaypoint sample = (ARRTreeWaypoint) this.sampleBiased();
    sample.setH(this.computeHeuristic(sample, this.getGoal()));
    sample.setP(this.computeCompositeHeuristic(sample));

    if (sample.equals(this.getGoal())) {
        target = Optional.of(sample);
    } else {
        int sampleAttempt = 0;
        // TODO: increase sampling attempts with lower cost bound
        while ((this.getCostBound() < sample.getP())
                && (this.getMaxSamplingAttempts() > sampleAttempt)) {
            // TODO: sampling space could be further constrained
            // according to cost bound
            sample = (ARRTreeWaypoint) this.sample();
            sample.setH(this.computeHeuristic(sample, this.getGoal()));
            sample.setP(this.computeCompositeHeuristic(sample));
            sampleAttempt++;
        }
        if (this.getCostBound() >= sample.getP()) {
            target = Optional.of(sample);
        }
    }

    return target;
}
```

Listing 7.14: SAFCS Core: ARRT Sampling

inital and final cost bias, respectively. The cost bound itself is clearly limited by the best possible cost—a direct connection between the start and goal POIs without any additional penalties (lines 27 to 31). It is only updated after having computed an improved `Trajectory` (line 13 of Listing 7.11). The cost bias, on the other hand, increases with each new planning attempt (line 19 of Listing 7.11) with the intention to improve the solution cost and, therewith, tighten the cost bound even more. Gradually shifting from distance to cost bias can be done without any consideration of the current cost bound. However, the SAFCS core `ARRTreePlanner` implements a somewhat tighter coupling shown in Listing 7.13. It performs a more pronounced cost bias shift if a certain cost bound has already been reached (lines 3 to 10) and otherwise increases the cost bias using a constant step (lines 12 to 17).

The sampling and extension steps shown in Listings 7.14 and 7.15, respectively, make use of the previously explained cost bound and weighted `CostSelector`. After taking a goal-biased sample according to one of the available sampling approaches presented in Section 7.2 (line 4) and computing its associated heuristics (lines 5 and 6), the sample's composite heuristic needs to satisfy the currently established cost bound (line 13) in order to be considered for an extension if it is not already the goal itself (line

```
protected Status extendToTarget(ARRTreeWaypoint target) {
    Status status = Status.TRAPPED;

    // establish priority order according to selection cost
    PriorityQueue<ARRTreeWaypoint> neighbors = new PriorityQueue<ARRTreeWaypoint>(
            Comparator.comparingDouble(new CostSelector(target)));
    // TODO: other nearest metrics (capabilities)
    neighbors.addAll((Set<ARRTreeWaypoint>) this.getPlanningContinuum()
            .findNearest(target, this.getNeighborLimit()));

    for (ARRTreeWaypoint neighbor : neighbors) {
        // TODO: single versus several extensions toward lower cost bound
        Optional<Edge> extension = this.createExtension(neighbor, target);

        if (extension.isPresent()) {
            // ensure all extensions satisfy the current cost bound
            ARRTreeWaypoint newest = this.getNewestWaypoint();
            if (newest.getF() <= this.getCostBound()) {
                if (newest.equals(target)) {
                    return Status.REACHED;
                } else {
                    return Status.ADVANCED;
                }
            } else {
                // remove too expensive vertex and associated edge
                if (newest.hasParent()) {
                    newest.getParent().removeChild(newest);
                    newest.removeParent();
                }
                this.getPlanningContinuum().removeVertex(newest);
                this.setNewestWaypoint(this.getStart());
            }
        }
    }

    return status;
}
```

Listing 7.15: SAFCS Core: ARRT Extension

8). The non-goal-biased sampling continues (lines 13 to 21) until a new sample satisfies the cost bound or a maximum number of sampling attempts have been reached. There are two possible improvement options to this sampling step that are revisited in Chapter 15. First, finding a sample that satisfies a tighter cost bound is less likely for the same number of sampling attempts. Second, sampling from a region within the `PlanningContinuum` that only contains samples featuring a composite heuristic that does not satisfy the current cost bound, is clearly not productive. Hence, the sampling space and number of sampling attempts could be more tightly coupled to the current cost bound.

Once a suitable new sample has been found, the RRT can be extended toward it from the vertex within the sample's neighborhood posessing the lowest weighted cost. The weighted `CostSelector` shown in Listing 7.12 can readily be applied to establish a pri-

ority order on the neighborhood (lines 5 and 6) in the extension step of Listing 7.15. The nearest neighbors in the RRT are, consequently, ordered according to the weighted `CostSelector` for the sampled target (lines 8 and 9). Currently, the SAFCS core refers to nearest in terms of distance only. It offers, however, a different method considering a metric that could incorporate `Aircraft Capabilities` to obtain a more performance-based concept of nearest neighbor, as discussed in Chapter 15. Once an extension toward the new sample has been created (line 13), it also has to satisfy the currently established cost bound (line 18) in order to be included in the RRT (lines 19 to 23). A violation of the cost bound constraint leads to the removal of the extension (lines 25 to 31) and proceeding with the next suitable neighbor according to the priority order.

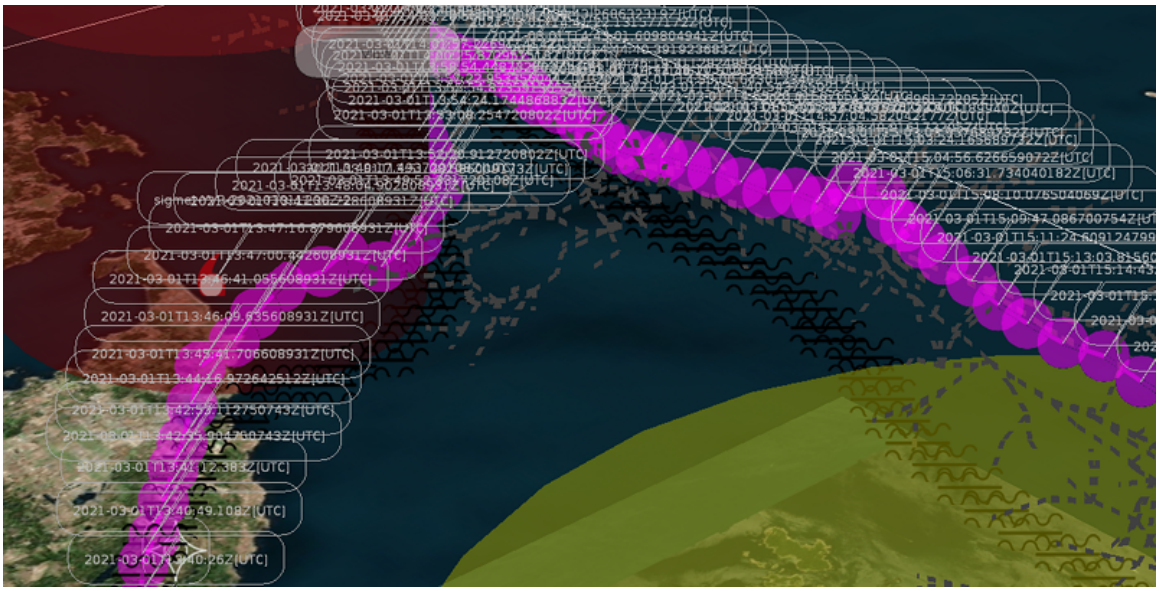


Figure 7.16: SAFCS UI: Anytime RRT

Figure 7.16 shows the final result of an `ARRTreePlanner` after performing multiple planning shots with increasing quality being displayed in our SAFCS UI. Notice how the computed `Trajectory` circumvents the moving `Obstacles` precisely without adding significant overhead in terms of distance. Recall that the yellow `Obstacle` moves toward the red one, as stated in previous examples, which introduces a planned climb at a position that will be affected by it at a future model time when the planned `Aircraft` transits the area.

More details about the SAFCS core ARRTreePlanner[6] can be found in our SAFCS software repository.

7.7 Anytime Dynamic RRT

The ADRRTreePlanner is a more sophisticated SAFCS core extension of the ARRTreePlanner discussed in Section 7.6 and shown in Figure 7.1. It applies the same heuristics as the ARRTreePlanner to guide the search in order to find and improve a Trajectory. It, furthermore, integrates the dynamic repair of the DRRTreePlanner into its elaboration step and can, therefore, adapt its solution to a partially known PlanningContinuum featuring dynamic Obstacles. Realizing the AnytimePlanner, DynamicPlanner, and LifelongPlanner interfaces, it reacts to any changes within its PlanningContinuum that are injected via the Obstacle reporting framework shown in Figure 5.16, and continues to run until explicitly terminated. Note that each dynamic repair follows the same multi-shot approach with a tightening cost bound, as discussed in Section 7.6 for the ARRTreePlanner.

Listing 7.16 shows the extended elaboration step of the ADRRTreePlanner. Notice the similarities between this and the elaboration steps of the DRRTreePlanner and ARRTreePlanner shown in Listings 7.8 and 7.11, respectively. It simply combines the dynamic repair with the incremental improvement of a solution. However, note that since multiple repair and improvement attempts are made that could lead to viable solutions being interleaved with risky probes, the number of risky probes has to be reset (line 13) once a feasible solution not featuring an infinite cost (line 10) has been found. The improvement invocation (line 9) is exactly the same as for the ARRTreePlanner and dispatches the corresponding method shown in Listing 7.11. However, the dynamic repair invocation dispatches a modified repair method with the replacing snippet shown in Listing 7.17.

Recall the three cases for dynamic multi-part repairs discussed in Section 7.5 and shown in Listing 7.8. Since the ADRRTreePlanner applies a different heuristic than the DRRTreePlanner based on a cost bias and bound, the second repairing case has to be modified. To that extent, the cost bound always has to reflect the situation following an injected change into the PlanningContinuum. After trimming any affected Edges (line 18), the cost is bound by the current solution if valid (line 21) or otherwise unbounded

[6] https://github.com/stephanheinemann/worldwind/tree/develop/src/main/java/com/cfar/swim/worldwind/planners/rrt/arrt

```
protected void elaborate(int partIndex) {
    // do not elaborate an exceeded risk policy solution beyond limit
    int riskyProbes = (this.getGoal().hasInfiniteCost()) ? 1 : 0;

    // proceed to next part only if fully improved and not in need of repair
    while (((!this.hasMaximumQuality() && (this.getMaxRiskyProbes() > riskyProbes))
            || this.needsRepair()) && !this.hasTerminated()) {
        if (!this.repair(partIndex)) break;
        this.improve(partIndex);
        if (this.getGoal().hasInfiniteCost()) {
            riskyProbes++;
        } else {
            riskyProbes = 0;
        }
        this.updateDynamicObstacles();
    }
    // backup after elaboration
    this.backup(partIndex);
}
```

Listing 7.16: SAFCS Core: ADRRT Elaboration

```
// ...
} else if (this.getStart().getEto().equals(partStart.getEto())) {
    // propagate potential cost change
    double deltaCost = partStart.getCost() - this.getStart().getCost();
    if (0d != deltaCost) {
        for (Position vertex : this.getPlanningContinuum().getVertices()) {
            ADRRTreeWaypoint waypoint = (ADRRTreeWaypoint) vertex;
            waypoint.setCost(waypoint.getCost() + deltaCost);
        }
    }

    // connect current to previous part
    if (partStart.hasParent()) {
        this.getStart().setParent(partStart.getParent());
    }

    // trim and re-grow tree if plan is affected
    this.trim();
    this.clearDynamicObstacles();
    // force improvement in case of dynamic removals
    this.setCostBound(this.getGoal().getCost());
    if (!this.hasValidPlan()) {
        // an invalid plan has an infinite cost bound
        this.getGoal().setInfiniteCost();
        this.setCostBound(this.getGoal().getCost());
        if (this.compute() && !this.getGoal().hasInfiniteCost()) {
            // trajectory without exceeding risk policy
            this.revisePlan(this.createTrajectory());
            this.updateCostBound();
        } else {
            // no trajectory or exceeded risk policy
            this.clearWaypoints();
        }
    }
} else {
// ...
```

Listing 7.17: SAFCS Core: ADRRT Repair

```
1 protected void handleSignificantChange() {
2     // TODO: purge entire tree or adjust distance and cost biases
3     // this.trim(this.getStart());
4     this.setDistanceBias(1d);
5     this.setCostBias(0d);
6 }
```

Listing 7.18: SAFCS Core: ADRRT Significant Change

(line 24 and 25). An invalid solution is equivalent to a non-existing solution and, hence, there must be no cost bound in this case. It should be emphasized how the combination of a multi-shot `AnytimePlanner` with a `DynamicPlanner` is now able to consider possible improvements, not only if the computed `Trajectory` is directly affected by dynamic changes but also in cases where adjacent regions in the `PlanningContinuum` have become less costly. Since an improvement always follows a dynamic repair (lines 8 and 9 in Listing 7.16) and the cost bound reflects the current situation after the injection of dynamic changes, consecutive shots may discover an improved solution. This is clearly an advantage over the dynamic behavior of the single-shot `DRRTreePlanner` and makes the `ADRRTreePlanner` the better choice for the online extension discussed in Section 7.8.

The different `DynamicPlanner` strategies for the determination and handling of significant changes in `DynamicEnvironments` have already been studied for CGS and RRTs in Sections 6.4 and 7.5, respectively. Listing 7.18 shows two possible handling strategies implemented by the `ADRRTreePlanner`. One option would be to simply purge the entire RRT and start planning from scratch (line 3). The other option only adjusts the distance and cost bias (lines 4 and 5), as if planning from scratch, but retains the trimmed RRT. Possible options for the determination and handling of significant changes are revisited in Chapter 15.

More details about the SAFCS core `ADRRTreePlanner`[7] can be found in our SAFCS software repository.

7.8 Online ADRRT

The `OADRRTreePlanner` is the most sophisticated extension in the SAFCS RRT family and plays a similar role as the `OADStarPlanner` in the SAFCS CGS family, representing one of the key algorithms that are managed by the SAFCS AM. Realizing an

[7] https://github.com/stephanheinemann/worldwind/tree/develop/src/main/java/com/cfar/swim/worldwind/planners/rrt/adrrt

OnlinePlanner, the OADRRTreePlanner has access to a Datalink source and associated Communication sequences, as shown in Figure 7.17, in order to monitor and affect the progress of a planned Aircraft presently executing a live mission.

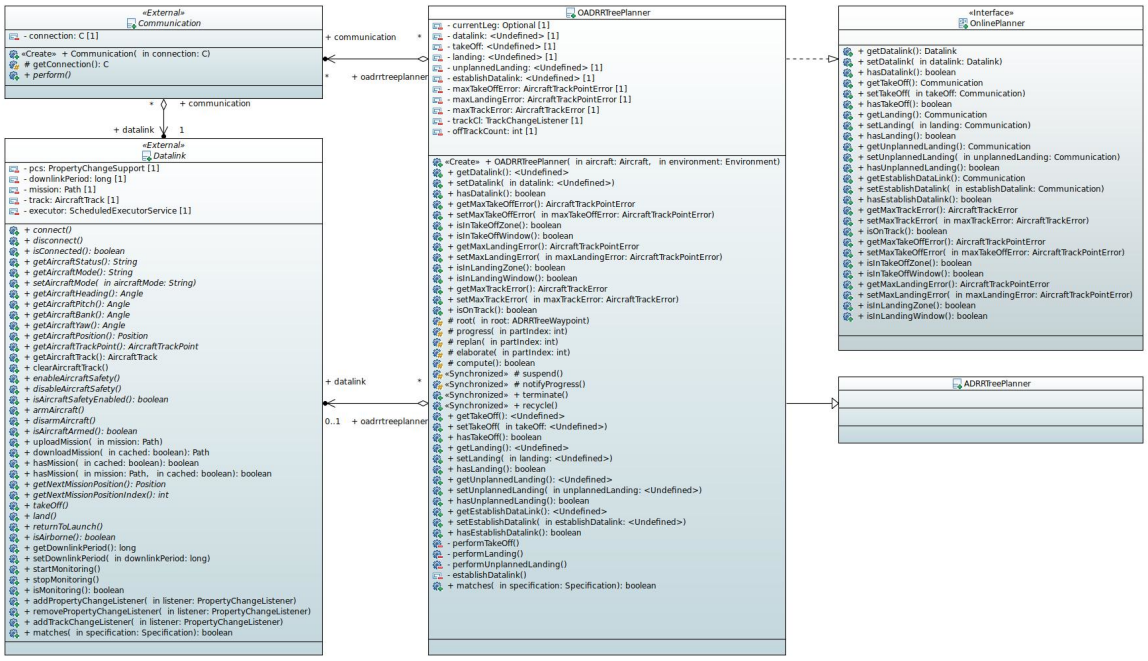


Figure 7.17: SAFCS Core: OADRRT Class Signatures

Recall that all offline Planners can only revise a Trajectory as long as the Aircraft has not departed its departure POI. Purely DynamicPlanners can, therefore, only handle changes to a DynamicEnvironment based on a stationary known Aircraft position. An OnlinePlanner additionally monitors the progress of its Datalink source in terms of an AircraftTrack comprising AircraftTrackPoints, as shown in Figure 5.22, and can be displayed in our SAFCS UI, as depicted in Figure 5.23. All statements made in Section 6.5 for the OADStarPlanner equally apply to the OADRRTreePlanner. Progressing for the OADRRTreePlanner essentially means removing all previously visited ADRRTreeWaypoints and re-rooting the RRT. The rooting method is shown in Listing 7.19 and simply removes all irrelevant vertices and dependent Edges from the PlanningContinuum. Rooting is intuitively the opposite of trimming where a branch of an existing RRT becomes the new RRT and all structures leading to that branch as well as their irrelevant dependents are eliminated. It is invoked for the next planned ADRRTreeWaypoint while the Datalink source remains on track within its RNP.

```
protected void root(ADRRTreeWaypoint root) {
    this.setStart(root);

    if (root.hasParent()) {
        ADRRTreeWaypoint parent = root.getParent();
        parent.removeChild(root);
        root.removeParent();
        while (parent.hasParent()) {
            parent = parent.getParent();
        }
        this.trim(parent);
    }
}
```

Listing 7.19: SAFCS Core: OADRRT Rooting

After the new root becomes the start POI of the OADRRTreePlanner (line 2), it is separated from its parent (lines 5 to 7). The RRT is then traversed from the parent of the new root to its existing root (lines 8 to 10) before trimming, that is, purging it (line 11). Reducing the number of relevant internal structures of a StructuredEnvironment, such as the PlanningContinuum, may, furthermore, reduce the computational cost of selective sampling during re-planning.

Recall that take-off and (planned or unplanned) landing Datalink Communications have to be performed by an OnlinePlanner. Maximum acceptable AircraftTrackErrors and AircraftTrackPointErrors have to be operationally defined to maintain the Datalink source within certain spatial and temporal boundaries of the planned Trajectory as well as to ensure the RNP during the individual phases of the flight, including the take-off and landing maneuvers. If off-track situations are recognized during the progress monitoring, then elaboration with re-planning has to be performed accordingly and ideally OPS has to be informed.

The elaboration step of an OADRRTreePlanner is shown in Listing 7.20 and extends the corresponding ADRRTreePlanner method with the handling of recognized off-track situations (lines 11 to 14). This extension is equivalent to the one discussed for the OADStarPlanner in Section 6.5 and invokes a re-planning attempt (line 13) to proceed directly to the next regularly planned ADRRTreeWaypoint and generate a new Trajectory based on the ETO when reaching it. Note that in case of an off-track situation during the execution of a multi-part plan, all subsequent parts are likely to be re-planned if the ETOs of connecting POIs cannot be kept consistent. Similar to the statements made with respect to dynamic repairs, the introduction of flexible Waypoint slots that correspond to the operationally RNP could potentially reduce the necessary re-planning effort and is revisited in Chapter 15.

As explained in Section 6.5, an airborne agent may suddenly and unexpectedly be exposed to unanticipated costs, such as imposed by in-flight turbulence, icing or closed airspaces. If these unanticipated costs exceed the established operational RiskPolicy, then the agent must be able to escape the perceived threat using a temporarily boosted RiskPolicy. Recall that RiskPolicy boosting is not applicable to situations where the goal of the mission exceeds the operational RiskPolicy, which would correspond to an undesired *pilot get-homitis* behavior. Instead, the mission has to be abandoned in these cases and the agent has to be guided to a suitable in-flight alternate, or instructed to perform an unplanned precautionary landing via the corresponding Datalink Communication. In-flight alternates and unplanned precautionary landings

```
protected void elaborate(int partIndex) {
    // limit deliberation for active part
    if (partIndex == this.getActivePart()) {
        this.setDeliberationStart(ZonedDateTime.now());
        this.setDeliberationEnd(
            this.getDeliberationStart().plus(this.getDeliberation()));
    } else {
        this.setDeliberationEnd(null);
    }

    if (!this.isOnTrack()) {
        this.offTrackCount = 0;
        this.replan(partIndex);
    }

    // do not elaborate an exceeded risk policy solution beyond limit
    int riskyProbes = (this.getGoal().hasInfiniteCost()) ? 1 : 0;

    // proceed to next part only if fully improved and not in need of repair
    while (this.canDeliberate() && !this.hasTerminated()
            && ((!this.hasMaximumQuality()
            && (this.getMaxRiskyProbes() > riskyProbes))
            || this.needsRepair())) {
        this.repair(partIndex);
        this.improve(partIndex);
        this.progress(partIndex);
        if (this.getGoal().hasInfiniteCost()) {
            riskyProbes++;
        } else {
            riskyProbes = 0;
        }
        this.updateDynamicObstacles();
    }
    // backup after elaboration
    this.backup(partIndex);
}
```

Listing 7.20: SAFCS Core: OADRRT Elaboration

are not yet implemented, further discussed in Chapter 15, and should incorporate suitable OPS interactions.

More details about progress monitoring, off-track determination, supported communication sequences, and other aspects of the SAFCS core `OADRRTreePlanner`[8] can be found in our SAFCS software repository.

7.9 Chapter Summary

In this chapter we studied some implementation details addressing important modifications and extensions to the supported members of the RRT family of algorithms. The presented modifications and extensions mainly relate to the integration of `CostIntervals` for SWIM `Obstacles` into the supported planning `Environments` and the computation of 4D single- and multi-part trajectories. We, furthermore, identified problematic areas and options for improvement, that shall be addressed in future research of our RRT core algorithms.

[8] https://github.com/stephanheinemann/worldwind/tree/develop/src/main/java/com/cfar/swim/worldwind/planners/rrt/oadrrt

Chapter 8

SAFCS Core — Probabilistic Roadmaps

The PRMs have yet to be reviewed and refactored carefully into the SAFCS core development branch. For now, more details on PRMs can be found on the associated feature branch[1].

[1] https://github.com/stephanheinemann/worldwind/tree/feature/continuum-planner/src/main/java/com/cfar/swim/worldwind/ai/prm

Chapter 9

SAFCS — Autonomic Manager

9.1 Managed Planners

The ManagedPlanners of the SAFCS are managable extensions to the key algorithms of their respective families in the SAFCS core. Currently, these Planners are the OADStarPlanner, the OADRRTreePlanner, and the OFADPRMPlanner representing the CGS, RRT, and PRM families, respectively. Figure 9.1 shows the additional interface signature that all ManagedPlanners are required to implement and further refines the diagram shown in Figure 5.26.

In addition to the required functionality of an AnytimePlanner, DynamicPlanner, LifelongPlanner, and OnlinePlanner, a ManagedPlanner shall report its achieved PlannerPerformance to its AutonomicManager. Being an OnlinePlanner, only one ManagedPlanner can be actively controlling the Datalink source of the managed planning Session at any time. Therefore, the AutonomicManager is responsible for assigning the leading *active* ManagedPlanner following a Performance competition while the remaining under-performing ManagedPlanners are put on *standby* until the next competition. Competitions between ManagedPlanners are initiated whenever a new Trajectory has to be computed, or if an existing Trajectory has to be revised following the injection or removal of dynamic Obstacles during the managed planning Session. Competitions include the tuning of each ManagedPlanner according to the (updated) Features of the active Scenario, and possibly considering the Reputation of the ManagedPlanner that has previously been persisted in the KnowledgeBase of the AutonomicManager in charge.

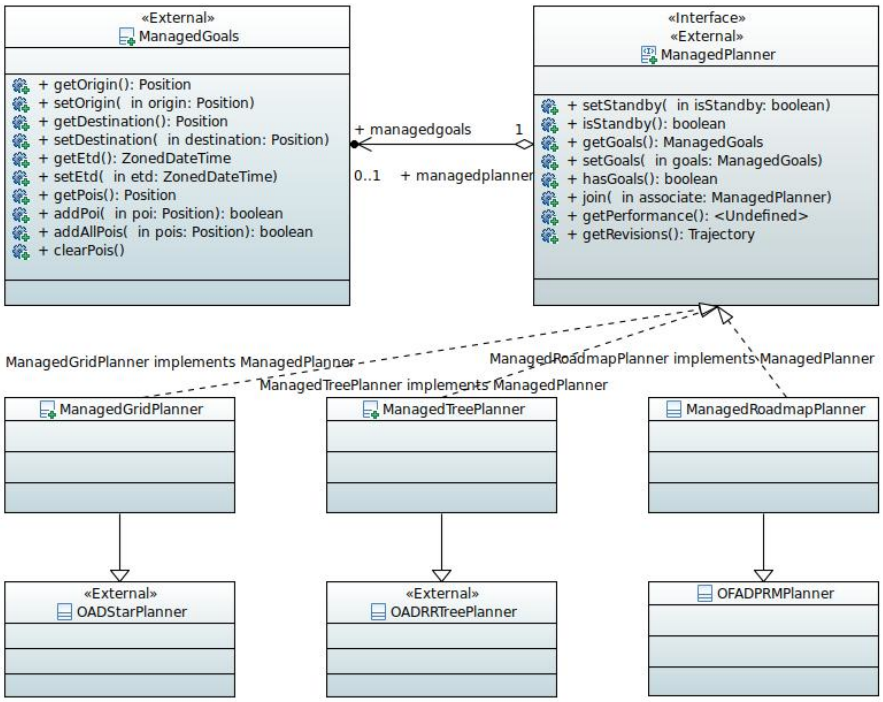


Figure 9.1: SAFCS AM: Managed Planners Signature

Listing 9.1 reveals the necessary implementation extensions to `ManagedPlanners` in terms of invocation as `Trajectory`-computing `Callables` that are invoked by an `ExecutorService` of their concrete `AutonomicManager`. It, furthermore, shows the fairly simple extensions for `Performance`-reporting and `Trajectory`-revisioning to allow for a possible assessment of the quality development of revised solutions. Lines 1 to 22 show the `call` method which is invoked by an `ExecuterService` of the `AutonomicManager`. Depending on whether or not intermediate POIs are present in the `ManagedGoals` specified by OPS, a single- or multi-part `Trajectory` planning is invoked in lines 6 to 9 or 11 to 15, respectively. This extension simply adds `call` to the invocation stack but otherwise does not change the implementation of the extended key algorithm.

The original `plan` methods are slightly modified, as shown in lines 24 to 33, to reset the `PlannerPerformance` of the `ManagedPlanner` before commencing the computation of a `Trajectory`. Resetting the performance is achieved by the `resetPerformance` method in lines 50 to 55 and includes clearing the existing `Trajectory` revisions and recording the start time of the computation. Recall that being an `AnytimePlanner` the `ManagedPlanner` computes `Trajectory` revisions incrementally to acknowledge the limitations of bounded rationality. The `PlannerPerformance`

```
public Trajectory call() throws IllegalStateException {
    Trajectory trajectory;

    if (this.hasGoals()) {
        if (this.getGoals().getPois().isEmpty()) {
            trajectory = this.plan(
                    this.getGoals().getOrigin(),
                    this.getGoals().getDestination(),
                    this.getGoals().getEtd());
        } else {
            trajectory = this.plan(
                    this.getGoals().getOrigin(),
                    this.getGoals().getDestination(),
                    this.getGoals().getPois(),
                    this.getGoals().getEtd());
        }
    } else {
        throw new IllegalStateException();
    }

    return trajectory;
}

public Trajectory plan(Position origin, Position destination, ZonedDateTime etd) {
    this.resetPerformance();
    return super.plan(origin, destination, etd);
}

public Trajectory plan(Position origin, Position destination,
    List<Position> waypoints, ZonedDateTime etd) {
    this.resetPerformance();
    return super.plan(origin, destination, waypoints, etd);
}

protected void revisePlan(Trajectory trajectory) {
    if (!trajectory.isEmpty() && !this.revisions.contains(trajectory)) {
        double normalizer = this.computePoiDistance(trajectory)
                / this.poiDistance;
        this.performance = new PlannerPerformance(
                this.planStartTime,
                new TrajectoryQuality(trajectory, normalizer),
                new DurationQuantity(Duration.between(
                        this.planStartTime, ZonedDateTime.now()),
                        1d / normalizer));
        this.revisions.add(trajectory);
        super.revisePlan(trajectory);
    }
}

protected void resetPerformance() {
    this.poiDistance = this.computePoiDistance();
    this.performance = PlannerPerformance.ZERO;
    this.revisions.clear();
    this.planStartTime = ZonedDateTime.now();
}

public PlannerPerformance getPerformance() {
    return this.performance;
}

public List<Trajectory> getRevisions() {
    return Collections.unmodifiableList(this.revisions);
}
```

Listing 9.1: SAFCS AM: Invoking Managed Planners

of each revision is, therefore, recorded accordingly before the aggregated revision history is extended and PlanRevisionListeners are notified in lines 35 to 48. Both PlannerPerformances and existing Trajectory revisions can be obtained from the notified PlanRevisionListeners by calling the provided getPerformance and getRevisions methods in lines 57 to 63, respectively. It should not be a surprise that an AutonomicManager adds its own PlanRevisionListener to its ManagedPlanners in order to obtain Trajectory revisions and PlannerPerformance updates. Recall that as a LifelongPlanner the plan and, hence, call methods do not terminate until the ManagedPlanner is terminated explicitly by its AutonomicManager. Thus, Trajectory revisions are necessary and more important than the final and returned Trajectory of the ManagedPlanner as a Callable.

Being a DynamicPlanner, a ManagedPlanner, furthermore, needs to re-initiate the incremental computation of a Trajectory whenever the Features of the active Scenario of the managed planning Session change, that is, when instructed by its AutonomicManager to join the competition for the best PlannerPerformance given the updated AC. In addition, as an OnlinePlanner, the presently *active* best-performing ManagedPlanner has to monitor the progress of the mission and upload its Trajectory to its Datalink source. These aspects of ManagedPlanners are detailed in Listings 9.3 and 9.2, respectively.

Listing 9.2 essentially reveals how an under-performing *standby* ManagedPlanner ceases to communicate with the Datalink source of the active Scenario of its managed planning Session. In particular, Trajectory revisions are not uploaded via its muted MissionLoader (line 4). The mission progress (lines 26 to 30) of the airborne agent and whether or not it isOnTrack (line 16 to 24), that is, within the specified operational tolerances, are not being monitored by a *standby* ManagedPlanner. Furthermore, all Datalink Communications, such as Connection establishment, take-off, landing, and unplanned landing Communications, are not performed while standing by (lines 32 to 54).

As the airborne agent progresses along its uploaded Trajectory, changes to the AC, that is, to the Features of the active Scenario of the managed planning Session, may require the AutonomicManager to instruct its *standby* ManagedPlanners to re-join their leading associate *active* ManagedPlanner in the competition for the best PlannerPerformance under the new conditions. Since the airborne agent has already progressed according to the (possibly multi-part) Trajectory of the *active*

```
public void setStandby(boolean isStandby) {
    if (this.isStandby != isStandby) {
        if (isStandby) {
            this.getMissionLoader().mute();
        } else {
            this.getMissionLoader().unmute();
        }
        this.isStandby = isStandby;
    }
}

public boolean isStandby() {
    return this.isStandby;
}

public boolean isOnTrack() {
    boolean isOnTrack = true;

    if (!this.isStandby()) {
        isOnTrack = super.isOnTrack();
    }

    return isOnTrack;
}

protected void progress(int partIndex) {
    if (!this.isStandby()) {
        super.progress(partIndex);
    }
}

protected void performTakeOff() {
    if (!this.isStandby()) {
        super.performTakeOff();
    }
}

protected void performLanding() {
    if (!this.isStandby()) {
        super.performLanding();
    }
}

protected void performUnplannedLanding() {
    if (!this.isStandby()) {
        super.performUnplannedLanding();
    }
}

protected void establishDatalink() {
    if (!this.isStandby()) {
        super.establishDatalink();
    }
}
```

Listing 9.2: SAFCS AM: Active and Standby Managed Planners

ManagedPlanner up to this point, the *standby* ManagedPlanners, consequently, need to join the next Waypoint in the active part of its current Trajectory.

As lines 44 to 59 of Listing 9.3 show, all ManagedPlanners are essentially suspended for any time period while not having received an explicit termination request, updates to the AC, or observed off-track situations. In case of a change to its AC, the AutonomicManager notifies all of its ManagedPlanners. Since in this situation a new competition and incremental computation of Trajectory revisions is initiated, the PlannerPerformance has to be reset again (line 52). After having committed to the dynamic changes (line 50), all *standby* ManagedPlanners join their associate *active* ManagedPlanner in the competition (line 53). To that end, the associate to be joined was previously announced by their AutonomicManager via the join method (lines 13 to 17).

The joining procedure of a *standby* ManagedPlanner involves several steps shown in lines 19 to 42. Obviously, only *standby* ManagedPlanners who have an *active* associate, that is still progressing toward a next Waypoint, are eligible to join the competition for the best PlannerPerformance (lines 20 and 21). Once the active part of a multi-part Trajectory and the next Waypoint within that part have been obtained from the associate (lines 22 and 23), all previous non-active parts of the mission, that, consequently, must have already been completed by the airborne agent, are erased (lines 25 to 28). Restoring the active part and aligning the start Waypoint with the next Waypoint to be visited includes the assignment of cost and ETO that had previously been computed by the associate *active* ManagedPlanner (lines 30 to 35). Finally, since the existing plan of the *standby* ManagedPlanner has become irrelevant, an entirely new Trajectory is computed (line 39) and the joining procedure is finished by removing the reference to its associate (line 40). Notice that re-planning during a joining procedure invokes exactly the same method that is invoked in an off-track situation described for OnlinePlanners in Chapters 6, 7, and 8, that is, the existing plan has become obsolete and cannot be repaired. After having computed the initial Trajectory, needsRepair (line 47) releases the ManagedPlanner from its suspension and an elaboration phase again incrementally improves and revises the solution.

It should be emphasized that while the *standby* ManagedPlanners are instructed to join their associate and re-plan before elaboration, the *active* ManagedPlanner skips the joining procedure and proceeds with the elaboration and included repair of its solution immediately. One may consider this approach unfair, but these are simply the established rules of our game and winners do not need to qualify for a race. After all,

```
protected synchronized ManagedPlanner getAssociate() {
    return this.associate;
}

protected synchronized void setAssociate(ManagedPlanner associate) {
    this.associate = associate;
}

protected synchronized boolean hasAssociate() {
    return (null != this.associate);
}

public synchronized void join(ManagedPlanner associate) {
    if (this.isStandby()) {
        this.setAssociate(associate);
    }
}

protected synchronized void joinAssociate() {
    if (this.isStandby() && this.hasAssociate()
            && this.getAssociate().hasNextWaypoint()) {
        Waypoint nextWaypoint = this.getAssociate().getNextWaypoint();
        int activePart = this.getAssociate().getActivePart();

        // clear all previous parts
        for (int part = 0; part < activePart; part++) {
            this.backups.get(part).clear();
        }

        // join active part
        this.restore(activePart);
        this.setStart(this.createWaypoint(nextWaypoint));
        this.getStart().setCost(nextWaypoint.getCost());
        this.getStart().setEto(nextWaypoint.getEto());
        this.backup(activePart);

        // re-plan from joined starting point
        this.restore(this.backups.size() - 1);
        this.replan(this.backups.size() - 1);
        this.setAssociate(null);
    }
}

protected synchronized void suspend() {
    try {
        // wait for termination, dynamic changes, off-track situations, or joins
        while (!this.hasTerminated() && !this.needsRepair() && this.isOnTrack()) {
            this.progress(this.backups.size() - 1);
            this.wait();
            this.updateDynamicObstacles();
            if (this.needsRepair() || !this.isOnTrack()) {
                this.resetPerformance();
                this.joinAssociate();
            }
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

Listing 9.3: SAFCS AM: Joining Managed Planners

repairing an existing solution may not necessarily be more efficient than planning from scratch in cases where the planning context or `StructuredEnvironment` are saturated with existing data. Furthermore, recall that a *significant* change to the established plan may very well lead to an implicit re-planning attempt during elaboration.

More details about `ManagedPlanners`,[1] can be found in our SAFCS software repository.

9.2 Discriminating Features

`Features` are the *sensors* of the SAFCS AM to become *context-aware* and, hence, essential to the concept of the SAFCS as an AE. The selection of `Features` of the actively executing or otherwise trained `Scenarios` of a managed planning `Session` requires domain knowledge about the `ManagedPlanners` employed by the SAFCS AM. However, a multitude of `Features` can be extracted from a `Scenario` that are not (equally) relevant to all `ManagedPlanners`. As such, future supported `ManagedPlanners` may very well be tuned on existing or require the extraction of new discriminating `Features`, that is, the introduction of new sensors.

The current set of extracted discriminating `Features` is shown in Listing 9.4. Presently, we have identified four main groups of `Features` that may affect the performance of `ManagedPlanners`. These groups are `feature.aircraft`, `feature.environment`, `feature.pois`, and `feature.policy`. The individual `Features` in each group are fairly self-explanatory, but it should be emphasized that most of them relate to `Obstacles` in the AC. In particular, the SAFCS AM is interested in their existence, cardinality, volume, distribution, density, and severity. Recall from Sections 3.1 and 5.2.4 that `Obstacles` may represent a manifold of temporal SWIM data, including airspaces, traffic, weather, and facilities. Consequently, `Obstacles` are highly unlikely to not at all be present in the `Scenario` of a managed planning `Session`. If, however, no `Obstacles` are sensed by the extraction of `Features`, then the complexities of planning are reduced to the `Capabilities` of the `Aircraft`, as explained in Section 5.2.2.

It should be fairly obvious, that the existence of `Obstacles` and their operational assessment within the AC affects the performance of any employed `ManagedPlanner`. Less obvious, on the other hand, is how the concrete `ManagedPlanner` is affected, and

[1] https://github.com/stephanheinemann/worldwind/tree/develop/src/main/java/com/cfar/swim/worldwind/planners/managed

```
feature.aircraft.altitude=aircraft altitude
feature.aircraft.speed.horizontal=aircraft horizontal speed
feature.aircraft.speed.vertical.climb=aircraft rate of climb
feature.aircraft.speed.vertical.descent=aircraft rate of descent
feature.aircraft.obstacles.distance.avg=aircraft average obstacle distance
feature.aircraft.obstacles.distance.max=aircraft maximum obstacle distance
feature.aircraft.obstacles.distance.min=aircraft minimum obstacle distance
feature.aircraft.obstacles.nearest.cost=aircraft nearest obstacle cost
feature.aircraft.obstacles.nearest.cost.policies=aircraft nearest obstacle policies cost
feature.aircraft.obstacles.nearest.volume=aircraft nearest obstacle volume
feature.aircraft.obstacles.nearest.volume.ratio=aircraft nearest obstacle volume ratio
feature.aircraft.safety.radius=aircraft safety sphere radius
feature.aircraft.safety.volume=aircraft safety sphere volume
feature.aircraft.safety.obstacles.count=aircraft safety sphere obstacle count
feature.aircraft.safety.obstacles.cost.policies=aircraft safety sphere policies obstacle cost
feature.aircraft.safety.obstacles.cost.avg=aircraft safety sphere average obstacle cost
feature.aircraft.safety.obstacles.cost.max=aircraft safety sphere maximum obstacle cost
feature.aircraft.safety.obstacles.cost.min=aircraft safety sphere minimum obstacle cost
feature.aircraft.safety.obstacles.cost.sum=aircraft safety sphere accumulated obstacle cost
feature.aircraft.safety.obstacles.volume.avg=aircraft safety sphere average obstacle volume
feature.aircraft.safety.obstacles.volume.max=aircraft safety sphere maximum obstacle volume
feature.aircraft.safety.obstacles.volume.min=aircraft safety sphere minimum obstacle volume
feature.aircraft.safety.obstacles.volume.sum=aircraft safety sphere accumulated obstacle volume

feature.environment.diameter=environment diameter
feature.environment.volume=environment volume
feature.environment.obstacles.count=environment obstacle count
feature.environment.obstacles.cost.policies=environment policies obstacle cost
feature.environment.obstacles.cost.avg=environment average obstacle cost
feature.environment.obstacles.cost.max=environment maximum obstacle cost
feature.environment.obstacles.cost.min=environment minimum obstacle cost
feature.environment.obstacles.cost.sum=environment accumulated obstacle cost
feature.environment.obstacles.volume.avg=environment average obstacle volume
feature.environment.obstacles.volume.max=environment maximum obstacle volume
feature.environment.obstacles.volume.min=environment minimum obstacle volume
feature.environment.obstacles.volume.sum=environment accumulated obstacle volume
feature.environment.obstacles.volume.ratio=environment obstacle volume ratio

feature.pois.distance.avg=consecutive POIs average distance
feature.pois.distance.max=consecutive POIs maximum distance
feature.pois.distance.min=consecutive POIs minimum distance
feature.pois.distance.sum=consecutive POIs accumulated distance
feature.pois.distance.vertical.avg=consecutive POIs average vertical distance
feature.pois.distance.vertical.max=consecutive POIs maximum vertical distance
feature.pois.distance.vertical.min=consecutive POIs minimum vertical distance
feature.pois.distance.vertical.sum=consecutive POIs accumulated vertical distance
feature.pois.diameter=POIs bounding box diameter
feature.pois.volume=POIs bounding box volume
feature.pois.obstacles.count=POIs bounding box obstacle count
feature.pois.obstacles.cost.policies=POIs bounding box policies obstacle cost
feature.pois.obstacles.cost.avg=POIs bounding box average obstacle cost
feature.pois.obstacles.cost.max=POIs bounding box maximum obstacle cost
feature.pois.obstacles.cost.min=POIs bounding box minimum obstacle cost
feature.pois.obstacles.cost.sum=POIs bounding box accumulated obstacle cost
feature.pois.obstacles.volume.avg=POIs bounding box average obstacle volume
feature.pois.obstacles.volume.max=POIs bounding box maximum obstacle volume
feature.pois.obstacles.volume.min=POIs bounding box minimum obstacle volume
feature.pois.obstacles.volume.sum=POIs bounding box accumulated obstacle volume
feature.pois.obstacles.volume.ratio=POIs bounding box obstacle volume ratio
feature.pois.environment.volume.ratio=POIs bounding box environment volume ratio
feature.pois.environment.obstacles.count.ratio=POIS bounding box environment obstacles count ratio
feature.pois.environment.obstacles.cost.policies.ratio=POIs bounding box environment policies obstacle cost ratio
feature.pois.environment.obstacles.cost.avg.ratio=POIs bounding box environment average obstacle cost ratio
feature.pois.environment.obstacles.cost.max.ratio=POIs bounding box environment maximum obstacle cost ratio
feature.pois.environment.obstacles.cost.min.ratio=POIs bounding box environment minimum obstacle cost ratio
feature.pois.environment.obstacles.cost.sum.ratio=POIs bounding box environment accumulated obstacle cost ratio
feature.pois.environment.obstacles.volume.ratio=POIs bounding box environment obstacles volume ratio

feature.policy.cost=operational cost policy
feature.policy.risk=operational risk policy
```

Listing 9.4: SAFCS AM: Features

what tuning parameters are at its disposal to improve its performance given a certain instance of Features. Sections 9.6 and 9.7 provide more insight into these topics. However, two simple RRT examples shall be given here: Let us assume that the AC does not feature any Obstacles, and that there are no restrictions regarding the Capabilities of the Aircraft. Then, a straight line from origin to destination would be the solution to this trivial problem. Recalling some of the tuning parameters of RRTs presented in Chapter 7, one may quickly deduce that the *goal bias* could be as high as 100%, and that the tree *extension* could cover the entire distance from origin to destination in a single *connect* step. Let us now assume that the AC features a relatively large Obstacle exceeding the operational RiskPolicy located directly in front of the Aircraft. Intuitively, an RRT now needs to apply a fairly low *goal bias* and favor the exploration of its planning Environment by applying a UNIFORM or GAUSSIAN sampling with shorter *extension* steps to be successful. Furthermore, many more iterations may be required to find a solution in this case, if any.

Since SAFCS Scenarios model $4D$ Environments, associated Features also exist in both space and time. Figure 9.2 shows the Environment of a managed Scenario with several embedded Obstacles at the consecutive time instances $T+0$, $T+1$, and $T+2$, respectively.

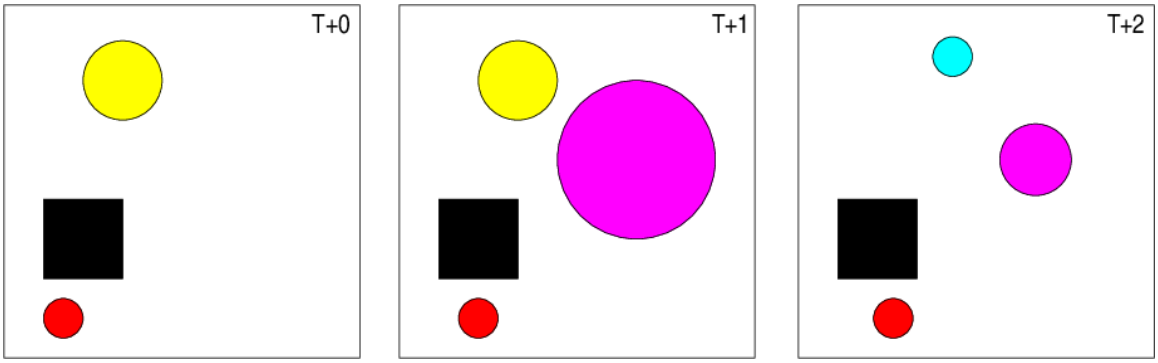


Figure 9.2: SAFCS AM: 4D Features

In this example, the black Obstacle is stationary and present in all time instances. The yellow Obstacle is also stationary but only present in time instances $T+0$ and $T+1$. The magenta Obstacle is stationary just like the cyan one and they are present during the time instances $T+1$ and $T+2$, and $T+2$ only, respectively. However, the magenta Obstacle changes its size in time. The red Obstacle is the only non-stationary one in this example as it moves from left to right and is present in all three time instances.

It should be apparent that extracting the Features of a managed Scenario can be short-sighted and overly optimistic if only one time instance is considered. In the opposite situation, an overly pessimistic set of Features would be extracted if too many time instances are taken into account. The relevant Features *horizon* is, therefore, an important management parameter and part of the operational constraints specified by OPS. It is a Duration that clearly depends on the volatility of the Environment of the managed Scenario and the progress of the agent within it. For instance, it would not be advisable to specify a Features horizon that covers a longer Duration than estimated for the completion of the mission itself. Likewise, it would be inappropriate to cover a single instance in time only.

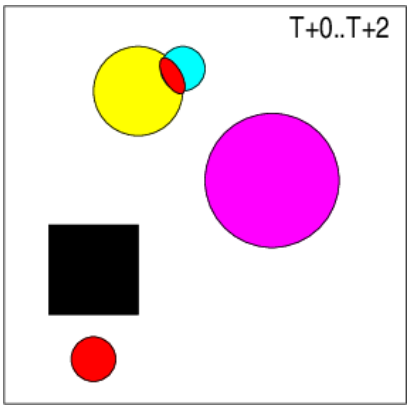


Figure 9.3: SAFCS AM: Features Horizon

The Features within the Features horizon can be constructed as shown in Figure 9.3. The SAFCS simply combines the Features of all time instances in a particular manner. Our approach ignores whether or not Obstacles overlap in space or time but recognizes if they are in fact the same Obstacle, such as the red one. This simplification obviously affects important Features, such as densities, volumes, and derived ratios. For example, both the yellow and cyan Obstacles are present during the Features horizon and they in fact intersect each other. This intersection does not merely affect the sum of all Obstacle volumes (e.g., `feature.environment.obstacles.volume.sum`) but also dependent ratios of these volumes (e.g., `feature.pois.environment.volume.ratio`). We, furthermore, do not compute intersecting volumes of Obstacles with the Environment, the POIs bounding box, or the Aircraft safety sphere, neither in space nor in time, but rather extract full minimum, maximum, and average volumes. When extracting accumulated volumes (e.g., `feature.environment.obstacles.volume.sum`),

we apply the average full volume of Obstacles that change their size in time, such as for the magenta one in Figure 9.3.

Without question, the Features horizon model can be elaborated to more precisely consider Obstacle movements and intersections to extract Features of higher quality. However, it should be recalled that Features are used by concrete AutonomicManagers to derive suitable Tunings for their ManagedPlanners. Whether a HeuristicAutonomicManager applies heuristics to the extracted Features or a SmacAutonomicManager considers the Reputation of ManagedPlanners based on an identified FeatureCategory and the best on average Tunings obtained during training, there is always room for error. We consider our simplified Features horizon model as currently sufficient for our experiments and are aware of the introduced *instrument error*. However, in the worst case, the extracted Features may not adequately represent the AC when, for example, many large Obstacles overlap significantly or merely touch certain spaces without actually restricting them.

Features can be associated to a FeatureCategory. A FeatureCategory essentially establishes covered ranges or constraints of Features. These ranges or constraints may then serve to discriminate different levels of Criticality or different FlightPhases, as shown in Figure 5.28. More sophisticated AutonomicManagers may associate arbitrary (unknown) Features with each other based on their FeatureCategory to apply promising Tunings of known Features of the same FeatureCategory. For example, the ManagedPlanner of a Scenario featuring a RiskPolicy-exceeding Obstacle density (e.g., feature.pois.obstacles.volume.ratio) of 25% may be tuned similarly to one featuring 26%. If an AutonomicManager performs training accordingly, then relevant FeatureCategories may be covered sufficiently to outperform otherwise heuristically guided approaches.

The SAFCS AM currently defines the Criticality levels *low*, *moderate*, *substantial*, *severe*, *critical*, and *fatal*. It, furthermore, defines the FlightPhases (or flight situations) *emergency*, *urgency*, *terminal*, *transition*, and *cruise*. Figure 9.4 increases the level of detail shown in Figure 5.28 with respect to FlightPhases and shows the relevant parts of the associated class signatures.

Applying domain knowledge, we have decided to establish FlightPhases based on the perceived Difficulty and Severity of their Features. The Difficulty of Features within their Features horizon $d(f)$ is described in terms of Obstacle scattering by Equation 9.1.

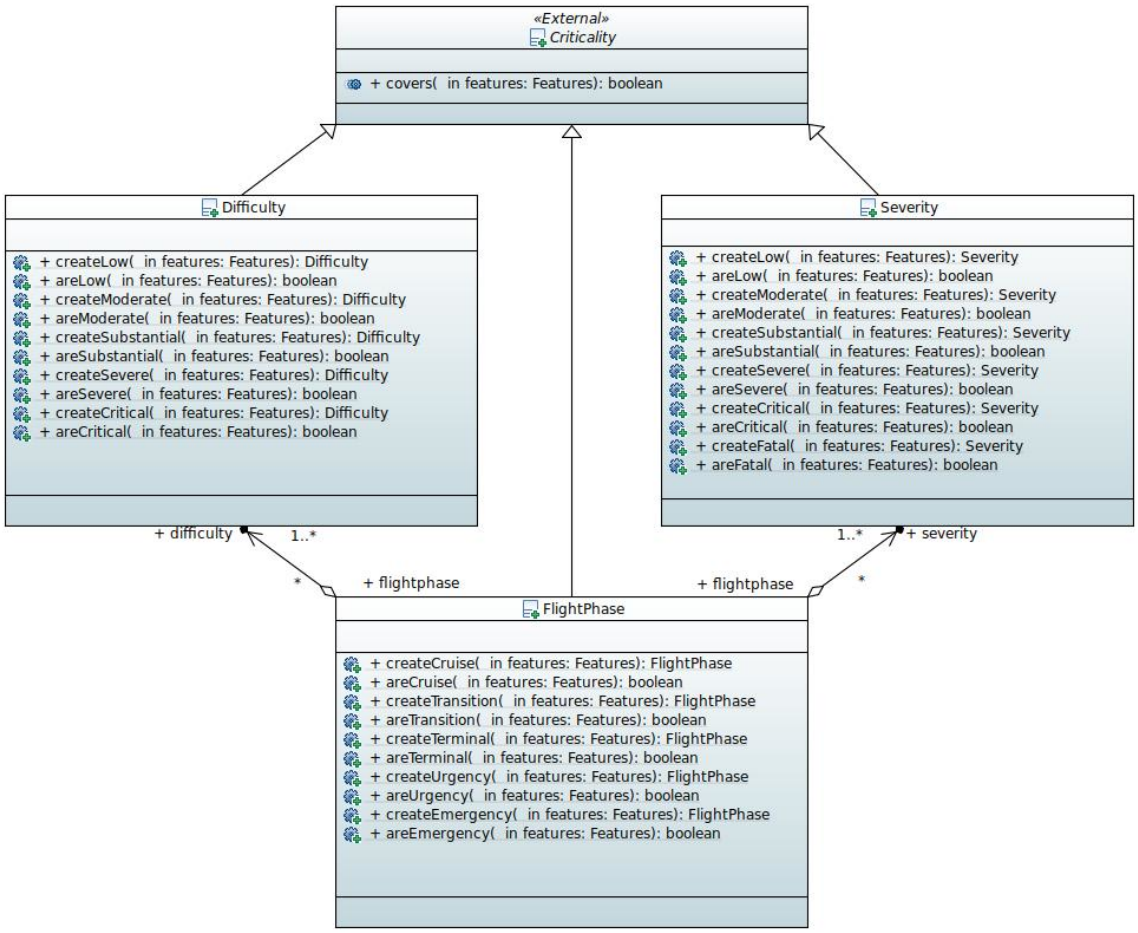


Figure 9.4: SAFCS AM: Criticality

$$\mathrm{d}(\text{feature}) = \frac{\text{feature.pois.obstacles.volume.avg}}{\text{feature.pois.volume} * \text{feature.pois.obstacles.volume.ratio}} \tag{9.1}$$

The equation is shown based on the sensed `Features` but could easily be transformed and simplified into a ratio of geometrical volume ratios. The first ratio is given by the average volume of `Obstacles` and the volume of the POI bounding box they reside in during the `Features` horizon. The second ratio is given by the accumulated volume of these `Obstacles` and the volume of the same POI bounding box they reside in during the `Features` horizon. While the first ratio tells us how much space on average is occupied by an `Obstacle` in the POI bounding box, the second ratio is an indicator of how

much space in total is occupied by all Obstacles in the POI bounding box. Recall that intersections between these Obstacles in space or time are not taken into account and accepted as *instrument error* for our experiments. Calculating the ratio of both of these ratios, that is, the average POI bounding box occupancy versus the total POI bounding box occupancy, gives us an idea about how *scattered* Obstacles are within the POI bounding box during the Features horizon. Therefore, a single large Obstacle occupying the entire POI bounding box would be considered less difficult than many smaller Obstacles scattered within it and vice versa.

A single large Obstacle is indeed less difficult than a higher number of smaller Obstacles for our SAFCS core Planners. For the basic forward A* algorithm, the heuristic will underestimate significantly in this case leading to a longer search. However, if the ARA* heuristic is inflated by the cost of the Obstacle, then a first solution is found quickly and the situation becomes equivalent to a no-Obstacle, non-inflated scenario. In both cases, the initial inflation could be set to the maximum or average Obstacle cost by the HeuristicAutonomicManager and improved by the SmacAutonomicManager based on the sensed Difficulty.

For RRT within a single large Obstacle, new samples will unlikely permit to improve an existing shorter solution. Therefore, the distance bias becomes more important than the cost bias, which is equivalent to the higher initial inflation of the ARA* heuristic.

The overall search in a less difficult Scenario may take longer, but leveraging the heuristic inflation and distance bias as mentioned may lead to rapidly available first solutions. Many smaller and scattered Obstacles, however, generally require a smaller heuristic inflation and distance bias from the beginning, and combining both situations may create the most challenging set of Features. In these cases, the applied heuristics can only be underestimating, which, consequently, leads to more exhausitive searches.

Table 9.1 shows how Difficulty ranges are categorized as *low*, *moderate*, *substantial*, *severe*, and *critical*, respectively. Chapter 11 shows pictures of evaluation scenarios that correspond to these Difficulty categories and visualize the same idea.

The Severity of a FlightPhase is another discriminating aggregation that captures the relevance and cost of Obstacles in the planning Environment. Low-cost Obstacles that are present in the Environment but do not affect the POI bounding box and, hence, the computed Trajectory, are considered to be less severe. Costly Obstacles that, on the other hand, penetrate the POI bounding box or even the Aircraft safety sphere are considered to be more severe.

Table 9.1: Difficulty

Difficulty Category	Difficulty Range
low	$d(f) > 0.8$
moderate	$d(f) \in (0.6, 0.8]$
substantial	$d(f) \in (0.4, 0.6]$
severe	$d(f) \in (0.2, 0.4]$
critical	$d(f) \in (0.0, 0,2]$

The `Severity` has an influence on the expansion of A* since higher costs in less difficult areas reduce the effect of the heuristic and may lead to a longer search for a first solution if the heuristic is not inflated accordingly. Similarly, more difficult high cost areas increase the effect of the heuristic and may lead to a faster search and better results if the heuristic remains more deflated. For RRT, it affects the choice of biases in particular as well as the sampling distribution. Nearby and expensive `Obstacles` may require a lower goal (distance) bias, more scattered sampling, and higher cost bias. In a low `Severity` scenario a higher goal (distance) bias, more focused sampling, and lower cost bias may be the better choices.

Table 9.2: Severity

Severity Category	Aircraft Safety Sphere	POI Bounding Box	Environment
low	$c \leq 0$	$c \leq 0$	$c \leq$ `RiskPolicy`
moderate	$c \leq 0$	$c \leq$ `RiskPolicy`	$c \in (0,$ `RiskPolicy`$]$
substantial	$c \leq$ `RiskPolicy`	$c \in [0,$ `RiskPolicy`$]$	$c > 0$
severe	$c \in [0,$ `RiskPolicy`$]$	$c > 0$	$c >$ `RiskPolicy`
critical	$c =$ `RiskPolicy`	$c >$ `RiskPolicy`	$c >$ `RiskPolicy`
fatal	$c >$ `RiskPolicy`	$\top$	$\top$

Table 9.2 shows how sensed `Obstacle` costs within the `Aircraft` safety sphere, the POI bounding box, and the planning `Environment` during the `Features` horizon are categorized as *low*, *moderate*, *substantial*, *severe*, *critical*, and *fatal*, respectively. Recall from Section 5.2.5 how costs are calculated for this discussion. Notice that costs can theoretically be negative in an `Environment` with a positive base cost that ensures the admissibility and consistency of heuristics. Negative cost could, therefore, indicate favorable or preferred areas. Furthermore, recall that the `RiskPolicy` is part of the operational goals and should not be exceeded as the airborne mission unfolds.

RiskPolicy-exceeding costs have to be avoided when computing a Trajectory. If the airborne agent is, however, unexpectedly exposed to RiskPolicy-exceeding costs, then a *fatal* Severity is assumed, and the responsible ManagedPlanner will have to boost the RiskPolicy temporarily to escape the perceived threat. Notice that, as opposed to the Difficulty categories, the Severtiy categories are not mutually exclusive. Therefore, particular Features may be associated with multiple Severity categories. For example, a cost vector of $(0, 0,$ RiskPolicy$)$ representing the cost assessment within the Aircraft safety sphere, the POI bounding box, and the Environment, respectively, is categorized to be *low*, *moderate* and *substantial* at the same time. This overlap is intentional as it may allow to associate more suitable PlannerTunings across different Severity categories. Chapter 11 shows pictures of evaluation scenarios that correspond to these Severity categories and visualize the same idea.

Table 9.3: Flight Phase

Flight Phase	Difficulty	Severity
cruise	$d(f) \in$ {low, moderate, substantial}	$s(f) \in$ {low, moderate, substantial}
transition	$d(f) \in$ {low, moderate, substantial}	$s(f) \in$ {severe}
terminal	$d(f) \in$ {severe, critical}	$s(f) \in$ {low, moderate, substantial, severe}
urgency	$\top$	$s(f) =$ critical
emergency	$\top$	$s(f) =$ fatal

Finally, a particular FlightPhase aggregates a number of Difficulty and Severity categories, as shown in Table 9.3. Since, Severity categories are not mutually exclusive, there is, consequently, overlap for FlightPhases as well. A FlightPhase covers Features within their horizon if these Features, in turn, are covered by at least one of the associated Difficulty and Severity categories. It should be emphasized that the associations shown in Table 9.3 are again the author's choice and our interpretation of the general Criticality of particular situations. In fact, a real-world flight phase, such as cruise, may obviously feature the full range of threat levels. However, we simply establish a reasonable mapping for the FlightPhase as FeatureCategory where the assigned names are irrelevant for the concrete AutonomicManager.

Regardless of the concrete interpretation, applied ranges or constraints, a FeatureCategory simply uses domain knowledge to categorize Features in order to achieve a similar PlannerPerformance for selected PlannerTunings. Therefore, an entirely different categorization of Features is possible.

```
1
2  ------------------------------------------------------------------------
3  aircraft altitude = 150.0
4  aircraft horizontal speed = 15.0
5  aircraft rate of climb = 2.0
6  aircraft rate of descent = 2.0
7  aircraft average obstacle distance = 22489.844831411352
8  aircraft maximum obstacle distance = 37292.46386824035
9  aircraft minimum obstacle distance = 6351.4405672079765
10 aircraft nearest obstacle cost = 40.0
11 aircraft nearest obstacle policies cost = 40.0
12 aircraft nearest obstacle volume = 1.5344180129087686E12
13 aircraft nearest obstacle volume ratio = 3.412360720438377E-7
14 aircraft safety sphere radius = 50.0
15 aircraft safety sphere volume = 523598.7755982988
16 aircraft safety sphere obstacle count = 1.0
17 aircraft safety sphere policies obstacle cost = 40.0
18 aircraft safety sphere average obstacle cost = 40.0
19 aircraft safety sphere maximum obstacle cost = 40.0
20 aircraft safety sphere minimum obstacle cost = 40.0
21 aircraft safety sphere accumulated obstacle cost = 40.0
22 aircraft safety sphere average obstacle volume = 1.5530106408658975E12
23 aircraft safety sphere maximum obstacle volume = 1.5716032688230261E12
24 aircraft safety sphere minimum obstacle volume = 1.5344180129087686E12
25 aircraft safety sphere accumulated obstacle volume = 1.5530106408658975E12
26 ------------------------------------------------------------------------
27 environment diameter = 124208.5824490695
28 environment volume = 7.413751454563458E13
29 environment obstacle count = 3.0
30 environment policies obstacle cost = Infinity
31 environment average obstacle cost = 46.666666666666664
32 environment maximum obstacle cost = 60.0
33 environment minimum obstacle cost = 40.0
34 environment accumulated obstacle cost = 140.0
35 environment average obstacle volume = 1.4892727468056912E12
36 environment maximum obstacle volume = 3.157983617564281E12
37 environment minimum obstacle volume = 5.911794173661902E11
38 environment accumulated obstacle volume = 5.302173675796367E12
39 environment obstacle volume ratio = 0.07151809321221134
40 ------------------------------------------------------------------------
41 consecutive POIs average distance = 52635.939797789586
42 consecutive POIs maximum distance = 70894.44238347997
43 consecutive POIs minimum distance = 34377.43721209921
44 consecutive POIs accumulated distance = 105271.87959557917
45 consecutive POIs average vertical distance = 425.0
46 consecutive POIs maximum vertical distance = 700.0
47 consecutive POIs minimum vertical distance = 150.0
48 consecutive POIs accumulated vertical distance = 850.0
49 POIs bounding box diameter = 71047.13536493029
50 POIs bounding box volume = 1.4117217677411138E12
51 POIs bounding box obstacle count = 1.0
52 POIs bounding box policies obstacle cost = 40.0
53 POIs bounding box average obstacle cost = 40.0
54 POIs bounding box maximum obstacle cost = 40.0
55 POIs bounding box minimum obstacle cost = 40.0
56 POIs bounding box accumulated obstacle cost = 40.0
57 POIs bounding box average obstacle volume = 1.5530106408658975E12
58 POIs bounding box maximum obstacle volume = 1.5716032688230261E12
59 POIs bounding box minimum obstacle volume = 1.5344180129087686E12
60 POIs bounding box accumulated obstacle volume = 1.5530106408658975E12
61 POIs bounding box obstacle volume ratio = 1.1000826624291975
62 POIS bounding box environment obstacles count ratio = 0.3333333333333333
63 POIs bounding box environment policies obstacle cost ratio = 0.0
64 POIs bounding box environment average obstacle cost ratio = 0.8571428571428572
65 POIs bounding box environment maximum obstacle cost ratio = 0.6666666666666666
66 POIs bounding box environment minimum obstacle cost ratio = 1.0
67 POIs bounding box environment accumulated obstacle cost ratio = 0.2857142857142857
68 POIs bounding box environment obstacles volume ratio = 0.292900748980585
69 POIs bounding box environment volume ratio = 0.01904193546807053
70 ------------------------------------------------------------------------
71 operational cost policy = 2.0
72 operational risk policy = 50.0
   ------------------------------------------------------------------------
```

Listing 9.5: SAFCS AM: Features Instance

Listing 9.5 shows a `Features` instance covering a 5 minutes horizon for the `Scenario` depicted in Figure 9.5. The example shows several `Obstacles` featuring different volumes and costs. Furthermore, it can be seen that the `Environment`, the POI bounding box, and the `Aircraft` safety sphere are affected differently by the `Obstacles` in this `Scenario`.



(a) SAFCS UI: Features Instance **(b)** SAFCS UI: Features Horizon

Figure 9.5: SAFCS UI: Features Instance

More details about `Features`,[2] `FeatureCategory`,[3] `Criticality`,[4] and `FlightPhase`[5] can be found in our SAFCS software repository.

9.3 Tuning Parameters

The `Tuning` parameters, that is, the `PlannerTunings` of `ManagedPlanners`, are the *actuators* (or *effectors*) of the SAFCS AM to effect its performance within its current AC. Thus, `PlannerTunings` together with the *selection* of active and standby `ManagedPlanners` are essential to realize the concept of the SAFCS as a self-adaptive AE.

[2] https://github.com/stephanheinemann/worldwind/blob/develop/src/main/java/com/cfar/swim/worldwind/managing/Features.java
[3] https://github.com/stephanheinemann/worldwind/blob/develop/src/main/java/com/cfar/swim/worldwind/managing/FeatureCategory.java
[4] https://github.com/stephanheinemann/worldwind/blob/develop/src/main/java/com/cfar/swim/worldwind/managing/Criticality.java
[5] https://github.com/stephanheinemann/worldwind/tree/develop/src/main/java/com/cfar/swim/worldwind/flight

Recall from Section 5.3.3 that a PlannerTuning aggregates both a Features instance, such as in Listing 9.5, and a Planner Specification. The Planner Specification, in turn, can be identified by the identifier of its FactoryProduct, that is, its ManagedPlanner, and aggregates the corresponding PlannerProperties as described in Section 5.2.1. It is precisely the concrete PlannerProperties that comprise the parameter configuration of its corresponding ManagedPlanner. Therefore, it should be emphasized that a Specification cannot only be used to *create* a concrete FactoryProduct via a capable Factory but also to *update* an existing FactoryProduct. Exactly this mechanism is employed by the SAFCS AutonomicManagers to update the PlannerProperties of their ManagedPlanners after a suitable Tuning has been established.

The tuning process itself is desribed for the dedicated HeuristicPlannerTuning and SmacPlannerTuning in Sections 9.6 and 9.7, respectively. However, the individual configuration parameters contained by the concrete PlannerProperties, their effects and dependencies shall be reviewed briefly. As explained in Section 9.1, the currently supported ManagedPlanners are the ManagedGridPlanner, the ManagedTreePlanner, and the ManagedRoadmapPlanner. Recall that each of these ManagedPlanners extends the most sophisticated key algorithm of its associated family, that is, CGS, RRTs, and PRMs, respectively. As such, the relevant configuration parameters to be tuned are the ones associated with each of these key algorithms and summarized in Table 9.4. The parameters related to operational policies and tolerances of the corresponding OnlinePlanners are ignored as they are not being tuned but merely forwarded by the AutonomicManager.

The semantics of the individual configuration parameters are explained in Chapters 6, 7, and 8, respectively. Many parameters relate to the expansion, exploration versus exploitation, and incremental improvement while others are more algorithm-specific and consider preferred regions or certain techniques and strategies. Note, however, that although some parameters are equally named, they may be constrained and interpreted differently across the supported ManagedPlanners. Furthermore, certain combinations of parameter values are unlikely to succeed in finding a solution, that is, a feasible and high-quality Trajectory, for a given problem, that is, a particular Features instance. As an example, consider the parameters epsilon and goalThreshold. The maximum expansion distance epsilon obviously needs to consider the POIs distances to be covered and, thus, depends on the extracted Features from the active Scenario of a managed planning Session. Furthermore, a large goal threshold distance goalThreshold,

Table 9.4: Tuning Parameters

ManagedPlanner	Parameter	Constraints
ManagedGridPlanner	maximumQuality $\in \mathbb{R}$	maximumQuality ≥ 1.0
	minimumQuality $\in \mathbb{R}$	minimumQuality $\geq$ maximumQuality
	qualityImprovement $\in \mathbb{R}$	qualityImprovement > 0.0
	significantChange $\in \mathbb{R}$	significantChange $\in [0.0, 1.0]$
ManagedTreePlanner	bias $\in \mathbb{Z}$	bias $\in [1, 100]$
	epsilon $\in \mathbb{R}$	epsilon > 0.0
	extension $\in$ {LINEAR, FEASIBLE}	$\top$
	goalThreshold $\in \mathbb{R}$	goalThreshold ≥ 0.0
	maxIterations $\in \mathbb{Z}$	maxIterations ≥ 1
	maximumQuality $\in \mathbb{R}$	minimumQuality $\leq$ maximumQuality ≤ 1.0
	minimumQuality $\in \mathbb{R}$	$0.0 \leq$ minimumQuality $\leq$ maximumQuality
	neighborLimit $\in \mathbb{Z}$	neighborLimit ≥ 1
	qualityImprovement $\in \mathbb{R}$	qualityImprovement > 0.0
	sampling $\in$ {UNIFORM, GAUSSIAN, ELLIPSOIDAL}	$\top$
	significantChange $\in \mathbb{R}$	significantChange $\in [0.0, 1.0]$
	strategy $\in$ {EXTEND, CONNECT}	$\top$
ManagedRoadmapPlanner	bias $\in \mathbb{Z}$	bias $\in [1, 100]$
	epsilon $\in \mathbb{R}$	epsilon > 0.0
	lambda $\in \mathbb{R}$	lambda $\in [0.0, 1.0]$
	maximumQuality $\in \mathbb{R}$	minimumQuality $\leq$ maximumQuality ≤ 1.0
	minimumQuality $\in \mathbb{R}$	$0.0 \leq$ minimumQuality $\leq$ maximumQuality
	queryMode $\in$ {SINGLE, MULTIPLE}	$\top$
	neighborLimit $\in \mathbb{Z}$	neighborLimit ≥ 1
	qualityImprovement $\in \mathbb{R}$	qualityImprovement > 0.0
	significantChange $\in \mathbb{R}$	significantChange $\in [0.0, 1.0]$

in particular one that is larger than `epsilon`, would be an inappropriate choice and may lead to a low-quality `Trajectory`.

Applying such domain knowledge about configuration parameters leads to the application of heuristics to form the building blocks of the `HeuristicAutonomicManager` explained in Section 9.6. Human experience can, furthermore, restrict the given constraints for each parameter in order to reduce the parameter configuration space to be searched by the `SmacAutonomicManager` explained in Section 9.7.

More details about the `Tuning`[6] and `PlannerTuning`[7] in particular can be found in our SAFCS software repository.

[6] https://github.com/stephanheinemann/worldwind/blob/develop/src/main/java/com/cfar/swim/worldwind/managing/Tuning.java

[7] https://github.com/stephanheinemann/worldwind/blob/develop/src/main/java/com/cfar/swim/worldwind/managing/PlannerTuning.java

9.4 Knowledge Base

The KnowledgeBase of the SAFCS AM component is a persistable Reputation that maps PlannerTunings to Collections of PlannerPerformances. Whereas recorded PlannerPerformances serve as the short-term *sensors* of the employed AutonomicManager itself, the persistable KnowledgeBase represents its long-term *sensor*. Together, both establish the required *self-awareness* on the primitive level required for *self-configuration* and *self-optimization* on the major level of the AE, as shown in Figure 2.13.

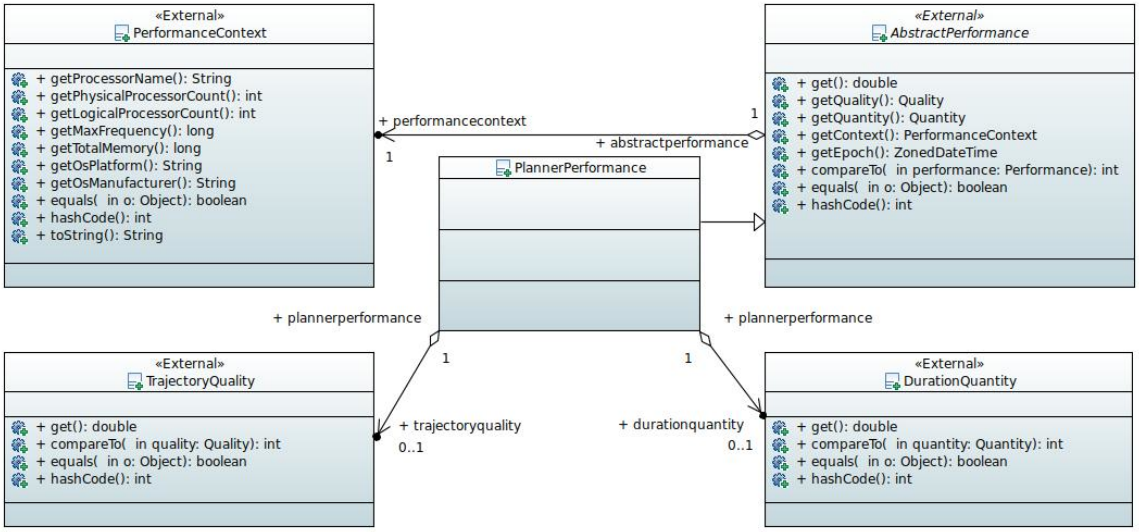


Figure 9.6: SAFCS AM: Performance Signature

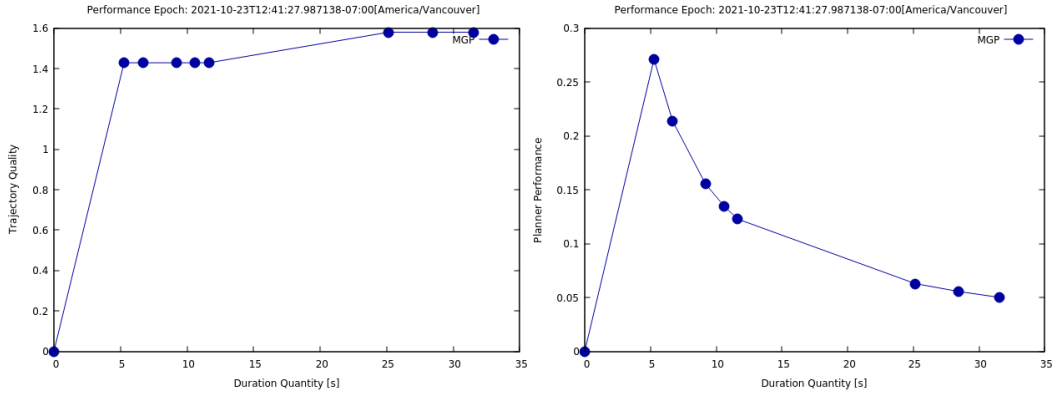
Recall from Section 5.3.4 that a PlannerPerformance comprises a TrajectoryQuality and a DurationQuantity. Figure 9.6 refines the diagram shown in Figure 5.30 and reveals additional aggregations as well as parts of the associated functional signatures. Note that all AbstractPerformances, such as the PlannerPerformance, are measured within a PerformanceContext. The PerformanceContext captures the system environment in which the concrete Performance has been achieved. Obviously, the measured PlannerPerformance depends on factors such as processor architecture, memory, and operating system. Associating Performances with their PerformanceContext, therefore, allows to persist, filter and compare Reputations of ManagedPlanners even if the KnowledgeBase is transferred between different execution environments. The presently supported set of

captured contextual data is considered sufficient to identify similar environments but can be extended easily. An example `PerformanceContext` is shown in Table 9.5.

Table 9.5: Performance Context

Performance Context Property Name	Performance Context Property Value
processor name	Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz
physical processor count	4
logical processor count	8
maximum logical processor frequency [Hz]	3400000000
total physical memory [bytes]	12483780608
operating system platform	LINUX
operating system manufacturer	GNU/Linux

All `AbstractPerformances`, furthermore, aggregate an `epoch`. The `epoch` can be any time instance in the past, but its purpose is to identify `PlannerPerformances` that originate from the same `ManagedPlanner` invocation. To this end, line 40 of Listing 9.1 sets the `epoch` to the `planStartTime` of the `ManagedPlanner`. Hence, all incremental revisions of a `Trajectory` and their associated `PlannerPerformances` can be joined to analyze them as part of the same invocation. Figure 9.7 shows an example of a `PlannerPerformance` plot representing multiple `Trajectory` revisions during a single invocation for a given set of `Features`.



(a) SAFCS AM: Trajectory Quality **(b)** SAFCS AM: Planner Performance

Figure 9.7: SAFCS AM: Performance Plot

Whereas the diagram in Figure 9.7a plots the `TrajectoryQuality` of consecutive `Trajectory` revisions published at the corresponding `DurationQuantity`, the

diagram in Figure 9.7b shows the resulting PlannerPerformance as the ratio of TrajectoryQuality over DurationQuantity. In this concrete example, the gained TrajectoryQuality over several Trajectory revisions is insufficient to maintain a constant PlannerPerformance after issuing the first Trajectory at approximately 5 seconds.

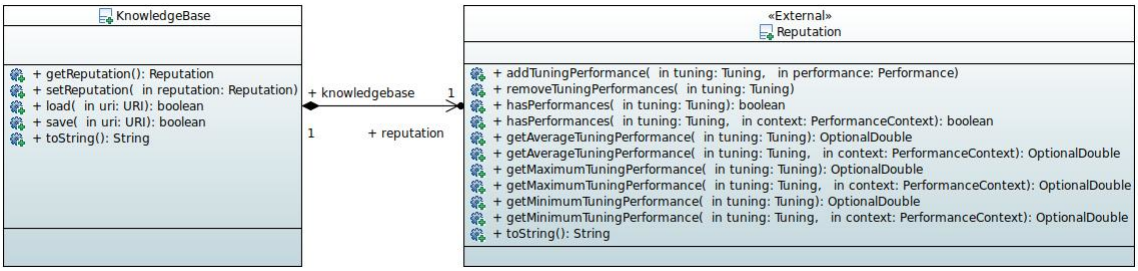


Figure 9.8: SAFCS AM: Reputation Signature

Figure 9.8 reveals the functional signatures of the SAFCS AM KnowledgeBase and its aggregated Reputation. As mentioned before, the KnowledgeBase and its contents can be persisted and shared between different independent managed planning Sessions and execution environments. The corresponding load and save methods are invoked by an AutonomicManager during its initialization and cleanup phases, respectively, to gain access to and extend the contained Reputation.

The Reputation itself offers several methods to access Performances that are associated with particular Tunings. Recall from Sections 5.3.3 and 9.3 that a Tuning, serving as the key of the Reputation, comprises both a Specification and Features. Thus, the Reputation associates Specifications with their Properties, that have been applied to Features, with the resulting Performances. It should be emphasized that it is a Collection of Performances that is stored for a particular Tuning since ManagedPlanners are not just only AnytimePlanners but also not necessarily deterministic. The consecutive invocations of a ManagedTreePlanner given a corresponding Planner Specification and PlannerTuning, for instance, most likely results in different PlannerPerformances when applied to the same Features. This statement is obviously true for all ManagedPlanners that represent key algorithms of probabilistic Planner families, such as RRTs and PRMs. Therefore, the Reputation offers additional methods to obtain average, maximum, and minimum Performances for a given Tuning.

The mentioned methods are, furthermore, available to query the Tuning Performances for a particular PerformanceContext only. Recall that the

KnowledgeBase stores Performances for different PerformanceContexts. Hence, an AutonomicManager needs to be able, to filter the Reputation for relevant and comparable Performances matching its present execution environment.

Although our SAFCS AM Reputation presently stores Tunings and associated Performances for ManagedPlanners only, other Tunings, such as EnvironmentTunings, are planned as future extensions and revisited in Chapter 15. The hierarchy of a HierarchicalEnvironment as well as the resolution of a MultiResultionEnvironment may significantly affect the Performance of a ManagedPlanner and should be part of an extended *effector* set.

More details about the KnowledgeBase[8] and Reputation[9] can be found in our SAFCS software repository.

9.5 Autonomic Managers

All concrete AutonomicManagers follow the same conceptual workflow, offer similar functionalities and differ mainly in *how* tuning is performed based on the sensed Features. Figure 9.9 exposes important parts of the functional signature of an AbstractAutonomicManager serving as the baseline implementation of all concrete AutonomicManagers and refines the diagram shown in Figure 5.25.

All AutonomicManagers manage a planning Session and, hence, the entry method to the manager's life cycle is called manage (similar to plan for Planners or call for ManagedPlanners) and is passed a reference to the managed Session either directly, or in form of an unique identifier. The managing life cycle consists of the initialize, run, and cleanup phases. Each phase is associated with a corresponding method, as shown in Figure 9.9.

The commented abstract managing life cycle shown in Figure 9.10 assumes that the concrete AbstractAutonomicManager is *executing* its managing functions based on the active source Scenario of the managed planning Session. More precisely, the active source Scenario describes the problem to be solved as well as contains the POIs representing the ManagedGoals to be accomplished. This is opposed to a concrete AbstractAutonomicManager that performs *training* in order to extend its

[8] https://github.com/stephanheinemann/worldwind/blob/develop/src/main/java/com/cfar/swim/worldwind/managing/KnowledgeBase.java

[9] https://github.com/stephanheinemann/worldwind/blob/develop/src/main/java/com/cfar/swim/worldwind/managing/Reputation.java

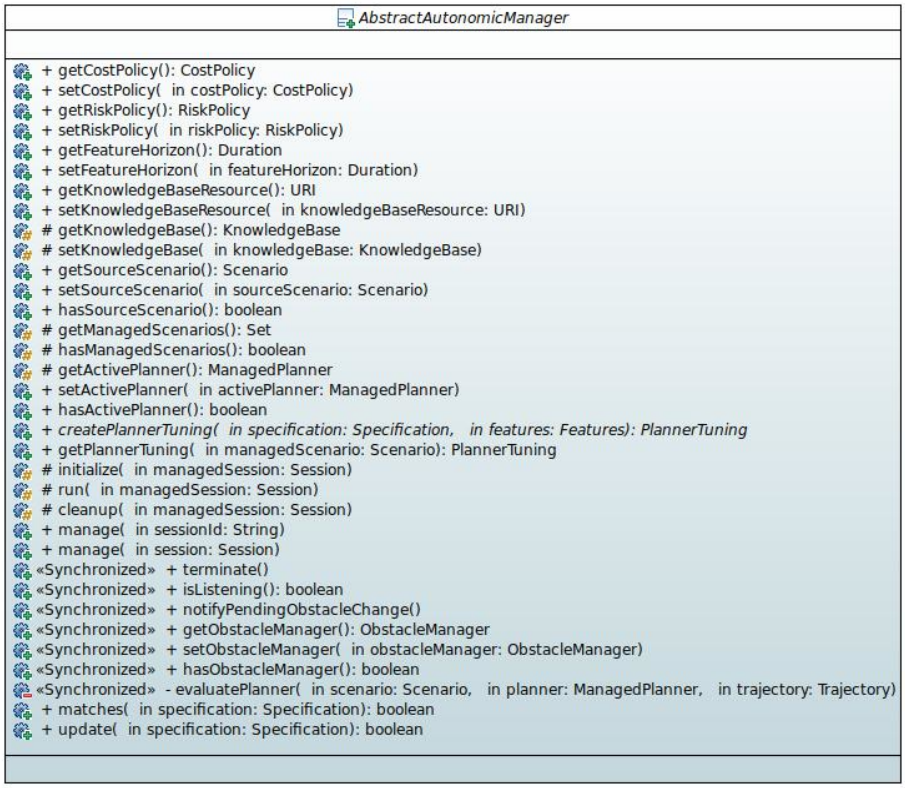


Figure 9.9: SAFCS AM: Abstract Autonomic Manager Signature

`KnowledgeBase`, which is not necessarily supported by all `AutonomicManagers`, but follows the same life cyle.

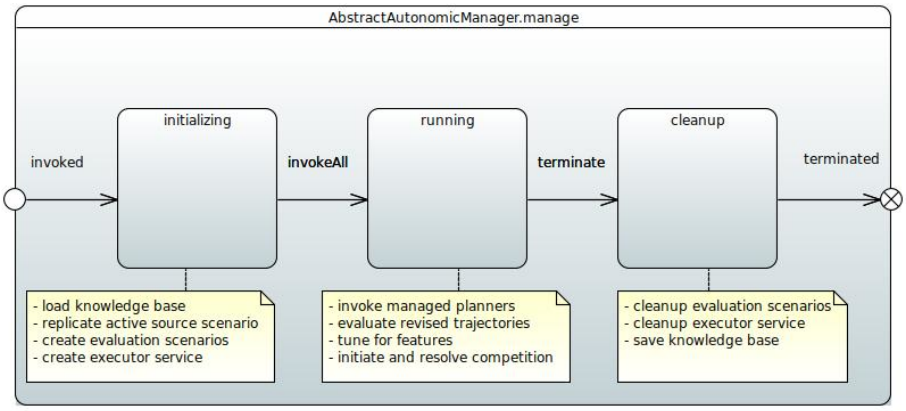


Figure 9.10: SAFCS AM: Abstract Autonomic Manager Life Cycle

The `initialize` phase of an *executing* `AbstractAutonomicManager` consists of loading the persisted `KnowledgeBase` and creating *evaluation* `Scenarios` that replicate

the ManagedGoals of the active source Scenario of the managed planning Session. Each evaluation Scenario is equipped with one of the registered ManagedPlanners and a supported Environment of the former. Thus, if ManagedPlanners support several planning Environments, then an evaluation Scenario is created for each of the valid combinations. The AbstractAutonomicManager then registers a PlanRevisionListener with each of the evaluated ManagedPlanners to obtain Trajectory revisions for subsequent evaluations. Furthermore, a PlannerTuning is created for the ManagedPlanners of each evaluation Scenario and tuned for the initial Features of the active source Scenario. Note that createPlannerTuning is the only abstract method shown in the method signature of the AbstractAutonomicManager in Figure 9.9. It is precisely the PlannerTuning that differs between concrete implementations of of the AbstractAutonomicManager and it is discussed in more detail in Sections 9.6 and 9.7 for that reason. Lastly, given that evaluation Scenarios have been created, an ExecutorService is instantiated that is capable of assigning them to separate processor cores in order to achieve a fair evaluation.

The run phase of an *executing* AbstractAutonomicManager triggers the invocation of all ManagedPlanners in order to achieve their ManagedGoals. Whenever a new or revised Trajectory is published by one of them, the associated PlannerPerformance is evaluated. Listing 9.6 shows the individual steps of this performance evaluation.

First, in lines 4 to 6, the currently active Tuning of the ManagedPlanner is obtained and its associated PlannerPerformance stored in the Reputation of the KnowledgeBase. After styling and showing the published Trajectory in the evaluation Scenario (lines 8 and 9), the currently best Trajectory of the active source Scenario (line 10) of the managed Session is compared with the former (lines 14 to 31). In order to compare trajectories fairly, only their common parts are taken into account. Thus, the comparison of the new (multi-part) Trajectory with the currently best one is constrained to the same start and covered intermediate POIs. Although it may have taken longer to compute an earlier part or better overall solution, the highest TrajectoryQuality is always considered for an update of the active source Scenario (line 32). More precisely, the published Trajectory replaces the currently best Trajectory if it is not empty, has a common start with the currently best Trajectory, and one of the following conditions is met (lines 29 to 31):

- the currently best Trajectory is empty, or

```
private synchronized void evaluatePlanner(
        Scenario scenario, ManagedPlanner planner, Trajectory trajectory) {
    // extending knowledge base
    PlannerTuning tuning = this.getPlannerTuning(scenario);
    this.getKnowledgeBase().getReputation().addTuningPerformance(
        tuning, planner.getPerformance());

    TrajectoryStylist.styleTrajectory(trajectory);
    scenario.setTrajectory(trajectory);
    Trajectory sourceTrajectory = getSourceScenario().getTrajectory();

    // comparison of trajectories can only be based on the same start
    // and end POIs using the same environment normalizer
    List<Waypoint> spois = sourceTrajectory.getPois();
    List<Waypoint> rpois = trajectory.getPois();
    int poiIndex = Math.min(spois.size(), rpois.size()) - 1;

    Trajectory st = sourceTrajectory.getSubTrajectory(poiIndex);
    TrajectoryQuality stq = new TrajectoryQuality(st);
    Trajectory et = trajectory.getSubTrajectory(poiIndex);
    TrajectoryQuality rtq = new TrajectoryQuality(et);

    boolean commonStart = true;
    if (!st.isEmpty() && !et.isEmpty()) {
        commonStart = st.getFirstWaypoint().equals(et.getFirstWaypoint());
    }

    // consider non-empty, improved, or longer revised trajectory
    if (!trajectory.isEmpty() && commonStart && (sourceTrajectory.isEmpty()
            || (0 > stq.compareTo(rtq))
            || ((0 == stq.compareTo(rtq)) && (rpois.size() >= poiIndex)))) {
        getSourceScenario().setTrajectory(trajectory);

        // select improved planner if necessary
        if (getActivePlanner() !=  planner) {
            getActivePlanner().setStandby(true);
            setActivePlanner(planner);
            getActivePlanner().setStandby(false);
            getActivePlanner().getMissionLoader().revisePlan(trajectory);
        }
    }
}
```

Listing 9.6: SAFCS AM: Performance Evaluation

```
1  public synchronized void notifyPendingObstacleChange() {
2      this.tunerExecutor.execute(new Runnable() {
3          public void run() {
4              if (hasObstacleManager()
5                      && !getObstacleManager().commitObstacleChange().isEmpty()) {
6                  getSourceScenario().clearTrajectory();
7
8                  // update all managed scenarios and planner tunings
9                  for (Scenario managedScenario : getManagedScenarios()) {
10                     PlannerTuning tuning = getPlannerTuning(managedScenario);
11                     tuning.getFeatures().extractFeatures(getSourceScenario());
12                     List<Properties<Planner>> candidates = tuning.tune();
13                     tuning.getSpecification().setProperties(candidates.get(0));
14
15                     ManagedPlanner managedPlanner = (ManagedPlanner)
16                         managedScenario.getPlanner();
17                     managedPlanner.update(tuning.getSpecification());
18                     if (getActivePlanner() != managedPlanner) {
19                         // join active planner in trajectory competition
20                         managedPlanner.join(getActivePlanner());
21                     }
22
23                     managedScenario.submitReplaceObstacles(
24                             getSourceScenario().getObstacles());
25                 }
26             }
27         }
28     });
29 }
```

Listing 9.7: SAFCS AM: Tuning and Competition Joining

- the new or revised `Trajectory` has a higher `TrajectoryQuality` constrained to common parts, or
- the new or revised `Trajectory` has the same `TrajectoryQuality` constrained to common parts but is longer and, hence, closer to achieving the `ManagedGoals`.

The `ManagedPlanner` delivering the highest quality `Trajectory` based on these conditions subsequently becomes the leading *active* `Planner` (lines 35 to 38) and its published `Trajectory` is uploaded to the `Datalink` target via its `MissionLoader` (line 39). The active `ManagedPlanner` remains active until a higher `TrajectoryQuality` is published, possibly following a restarted competition after a `Features` update of the AC.

The `AutonomicManager` is notified about pending `Obstacle` changes in the active source `Scenario` of its managed planning `Session`, as shown in Listing 9.7. As a result of such a dynamic change, the currently best `Trajectory` of the source `Scenario` is cleared (line 6). Then, the updated `Features` of the modified source `Scenario` are extracted, and `PlannerTunings` associated with the `ManagedPlanners` tuned accordingly (lines 10 to 17). Recall from Sections 5.3.3 and 9.3 that the concrete `PlannerTuning`,

such as the HeuristicPlannerTuning and the SmacPlannerTuning, determine how their associated ManagedPlanner is tuned for certain Features.

Following the extraction of Features, the tuning of PlannerTunings, and the the update of tuned Planner Specifications, the competition between the ManagedPlanners of the AbstractAutonomicManager is initiated again. As explained in Section 9.1, the currently leading *active* ManagedPlanner may proceed with the repair of its established plan immediately (lines 23 to 24). The currently following *standby* ManagedPlanners, on the other hand, are first instructed to join their associate in the competition (lines 18 to 21), by re-planning, before continuing to elaborate on their solutions. Elaboration again leads to the revision of computed solutions and the associated performance evaluation of Listing 9.6. Since ManagedPlanners are LifelongPlanners, the process repeats until explicitly terminated. It consists of continuously sensing and extracting changed Features from the AC, tuning ManagedPlanners accordingly, and selecting the best of the competitors to upload its solution via the Datalink in order to achieve the ManagedGoals. During this process, the KnowledgeBase is extended continuously and may improve the performance of an AutonomicManager in future Sessions. The run phase of the AbstractAutonomicManager, therefore, realizes the concept of the adaptation loop shown in Figure 2.12.

Finally, after the explicitly requested termination, the cleanup phase of an *executing* AbstractAutonomicManager removes all evaluation Scenarios, shuts down its ExecutorService, and stores the extended KnowledgeBase for future use.

More details about the AutonomicManager[10] and the abstract derived AbstractAutonomicManager[11] can be found in our SAFCS software repository.

9.6 Heuristic Autonomic Manager

The HeuristicAutonomicManager is the simplest concrete extension of the currently supported AutonomicManagers. It behaves exactly as the AbstractAutonomicManager described in Section 9.5 without overriding its functionality except for the required concrete createPlannerTuning and being able to identify itself. Therefore, the

[10] https://github.com/stephanheinemann/worldwind/blob/develop/src/main/java/com/cfar/swim/worldwind/managers/AutonomicManager.java

[11] https://github.com/stephanheinemann/worldwind/blob/develop/src/main/java/com/cfar/swim/worldwind/managers/AbstractAutonomicManager.java

HeuristicAutonomicManager features the fairly slim additional method signature shown in Figure 9.11.

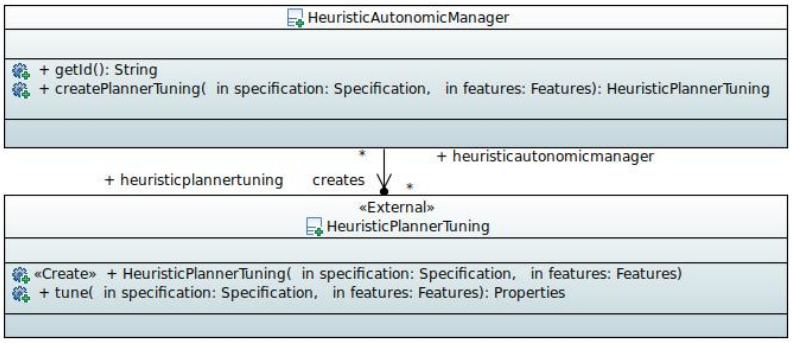


Figure 9.11: SAFCS AM: Heuristic Autonomic Manager Tuning

All concrete AutonomicManagers are required to provide their associated concrete PlannerTuning and, hence, the HeuristicAutonomicManager creates a HeuristicPlannerTuning based on a ManagedPlanner Specification and a set of Features. The correspondingly tuned initial Properties can be obtained by invoking the inherited parameterless method tune of its abstract PlannerTuning shown in Figure 5.29. The tuned Properties of the ManagedPlanner for modified Features can subsequently be obtained by invoking the corresponding parameterized method if necessary. Recall from Section 9.5 how modified Features are extracted from the source Scenario of the managed planning Session. Listing 9.7 shows the extraction of these Features to modify the Tuning and set the tuned Properties of the ManagedPlanner in lines 10 to 17.

The HeuristicAutonomicManager implements a trivial *model-free* approach that only supports *execution*. As such, it does not create an internal model of the AC, nor does it make use of the extended KnowledgeBase. Its HeuristicPlannerTuning rather simply applies heuristics to create presumably suitable Properties for its ManagedPlanners based on the sensed Features. Without showing all of the currently implemented heuristics, a few examples of the ManagedTreePlanner shall be provided here.

The first heuristic for the ManagedTreePlanner is shown in Listing 9.8 and covers an Obstacle-free Environment. The Properties in this situation are tuned to achieve a rapid progress toward the goal. In particular, the maximum extension distance epsilon is set to the maximum POI distance as extracted from the Features and a CONNECT Strategy is selected to reach the goal in possibly a single step. No goal threshold is

```
// ...
// features without obstacles
if (0d == features.get(Features.FEATURE_POIS_OBSTACLES_COUNT)) {
    // sampling
    properties.setSampling(Sampling.ELLIPSOIDAL);
    // goal bias
    // TODO: consider lower bias (feature horizon may not cover all obstacles)
    properties.setBias(100); // ignore terrain and capabilities
    // epsilon - extension distance
    properties.setEpsilon(features.get(Features.FEATURE_POIS_DISTANCE_MAX));
    // extension strategy and technique
    properties.setStrategy(Strategy.CONNECT);
    properties.setExtension(Extension.FEASIBLE);
    // goal threshold
    properties.setGoalThreshold(0d);
    // maximum iterations
    // TODO: consider more iterations (feature horizon may not cover all obstacles)
    properties.setMaxIterations(1); // ignore terrain and capabilities
    // neighborhood
    properties.setNeighborLimit(1);
    // quality
    properties.setMinimumQuality(0.0d);
    properties.setMaximumQuality(0.0d);
    properties.setQualityImprovement(0.001d);
    properties.setSignificantChange(0.01d);
} else { // ...
```

Listing 9.8: SAFCS AM: Obstacle-Free Heuristic Tuning

accounted for. The neighborhood is reduced to a minimum. The minimum initial quality equals the maximum final quality and, thus, no incremental solutions are required. Optimistically disregarding possible terrain, which could potentially be excluded by AIXM `Obstacles` but is currently not captured as part of the `Features`, and `Aircraft Capabilities`, the goal bias is set to 100% and the number of iterations equals 1. This is clearly an optimistic example which, in particular, ignores the fact that an extracted set of `Features` within a horizon may not cover all relevant `Features` in space and time. A less optimistic tuning would obviously be the better and more flexible choice.

Listing 9.9 shows some of the heuristics applied to the `ManagedTreePlanner` in case of an `Obstacle`-cluttered `Environment`. In this example, both the `Sampling` and goal bias values are based on the derived `FlightPhase` of the `Scenario` `Features` within the horizon. The maximum extension distance `epsilon` is reduced to 5% of the maximum POI distance of the extracted `Features`. Depending on the sensed `Difficulty` of the `Scenario`, a `CONNECT Strategy` is selected in the *low* or *moderate* case. Otherwise an `EXTEND Strategy` is assumed to be more successful. Similarly, the number of iterations depends on the derived `FlightPhase` and increases from 500 for *cruise* to a maximum of 2000. The more difficult the `Scenario`, the more sampling neighbors are assumed to result in a more successful search. The neighborhood of samples is, therefore, correspondingly adjusted to a minimum of 1 and a maximum of 5 neighbors.

It should be emphasized that these and other implemented heuristics are simply the author's choice and the result of experimentation (Ferreira [BF19] and Rosa [dCR19]). They are not intended to represent or approach an optimal `PlannerTuning`. However, they may serve as a `Performance` baseline and deliver an initial *incumbent* configuration for more sophisticated `AutonomicManagers` that support a *training* mode, such as the `SmacAutonomicManager`.

More details about the `HeuristicAutonomicManager`[12] and related classes can be found in our SAFCS software repository.

[12] https://github.com/stephanheinemann/worldwind/tree/develop/src/main/java/com/cfar/swim/worldwind/managers/heuristic

```
// ...
} else {
    // features with obstacles
    // sampling
    if (FlightPhase.areCruise(features) || FlightPhase.areTransition(features)) {
        properties.setSampling(Sampling.ELLIPSOIDAL);
    } else {
        properties.setSampling(Sampling.UNIFORM);
    }

    // goal bias
    if (FlightPhase.areCruise(features)) {
        properties.setBias(10);
    } else if (FlightPhase.areTransition(features)) {
        properties.setBias(5);
    } else if (FlightPhase.areTerminal(features)) {
        properties.setBias(3);
    } else {
        properties.setBias(1);
    }

    // epsilon - extension distance
    properties.setEpsilon(0.05d * features.get(Features.FEATURE_POIS_DISTANCE_MAX));

    // goal threshold
    properties.setGoalThreshold(properties.getMaxLandingHorizontalError() / 2d);

    // extension strategy and technique
    if (Difficulty.areLow(features) || Difficulty.areModerate(features)) {
        properties.setStrategy(Strategy.CONNECT);
    } else {
        properties.setStrategy(Strategy.EXTEND);
    }
    properties.setExtension(Extension.FEASIBLE);

    // maximum iterations
    // reduce search time in less critical situations for multi-part problems
    if (FlightPhase.areCruise(features)) {
        properties.setMaxIterations(500);
    } else if (FlightPhase.areTransition(features)) {
        properties.setMaxIterations(1000);
    } else if (FlightPhase.areTerminal(features)) {
        properties.setMaxIterations(1500);
    } else {
        properties.setMaxIterations(2000);
    }

    // neighborhood
    if (Difficulty.areLow(features)) {
        properties.setNeighborLimit(1);
    } else if (Difficulty.areModerate(features)) {
        properties.setNeighborLimit(2);
    } else if (Difficulty.areSubstantial(features)) {
        properties.setNeighborLimit(3);
    } else if (Difficulty.areSevere(features)) {
        properties.setNeighborLimit(4);
    } else {
        properties.setNeighborLimit(5);
    }

    // ...
}
```

Listing 9.9: SAFCS AM: Obstacle-Cluttered Heuristic Tuning

9.7 SMAC Autonomic Manager

Recall from Section 3.3 that SMAC was introduced by Hutter, Hoos, and Leyton-Brown [HHLB11], and subsequently implemented as Python[13] and Java[14] libraries. We have modified and packaged the Java version and its dependencies as required to make them available as Maven[15] artifacts. This allows for easier dependency management and integration into our SAFCS. The following discussion is limited to the integration of SMAC into the SAFCS, and more implementation-specific details can be found in the available SMAC user manual[16] and on the respective software repositories.

SMAC evaluates an algorithm on a set of `ProblemInstances` having certain `Features` to obtain a *best-on-average* `ParameterConfiguration` using a `TargetAlgorithmEvaluator`. For our problem domain and the integration into our SAFCS AM, the algorithm becomes the `ManagedPlanner`, a `ProblemInstance` is represented by a planning `Scenario` and its extracted `Features`, and the `ParameterConfiguration` is part of our `PlannerTuning`—more precisely, the `Properties` of its `Specification`. Finally, a `SmacPlannerEvaluator` extends the `TargetAlgorithmEvaluator` class hierarchy and performs the evaluation runs.

The `SmacAutonomicManager` is a second and more sophisticated *model-based* `AutonomicManager` of the SAFCS AM component that supports both the *executing* mode available to all concrete `AbstractAutonomicManagers` as well as a *training* mode to extend its `KnowledgeBase` and potentially perform better while *executing* subsequently. Therefore, the `SmacAutonomicManager` aggregates a `SmacManagerMode` that can be either `EXECUTING` or `TRAINING`, as shown in Figure 9.12.

While operating in `TRAINING` mode, the `SmacAutonomicManager` adheres to the same life cycle of an `AbstractAutonomicManager` explained in Section 9.5. Recall that this life cycle comprises the `initialize`, `run`, and `cleanup` phases. However, whereas the `SmacAutonomicManager` reverts to the behavior of the `AbstractAutonomicManager` while `EXECUTING`, it performs different steps in each of the phases while `TRAINING`.

The `initialize` phase of a `SmacAutonomicManager` in `TRAINING` consists of instantiating and configuring `SequentialModelBasedAlgorithmConfigurations` for all registered `ManagedPlanners` that support any of the available training `Scenarios`. Put differently, instead of replicating the currently active `Scenario` as the source

[13] https://github.com/automl/SMAC3
[14] https://github.com/SJrX/SMAC
[15] https://search.maven.org/search?q=com.uvic-cfar.swim.ubc.cs
[16] https://www.cs.ubc.ca/labs/beta/Projects/SMAC/

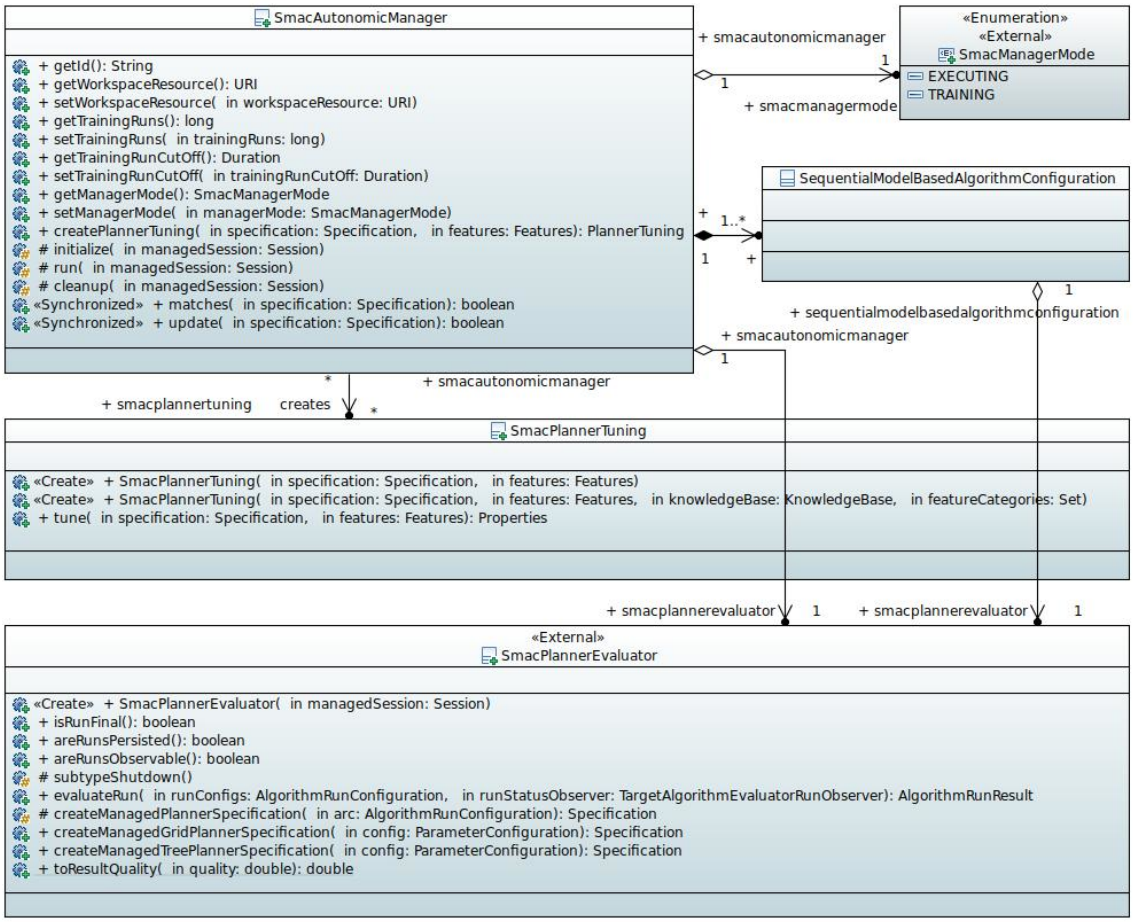


Figure 9.12: SAFCS AM: SMAC Autonomic Manager Tuning

`Scenario` of the managed planning `Session`, *all* currently defined `Scenarios` are employed for training using the registered and supporting `ManagedPlanners` available. Since training is performed for each of each of these `ManagedPlanners` on their supported `Scenarios` representing a set of `ProblemInstances`, exactly one dedicated `SequentialModelBasedAlgorithmConfiguration` instance is instantiated and configured per `ManagedPlanner`.

The configuration of a `SequentialModelBasedAlgorithmConfiguration` consists of defining appropriate `SMACOptions`. These include the applicable `ParameterConfigurationSpaces` to be applied as well as numerous other options, such as the number of `trainingRuns` to perform, the `trainingRunCutOff Durations` to observe, and the `workspaceResource` to log training activities at.

Example `ParameterConfigurationSpaces` for the `ManagedGridPlanner` and `ManagedTreePlanner` are shown in Listings 9.10 and 9.11, respectively, and demon-

```
# name type range default
minimumQuality real [1.0, 100.0] [100.0]
maximumQuality real [1.0, 100.0] [1.0]
qualityImprovement real [0.1, 100.0] [20.0]
# forbidden clauses
{(minimumQuality < maximumQuality)
    || ((qualityImprovement - 0.1) > (minimumQuality - maximumQuality))}
```

Listing 9.10: SAFCS AM: MGP SMAC Parameter Configuration Space

```
# name type range default
sampling categorical {ELLIPSOIDAL, UNIFORM, GAUSSIAN} [UNIFORM]
strategy categorical {EXTEND, CONNECT} [EXTEND]
#extension categorical {LINEAR, FEASIBLE} [FEASIBLE]
maxIterations integer [1, 2000] [1000]
epsilon real [1, 1000] [500]
bias integer [0, 100] [5]
#goalThreshold real [0.0, 500] [50]
neighborLimit integer [1, 10] [5]
initialCostBias real [0.0, 1.0] [0.0]
finalCostBias real [0.0, 1.0] [1.0]
improvementFactor real [0.001, 1.0] [0.2]

# forbidden clauses
#{epsilon < goalThreshold}
{(initialCostBias > finalCostBias)
    || ((improvementFactor - 0.001) > (finalCostBias - initialCostBias))}
```

Listing 9.11: SAFCS AM: MTP SMAC Parameter Configuration Space

strate how ParameterConfigurations can be constrained further by applying domain knowledge as necessary to limit the search space. A ParameterConfigurationSpace file consists of the four columns name, type, range, and default. The Tuning parameters shown in Table 9.4, or a selection thereof, are simply transferred into this file, typed and constrained as necessary. Although the default column could serve to establish the initial *incumbent* ParameterConfiguration, our SAFCS defaults to its HeuristicPlannerTuning described in Section 9.6 as a baseline.

The SMACOptions and ParameterConfigurationSpace are then encapsulated into an AlgorithmExecutionConfiguration, which together with the SmacPlannerEvaluator, serve as the essential building blocks for all SequentialModelBasedAlgorithmConfigurations that are hosted by the SmacAutonomicManager. Having created the required SequentialModelBasedAlgorithmConfigurations, the persisted KnowledgeBase is then loaded and, thus, ready for extension during the subsequent training.

The run phase of a SmacAutonomicManager is shown in Listing 9.12. As already explained, the SmacAutonomicManager reverts to the behavior of the AbstractAutonomicManager while EXECUTING (lines 2 and 3) but performs evaluations of ManagedPlanners on training Scenarios while in TRAINING. The training is performed sequentially for each ManagedPlanner (lines 6 to 10) resulting in its best-on-average *incumbent* ParameterConfiguration (line 11) for the supported set of ProblemInstances, that is, training Scenarios with extracted Features. Recall from Section 3.3.1 that SMAC internally builds a random forest of regression trees for variance reduction and combines randomly selected ParameterConfigurations from the available ParameterConfigurationSpace with local search in the neighborhood of promising ParameterConfigurations following the ideas of Algorithm 12. Hence, the SequentialModelBasedAlgorithmConfiguration invokes our SmacPlannerEvaluator several times during each training session, passing in the ParameterConfigurations to be evaluated. The number of these evaluations and their associated time limit is specified by the trainingRuns and trainingRunCutOff Properties of the SmacAutonomicManager, respectively, and provided via its Specification, as shown in Figure 5.32b.

Once the best-on-average *incumbent* ParameterConfiguration has been obtained, its associated Performances are stored in the KnowledgeBase of our SAFCS AM (lines 14 to 44). To this end, the *incumbent* is first translated into a Specification of the ManagedPlanner (lines 18 to 28) to create an associated

```
protected void run(Session managedSession) {
    if (SmacManagerMode.EXECUTING == this.getManagerMode()) {
        super.run(managedSession);
    } else if (SmacManagerMode.TRAINING == this.getManagerMode()) {
        // run training session
        for (String plannerId : this.smacs.keySet()) {
            SequentialModelBasedAlgorithmConfiguration smac = this.smacs
                .get(plannerId);
            // run SMAC
            smac.run();
            ParameterConfiguration incumbent = smac.getIncumbent();

            // extend knowledge base
            Set<ProblemInstance> problems = smac.getRunHistory()
                .getUniqueInstancesRan();

            for (ProblemInstance problem : problems) {
                Specification<Planner> managedPlannerSpecification = null;
                if (Specification.PLANNER_MGP_ID.equals(plannerId)) {
                    managedPlannerSpecification = this.plannerEvaluator
                            .createManagedGridPlannerSpecification(incumbent);
                } else if (Specification.PLANNER_MTP_ID.equals(plannerId)) {
                    managedPlannerSpecification = this.plannerEvaluator
                            .createManagedTreePlannerSpecification(incumbent);
                } else if (Specification.PLANNER_MRP_ID.equals(plannerId)) {
                    managedPlannerSpecification = this.plannerEvaluator
                            .createManagedRoadmapPlannerSpecification(incumbent);
                }

                PlannerTuning tuning = this.createPlannerTuning(
                        managedPlannerSpecification,
                        (Features) problem.getFeatures());
                double cost = smac.getRunHistory().getEmpiricalCost(
                        incumbent,
                        Collections.singleton(problem),
                        this.getTrainingRunCutOff().toSeconds());
                NumericPerformance performance = new NumericPerformance(
                        ZonedDateTime.now(),
                        new NumericQuality(SmacPlannerEvaluator
                            .toResultQuality(cost)),
                        new NumericQuantity(1));
                this.getKnowledgeBase().getReputation()
                    .addTuningPerformance(tuning, performance);
            }
        }
    }
}
```

Listing 9.12: SAFCS AM: SMAC Training

PlannerTuning (lines 30 to 32). Recall that createPlannerTuning is implementation-specific and creates a SmacPlannerTuning in this case. Next, the empirical cost, that is, the TrajectoryQuality measure of the *incumbent* as determined by our SmacPlannerEvaluator, is retrieved for each of the ProblemInstances in the training set (lines 33 to 36). Finally, a normalized NumericPerformance that corresponds to the retrieved cost is stored (lines 37 to 43). It should be emphasized that a NumericPerformance does not aggregate a TrajectoryQuality and DurationQuantity such as the PlannerPerformance. Although the SMAC framework allows for additional run data, such as a Trajectory string to be included in the result of the SmacPlannerEvaluator, the evaluator simply reports the resulting TrajectoryQuality measure instead. As will be clarified shortly, this is sufficient as the applied SmacPlannerTuning compares stored Performances only and disregards the concrete Quality and Quantity components while the SmacAutonomicManager is EXECUTING.

Finally, during the cleanup phase of the SmacAutonomicManager, the extended KnowledgeBase is saved for future use in TRAINING or while EXECUTING.

As already mentioned, the SmacAutonomicManager reverts to the AbstractAutonomicManager while EXECUTING a managed planning Session. The difference between the HeuristicAutonomicManager presented in Section 9.6 and the SmacAutonomicManager is, therefore, reduced to their respective PlannerTuning implementations. In fact, the SmacPlannerTuning shown in Figure 9.12 extends the HeuristicPlannerTuning and defaults to the behavior of the former, if no suitable PlannerTunings can be found in the KnowledgeBase.

The KnowledgeBase contains the Reputation, that is, the Collections of Performances, of tuned ManagedPlanners for solving Scenarios described by Features. Recall that the Reputation is not only extended during TRAINING but also while EXECUTING. Therefore, the stored Performances can be instances of PlannerPerformances stored by an AbstractAutonomicManager implementation, such as the HeuristicAutonomicManager and SmacAutonomicManager, while EXECUTING, or NumericPerformances stored by the SmacAutonomicManager during TRAINING. The concrete PlannerTunings mapping to these Collections of Performances can, therefore, be instances of the HeuristicPlannerTuning or SmacPlannerTuning.

The SmacPlannerTuning, that is applied by the SmacAutonomicManager while EXECUTING, attempts to identify PlannerTunings, stored in the Reputation of the

KnowledgeBase, whose Features are covered by the same FeatureCategory as the extracted Features of the active source Scenario of the managed planning Session. Recall from Section 5.3.3 that a FeatureCategory defines ranges or constraints on covered Features and may take the form of a level of Criticality or a particular FlightPhase. If no FeatureCategory covers the extracted Features, or the Features of stored PlannerTunings are not covered by any associated FeatureCategory, then the SmacPlannerTuning reverts to the HeuristicPlannerTuning it inherits from. Listing 9.13 reveals the full tuning procedure of the SmacPlannerTuning.

If there is no KnowledgeBase or no defined FeatureCategory (line 4), the SmacPlannerTuner reverts to the HeuristicPlannerTuner (line 6). Otherwise the Reputation of the KnowledgeBase is obtained (line 8) and the contained PlannerTunings are filtered for the ones that can be associated with the ManagedPlanner to be tuned (lines 9 to 13). The FeatureCategory items that cover the extracted Features of the replicated source Scenario are then identified (lines 15 to 17). If no FeatureCategory covers the extracted Features, then again the SmacPlannerTuner reverts to the HeuristicPlannerTuner (line 20). Otherwise, the previously filtered PlannerTunings are once again filtered for those that are covered by any of the found FeatureCategory items (lines 23 to 28). If no such PlannerTunings in the Reputation exist, then the SmacPlannerTuner defaults to the HeuristicPlannerTuner as before (line 31). Otherwise, the PlannerTunings in the Reputation that can be associated to one of the same FeatureCategory items as the extracted Features, are collected and sorted according to their maximum applicable Performance (lines 34 to 57). Notice that only Performances of the current PerformanceContext, that is, execution environment, are taken into account for the selection of a suitable PlannerTuning (lines 37, 42, and 44).

More details about the SmacAutonomicManager[17] and related classes can be found in our SAFCS software repository.

[17] https://github.com/stephanheinemann/worldwind/tree/develop/src/main/java/com/cfar/swim/worldwind/managers/smac

```
public List<Properties<Planner>> tune(
        Specification<Planner> specification, Features features) {
    List<Properties<Planner>> properties = new ArrayList<>();
    if ((null == this.knowledgeBase) || (null == this.featureCategories)) {
        // revert to heuristic tuning
        properties = super.tune(specification, features);
    } else {
        Reputation reputation = this.knowledgeBase.getReputation();
        List<PlannerTuning> tunings = reputation.keySet().stream()
            .filter(t -> (t instanceof PlannerTuning))
            .map(t -> (PlannerTuning) t)
            .filter(t -> t.getSpecification().getId().equals(specification.getId()))
            .collect(Collectors.toUnmodifiableList());
        // apply tuning for associated feature categories
        List<FeatureCategory> categories = this.featureCategories.stream()
            .filter(c -> c.covers(features))
            .collect(Collectors.toUnmodifiableList());
        if (categories.isEmpty()) {
            // revert to heuristic tuning
            properties = super.tune(specification, features);
        } else {
            // associated covered feature categories
            List<PlannerTuning> categoryTunings = new ArrayList<>();
            for (FeatureCategory category : categories) {
                categoryTunings.addAll(tunings.stream()
                    .filter(t -> category.covers(t.getFeatures()))
                    .collect(Collectors.toUnmodifiableList()));
            }
            if (categoryTunings.isEmpty()) {
                // revert to heuristic tuning
                properties = super.tune(specification, features);
            } else {
                // apply feature category tunings
                properties.addAll(categoryTunings
                    .stream()
                    .filter(t -> reputation.hasPerformances(
                            t, PerformanceContext.CURRENT, NumericPerformance.class))
                    .sorted(new Comparator<PlannerTuning>() {
            public int compare(PlannerTuning t1, PlannerTuning t2) {
                int result = 0;
                OptionalDouble p1 = reputation.getMaximumTuningPerformance(
                        t1, PerformanceContext.CURRENT, NumericPerformance.class);
                OptionalDouble p2 = reputation.getMaximumTuningPerformance(
                        t2, PerformanceContext.CURRENT, NumericPerformance.class);
                if (p1.isPresent() || p2.isPresent()) {
                    if (p1.isPresent() && p2.isEmpty()) {
                        result = -1;
                    } else if (p1.isEmpty() && p2.isPresent()) {
                        result = 1;
                    } else {
                        result = (p1.getAsDouble() < p2.getAsDouble()) ? 1 : -1;
                    }
                }
                return result;
            }})
                    .map(t -> t.getSpecification().getProperties())
                    .collect(Collectors.toUnmodifiableList()));
                if (properties.isEmpty()) {
                    // revert to heuristic tuning
                    properties = super.tune(specification, features);
                }
            }
        }
    }
    return properties;
}
```

Listing 9.13: SAFCS AM: SMAC Tuning

9.8 Chapter Summary

In this chapter we presented the main concepts and selected functional aspects of our SAFCS Autonomic Manager. In particular, we covered the topics of sensing and actuating in terms of extracting `Features` from a managed `Scenario` within an operationally defined horizon as well as in terms of tuning and selecting an active `ManagedPlanner` following a `Performance` competition. The concrete Autonomic Manager is responsible for the tuning and may apply heuristics, machine learning, or a combination of both using an established `KnowledgeBase` of `Reputations` with the intention to improve the overall `Performance` of the SAFCS as an Autonomic Element.

Chapter 10

SAFCS — User Interface

Our SAFCS UI enables and facilitates control, supervision, analysis, testing, and communication of features of the underlying SAFCS core. It has become an invaluable tool to support the development of the core as well as to simulate OPS use-cases. It can, furthermore, serve as a standalone GCS for test flights.

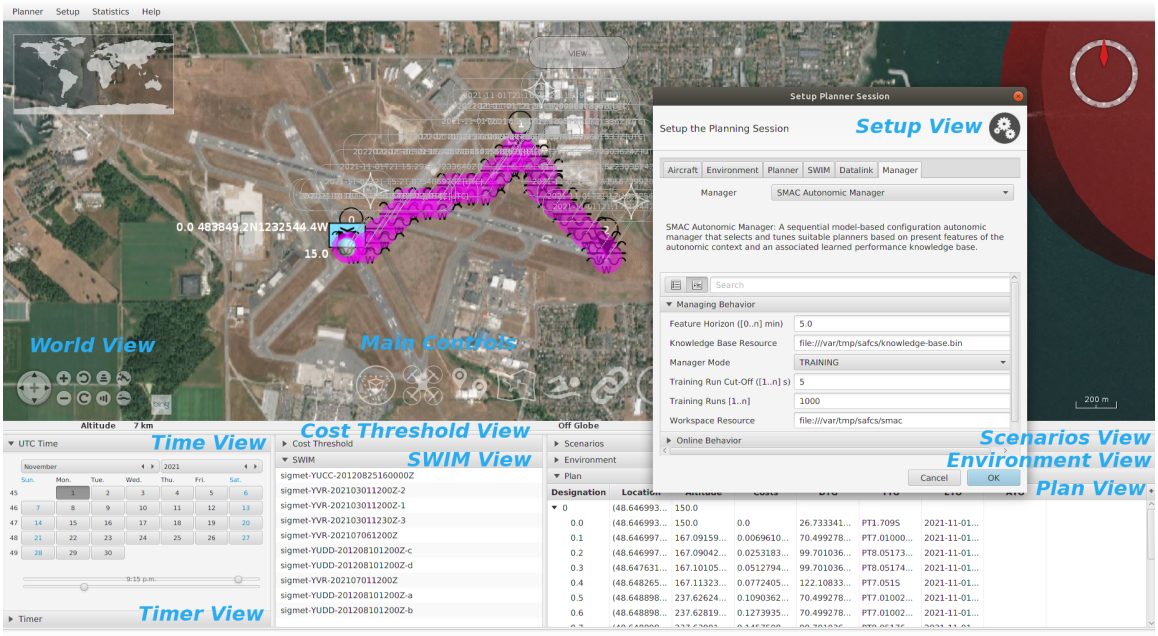


Figure 10.1: SAFCS UI: Views

The SAFCS UI combines several `Views` realizing the required OPS use-cases into one interface, as shown in Figure 10.1. Recall from Section 5.4 that each `View` is realized in a separate package with its corresponding `Presenter` following the *convention over con-*

figuration opinionated approach of the Afterburner.fx MVP framework. This approach makes the SAFCS UI easily testable, maintainable and extendable.

The `Views` currently supported by the SAFCS UI are

- the `WorldView`,
- the `TimeView`,
- the `TimerView`,
- the (Cost) `ThresholdView`,
- the `SwimView`,
- the `ScenarioView`,
- the `EnvironmentView`,
- the `PlanView`,
- the `SetupView`, and
- the `PlannerView` combining all previous `Views`.

10.1 World

The `WorldView` integrates the NASA Java WorldWind virtual globe Software Development Kit (SDK)[1] to display and interact with geographic data featuring high-resolution imagery and terrain. NASA WorldWind, furthermore, supports many of the frequently used geometric shapes and associated operations to serve as a foundation for our extensions, including `Environments`, `Aircraft`, SWIM `Obstacles`, `Trajectory Waypoints`, `Datalink TrackPoints`, and their corresponding symbology. Since the SAFCS requires more sophisticated intersection and collision checking than offered by the NASA WorldWind SDK, we also integrated the physics engine JBullet[2] into the SAFCS core. Although the physics engine is currently only employed to determine `Obstacle` collisions, it might be an interesting option to simulate physical models in the future.

[1] `https://worldwind.arc.nasa.gov/java/`
[2] `http://jbullet.advel.cz/`



Figure 10.2: SAFCS UI: World

Figure 10.2 shows the `WorldView` of our SAFCS UI displaying the area near the Victoria International Airport.[3] The `View` comprises controls that are included by the original SDK, such as the ones in its top and lower left corners. These controls allow to display a desired location, move, zoom, rotate, or pan the perspective of the `View` as well as to exaggerate existing terrain. The controls at the bottom of the view have been added to provide the interface for our desired functionality. Depending on the particular control functionality, the `WorldMode` may become one of `VIEW`, `AIRCRAFT`, `ENVIRONMENT`, `WAYPOINT`, `PLANNING`, `LOADING`, `UPLOADING`, `LAUNCHING`, `LANDING`, or `MANAGING`, and is displayed in the status bubble at the top of the `WorldView`.

More details about the `WorldView`, `WorldPresenter` and related classes can be found in the `world`[4] package in our SAFCS UI software repository.

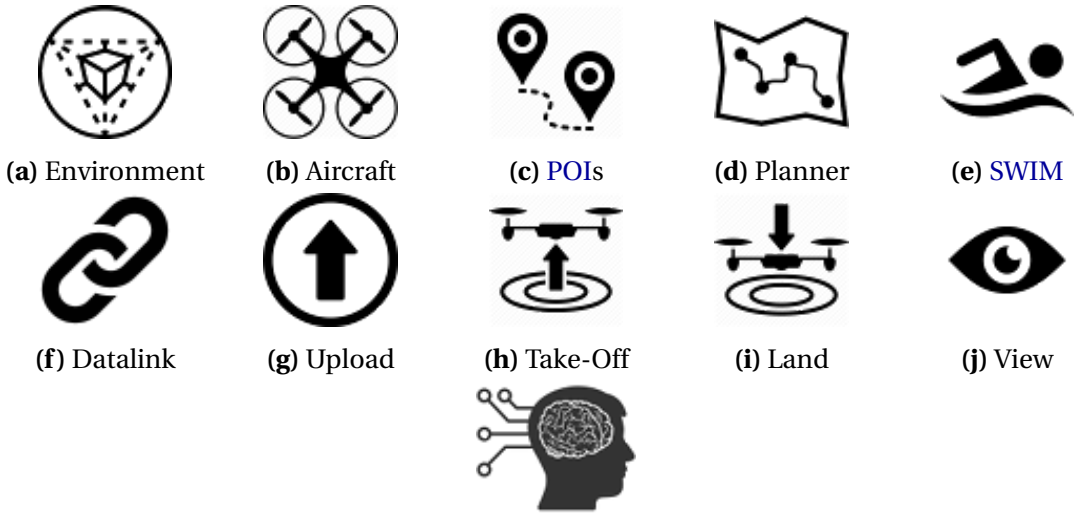
10.2 Main Controls and Setup

The main controls of the `WorldView` are shown in Figure 10.3. They are sequenced at the bottom of the `WorldView` from left to right to support the natural workflow from defining operational goals to the invocation of AMs. Most controls follow the main control philosophy to execute a function on mouse left-click or open the corresponding `SetupView` tab to define associated `Specifications` on mouse right-click.

The *Environment* control shown in Figure 10.3a enables OPS to geo-fence the sector and define the `Specification` of a planning `Environment`. The `Specification` of an

[3] https://www.victoriaairport.com/

[4] https://github.com/stephanheinemann/worldwind-ui/tree/develop/src/main/java/com/cfar/swim/worldwind/ui/world



(a) Environment **(b)** Aircraft **(c)** POIs **(d)** Planner **(e)** SWIM

(f) Datalink **(g)** Upload **(h)** Take-Off **(i)** Land **(j)** View

(k) Manager

Figure 10.3: SAFCS UI: Main Controls

`Environment` depends on the concrete `Environment` class, as explained in Section 5.2.3, and comprises `Properties` that define its upper and lower boundaries as well as its resolution. Once the planning `Environment` has been constructed, an `Aircraft` may be placed inside it to define the planned departure position after take-off.

The *Aircraft* control shown in Figure 10.3b enables OPS to define the `Specification` of an `Aircraft` and place it at a 3D position within the constructed planning `Environment`. Recall from Section 5.2.2 that the `Properties` of an `Aircraft` `Specification` comprise its `Capabilities`. The `Capabilities` can be compared to performance values published by the manufacturer in the corresponding Pilot Operating Handbook (POH). They are used by `Planners` to determine the feasibility of a `Trajectory` and estimate the ETOs of the intermediate `Waypoints`.

The POIs are special intermediate and final `Waypoints` of a (multi-part) `Trajectory` and can be defined as mission goals using the POIs control shown in Figure 10.3c. Currently, the SAFCS does not support the dedicated `Specification` of POI `Properties`. However, as further discussed in Chapter 15, different POI types and actions with their associated time slots are envisioned as an extension. These POI types would include alternate aerodromes (goals) for diversions if the operational `RiskPolicy` is exceeded during an airborne mission.

If OPS elects to specify and configure a concrete `Planner` instead of supervising the activities of an `AutonomicManager`, then the *Planner* control shown in Figure 10.3d can be used to define the corresponding `Planner` `Specification` and to invoke the created

Planner. Recall from Section 5.2.5 that the Properties of all Planner Specifications comprise an operational RiskPolicy and CostPolicy. These policies significantly affect the performance and computed Trajectory of a Planner. Together with the Environment, Aircraft, and POI Waypoints, they define and constrain the operational goals of the planning Scenario.

The SWIM control shown in Figure 10.3e enables OPS to connect the SAFCS with SWIM data providers and subscribe to data feeds for AIXM, FIXM, WXXM/IWXXM, and AMXM. Currently, following an FAA interface change, LiveSwimConnections are not supported, and the SAFCS UI can be used with only a rudimentary SimulatedSwimConnection to probabilistically inject SWIM Obstacles. Establishing the connection to the FAA SWIM infrastructure is, therefore, revisited in Chapter 15.

The *Datalink* control shown in Figure 10.3f is used to establish a Datalink Connection to the managed agent. A Datalink is required when invoking OnlinePlanners and, consequently, ManagedPlanners via AutonomicManagers. The Datalink downlinks an AircraftTrack consisting of AircraftTrackPoints, as explained in Section 5.2.5 and shown in Figure 5.23. AircraftTrackPoints allow OPS to monitor the progress of a planning Session and select different views, including an FPV to simulate the agent forward view if no dedicated video downlink is available.

The *Upload* (or uplink) control shown in Figure 10.3g enables OPS to manually upload a Trajectory computed by an offline Planner to the Datalink target. Since OnlinePlanners and, consequently, *active* ManagedPlanners upload Trajectory revisions implicitly, this control is required only for the most basic modes of operations.

The *Take-Off* and *Land* controls shown in Figure 10.3h and 10.3i, respectively, enable OPS to manually initiate take-off and landing manoeuvers. These controls are required for offline Planners as well as to support interventions by OPS if an immediate landing or return-to-launch is considered necessary during an OnlinePlanner or ManagedPlanner Session. Under normal conditions, however, OnlinePlanners and, consequently, *active* ManagedPlanners initiate their take-off and landing manoeuvers automatically via Datalink Communications and only request OPS confirmations.

The *View* control shown in Figure 10.3j supports a variety of views both on the planned Aircraft as well as the tracked Datalink source. The supported ViewModes include top-down (above), chase, and FPV.

Finally, the *Manager* control shown in Figure 10.3k enables OPS to configure and initiate a fully autonomous mission once the necessary ManagedGoals have been defined. The corresponding SetupView allows to select the desired AutonomicManager

Specification and configure its Properties. While *executing* and as explained in Chapter 9, the concrete AutonomicManager automatically performs Datalink Communications via its *active* ManagedPlanner to upload Trajectory revisions and perform take-off and landing manoeuvers as necessary. It selects and tunes ManagedPlanners according to their measured Performance under operational policies within a Scenario described by extracted discriminating Features.



Figure 10.4: SAFCS UI: Activity

Whenever the SAFCS is actively performing a function as a response to the operation of the described main controls, the WorldView displays the activity in form of highlighted controls and a progress circle, as shown in Figure 10.4. The example shows a connected Datalink and an executing (or training) AutonomicManager.

More details about the main controls of the WorldView, the SetupView, the SetupPresenter and related classes can be found in the world[5] and setup[6] packages in our SAFCS UI software repository.

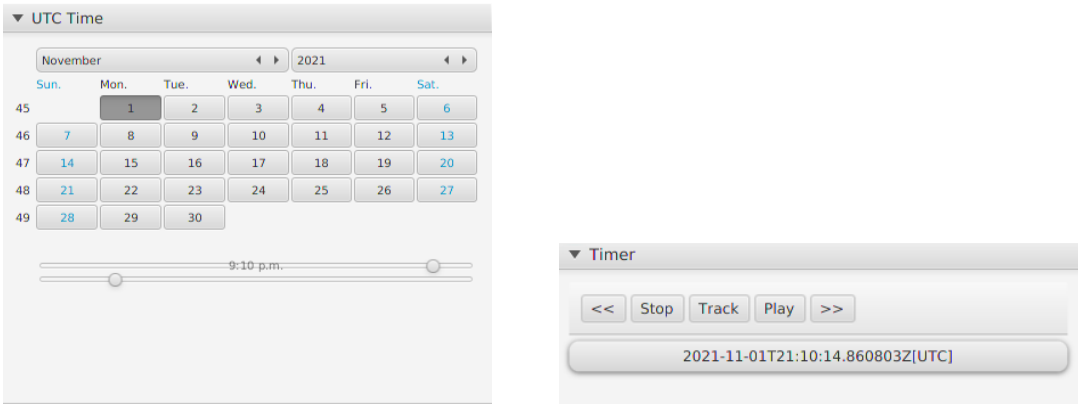
10.3 Time and Timer

The bottom left accordion of the SAFCS UI contains both the TimeView and TimerView shown in Figure 10.5. All SAFCS missions are planned and executed in a 4D Environment. Therefore, the aggregating planning Scenario can be set to any instance in time. The time control allows for the selection of year, month, day and UTC. Usually

[5] https://github.com/stephanheinemann/worldwind-ui/tree/develop/src/main/java/com/cfar/swim/worldwind/ui/world

[6] https://github.com/stephanheinemann/worldwind-ui/tree/develop/src/main/java/com/cfar/swim/worldwind/ui/setup

a time instance in the near future is set in the `TimeView` and serves as the ETD for the invoked `Planner` or `AutonomousManager`. However, a time instance in the past is very well possible and enables OPS to replay and assess certain situations. Depending on the selected time, temporal SWIM data may or may not be present and become relevant for the planning `Session`.



(a) SAFCS UI: Time **(b)** SAFCS UI: Timer

Figure 10.5: SAFCS UI: Time and Timer

Time can also be tracked, played, rewound or fast-forwarded using the `TimerControl`. For example, consider an ETD a few hours in the future, that could have been used to compute a `Trajectory`. By playing the time starting at the ETD, the SAFCS UI `WorldView` can display the planned progress of the `Aircraft` as well as the presence and movements of any embedded temporal SWIM `Obstacles`. Thus, using the `TimerView`, the currently experienced or forecast SWIM data and the resulting planned `Aircraft` position on the computed `Trajectory` can be easily visualized in 4D. Finally, being able to track the current time enables OPS to compare the planned progress of the `Aircraft` with that of the `Datalink` source during an online or managed planning `Session`. Established operational tolerances, such as acceptable and actual `AircraftTrackPointErrors` and `AircraftTrackErrors`, can be monitored as the mission unfolds.

More details about the `TimeView`, `TimePresenter`, `TimerView`, `TimerPresenter` and related classes can be found in the `time`[7] and `timer`[8] packages in our SAFCS UI software repository.

[7] https://github.com/stephanheinemann/worldwind-ui/tree/develop/src/main/java/com/cfar/swim/worldwind/ui/time

[8] https://github.com/stephanheinemann/worldwind-ui/tree/develop/src/main/java/com/cfar/swim/worldwind/ui/timer

10.4 SWIM Data and Cost Threshold

The bottom center accordion of the SAFCS UI contains both the cost `ThresholdView` and `SwimView`, as shown in Figure 10.6. The `SwimView` lists the identifiers of all currently loaded SWIM data sets. These temporal SWIM data sets can be explicitly loaded, unloaded, enabled, and disabled by OPS. They can also be embedded automatically by a connected `SwimConnection`.



(a) SAFCS UI: SWIM Data **(b)** SAFCS UI: Cost Threshold

Figure 10.6: SAFCS UI: SWIM Data and Cost Threshold

Both `Environments` and `Obstacles` are associated with costs. The `Environment` *base* cost is added to the costs of active `CostIntervals` of embedded SWIM `Obstacles` in order to determine the transition (step) costs of the planned `Aircraft`, as described in Sections 5.2.3 and 5.2.5. Since the `Environment` internal structure, consisting of, for example, `PlanningGrids` or `Edges`, and the embedded `Obstacles` may clutter the `WorldView` significantly, the `ThresholdView` offers a slider to selectively show only `Environments` and `Obstacles` that exceed a selected cost threshold. OPS may, for example, elect to show only those parts of a planning `Environment` that exceed the operational `RiskPolicy` and, therewith, declutter the `WorldView` substantially.

More details about the `SwimView`, `SwimPresenter`, `ThresholdView`, `ThresholdPresenter` and related classes can be found in the `swim`[9] and `threshold`[10] packages in our SAFCS UI software repository.

[9] https://github.com/stephanheinemann/worldwind-ui/tree/develop/src/main/java/com/cfar/swim/worldwind/ui/swim

[10] https://github.com/stephanheinemann/worldwind-ui/tree/develop/src/main/java/com/cfar/swim/worldwind/ui/threshold

10.5 Scenarios

The `ScenarioView` is contained in the bottom right accordion of the SAFCS UI and shown in Figure 10.7. It lists the identifiers of all available `Scenarios` that are hosted by the planning `Session`. `Scenarios` can be added, removed and edited to aggregate `FactoryProducts`, such as `Environments`, `Aircraft`, `Planners`, and `Connections`, as explained in Section 5.2.1. At least, the *default* `Scenario` is present at all times. The currently *active* `Scenario` is shown in bold.



Figure 10.7: SAFCS UI: Scenarios

Recall that `Scenarios` satisfy several use-cases. They support independent missions as part of the same planning `Session`. In the context of a managed `Session` they can be employed by an `AutonomicManager` both as evaluation `Scenarios` while `EXECUTING` and training `Scenarios` while in `TRAINING`, as explained in Chapter 9.

More details about the `ScenarioView`, `ScenarioPresenter` and related classes can be found in the `scenario`[11] package in our SAFCS UI software repository.

10.6 Environment

The `EnvironmentView` is the second `View` contained in the bottom right accordion of the SAFCS UI and shown in Figure 10.8.

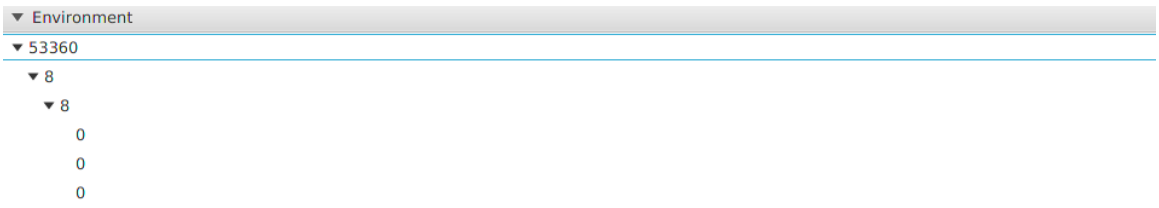


Figure 10.8: SAFCS UI: Environment

The `EnvironmentView` always shows the hierarchy, if any, of the `HierarchicalEnvironment` of the currently *active* `Scenario`. It enables OPS to

[11] https://github.com/stephanheinemann/worldwind-ui/tree/develop/src/main/java/com/cfar/swim/worldwind/ui/scenario

manually refine or coarsen certain parts of the hierarchy and modify a multi-resolution `Environment`, such as the `PlanningGrid`. The `EnvironmentView` may, furthermore, reveal the complexity of the planning task to developers by, for example, showing the number of voxels in a `PlanningGrid` (53360).

More details about the `EnvironmentView`, `EnvironmentPresenter` and related classes can be found in the `environment`[12] package in our SAFCS UI software repository.

10.7 Plan

The `PlanView` is the third `View` contained in the bottom right accordion of the SAFCS UI and shown in Figure 10.9.

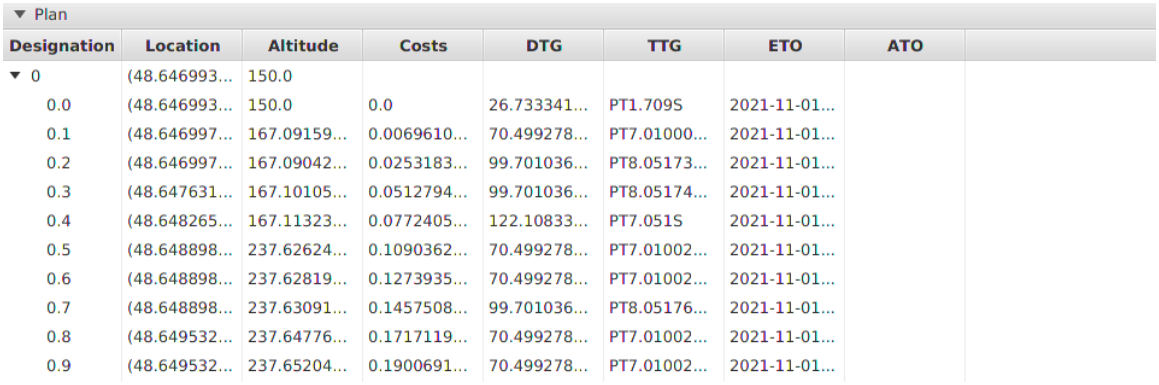


Figure 10.9: SAFCS UI: Plan

The `PlanView` shows the currently computed (revised) `Trajectory` that is also displayed graphically in the `WorldView` in form of a numeric table. The table consists of generated `Waypoint` designators, locations, altitudes, costs, Distances To Go (DTGs), Times To Go (TTGs), ETOs, and Actual Times Over (ATOs). This enables OPS to assess the achieved `TrajectoryQuality` for individual parts of the `Trajectory` and replicates the navigation log as a result of traditional flight planning.

More details about the `PlanView`, `PlanPresenter` and related classes can be found in the `plan`[13] package in our SAFCS UI software repository.

[12] https://github.com/stephanheinemann/worldwind-ui/tree/develop/src/main/java/com/cfar/swim/worldwind/ui/environment

[13] https://github.com/stephanheinemann/worldwind-ui/tree/develop/src/main/java/com/cfar/swim/worldwind/ui/plan

10.8 Chapter Summary

In this chapter we briefly discussed the main `Views` of our SAFCS UI component. We described the purpose of each `View` and mentioned the associated functionality in the SAFCS core. In particular, the main controls located at the bottom of the `WorldView` allow OPS to modify `Specifications` of created `FactoryProducts` as well as the definition of operational goals and policies. They can be used to control an airborne mission manually or supervise the latter in an autonomously managed mode.

Part III

Evaluation, Vehicle Integration and Experiments

Chapter 11

Evaluation

The evaluation of our SAFCS could be manifold and include the discussion of the architecture in terms of the suitability and soundness of its interfaces, its maintainability and extensibility. In fact, during its development and apart from continuous refactoring, the SAFCS core architecture never had to be re-engineered and allowed other contributors to include their extensions without major obstacles and perform their own successful research and evaluations (Ferreira [BF19] and Rosa [dCR19]).

However, our evaluation shall be limited to the performance of the currently supported `AutonomicManagers` with respect to planning `Scenarios` of varying `Features`. These `Features` shall be covered by our established `FlightPhases` *cruise*, *transition*, *terminal*, *urgency*, and *emergency*. As discussed in Section 9.2, a `FlightPhase` aggregates a set of `Difficulty` and `Severity` associations and, therefore, serves as a `FeatureCategory` itself.

The performance of an `AutonomicManager` is evaluated for five `Scenarios` per `FlightPhase` featuring different `Difficulty` and `Severity` combinations. First, the `HeuristicAutonomicManager`, explained in Section 9.6, shall be evaluated to confirm that `ManagedPlanners` can be selected and tuned based on extracted `Features` of the evaluation `Scenarios` and result in a reasonable `TrajectoryQuality` and `PlannerPerformance`. This first step establishes a performance-baseline to compare other `AutonomicManagers` against and may potentially help to find more suitable heuristics for an improved version of the `HeuristicAutonomicManager`.

Second, the `SmacAutonomicManager`, explained in Section 9.7, is trained on the combined evaluation `Scenarios` of each `FlightPhase` to determine the *best-on-average* `PlannerTuning`, respectively. That is, each evaluation `Scenario` represents a *problem instance*, using SMAC terminology, in a set of five for its associated `FlightPhase`. The

evaluation shall then discover whether or not a best-on-average SmacPlannerTuning can outperform our selected HeuristicPlannerTuning that is applied to individual Scenarios. If so, it may help to reason about and find improved heuristics that are derived from individual Features as opposed to from covered FlightPhases.

11.1 Evaluation Criteria

In order to assess the performance of different AutonomicManagers with respect to individual Scenarios with particular Features, it is necessary to *normalize* both the TrajectoryQuality and DurationQuantity. This normalization allows to account for multi-part trajectories, that is, partial trajectories for multiple intermediate POIs as generated by AnytimePlanners. Recall from Sections 5.3.2 and 9.1 that all ManagedPlanners are specialized AnytimePlanners. Hence, the TrajectoryQuality $Q(\mathcal{T},\pi)$ of a Trajectory $\mathcal{T}$ in a Scenario (problem instance) π, which is defined as the inverse of the its normalized cost in the Scenario, that is, $(c(\pi)(\mathcal{T}))^{-1}$, has to be multiplied by the normalizer $\eta(\mathcal{T},\pi)$ as shown in Equation 11.1.

$$\tilde{Q}(\mathcal{T},\pi) = \eta(\mathcal{T},\pi)Q(\mathcal{T},\pi) = \frac{\eta(\mathcal{T},\pi)}{c(\pi)(\mathcal{T})} = \frac{\sum_{0}^{l(\mathcal{T},\pi)-1} h(\pi)(p_i(\pi),p_{i+1}(\pi))}{c(\pi)(\mathcal{T}) \sum_{0}^{n(\pi)-1} h(\pi)(p_i(\pi),p_{i+1}(\pi))} \tag{11.1}$$

We, therefore, define the normalizer $\eta(\mathcal{T},\pi)$ of a Trajectory $\mathcal{T}$ in a Scenario π as the fraction of the sum of the heuristics $h(\pi)$ (normalized Euclidean distances in π) of consecutive Scenario POIs $(p_i(\pi),p_{i+1}(\pi))$ up to the last POI index $l(\mathcal{T},\pi)$ of the computed Trajectory $\mathcal{T}$ over the sum of the heuristics of consecutive POIs $(p_i(\pi),p_{i+1}(\pi))$ in the complete Trajectory containing all $n(\pi)$ POIs of the Scenario π including both the start and goal POIs. It is important to emphasize that the resulting normalized TrajectoryQuality $\tilde{Q}(\mathcal{T},\pi)$ serves to compare the generated Specification Properties of ManagedPlanners for one particular Scenario π and its Features only. Since different Scenarios with their respective Features permit their own (normalized) optimal TrajectoryQuality $\tilde{Q}^*(\mathcal{T},\pi)$, it is *not* possible to compare the latter across different Scenarios. This optimal TrajectoryQuality

is *unknown* and irrelevant to compare different AutonomicManagers with respect to their generated Specification Properties as they are normalized to the same theoretically best possible TrajectoryQuality using the heuristic as described. Again, the normalized TrajectoryQuality $\tilde{Q}(\mathcal{T},\pi)$ is employed to compare different AutonomicManagers with respect to their generated Specification Properties for each particular Scenario π only. It is *not* intended for the comparison of a normalized TrajectoryQuality $\tilde{Q}(\mathcal{T}_1,\pi_1)$ of a Scenario (problem instance) π_1 with another normalized TrajectoryQuality $\tilde{Q}(\mathcal{T}_2,\pi_2)$ of a different Scenario π_2. Furthermore, it is *not* our goal and should be avoided to compare the TrajectoryQuality of different ManagedPlanners with respect to the same Scenario π as different ManagedPlanners can be configured differently and our AutonomicManagers may not tune all possible parameters to allow for a fair comparison. For example, the evaluated AutonomicManagers do *not* adjust the resolution and hierarchy of the planning Environment, which can be considered a disadvantage for the ManagedGridPlanner compared to the ManagedTreePlanner whose maximum step distance epsilon is part of its Specification Properties and may, therefore, adapt better to different Scenarios.

$$\tilde{D}(\mathcal{T},\pi) = \eta(\mathcal{T},\pi)^{-1} D(\mathcal{T},\pi) = \frac{D(\mathcal{T},\pi)}{\eta(\mathcal{T},\pi)} = \frac{D(\mathcal{T},\pi) \sum_{0}^{n(\pi)-1} h(\pi)(p_i(\pi), p_{i+1}(\pi))}{\sum_{0}^{l(\mathcal{T},\pi)-1} h(\pi)(p_i(\pi), p_{i+1}(\pi))} \tag{11.2}$$

Similarly, the normalized DurationQuantity $\tilde{D}(\mathcal{T},\pi)$ to compute a (partial multi-part) Trajectory $\mathcal{T}$ in Scenario π is derived as shown in Equation 11.2. This time, the DurationQuantity $D(\mathcal{T},\pi)$ of a Trajectory $\mathcal{T}$ computed in Scenario π is divided by the previously established normalizer. Thus, a normalized DurationQuantity $\tilde{D}(\mathcal{T}_1,\pi)$ to compute a (partial multi-part) Trajectory $\mathcal{T}_1$ with a normalized TrajectoryQuality $\tilde{Q}(\mathcal{T}_1,\pi)$ can be compared to a normalized DurationQuantity $\tilde{D}(\mathcal{T}_2,\pi)$ and normalized TrajectoryQuality $\tilde{Q}(\mathcal{T}_2,\pi)$ for a different Trajectory $\mathcal{T}_2$ within the same Scenario π as a result of two arbitrary Specification Properties for the same ManagedPlanner.

The quotient of the normalized TrajectoryQuality $\tilde{Q}(\mathcal{T},\pi)$ and the required normalized DurationQuantity $\tilde{D}(\mathcal{T},\pi)$ to compute the Trajectory $\mathcal{T}$ in Scenario π

then gives the normalized Performance $\tilde{P}(\mathcal{T},\pi)$ of a ManagedPlanner that has been configured by an AutonomicManager via its Specification Properties of its concrete PlannerTuning, as described in Sections 5.3.3 and 9.3. The normalized Performance of an automatically tuned ManagedPlanner is shown in the corresponding Equation 11.3.

$$\tilde{P}(\mathcal{T},\pi) = \frac{\tilde{Q}(\mathcal{T},\pi)}{\tilde{D}(\mathcal{T},\pi)} \tag{11.3}$$

For each of the five Scenarios π per FlightPhase, 10 runs are performed to assess the progressive normalized TrajectoryQuality $\tilde{Q}(\mathcal{T},\pi)$ and normalized PlannerPerformance $\tilde{P}(\mathcal{T},\pi)$ over the unnormalized DurationQuantity $D(\mathcal{T},\pi)$ of the corresponding computed Trajectory $\mathcal{T}$. Since the ManagedGridPlanner is a *deterministic* heuristic Planner, we expect only slight tolerances for the DurationQuantity $D(\mathcal{T},\pi)$ stemming from the scheduling applied by the runtime environment of our PerformanceContext. The ManagedTreePlanner is, however, a *non-determinisitc* Planner and several runs are required to derive representative mean and median values for its normalized TrajectoryQuality $\tilde{Q}(\mathcal{T},\pi)$ and normalized PlannerPerformance $\tilde{P}(\mathcal{T},\pi)$. The higher the number of evaluation runs, the more confidence can obviously be put into the results of such a derivation. However, within the scope of this dissertation we limit this number to 10 as major variations between PlannerPerformances that result from differently tuning the ManagedPlanners should become readily apparent, if any.

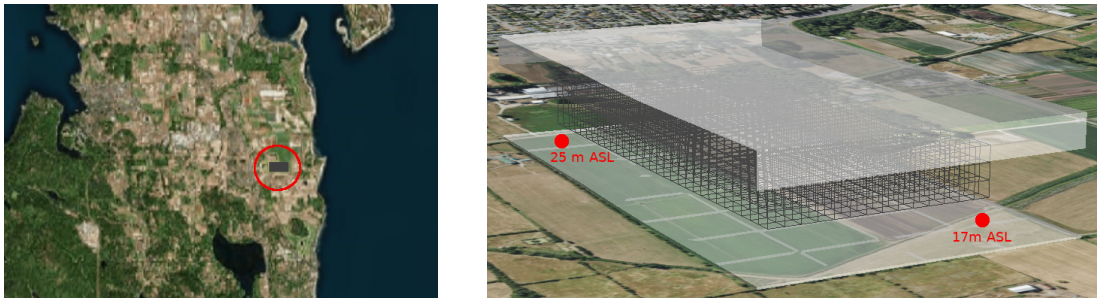
Our evaluation is not concerned with other possible criteria such as the number of computed Trajectory Waypoints or the required heading or altitude changes within the former, as these criteria are mainly interesting for *any-angle* Planners and their potential tuning parameters, if any. Furthermore, we limit our evaluation to offline and non-dynamic cases as the observed PlannerPerformances equally apply to dynamic repairs and replanning, which is always accompanied by the extraction of the changed Features from a Scenario π and the creation of a corresponding PlannerTuning.

The evaluation is performed within the PerformanceContext shown in Table 9.5 using a Java SE 11 LTS runtime environment.[1]

[1] https://www.oracle.com/java/technologies/java-se-support-roadmap.html

11.2 Evaluation Scenarios

All evaluation Scenarios are delimited by the University of Victoria (UVic) Center for Aerospace Research (CfAR) flight test area approximately 10 kilometers south-south east of the Victoria International Airport (CYYJ) located between the Patricia Bay Highway and the Haro Strait coastline of the Saanich Peninsula at 48.5561 degrees latitude and −123.3814 degrees longitude, as shown in Figure 11.1.



(a) UVic CfAR Flight Test Area Top View **(b)** UVic CfAR Flight Test Area Side View

Figure 11.1: UVic CfAR Flight Test Area

With an approximate elevation of 17 meters Above Mean Sea Level (ASL) in the north-east and 25 meters ASL in the south-west corners of the flight test area, all evaluation Scenarios occupy an identical bounding box with the vertical limits of 35 meters ASL up to and including 110 meters ASL. These limits provide a sufficient ground clearance of at least 10 meters and a generous vertical separation of approximately 30 meters (≈ 100 feet) from the 400 feet Above Ground Level (AGL) ceiling required by the applicable Transport Canada (TC) Special Flight Operations Certificate (SFOC). The elevation figures have been determined from importing a 1 meter resolution Light Detection and Ranging (LIDAR) elevation model into the SAFCS. This high-resolution LIDAR elevation model has been made available by the Government of Canada to cover, among other products, individual 10^2 square kilometer tiles that can be downloaded individually[2] or automatically[3] from a Web Map Service (WMS) server on-demand. The theoretical evaluation would not have to be limited to but is conducted based on Scenarios in this flight test area to facilitate and prepare the intended practical experiments.

As indicated in Section 11.1, the evaluation shall assess the achieved relative normalized TrajectoryQuality and normalized PlannerPerformance of ManagedPlanners

[2] https://open.canada.ca/data/en/dataset/957782bf-847c-4644-a757-e383c0057995

[3] https://datacube.services.geo.ca/ows/elevation?service=wms&request=GetCapabilities

based on PlannerTunings established by AutonomicManagers. The Environment resolution and hierarchy is, however, kept constant for all Scenarios since its adaptation is currently not implemented. We selected a PlanningGrid for the ManagedGridPlanner with a division of 40 cubic voxels from the west to the east boundary of the flight test area. Given a total distance of approximately 700 meters, each voxel, therefore, occupies approximately 17.5^3 cubic meters. Since the internal structure of a PlanningContinuum is created by growing a tree that consists of branches with a limited length of epsilon, there are no equivalent restrictions for ManagedTreePlanners. Therefore, the TrajectoryQuality and resulting PlannerPerformance of different ManagedPlanners should not be compared to draw conclusions about the tuning AutonomicManager. Rather, different AutonomicManagers should be compared with respect to the same ManagedPlanner operating in the same Scenario.

Furthermore, the selected Aircraft for the evaluation is the small quadrotor UAV Iris with *constant* Capabilities. All planned velocities, both horizontal and vertical, are limited to 1 meter per second. Hence, the evaluation Scenarios can be transited within approximately 12 minutes. We have selected this relatively slow speed to be able to simulate longer missions covering larger areas and featuring different FlightPhases with dynamic SWIM Obstacles. At the same time, the relatively slow speed allows for timely interventions by a flight test crew if necessary. Again, the theoretical evaluation would not have to be limited this way but serves to facilitate and prepare the transition to practical experiments.

Table 11.1: Obstacle Cost Color Map

0	5	10	15	20	25	30	35	40	45	50	55	60	65	70	75

Lastly, the operational CostPolicy and RiskPolicy covered in Section 5.2.5 are set to AVERAGE and SAFETY, respectively. The Environments of our evaluation Scenarios possess a constant base cost of 1, that is, the normalized distance covered in each step is multiplied by 1 if no Obstacles are present. The embedded SWIM Obstacle costs are, furthermore, limited to be within $(0.0, 100.0]$ and, therefore, non-negative. Thus, our applied heuristic (normalized Euclidean distance) remains consistent. Costs in the Environment at a particular Scenario time instance are visualized by assigning colors to SWIM Obstacle shapes, affected PlanningGrid voxels, and PlanningContinuum Edges as summarized in Table 11.1 showing the associated lower cost thresholds,

respectively. Recall that given our selected RiskPolicy SAFETY, a space in time with an accumulated cost of more than 50 cannot voluntarily be penetrated by a ManagedPlanner. Lower cost areas can be penetrated but may or may not lead to overall higher Trajectory costs and, hence, lower TrajectoryQuality depending on a necessary detour. The Features horizon of our AutonomicManagers, as explained in Section 9.2, is set to a constant of 4 minutes, that is, approximately $\frac{1}{3}$ of the time required to transition the Environment of an evaluation Scenario.

For each evaluation Scenario π we first provide two pictures, either top-down or perspective as appropriate, at the ETD possibly indicating the dynamic aspects of SWIM Obstacles if relevant. We then present two tables summarizing the Criticality assessment of the evaluated Scenario based on the extracted Features, including the determined Difficulty, Severity, and resulting FlightPhase assignments. Recall from Sections 9.2 and 9.3, that an AutonomicManager establishes a PlannerTuning based on this assessment. We cover these PlannerTunings in two additional tables for the ManagedGridPlanner and ManagedTreePlanner, respectively. Lastly, for each evaluation Scenario π and ManagedPlanner combination we then present and briefly discuss the following four diagrams for the applied AutonomicManager:

- $\tilde{Q}(\mathcal{T},\pi)$ over $D(\mathcal{T},\pi)$ for 10 runs including the mean
- $\tilde{P}(\mathcal{T},\pi)$ over $D(\mathcal{T},\pi)$ for 10 runs including the mean
- $\tilde{Q}(\mathcal{T},\pi)$ statistics for 10 runs including the median
- $\tilde{P}(\mathcal{T},\pi)$ statistics for 10 runs including the median

As already mentioned, we first present the results for the HeuristicAutonomicManager to establish a normalized TrajectoryQuality and normalized PlannerPerformance baseline with ten manual runs for two ManagedPlanners per Scenario and five Scenarios per FlightPhase resulting in 500 TrajectoryQuality and PlannerPerformance measurements in total. We then perform 1000 automatic SMAC evaluation training runs for each FlightPhase and ManagedPlanner combination using a tight cut-off with respect to the established baseline DurationQuantity of the HeuristicAutonomicManager. If the obtained best-on-average PlannerPerformance (*incumbent* configuration) of the SmacAutonomicManager is able to outperform the HeuristicAutonomicManager for the training Scenarios, we proceed to randomly generating test Scenarios and

comparing the individual PlannerTunings of the HeuristicAutonomicManager with the best-on-average PlannerTunings. The latter are stored in the Reputation KnowledgeBase of the SmacAutonomicManager, as explained in Section 9.4.

11.3 Cruise

The *cruise* FlightPhase evaluation and training is performed with five associated Scenarios. Recall from Section 9.2 that no RiskPolicy-exceeding SWIM Obstacles may be present within the Aircraft safety sphere and POI bounding box for this FlightPhase. Only the remaining parts of the planning Environment may feature such RiskPolicy-exceeding Obstacles. As a consequence, the *cruise* FlightPhase permits feasible straight legs between the consecutive POIs of the corresponding Scenario. However, the TrajectoryQuality can potentially be improved by avoiding existing non-RiskPolicy-exceeding SWIM Obstacles depending on the required detour. Figure 11.2 shows a ManagedTreePlanner for a typical *cruise* FlightPhase with the Aircraft exposed to an all-enclosing low-cost SWIM airspace (cyan) and RRT probes (dashed cyan) as well as the currently computed and incrementally improving Trajectory (magenta).

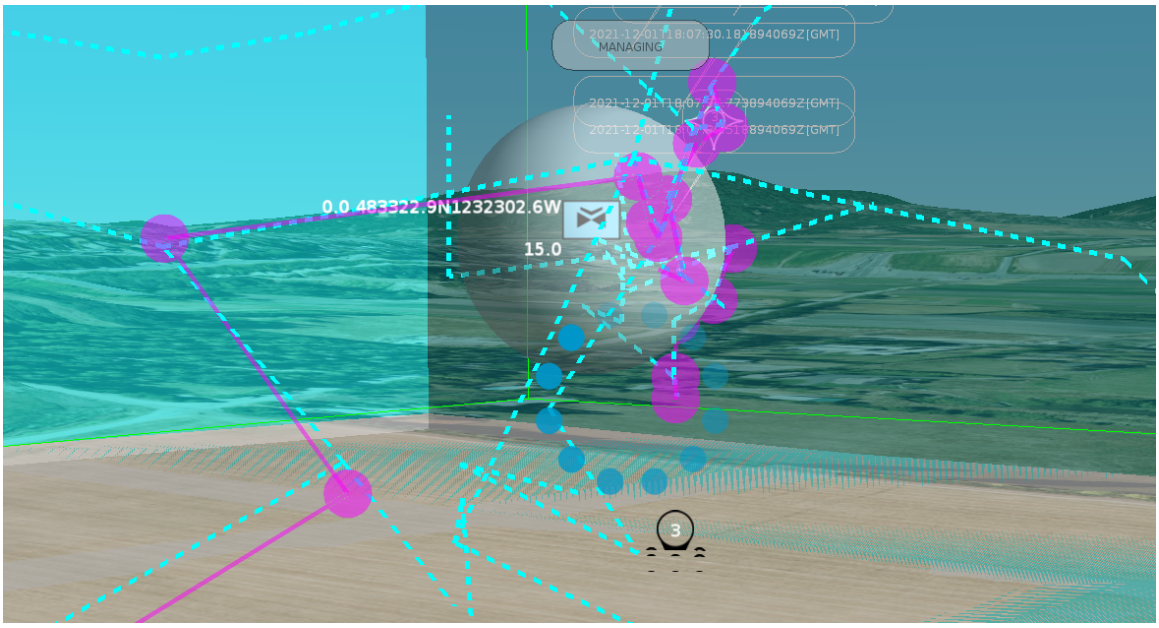


Figure 11.2: Managed Cruise Scenario

11.3.1 Cruise Scenario 1

Cruise Scenario 1 is a SWIM Obstacle-free Scenario with only the start and goal POIs at an altitude of 50 meters ASL each, that is, no intermediate POIs are present. The Scenario is shown in Figures 11.3 for the PlanningGrid and PlanningContinuum, respectively.

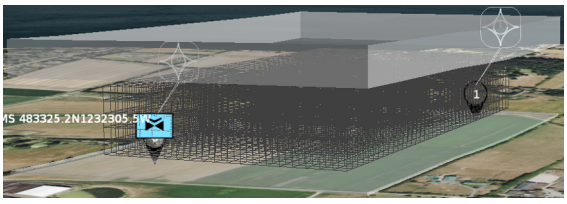
Without any SWIM Obstacles within the Aircraft safety sphere, POI bounding box, and remaining Environment, both the Severity and Difficulty of this Scenario are considered to be *low*. The *low* Severity and Difficulty result in the *cruise* FlightPhase association shown in Tables 11.2.

Since there are no SWIM Obstacles, a feasible straight leg represents the highest TrajectoryQuality. Such a straight leg can be produced by our ManagedTreePlanner given the appropriate PlannerTuning. In contrast, since the PlanningGrid of our ManagedGridPlanner is kept at a constant division of 40 cubic voxels, as described in Section 11.2, as well as being no any-angle Planner, it cannot produce the same TrajectoryQuality.

The HeuristicAutonomicManager applies the HeuristicPlannerTunings shown in Tables 11.3. It is instantly apparent that the HeuristicPlannerTuning for the ManagedGridPlanner is not as advantageous as the one for the ManagedTreePlanner. Selecting the Environment base cost to be the inflation factor essentially reduces the ManagedGridPlanner to a simple A* algorithm. Without a higher inflation A* progresses slowly through the relatively high-resolution grid. Clearly, a more aggressive inflation without any deflation or a lower resolution PlanningGrid would improve the PlannerPerformance of the ManagedGridPlanner in this Scenario. The ManagedTreePlanner, in contrast, is tuned to deterministically sample the goal POI and extend to it in a single step. Notice that this approach might be too optimistic given that the Features horizon only covers only approximately $\frac{1}{3}$ of the ETE. Therefore, RiskPolicy-exceeding SWIM Obstacles beyond the Features horizon may very well prevent the computation of a feasible Trajectory using a goal bias of 100%. Thus, a lower value would generally be more appropriate and allow for detours if required.

Given these two HeuristicPlannerTunings, we can confirm the deterministic normalized TrajectoryQuality of both ManagedPlanners in Figures 11.4. As mentioned before, the achieved normalized TrajectoryQuality of the ManagedGridPlanner has to be lower than the one of the ManagedTreePlanner due to the applied constant PlanningGrid division. The DurationQuantity of both ManagedPlanners is

also significantly different as a result of the poorer `HeuristicPlannerTuning` of the `ManagedGridPlanner`. Notice that while the `TrajectoryQuality` is a constant for all 10 runs of each `ManagedPlanner`, the `DurationQuantity` changes slightly as a result of the scheduling policy of our runtime environment, which leads to a decreasing normalized `PlannerPerformance` with increasing `DurationQuantity`.



(a) Cruise 1 Grid



(b) Cruise 1 Continuum

Figure 11.3: Cruise 1 Scenario

Table 11.2: Cruise 1 Feature Categories

(a) Cruise 1 Severity

Cost Assessment	ASS	POI	ENV
Maximum	0.0	0.0	0.0
Minimum	0.0	0.0	0.0
Risk Policy	0.0	0.0	0.0

(b) Cruise 1 Criticality

Criticality	Level
Difficulty	low (∞)
Severity	low
Flight Phase	cruise

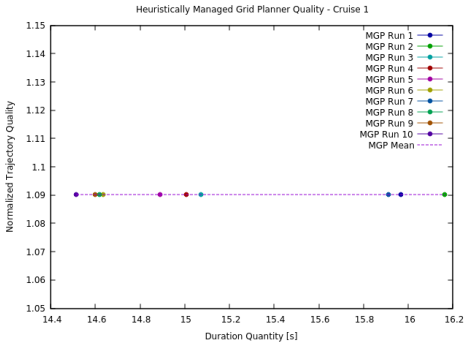
Table 11.3: Cruise 1 Heuristic Tunings

(a) Cruise 1 MGP Heuristic Tuning

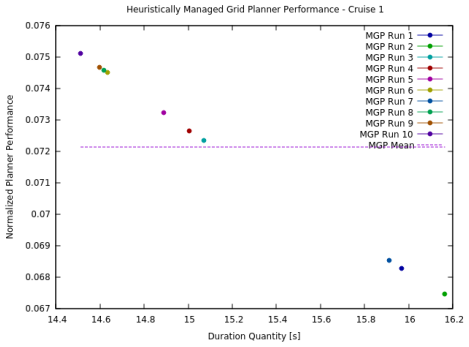
Property	Value
`costPolicy`	`AVERAGE`
`riskPolicy`	`SAFETY`
`minimumQuality`	1.0
`maximumQuality`	1.0
`qualityImprovement`	0.1
`significantChange`	0.01

(b) Cruise 1 MTP Heuristic Tuning

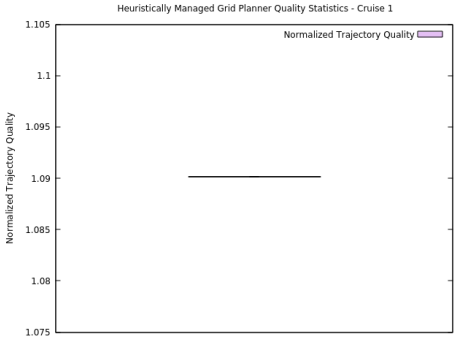
Property	Value
`costPolicy`	`AVERAGE`
`riskPolicy`	`SAFETY`
`maximumIterations`	1
`bias`	100
`epsilon`	561.1581...
`neighborLimit`	1
`goalThreshold`	0.0
`extension`	`FEASIBLE`
`strategy`	`CONNECT`
`sampling`	`ELLIPSOIDAL`
`minimumQuality`	0.0
`maximumQuality`	0.0
`qualityImprovement`	0.001
`significantChange`	0.01



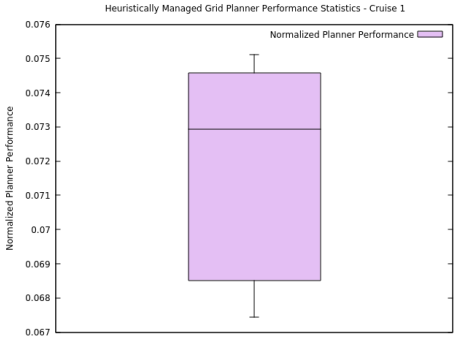
(a) H-AM MGP Quality



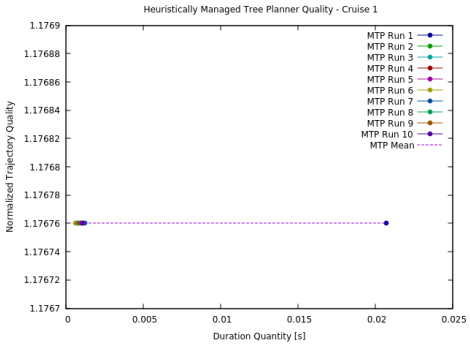
(b) H-AM MGP Performance



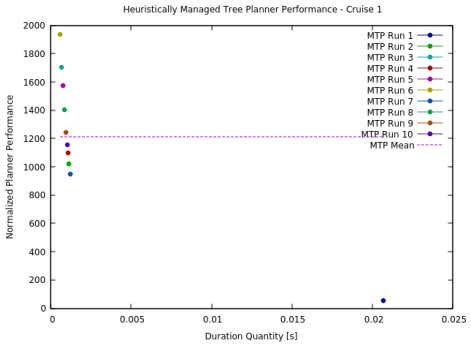
(c) H-AM MGP Quality Statistics



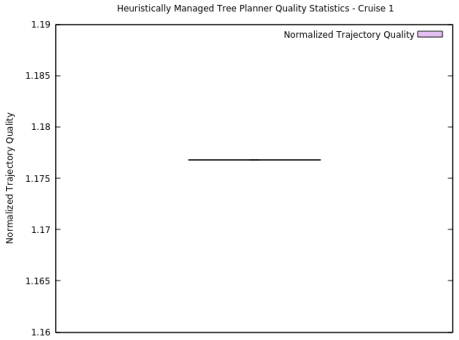
(d) H-AM MGP Performance Statistics



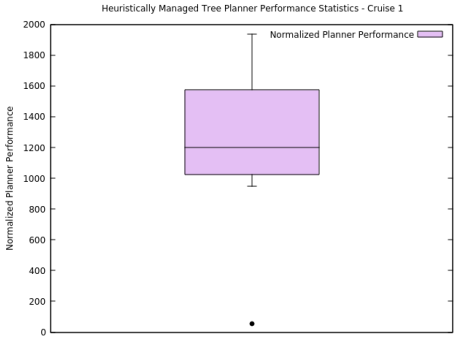
(e) H-AM MTP Quality



(f) H-AM MTP Performance



(g) H-AM MTP Quality Statistics



(h) H-AM MTP Performance Statistics

Figure 11.4: H-AM Quality and Performance: Cruise 1

11.3.2 Cruise Scenario 2

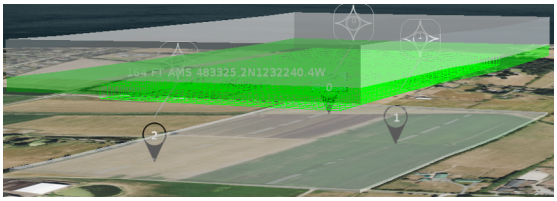
Cruise Scenario 2 consists of an Environment featuring a single large non-RiskPolicy-exceeding SWIM Obstacle that does not penetrate the Aircraft safety sphere or POI bounding box. The Scenario consists of one intermediate POI at an altitude of 60 meters ASL that has to be visited when traveling from the start to the goal POI, each at an altitude of 50 meters ASL, and is shown in Figures 11.5 from different directions for the PlanningGrid and PlanningContinuum, respectively.

With only one large SWIM Obstacle of cost 20 (light green) within the remaining Environment, the Difficulty of this Scenario is considered to be *low* (∞) while its Severity is *low* to *substantial*, resulting in the *cruise* FlightPhase association shown in Tables 11.4. Since there are no SWIM Obstacles within the POI bounding box, a feasible straight leg represents the highest TrajectoryQuality and the same statements made for Cruise Scenario 1 in Section 11.3.1 apply.

The HeuristicAutonomicManager applies the HeuristicPlannerTunings shown in Tables 11.5. It is again apparent that the HeuristicPlannerTuning for the ManagedGridPlanner is not as advantageous as the one for the ManagedTreePlanner for the same reasons as discussed in Section 11.3.1. However, with consecutive POIs being closer to each other, the ManagedGridPlanner has to examine a relatively smaller portion of the PlanningGrid using the Environment base-cost-deflated heuristic for each part of the computed multi-part Trajectory. The ManagedTreePlanner is again tuned to deterministically sample the intermediate and goal POIs and extend to them in a single step.

Given these two HeuristicPlannerTunings, we can confirm the deterministic normalized TrajectoryQuality of both ManagedPlanners in Figures 11.6 with the same properties as discussed in Section 11.3.1. However, having one intermediate POI leads to consecutive computations of a partial and complete Trajectory with their respective TrajectoryQuality. Since the computed overall Trajectory has an increased length, its normalized TrajectoryQuality has to be lower than for Cruise Scenario 1 of Section 11.3.1. Furthermore, the normalized TrajectoryQuality of the goal POI now depends on the one measured for the intermediate POI and, hence, both are connected with a line. The normalized TrajectoryQuality of both ManagedPlanners essentially remains constant, that is, two straight legs in case of the ManagedTreePlanner and two grid-constrained legs that differ only minimally due to the positions of the POIs within the PlanningGrid in case of the ManagedGridPlanner. Again, the decreased overall

DurationQuantity in case of the ManagedGridPlanner can be explained with the reduced search space for each Trajectory part using a deflated heuristic. The increased DurationQuantity for the second POI in case of the ManagedTreePlanner relates to computational overhead when transitioning from one part of a multi-part Trajectory to the next.



(a) Cruise 2 Grid



(b) Cruise 2 Continuum

Figure 11.5: Cruise 2 Scenario

Table 11.4: Cruise 2 Feature Categories

(a) Cruise 2 Severity

Cost Assessment	ASS	POI	ENV
Maximum	0.0	0.0	20.0
Minimum	0.0	0.0	20.0
Risk Policy	0.0	0.0	20.0

(b) Cruise 2 Criticality

Criticality	Level
Difficulty	low (∞)
Severity	low, moderate, substantial
Flight Phase	cruise

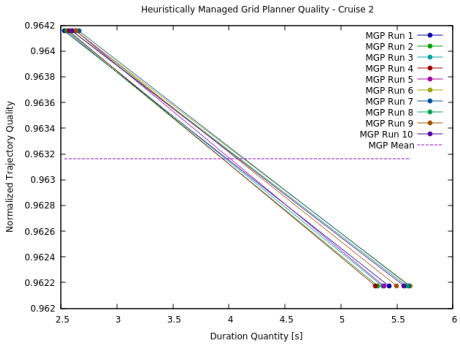
Table 11.5: Cruise 2 Heuristic Tunings

(a) Cruise 2 MGP Heuristic Tuning

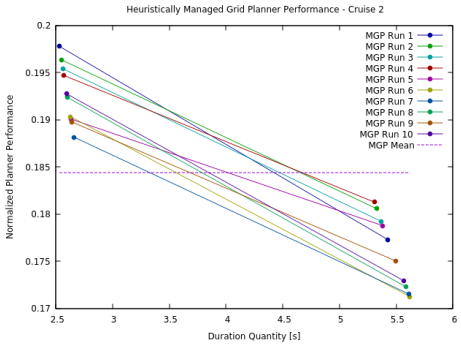
Property	Value
costPolicy	AVERAGE
riskPolicy	SAFETY
minimumQuality	1.0
maximumQuality	1.0
qualityImprovement	0.1
significantChange	0.01

(b) Cruise 2 MTP Heuristic Tuning

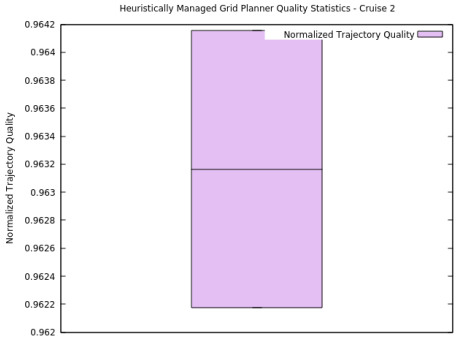
Property	Value
costPolicy	AVERAGE
riskPolicy	SAFETY
maximumIterations	1
bias	100
epsilon	336.6866...
neighborLimit	1
goalThreshold	0.0
extension	FEASIBLE
strategy	CONNECT
sampling	ELLIPSOIDAL
minimumQuality	0.0
maximumQuality	0.0
qualityImprovement	0.001
significantChange	0.01



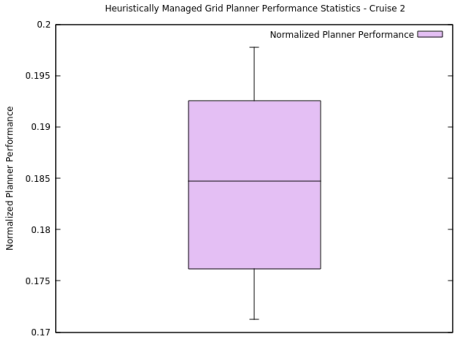
(a) H-AM MGP Quality



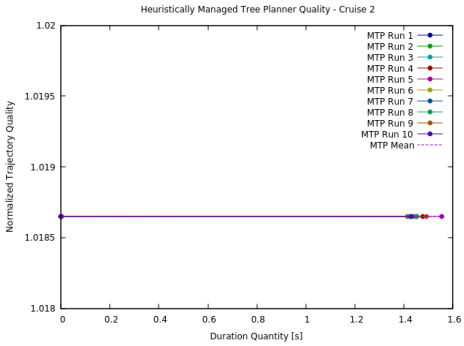
(b) H-AM MGP Performance



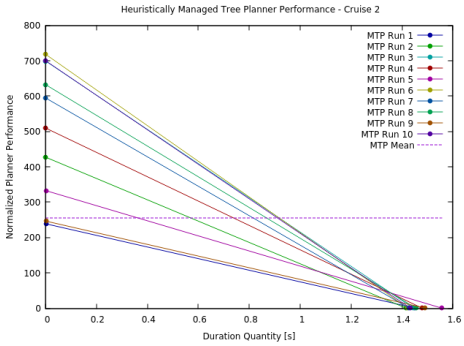
(c) H-AM MGP Quality Statistics



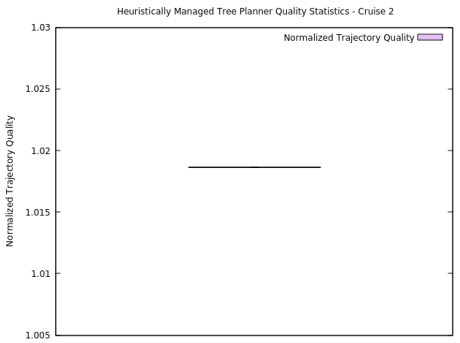
(d) H-AM MGP Performance Statistics



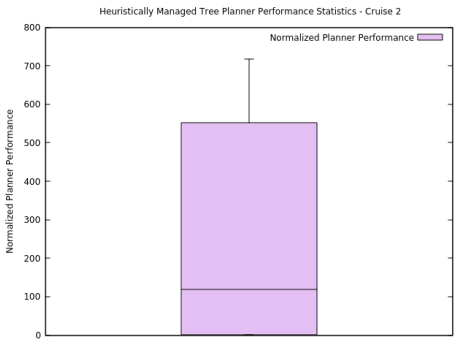
(e) H-AM MTP Quality



(f) H-AM MTP Performance



(g) H-AM MTP Quality Statistics



(h) H-AM MTP Performance Statistics

Figure 11.6: H-AM Quality and Performance: Cruise 2

11.3.3 Cruise Scenario 3

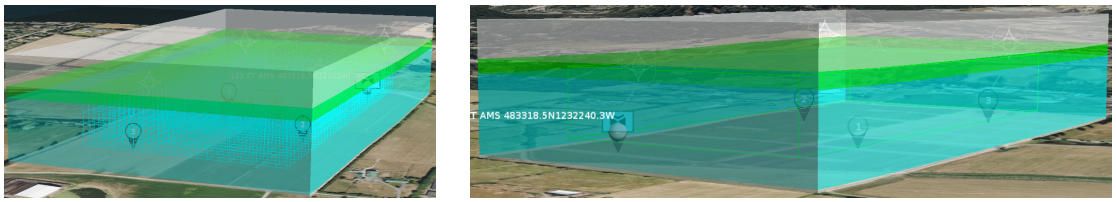
Cruise Scenario 3 consists of an Environment featuring two large non-RiskPolicy-exceeding SWIM Obstacles, one of which penetrates both the Aircraft safety sphere and POI bounding box as it, together with the first Obstacle, occupies the entire Environment and cannot be avoided. The Scenario consists of two intermediate POIs, each at an altitude of 50 meters ASL, that have to be visited in sequence when traveling from the start to the goal POI, each at an altitude of 40 meters ASL, and is shown in Figures 11.7 from different directions for the PlanningGrid and PlanningContinuum, respectively.

With only two large SWIM Obstacles of cost 20 (light green) and 5 (light cyan) within the Environment, the Difficulty of this Scenario is still considered to be *low* (1.0) while its Severity is now *substantial*, resulting in the *cruise* FlightPhase association shown in Tables 11.6. Even with the large SWIM Obstacles within the POI bounding box, a feasible straight leg still represents the highest TrajectoryQuality and the same statements made for Cruise Scenario 1 in Section 11.3.1 apply.

The HeuristicAutonomicManager applies the HeuristicPlannerTunings shown in Tables 11.7. The Obstacle-contaminated POI bounding box now leads to a different tuning approach that considers more Features of the planning Environment, as explained in Section 9.3. In particular, the heuristic inflation factor of the ManagedGridPlanner is adjusted to reduce the search space in a Scenario of lower Difficulty and higher Severity. Similarly, the ManagedTreePlanner is tuned to be non-deterministic and allows for goal-biased probabilistic samples and shorter tree extensions. However, the distance bias is very high and only decreased to 75% (25% cost bias). The goal bias is still considerable at 10%, and the sampling neighborhood is limited to only 1.

Given these two HeuristicPlannerTunings, we can confirm the deterministic normalized TrajectoryQuality of the ManagedGridPlanner as well as the non-deterministic normalized TrajectoryQuality of the ManagedTreePlanner in Figures 11.8. Having two intermediate POIs, consequently, leads to consecutive computations of two partial and one complete Trajectory with their respective TrajectoryQuality. Hence, at least 3 TrajectoryQuality points are shown in the corresponding diagrams. However, since our ManagedPlanners are specialized AnytimePlanners, multiple TrajectoryQuality points may be the result of incrementally improving one part of the multi-part Trajectory before proceeding to the next. Therefore, whenever a de-

creasing TrajectoryQuality is found, it is associated with the progression to a subsequent part. The ManagedTreePlanner is again constrained by the PlanningGrid, which leads to a slightly decreased normalized TrajectoryQuality during the initial climb and final descent compared to the intermediate level leg. As opposed to the ManagedTreePlanner, it does not find any incremental partial Trajectory with higher normalized TrajectoryQuality. Recall the discussion about the *effective* versus set sub-optimality bound in Section 6.3 for the corresponding rationale. The relatively low DurationQuantity for both ManagedPlanners is the result of the high heuristic inflation factor and distance bias, respectively.



(a) Cruise 3 Grid **(b)** Cruise 3 Continuum

Figure 11.7: Cruise 3 Scenario

Table 11.6: Cruise 3 Feature Categories

(a) Cruise 3 Severity

Cost Assessment	ASS	POI	ENV
Maximum	5.0	5.0	20.0
Minimum	5.0	5.0	5.0
Risk Policy	5.0	5.0	25.0

(b) Cruise 3 Criticality

Criticality	Level
Difficulty	low (1.0)
Severity	substantial
Flight Phase	cruise

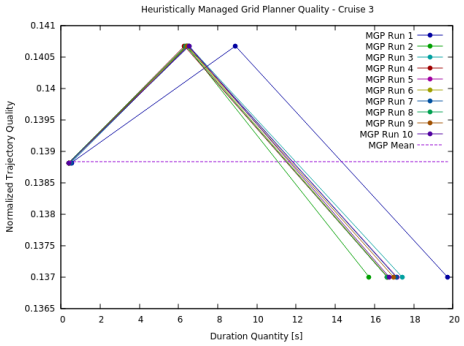
Table 11.7: Cruise 3 Heuristic Tunings

(a) Cruise 3 MGP Heuristic Tuning

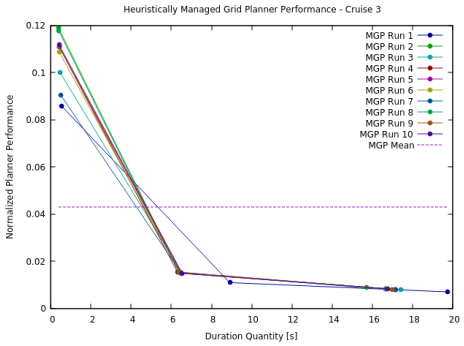
Property	Value
costPolicy	AVERAGE
riskPolicy	SAFETY
minimumQuality	6.0
maximumQuality	4.5
qualityImprovement	0.3
significantChange	0.5

(b) Cruise 3 MTP Heuristic Tuning

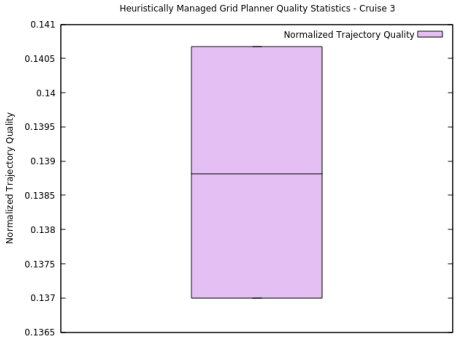
Property	Value
costPolicy	AVERAGE
riskPolicy	SAFETY
maximumIterations	500
bias	10
epsilon	13.5917...
neighborLimit	1
goalThreshold	2.5
extension	FEASIBLE
strategy	CONNECT
sampling	ELLIPSOIDAL
minimumQuality	0.0
maximumQuality	0.25
qualityImprovement	0.05
significantChange	0.5



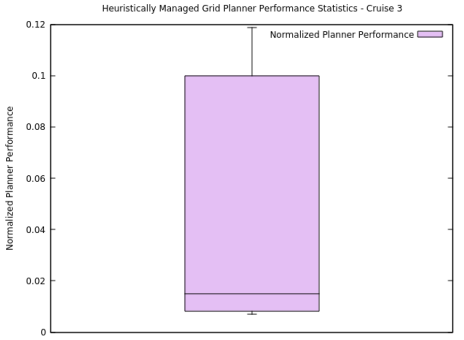
(a) H-AM MGP Quality



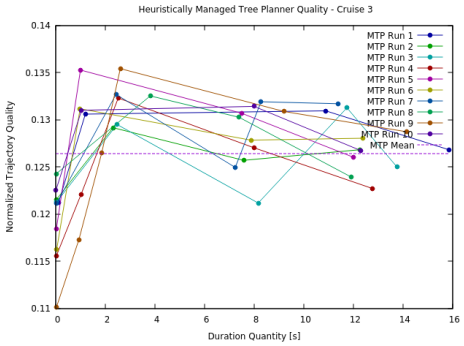
(b) H-AM MGP Performance



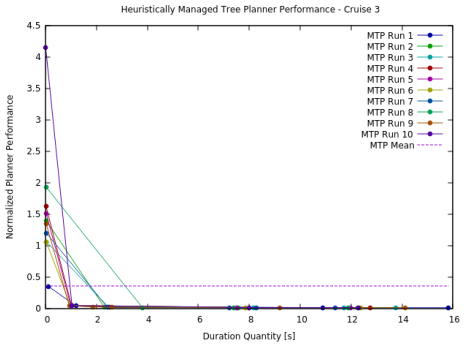
(c) H-AM MGP Quality Statistics



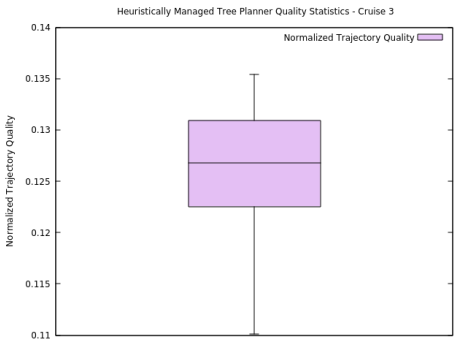
(d) H-AM MGP Performance Statistics



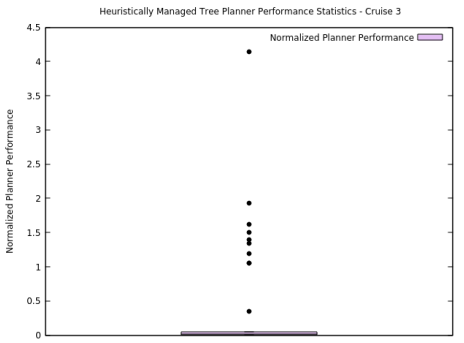
(e) H-AM MTP Quality



(f) H-AM MTP Performance



(g) H-AM MTP Quality Statistics



(h) H-AM MTP Performance Statistics

Figure 11.8: H-AM Quality and Performance: Cruise 3

11.3.4 Cruise Scenario 4

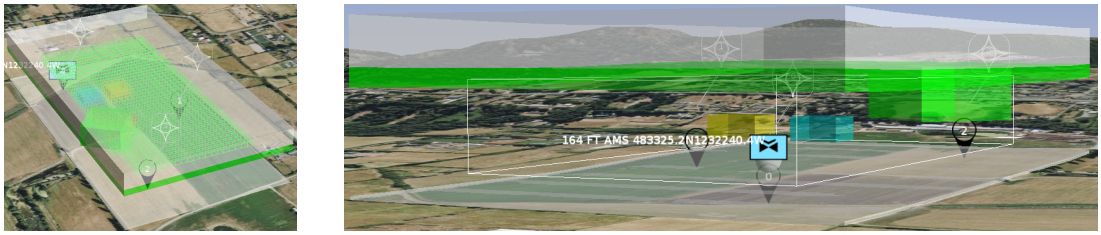
Cruise Scenario 4 consists of an Environment featuring one large as well as several smaller non-RiskPolicy-exceeding SWIM Obstacles, two of which penetrate the POI bounding box. The Scenario consists of one intermediate POI at an altitude of 60 meters ASL that has to be visited when traveling from the start to the goal POI, each at an altitude of 50 meters ASL, and is shown in Figures 11.7 from different directions for the PlanningGrid and PlanningContinuum, respectively.

With one large and three smaller SWIM Obstacles of cost 20 (light green), 10 (cyan), 20 (light green) and 40 (yellow), respectively, within the Environment, the Difficulty of this Scenario increases to *substantial* (0.5) while its Severity is now *substantial* to *severe*, resulting in the *cruise* and *transition* FlightPhase associations shown in Tables 11.8. As the cyan and yellow SWIM Obstacles contaminate the POI bounding box (at lower altitudes) within the Features horizon, resulting in a RiskPolicy-assessed cost of 50, a feasible straight first leg does not represent the highest TrajectoryQuality anymore. The remaining planning Environment is affected by all (including the higher altitude) SWIM Obstacles within the Features horizon exceeding the operational RiskPolicy. Recall the discussed cost computation in Section 5.2.5 to derive the associated values in Table 11.8a.

The HeuristicAutonomicManager applies the HeuristicPlannerTunings shown in Tables 11.9. The Obstacle-contaminated POI bounding box leads to a tuning approach that considers more Features of the planning Environment, as explained in Section 9.3. In particular, the heuristic inflation factor of the ManagedGridPlanner is adjusted to reflect the relevant SWIM Obstacle costs within the contaminated POI bounding box. Similarly, the ManagedTreePlanner is tuned to be non-deterministic and to apply an EXTEND growth Strategy while considering a slightly larger sampling neighborhood of 3.

Given these two HeuristicPlannerTunings, we can confirm the deterministic normalized TrajectoryQuality of the ManagedGridPlanner as well as the non-deterministic normalized TrajectoryQuality of the ManagedTreePlanner in Figures 11.10. Having one intermediate POI, consequently, leads to consecutive computations of one partial and one complete Trajectory with their respective TrajectoryQuality. Thus, at least 2 TrajectoryQuality points are shown in the corresponding diagrams with potentially additional ones representing the incremental improvement of a single part. Since only the first part is affected by SWIM Obstacles, the normalized

`TrajectoryQuality` improvement shown in the corresponding diagrams is expected. The relatively low `DurationQuantity` for the `ManagedGridPlanner` is the result of the high heuristic inflation factor and, although not for this `Scenario`, may turn out to be too aggressive to avoid lower cost `Obstacles`.



(a) Cruise 4 Grid **(b)** Cruise 4 Continuum

Figure 11.9: Cruise 4 Scenario

Table 11.8: Cruise 4 Feature Categories

(a) Cruise 4 Severity

Cost Assessment	ASS	POI	ENV
Maximum	0.0	40.0	40.0
Minimum	0.0	10.0	10.0
Risk Policy	0.0	50.0	∞

(b) Cruise 4 Criticality

Criticality	Level
Difficulty	substantial (0.5)
Severity	substantial, severe
Flight Phase	cruise, transition

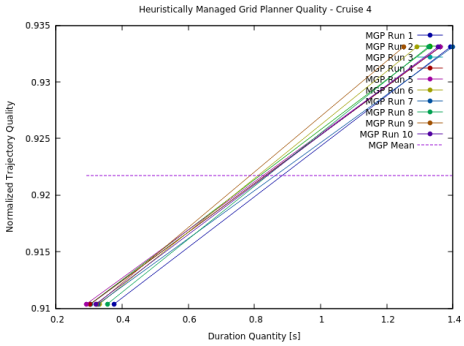
Table 11.9: Cruise 4 Heuristic Tunings

(a) Cruise 4 MGP Heuristic Tuning

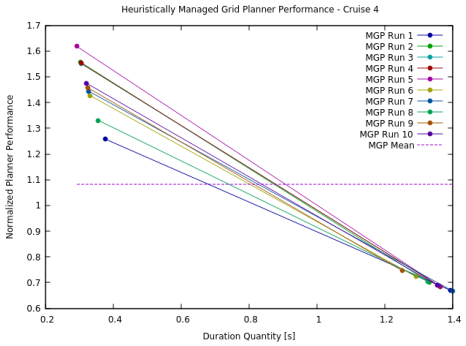
Property	Value
`costPolicy`	`AVERAGE`
`riskPolicy`	`SAFETY`
`minimumQuality`	26.0
`maximumQuality`	19.5
`qualityImprovement`	1.3
`significantChange`	0.5

(b) Cruise 4 MTP Heuristic Tuning

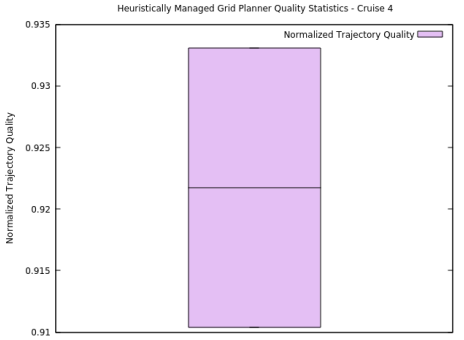
Property	Value
`costPolicy`	`AVERAGE`
`riskPolicy`	`SAFETY`
`maximumIterations`	500
`bias`	10
`epsilon`	16.8343...
`neighborLimit`	3
`goalThreshold`	2.5
`extension`	`FEASIBLE`
`strategy`	`EXTEND`
`sampling`	`ELLIPSOIDAL`
`minimumQuality`	0.0
`maximumQuality`	0.25
`qualityImprovement`	0.05
`significantChange`	0.5



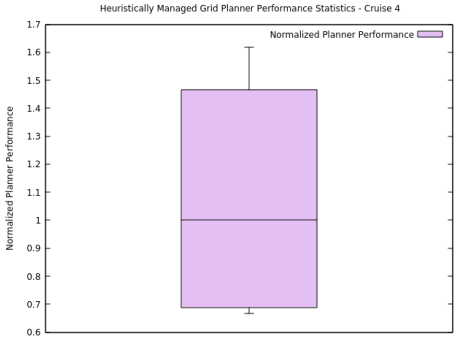
(a) H-AM MGP Quality



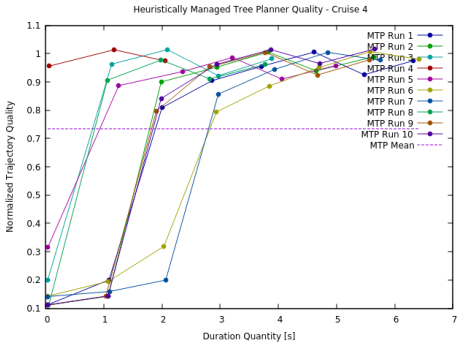
(b) H-AM MGP Performance



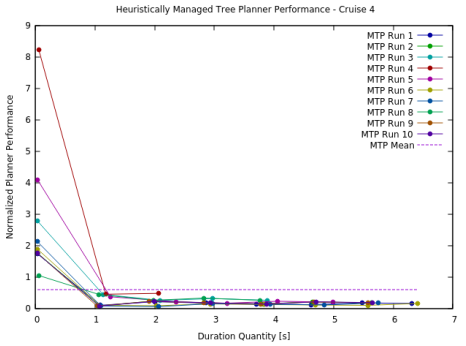
(c) H-AM MGP Quality Statistics



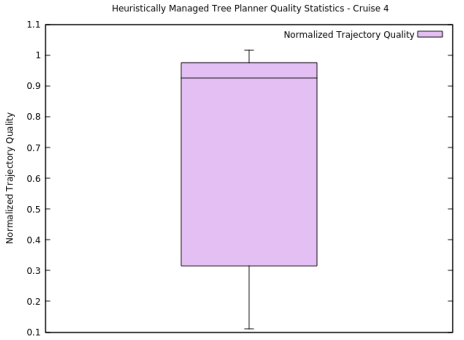
(d) H-AM MGP Performance Statistics



(e) H-AM MTP Quality



(f) H-AM MTP Performance



(g) H-AM MTP Quality Statistics



(h) H-AM MTP Performance Statistics

Figure 11.10: H-AM Quality and Performance: Cruise 4

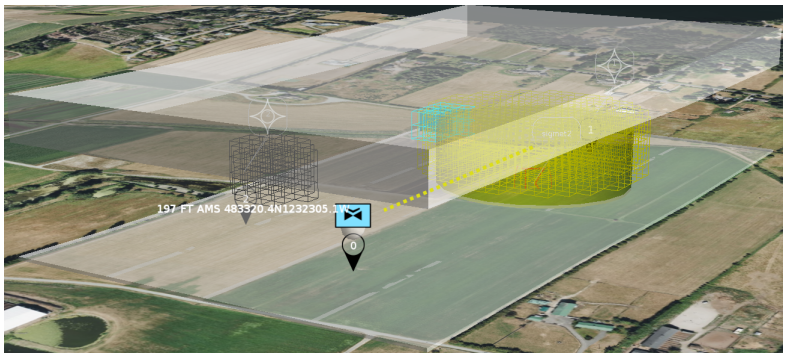
11.3.5 Cruise Scenario 5

Cruise Scenario 5 consists of an Environment featuring one larger moving as well as two smaller stationary non-RiskPolicy-exceeding SWIM Obstacles, all of which penetrate the POI bounding box. The Scenario consists of one intermediate POI at an altitude of 40 meters ASL that has to be visited when traveling from the start to the goal POI, at altitudes of 60 and 50 meters ASL, respectively, and is shown in Figures 11.11 from different directions for both the PlanningGrid and PlanningContinuum.

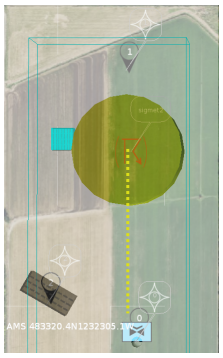
With the stationary two smaller and single moving larger SWIM Obstacles of cost 3 (gray), 7 (light cyan), and 40 (yellow), respectively, within the Environment, the Difficulty of this Scenario becomes *moderate* (0.7756...) while its Severity is *moderate* to *substantial*, resulting in the *cruise* FlightPhase association shown in Tables 11.10. Since all present SWIM Obstacles contaminate the POI bounding box within the Features horizon, the same RiskPolicy-assessed cost of 50 results for the entire Environment. Feasible straight legs do not represent the highest TrajectoryQuality as the yellow and light cyan Obstacle can be avoided fully and the gray one, surrounding the goal POI, at least partially.

The HeuristicAutonomicManager applies the HeuristicPlannerTunings shown in Tables 11.11. The Obstacle-contaminated POI bounding box leads to a tuning approach that considers more Features of the planning Environment, as explained in Section 9.3. In particular, the heuristic inflation factor of the ManagedGridPlanner is adjusted to reflect the relevant SWIM Obstacle costs within the contaminated POI bounding box. Similarly, the ManagedTreePlanner is tuned to be non-deterministic considering a relatively small sampling neighborhood of 2.

Given these two HeuristicPlannerTunings, we can confirm the deterministic normalized TrajectoryQuality of the ManagedGridPlanner as well as the non-deterministic normalized TrajectoryQuality of the ManagedTreePlanner in Figures 11.12. Having one intermediate POI, consequently, leads to consecutive computations of one partial and one complete Trajectory with their respective TrajectoryQuality. Thus, at least 2 TrajectoryQuality points are shown in the corresponding diagrams with potentially additional ones representing the incremental improvement of a single part. The ManagedGridPlanner still avoids all Obstacles within a relatively short DurationQuantity albeit the fairly aggressive heuristic inflation. The higher normalized TrajectoryQuality of the ManagedTreePlanner is again the result of not being constrained by a discretized planning Environment, such as the PlanningGrid.



(a) Cruise 5 Grid



(b) Cruise 5 Continuum

Figure 11.11: Cruise 5 Scenario

Table 11.10: Cruise 5 Feature Categories

(a) Cruise 5 Severity

Cost Assessment	ASS	POI	ENV
Maximum	0.0	40.0	40.0
Minimum	0.0	3.0	3.0
Risk Policy	0.0	50.0	50.0

(b) Cruise 5 Criticality

Criticality	Level
Difficulty	moderate (0.7756…)
Severity	moderate, substantial
Flight Phase	cruise

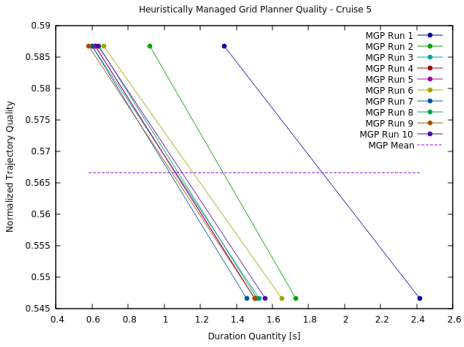
Table 11.11: Cruise 5 Heuristic Tunings

(a) Cruise 5 MGP Heuristic Tuning

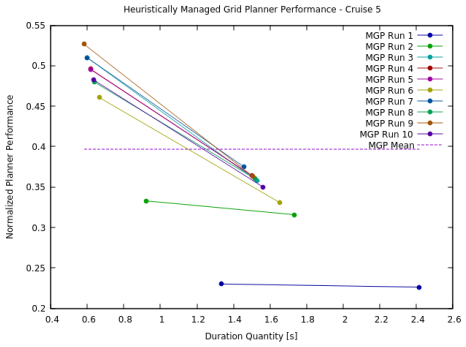
Property	Value
`costPolicy`	`AVERAGE`
`riskPolicy`	`SAFETY`
`minimumQuality`	$17.\overline{6}$
`maximumQuality`	13.25
`qualityImprovement`	$0.88\overline{3}$
`significantChange`	0.5

(b) Cruise 5 MTP Heuristic Tuning

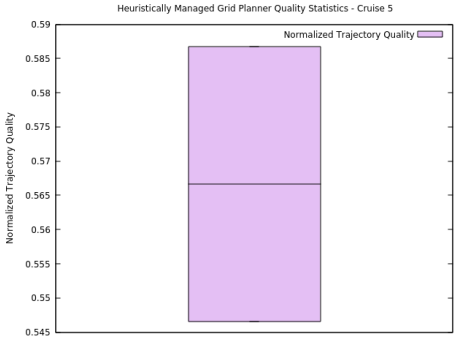
Property	Value
`costPolicy`	`AVERAGE`
`riskPolicy`	`SAFETY`
`maximumIterations`	500
`bias`	10
`epsilon`	24.7876…
`neighborLimit`	2
`goalThreshold`	2.5
`extension`	`FEASIBLE`
`strategy`	`CONNECT`
`sampling`	`ELLIPSOIDAL`
`minimumQuality`	0.0
`maximumQuality`	0.25
`qualityImprovement`	0.05
`significantChange`	0.5



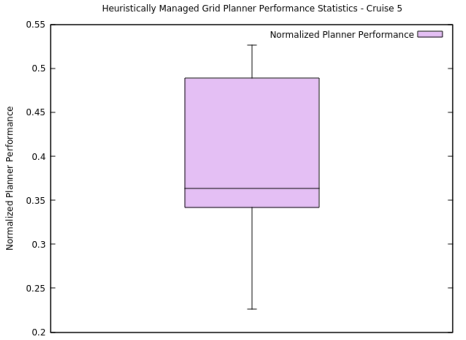
(a) H-AM MGP Quality



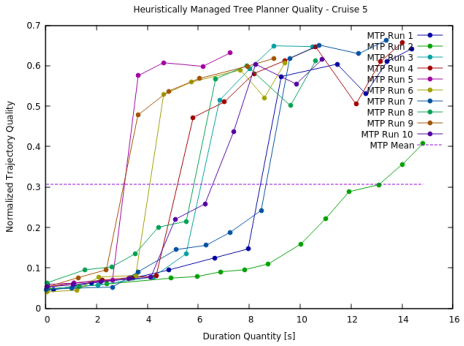
(b) H-AM MGP Performance



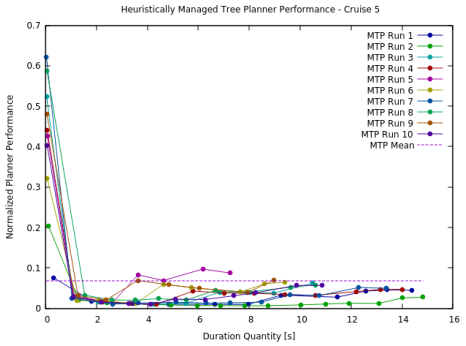
(c) H-AM MGP Quality Statistics



(d) H-AM MGP Performance Statistics



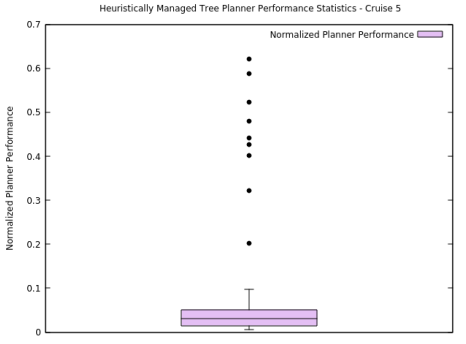
(e) H-AM MTP Quality



(f) H-AM MTP Performance



(g) H-AM MTP Quality Statistics



(h) H-AM MTP Performance Statistics

Figure 11.12: H-AM Quality and Performance: Cruise 5

11.4 Transition

The *transition* `FlightPhase` evaluation and training is performed with five associated `Scenarios`. Recall from Section 9.2 that this `FlightPhase` is equivalent to the *cruise* `FlightPhase` in terms of the sensed `Difficulty` from the extracted `Features` of a given `Scenario`. However, its higher *severe* `Severity` allows `RiskPolicy`-exceeding SWIM `Obstacles` within the POI bounding box for this `FlightPhase`, which may require a feasible detour or prevent the computation of a `Trajectory` that satisfies the operational `RiskPolicy` altogether. Figure 11.13 shows a `ManagedGridPlanner` for a typical *transition* `FlightPhase` with the currently planned `Aircraft` `Trajectory` (magenta) avoiding a moving `RiskPolicy`-exceeding adverse SWIM weather space (red) and its affected voxels of the `PlanningGrid`.

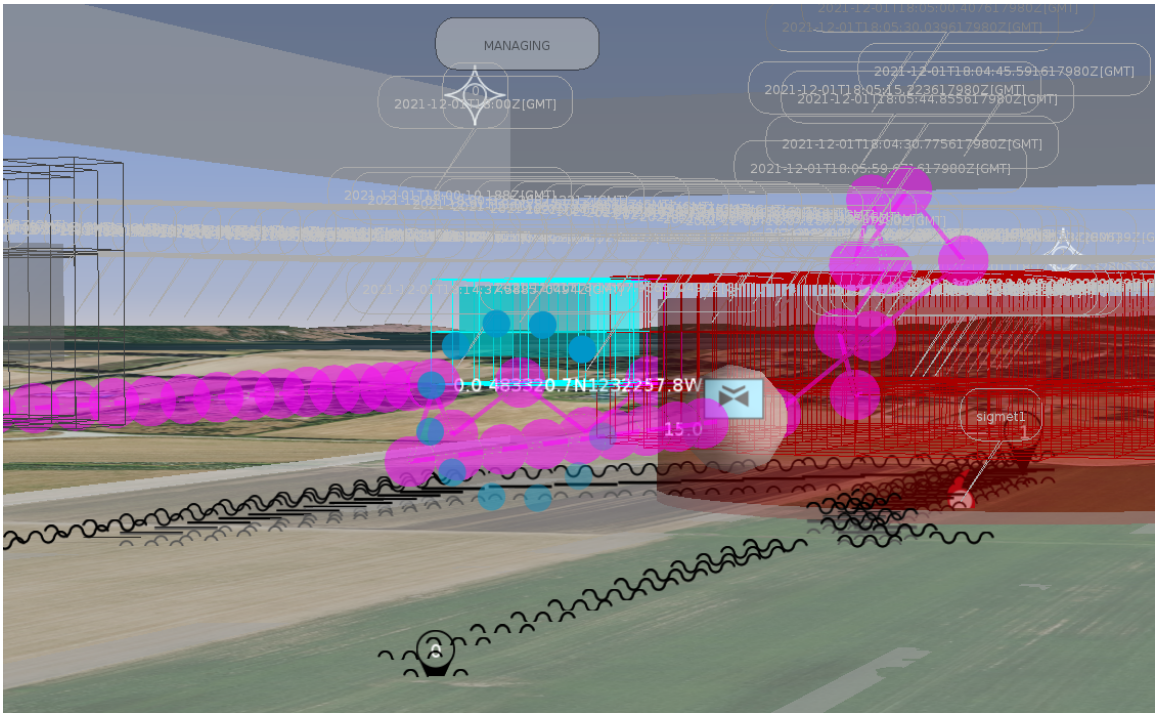


Figure 11.13: Managed Transition Scenario

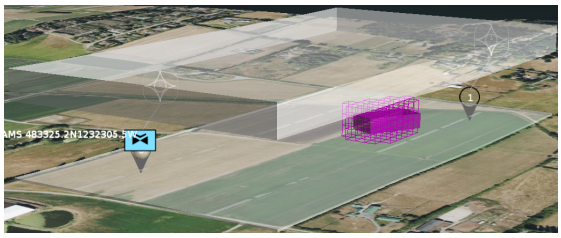
11.4.1 Transition Scenario 1

Transition Scenario 1 consists of an Environment featuring one stationary small RiskPolicy-exceeding SWIM Obstacle which penetrates the POI bounding box. The Scenario consists of only the start and goal POIs, each at an altitude of 50 meters ASL, and is shown in Figures 11.14 from different directions for the PlanningGrid and PlanningContinuum, respectively.

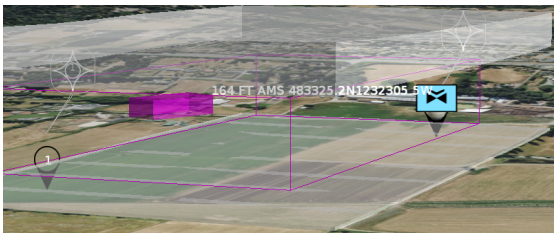
With only the single small stationary but expensive SWIM Obstacle of cost 70 (magenta) within the POI bounding box, the Difficulty of this Scenario is considered to be *low* (1.0) while its Severity is *severe*, resulting in the *transition* FlightPhase association shown in Tables 11.12. Since the present SWIM Obstacle contaminates the POI bounding box within the Features horizon, the same RiskPolicy-assessed cost of 70 results for the entire Environment. Feasible straight legs do not represent the highest TrajectoryQuality and may not be possible without exceeding the operational RiskPolicy.

The HeuristicAutonomicManager applies the HeuristicPlannerTunings shown in Tables 11.13. The Obstacle-contaminated POI bounding box leads to a tuning approach that considers more Features of the planning Environment, as explained in Section 9.3. In particular, the heuristic inflation factor of the ManagedGridPlanner is adjusted to reflect the RiskPolicy-exceeding SWIM Obstacle costs within the contaminated POI bounding box and deflate this factor incrementally to achieve a 50% tighter sub-optimality bound. Similarly, the ManagedTreePlanner is tuned to be non-deterministic using a maximum of 1000 iterations to compute a feasible Trajectory while proceeding from a 100% to a 50% distance bias (50% cost bias). Its goal bias is tuned to a moderate 5% and the neighborhood to the minimum of 1 given the *low* Difficulty of this Scenario.

Given these two HeuristicPlannerTunings, we can confirm the deterministic normalized TrajectoryQuality of the ManagedGridPlanner as well as the non-deterministic normalized TrajectoryQuality of the ManagedTreePlanner in Figures 11.15. Having no intermediate POIs immediately leads to the computation of the complete Trajectory, which in the case of the ManagedTreePlanner takes one or multiple non-deterministic steps with increasing normalized TrajectoryQuality. The slightly higher normalized TrajectoryQuality of the ManagedTreePlanner is again the result of not being constrained by a discretized planning Environment, such as the PlanningGrid.



(a) Transition 1 Grid



(b) Transition 1 Continuum

Figure 11.14: Transition 1 Scenario

Table 11.12: Transition 1 Feature Categories

(a) Transition 1 Severity

Cost Assessment	ASS	POI	ENV
Maximum	0.0	70.0	70.0
Minimum	0.0	70.0	70.0
Risk Policy	0.0	∞	∞

(b) Transition 1 Criticality

Criticality	Level
Difficulty	low (1.0)
Severity	severe
Flight Phase	transition

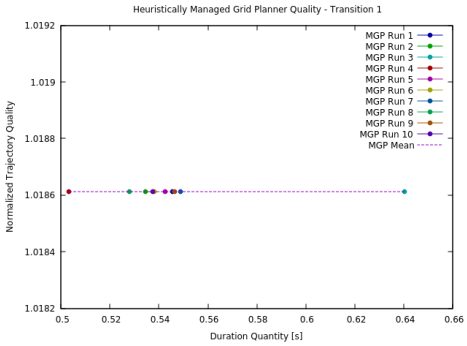
Table 11.13: Transition 1 Heuristic Tunings

(a) Transition 1 MGP Heuristic Tuning

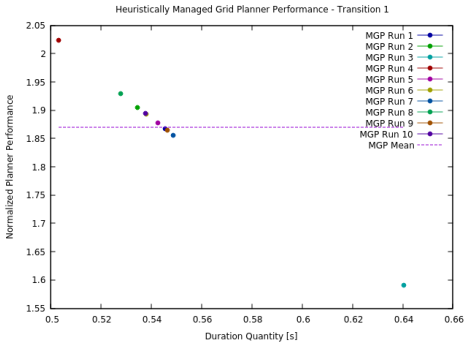
Property	Value
`costPolicy`	`AVERAGE`
`riskPolicy`	`SAFETY`
`minimumQuality`	71.0
`maximumQuality`	35.5
`qualityImprovement`	7.1
`significantChange`	0.5

(b) Transition 1 MTP Heuristic Tuning

Property	Value
`costPolicy`	`AVERAGE`
`riskPolicy`	`SAFETY`
`maximumIterations`	1000
`bias`	5
`epsilon`	28.0579…
`neighborLimit`	1
`goalThreshold`	2.5
`extension`	`FEASIBLE`
`strategy`	`CONNECT`
`sampling`	`ELLIPSOIDAL`
`minimumQuality`	0.0
`maximumQuality`	0.5
`qualityImprovement`	0.1
`significantChange`	0.5



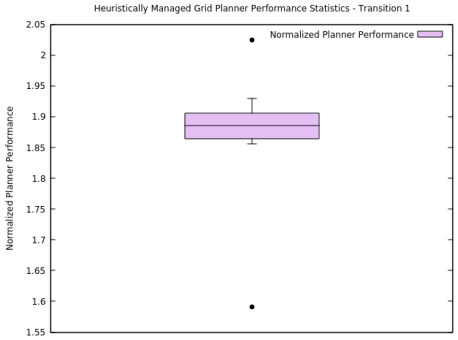
(a) H-AM MGP Quality



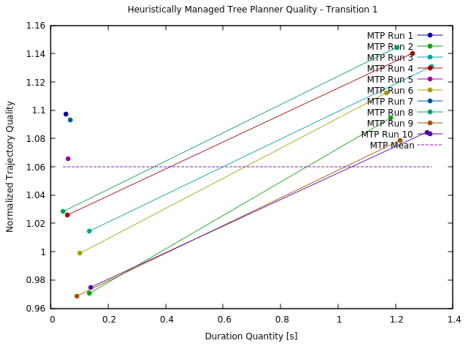
(b) H-AM MGP Performance



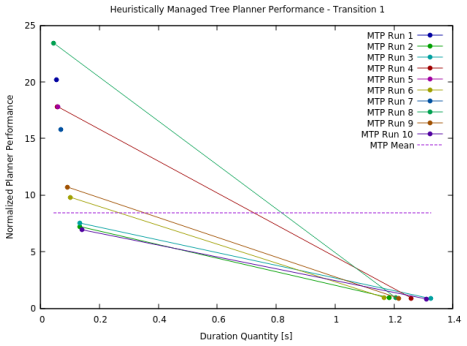
(c) H-AM MGP Quality Statistics



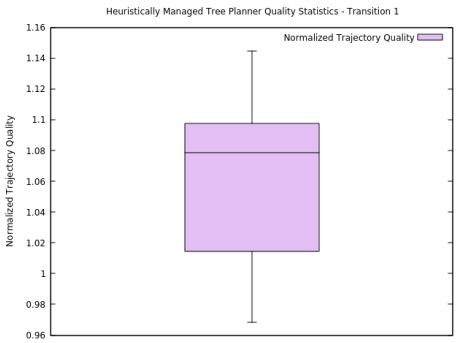
(d) H-AM MGP Performance Statistics



(e) H-AM MTP Quality



(f) H-AM MTP Performance



(g) H-AM MTP Quality Statistics



(h) H-AM MTP Performance Statistics

Figure 11.15: H-AM Quality and Performance: Transition 1

11.4.2 Transition Scenario 2

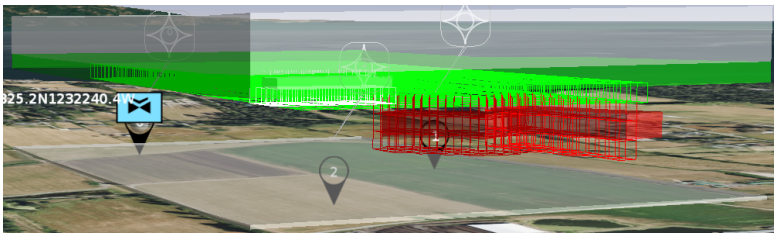
Transition Scenario 2 consists of an Environment featuring one large lower-cost and three smaller RiskPolicy-exceeding SWIM Obstacles, two of which penetrate the POI bounding box. The Scenario consists of one intermediate POI at an altitude of 70 meters ASL that has to be visited when traveling from the start to the goal POI, each at an altitude of 50 meters ASL, and is shown in Figures 11.16 from different directions for the PlanningGrid and PlanningContinuum, respectively.

With several smaller RiskPolicy-exceeding SWIM Obstacles of cost 50 (light red), 60 (dark red), and 80 (white), respectively, in addition to the large lower-cost Obstacle of cost 20 (light green), the Difficulty of this Scenario increases to *substantial* (0.5) while its Severity remains *severe*, resulting in the *transition* FlightPhase association shown in Tables 11.14. Since only the two red SWIM Obstacles at lower altitudes contaminate the POI bounding box within the Features horizon, its minimum and maximum costs differ from the enclosing Environment. However, both accumulated costs exceed the operational RiskPolicy, and a straight second leg is not possible without exceeding it.

The HeuristicAutonomicManager applies the HeuristicPlannerTunings shown in Tables 11.15. The Obstacle-contaminated POI bounding box leads to a tuning approach that considers more Features of the planning Environment, as explained in Section 9.3. In particular, the heuristic inflation factor of the ManagedGridPlanner is adjusted to reflect the RiskPolicy-exceeding SWIM Obstacle costs within the contaminated POI bounding box and deflate this factor incrementally to achieve a 50% tighter sub-optimality bound. Similarly, the ManagedTreePlanner is tuned to be non-deterministic using a maximum of 1000 iterations to compute a feasible Trajectory while proceeding from a 100% to a 50% distance bias (50% cost bias). Its goal bias is tuned to a moderate 5%. Due to the increased Difficulty, the sampling neighorhood is increased to 3 while using an EXTEND growth Strategy.

Given these two HeuristicPlannerTunings, we can confirm the deterministic normalized TrajectoryQuality of the ManagedGridPlanner as well as the non-deterministic normalized TrajectoryQuality of the ManagedTreePlanner in Figures 11.17. Having one intermediate POI, consequently, leads to consecutive computations of one partial and one complete Trajectory with their respective TrajectoryQuality. Thus, at least 2 TrajectoryQuality points are shown in the corresponding diagrams with potentially additional ones representing the incremental improvement of a single

part. The slightly higher normalized TrajectoryQuality of the ManagedTreePlanner is again the result of not being constrained by a discretized planning Environment, such as the PlanningGrid. The DurationQuantity to compute the full Trajectory appears to be very good in both cases due to the high heuristic inflation and distance bias, respectively, and has to be successful without the existence of other non-RiskPolicy-exceeding SWIM Obstacles within the POI bounding box.



(a) Transition 2 Grid



(b) Transition 2 Continuum

Figure 11.16: Transition 2 Scenario

Table 11.14: Transition 2 Feature Categories

(a) Transition 2 Severity

Cost Assessment	ASS	POI	ENV
Maximum	0.0	60.0	80.0
Minimum	0.0	50.0	20.0
Risk Policy	0.0	∞	∞

(b) Transition 2 Criticality

Criticality	Level
Difficulty	substantial (0.5)
Severity	severe
Flight Phase	transition

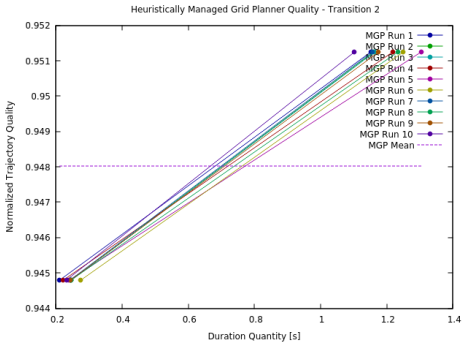
Table 11.15: Transition 2 Heuristic Tunings

(a) Transition 2 MGP Heuristic Tuning

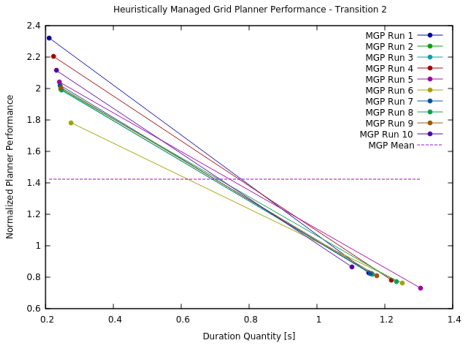
Property	Value
`costPolicy`	`AVERAGE`
`riskPolicy`	`SAFETY`
`minimumQuality`	56.0
`maximumQuality`	28.0
`qualityImprovement`	5.6
`significantChange`	0.5

(b) Transition 2 MTP Heuristic Tuning

Property	Value
`costPolicy`	`AVERAGE`
`riskPolicy`	`SAFETY`
`maximumIterations`	1000
`bias`	5
`epsilon`	16.8566…
`neighborLimit`	3
`goalThreshold`	2.5
`extension`	`FEASIBLE`
`strategy`	`EXTEND`
`sampling`	`ELLIPSOIDAL`
`minimumQuality`	0.0
`maximumQuality`	0.5
`qualityImprovement`	0.1
`significantChange`	0.5



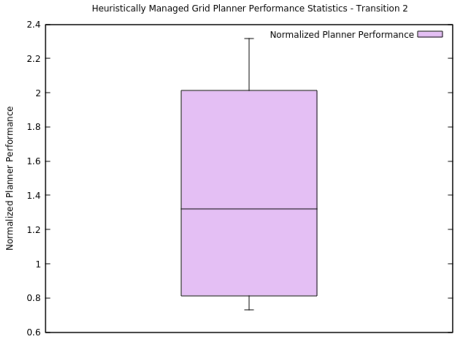
(a) H-AM MGP Quality



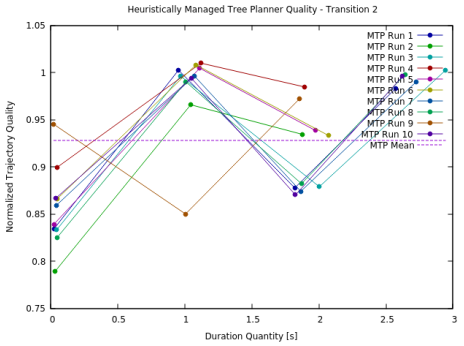
(b) H-AM MGP Performance



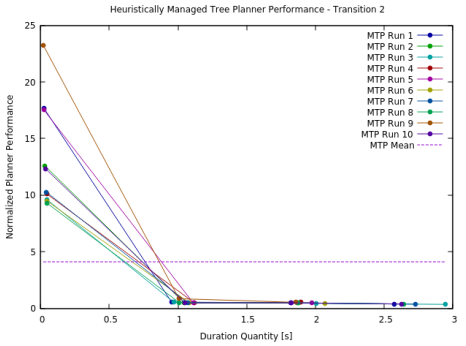
(c) H-AM MGP Quality Statistics



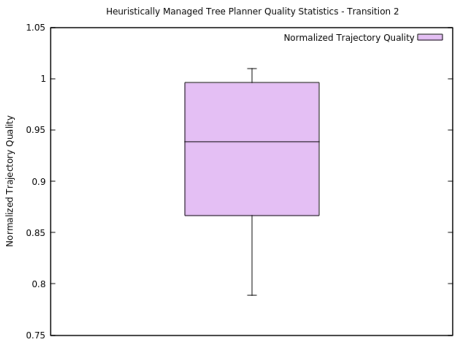
(d) H-AM MGP Performance Statistics



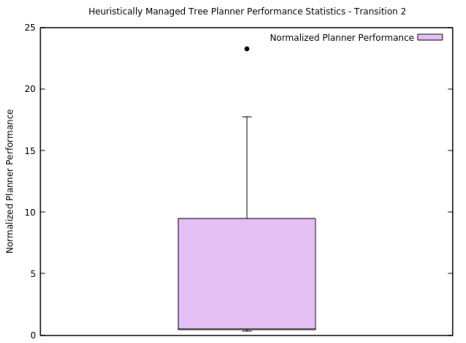
(e) H-AM MTP Quality



(f) H-AM MTP Performance



(g) H-AM MTP Quality Statistics



(h) H-AM MTP Performance Statistics

Figure 11.17: H-AM Quality and Performance: Transition 2

11.4.3 Transition Scenario 3

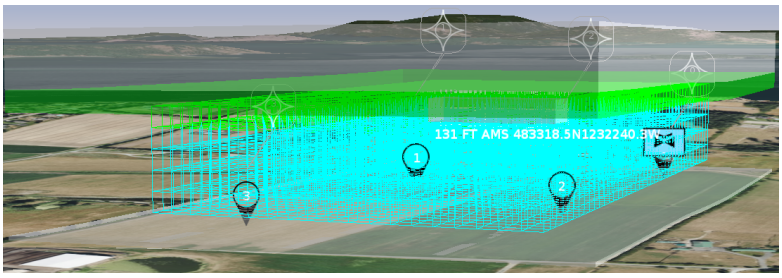
Transition Scenario 3 consists of an Environment featuring two large non-RiskPolicy-exceeding SWIM Obstacles, one of which penetrates both the Aircraft safety sphere and POI bounding box as it, together with the first Obstacle, occupies the entire Environment and cannot be avoided. The Scenario, furthermore, contains one small RiskPolicy-exceeding SWIM obstacle that penetrates the POI bounding box. It comprises two intermediate POIs, each at an altitude of 85 meters ASL, that have to be visited in sequence when traveling from the start to the goal POI, each at an altitude of 40 meters ASL, and is shown in Figures 11.18 from different directions for the PlanningGrid and PlanningContinuum, respectively.

With two large SWIM Obstacles of cost 20 (light green) and 5 (light cyan) within the Environment in addition to the single small RiskPolicy-exceeding SWIM Obstacle of cost 80 (white) within the POI bounding box, the Difficulty of this Scenario is still considered to be *substantial* (0.5) while its Severity is *severe*, resulting in the *transition* FlightPhase association shown in Tables 11.16. Due to the expensive white Obstacle, a straight second leg is not feasible without exceeding the operational RiskPolicy.

The HeuristicAutonomicManager applies the HeuristicPlannerTunings shown in Tables 11.17. The Obstacle-contaminated POI bounding box leads to a tuning approach that considers more Features of the planning Environment, as explained in Section 9.3. Since the Difficulty and Severity assessments are the same as for Transition Scenario 2 of Section 11.4.2, the same statements apply with respect to tuning the ManagedPlanners. Similar to Cruise Scenario 3 of Section 11.3.3, the large unavoidable SWIM Obstacles would cause the deflated heuristics of both ManagedPlanners to under-estimate the expected cost significantly and, therefore, has to remain partially inflated to obtain a solution within a reasonable DurationQuantity. The Difficulty of the Scenario would have to increase more to justify a higher deflation.

Given these two HeuristicPlannerTunings, we can confirm the deterministic normalized TrajectoryQuality of the ManagedGridPlanner as well as the non-deterministic normalized TrajectoryQuality of the ManagedTreePlanner in Figures 11.19. Having two intermediate POIs, consequently, leads to consecutive computations of two partial and one complete Trajectory with their respective TrajectoryQuality. Hence, at least 3 TrajectoryQuality points are shown in the corresponding diagrams. The slightly varying normalized TrajectoryQuality for each part of the computed Trajectory is the result of PlanningGrid constraints, al-

titude changes, Obstacle avoidance, and non-deterministic sampling as applicable. The under-estimating heuristic together with the increased maximum iterations for the *transition* FlightPhase cause a significant increase for the DurationQuantity of the ManagedTreePlanner, which is even more pronounced for more difficult FlightPhases. The more difficult a FlightPhase, the more the heuristic (distance bias) of the ManagedPlanners are deflated by the HeuristicPlannerTuning and, in case of the ManagedTreePlanner, the more iterations are attempted to find a solution. Hence, a Scenario with higher Difficulty (many smaller scattered Obstacles) in the presence of large unavoidable RiskPolicy-exhausting Obstacles represents a challenging situations for all ManagedPlanners, as a heuristic deflation and more attempts are required.



(a) Transition 3 Grid



(b) Transition 3 Continuum

Figure 11.18: Transition 3 Scenario

Table 11.16: Transition 3 Feature Categories

(a) Transition 3 Severity

Cost Assessment	ASS	POI	ENV
Maximum	5.0	80.0	80.0
Minimum	5.0	5.0	5.0
Risk Policy	5.0	∞	∞

(b) Transition 3 Criticality

Criticality	Level
Difficulty	substantial (0.5)
Severity	severe
Flight Phase	transition

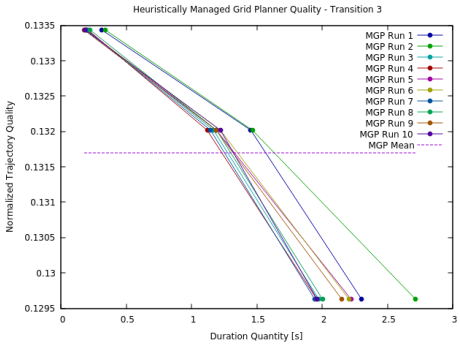
Table 11.17: Transition 3 Heuristic Tunings

(a) Transition 3 MGP Heuristic Tuning

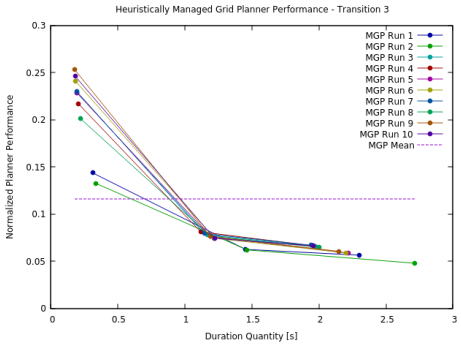
Property	Value
`costPolicy`	`AVERAGE`
`riskPolicy`	`SAFETY`
`minimumQuality`	43.5
`maximumQuality`	21.75
`qualityImprovement`	4.35
`significantChange`	0.5

(b) Transition 3 MTP Heuristic Tuning

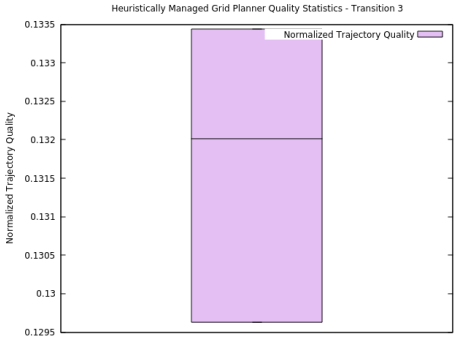
Property	Value
`costPolicy`	`AVERAGE`
`riskPolicy`	`SAFETY`
`maximumIterations`	1000
`bias`	5
`epsilon`	13.5918…
`neighborLimit`	3
`goalThreshold`	2.5
`extension`	`FEASIBLE`
`strategy`	`EXTEND`
`sampling`	`ELLIPSOIDAL`
`minimumQuality`	0.0
`maximumQuality`	0.5
`qualityImprovement`	0.1
`significantChange`	0.5



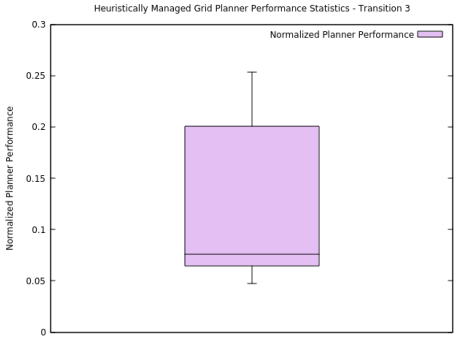
(a) H-AM MGP Quality



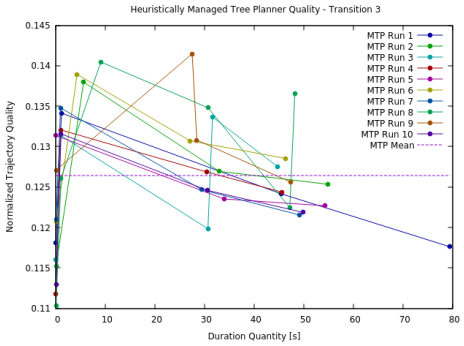
(b) H-AM MGP Performance



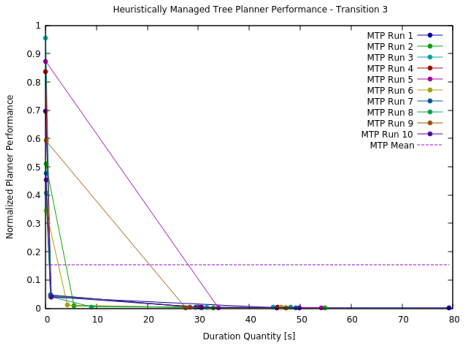
(c) H-AM MGP Quality Statistics



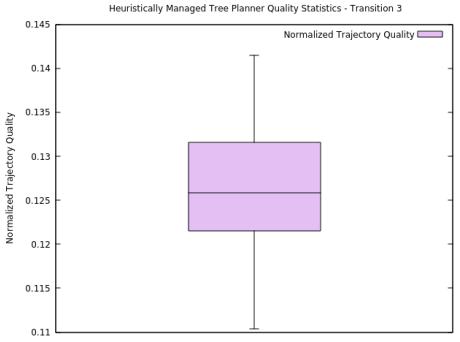
(d) H-AM MGP Performance Statistics



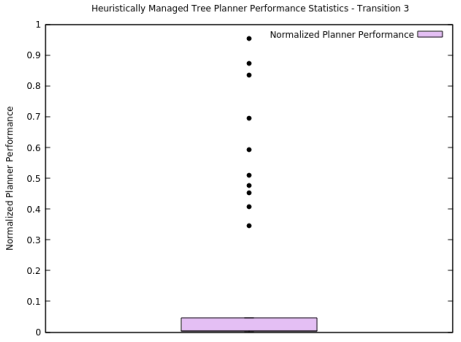
(e) H-AM MTP Quality



(f) H-AM MTP Performance



(g) H-AM MTP Quality Statistics



(h) H-AM MTP Performance Statistics

Figure 11.19: H-AM Quality and Performance: Transition 3

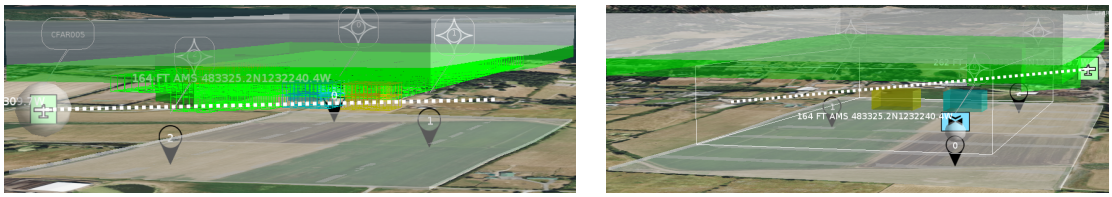
11.4.4 Transition Scenario 4

Transition Scenario 4 consists of an Environment featuring one large as well as several smaller non-RiskPolicy-exceeding SWIM Obstacles, three of which penetrate the POI bounding box, including one that represents another moving aircraft. The Scenario consists of one intermediate POI at an altitude of 60 meters ASL that has to be visited when traveling from the start to the goal POI, each at an altitude of 50 meters ASL, and is shown in Figures 11.20 from different directions for the PlanningGrid and PlanningContinuum, respectively.

With one large and four smaller SWIM Obstacles of cost 20 (light green), 10 (cyan), 20 (light green), 40 (yellow), and 100 (white aircraft), respectively, within the Environment, the Difficulty of this Scenario remains *substantial* (0.4963...) while its Severity is *severe*, resulting in the *transition* FlightPhase associations shown in Tables 11.18. As the cyan, yellow and white SWIM Obstacles contaminate the POI bounding box within the Environment, the RiskPolicy is exceeded in both cases.

The HeuristicAutonomicManager applies the HeuristicPlannerTunings shown in Tables 11.19. The Obstacle-contaminated POI bounding box leads to a tuning approach that considers more Features of the planning Environment, as explained in Section 9.3. Since the Difficulty and Severity assessments are the same as for Transition Scenarios 2 and 3 of Sections 11.4.2 and 11.4.3, respectively, the same statements apply with respect to tuning the ManagedPlanners.

Given these two HeuristicPlannerTunings, we can confirm the deterministic normalized TrajectoryQuality of the ManagedGridPlanner as well as the non-deterministic normalized TrajectoryQuality of the ManagedTreePlanner in Figures 11.21. Having one intermediate POI, consequently, leads to consecutive computations of one partial and one complete Trajectory with their respective TrajectoryQuality. Thus, at least 2 TrajectoryQuality points are shown in the corresponding diagrams with potentially additional ones representing the incremental improvement of a single part. Since the white moving aircraft affects the computed Trajectory within the discrete PlanningGrid more than within the PlanningContinuum, the resulting normalized TrajectoryQuality of the ManagedGridPlanner, particularly for the first leg, is significantly lower.



(a) Transition 4 Grid **(b)** Transition 4 Continuum

Figure 11.20: Transition 4 Scenario

Table 11.18: Transition 4 Feature Categories

(a) Transition 4 Severity

Cost Assessment	ASS	POI	ENV
Maximum	0.0	100.0	100.0
Minimum	0.0	10.0	10.0
Risk Policy	0.0	∞	∞

(b) Transition 4 Criticality

Criticality	Level
Difficulty	substantial (0.4963…)
Severity	severe
Flight Phase	transition

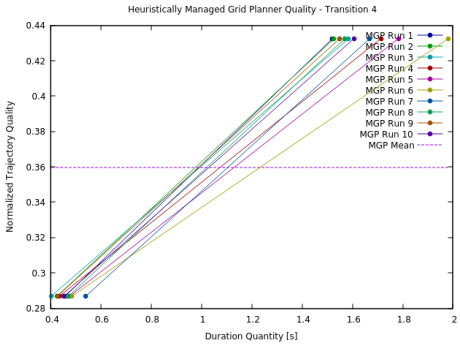
Table 11.19: Transition 4 Heuristic Tunings

(a) Transition 4 MGP Heuristic Tuning

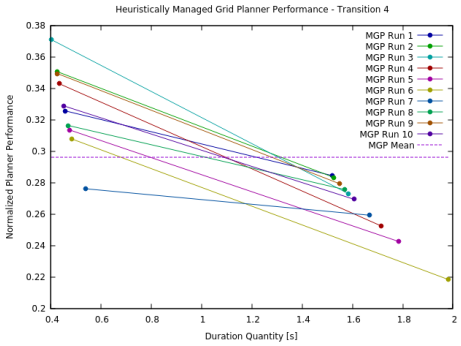
Property	Value
`costPolicy`	`AVERAGE`
`riskPolicy`	`SAFETY`
`minimumQuality`	51.0
`maximumQuality`	25.5
`qualityImprovement`	5.1
`significantChange`	0.5

(b) Transition 4 MTP Heuristic Tuning

Property	Value
`costPolicy`	`AVERAGE`
`riskPolicy`	`SAFETY`
`maximumIterations`	1000
`bias`	5
`epsilon`	16.8343…
`neighborLimit`	3
`goalThreshold`	2.5
`extension`	`FEASIBLE`
`strategy`	`EXTEND`
`sampling`	`ELLIPSOIDAL`
`minimumQuality`	0.0
`maximumQuality`	0.5
`qualityImprovement`	0.1
`significantChange`	0.5



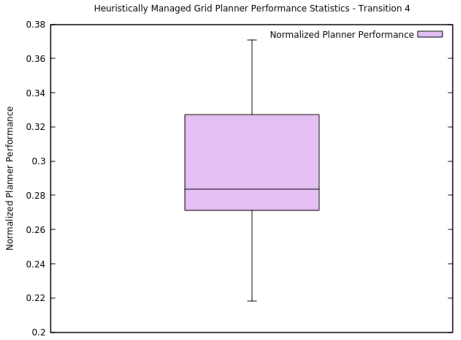
(a) H-AM MGP Quality



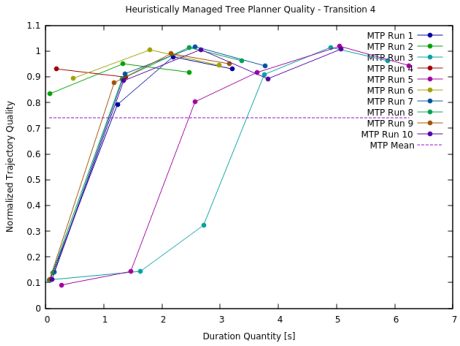
(b) H-AM MGP Performance



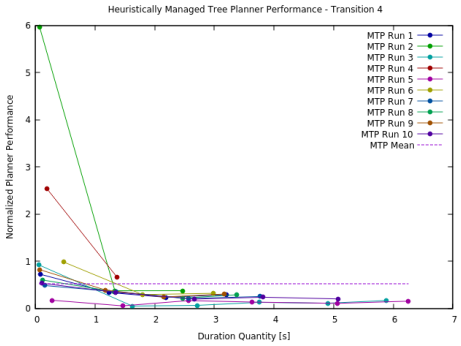
(c) H-AM MGP Quality Statistics



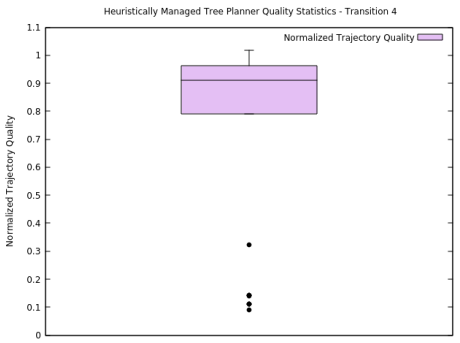
(d) H-AM MGP Performance Statistics



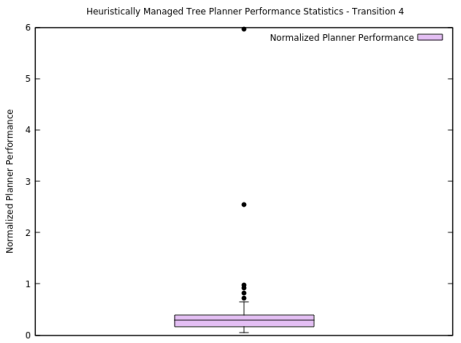
(e) H-AM MTP Quality



(f) H-AM MTP Performance



(g) H-AM MTP Quality Statistics



(h) H-AM MTP Performance Statistics

Figure 11.21: H-AM Quality and Performance: Transition 4

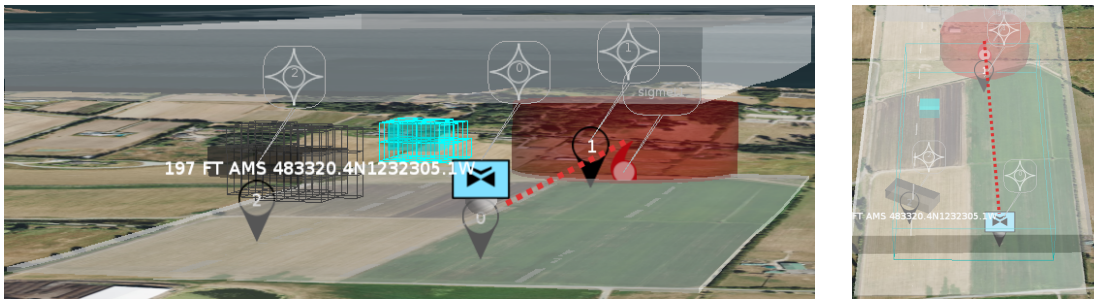
11.4.5 Transition Scenario 5

Transition Scenario 5 consists of an Environment featuring one larger moving RiskPolicy-exceeding as well as two smaller stationary non-RiskPolicy-exceeding SWIM Obstacles, all of which penetrate the POI bounding box. The Scenario comprises one intermediate POI at an altitude of 40 meters ASL that has to be visited when traveling from the start to the goal POI, at altitudes of 60 and 50 meters ASL, respectively, and is shown in Figures 11.22 from different directions for both the PlanningGrid and PlanningContinuum.

With the stationary two smaller and single moving larger SWIM Obstacles of cost 3 (gray), 7 (light cyan), and 60 (dark red), respectively, within the Environment, the Difficulty of this Scenario becomes *moderate* (0.7351...) while its Severity remains *severe*, resulting in the *transition* FlightPhase association shown in Tables 11.20. Since all present SWIM Obstacles contaminate the POI bounding box within the enclosing Environment, both possess the same RiskPolicy-exceeding cost assessment.

The HeuristicAutonomicManager applies the HeuristicPlannerTunings shown in Tables 11.21. The Obstacle-contaminated POI bounding box leads to a tuning approach that considers more Features of the planning Environment, as explained in Section 9.3. In particular, the heuristic inflation factor of the ManagedGridPlanner is adjusted to reflect the relevant SWIM Obstacle costs within the contaminated POI bounding box. Similarly, the ManagedTreePlanner is tuned to be non-deterministic considering a relatively small sampling neighborhood of 2 while using a CONNECT growth Strategy.

Given these two HeuristicPlannerTunings, we can confirm the deterministic normalized TrajectoryQuality of the ManagedGridPlanner as well as the non-deterministic normalized TrajectoryQuality of the ManagedTreePlanner in Figures 11.23. Having one intermediate POI, consequently, leads to consecutive computations of one partial and one complete Trajectory with their respective TrajectoryQuality. Thus, at least 2 TrajectoryQuality points are shown in the corresponding diagrams with potentially additional ones representing the incremental improvement of a single part. Both ManagedPlanners have to avoid the RiskPolicy-exceeding red moving Obstacle and may partially avoid the gray lower-cost Obstacle surrounding the goal POI. Their normalized TrajectoryQuality is similar, with the ManagedGridPlanner being constrained to its PlanningGrid.



(a) Transition 5 Grid **(b)** Transition 5 Continuum

Figure 11.22: Transition 5 Scenario

Table 11.20: Transition 5 Feature Categories

(a) Transition 5 Severity

Cost Assessment	ASS	POI	ENV
Maximum	0.0	60.0	60.0
Minimum	0.0	3.0	3.0
Risk Policy	0.0	∞	∞

(b) Transition 5 Criticality

Criticality	Level
Difficulty	moderate (0.7351…)
Severity	severe
Flight Phase	transition

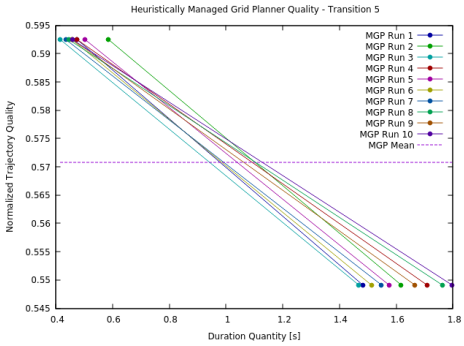
Table 11.21: Transition 5 Heuristic Tunings

(a) Transition 5 MGP Heuristic Tuning

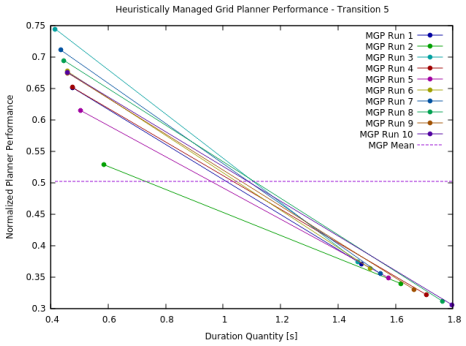
Property	Value
`costPolicy`	`AVERAGE`
`riskPolicy`	`SAFETY`
`minimumQuality`	$24.\overline{3}$
`maximumQuality`	$12.1\overline{6}$
`qualityImprovement`	$2.4\overline{3}$
`significantChange`	0.5

(b) Transition 5 MTP Heuristic Tuning

Property	Value
`costPolicy`	`AVERAGE`
`riskPolicy`	`SAFETY`
`maximumIterations`	1000
`bias`	5
`epsilon`	24.7876…
`neighborLimit`	2
`goalThreshold`	2.5
`extension`	`FEASIBLE`
`strategy`	`CONNECT`
`sampling`	`ELLIPSOIDAL`
`minimumQuality`	0.0
`maximumQuality`	0.5
`qualityImprovement`	0.1
`significantChange`	0.5



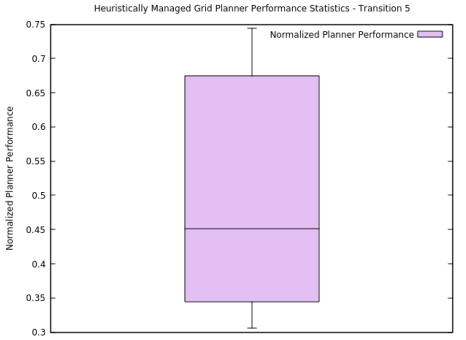
(a) H-AM MGP Quality



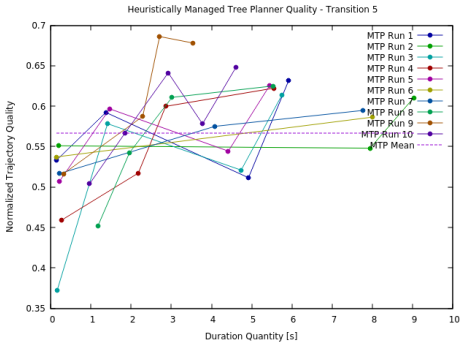
(b) H-AM MGP Performance



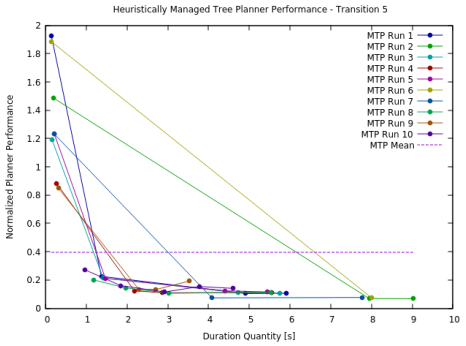
(c) H-AM MGP Quality Statistics



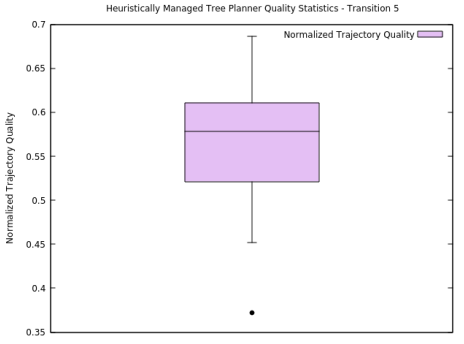
(d) H-AM MGP Performance Statistics



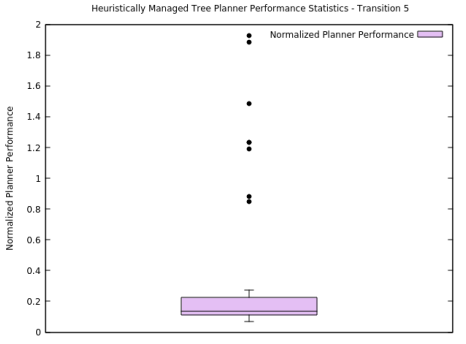
(e) H-AM MTP Quality



(f) H-AM MTP Performance



(g) H-AM MTP Quality Statistics



(h) H-AM MTP Performance Statistics

Figure 11.23: H-AM Quality and Performance: Transition 5

11.5 Terminal

The *terminal* FlightPhase evaluation and training is performed with five associated Scenarios. Recall from Section 9.2 that this FlightPhase covers the higher levels of Difficulty *severe* and *critical* from the extracted Features of a given Scenario while the Severity spans from *low* to *substantial*, only excluding the *critical* and *fatal* assessments. The *terminal* FlightPhase, therefore, requires a more extensive search for a Trajectory of higher TrajectoryQuality in a Features horizon with more scattered SWIM Obstacles of varying costs. However, the Aircraft safety sphere is not yet penetrated with any RiskPolicy-exceeding Obstacles. Generally, the incremental approach of an AnytimePlanner helps in these situations to provide an earlier solution that avoids all RiskPolicy-exceeding Obstacles and improves during the continued elaboration offering less costly detours. Figure 11.24 shows a ManagedTreePlanner for a typical *terminal* FlightPhase with the Aircraft exposed to a SWIM Obstacle-cluttered PlanningContinuum featuring both stationary airspaces (varying colors) as well as moving adverse weather (yellow) and traffic (white helicopter). Its RRT probes (dashed colored) as well as the currently computed and incrementally improving Trajectory (magenta) are also shown.

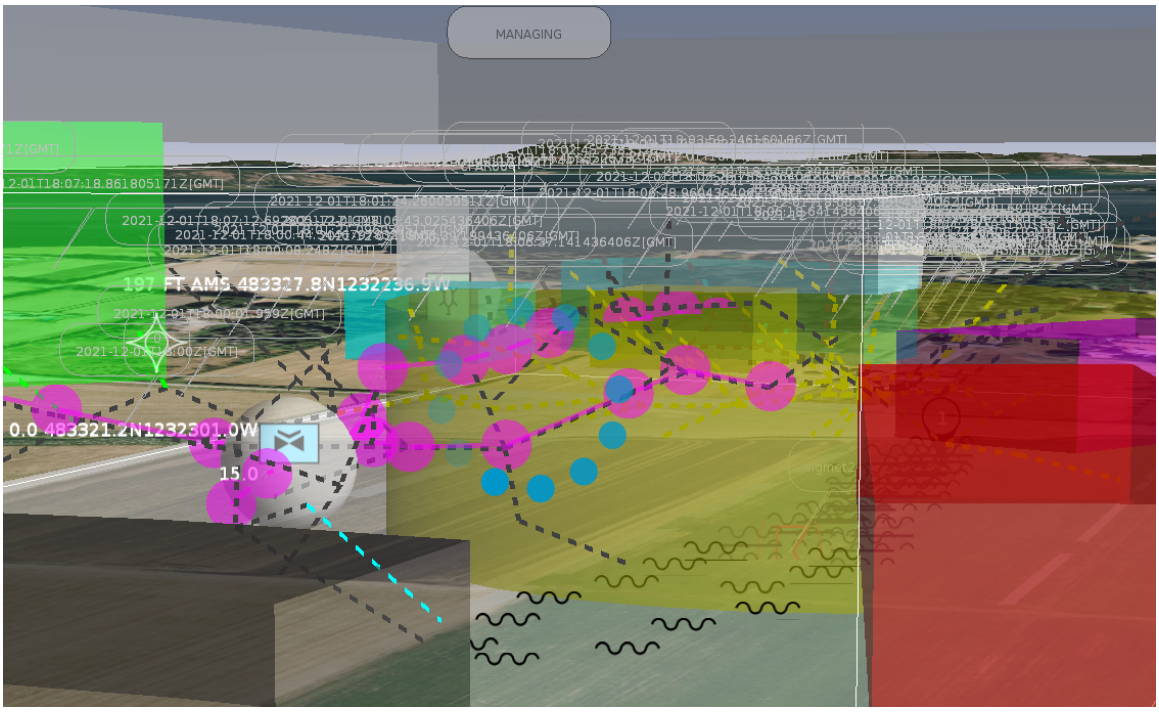


Figure 11.24: Managed Terminal Scenario

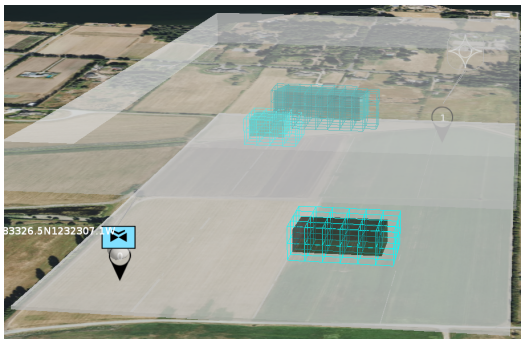
11.5.1 Terminal Scenario 1

Terminal Scenario 1 consists of an Environment featuring three stationary small non-RiskPolicy-exceeding SWIM Obstacles which penetrate the POI bounding box. The Scenario consists of only the start and goal POIs, each at an altitude of 50 meters ASL, and is shown in Figures 11.25 from different directions for the PlanningGrid and PlanningContinuum, respectively.

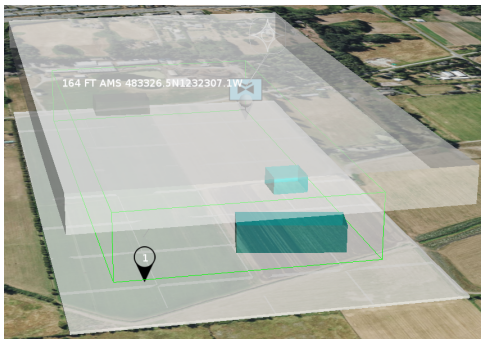
With only the three small stationary SWIM Obstacles of cost 4 (gray with a light cyan grid due to its base cost of 1), 7 (light cyan), and 13 (cyan), respectively, within the POI bounding box and no larger Obstacle to compensate the scatteredness within the Features horizon of this Scenario, its Difficulty is considered to be *severe* ($0.\overline{3}$) while its Severity is *moderate* to *substantial*, resulting in the *terminal* FlightPhase association shown in Tables 11.22. Since the present SWIM Obstacles contaminate the POI bounding box, the same RiskPolicy-assessed cost of 24 results for the entire Environment.

The HeuristicAutonomicManager applies the HeuristicPlannerTunings shown in Tables 11.23. The Obstacle-contaminated POI bounding box leads to a tuning approach that considers more Features of the planning Environment, as explained in Section 9.3. In particular, the heuristic inflation factor of the ManagedGridPlanner is adjusted to reflect the SWIM Obstacle costs within the contaminated POI bounding box and deflate this factor incrementally to achieve a 75% tighter sub-optimality bound. Similarly, the ManagedTreePlanner is tuned to be non-deterministic using a maximum of 1500 iterations to compute a feasible Trajectory while proceeding from a 100% to a 25% distance bias (75% cost bias). Its goal bias is tuned to a lower 3% and the sampling neighborhood to a moderate 4 due to the *severe* Difficulty of this Scenario. Notice, furthermore, that the Sampling approach is now UNIFORM using an EXTEND growth Strategy to favor exploration within the PlanningContinuum.

Given these two HeuristicPlannerTunings, we can confirm the deterministic normalized TrajectoryQuality of the ManagedGridPlanner as well as the non-deterministic normalized TrajectoryQuality of the ManagedTreePlanner in Figures 11.26. Having no intermediate POIs immediately leads to the computation of the complete Trajectory, which in the case of the ManagedTreePlanner takes one or multiple non-deterministic steps with increasing normalized TrajectoryQuality. The occasionally slightly higher normalized TrajectoryQuality of the ManagedTreePlanner is the result of the non-constraining PlanningContinuum.



(a) Terminal 1 Grid



(b) Terminal 1 Continuum

Figure 11.25: Terminal 1 Scenario

Table 11.22: Terminal 1 Feature Categories

(a) Terminal 1 Severity

Cost Assessment	ASS	POI	ENV
Maximum	0.0	13.0	13.0
Minimum	0.0	4.0	4.0
Risk Policy	0.0	24.0	24.0

(b) Terminal 1 Criticality

Criticality	Level
Difficulty	severe (0.3333…)
Severity	moderate, substantial
Flight Phase	terminal

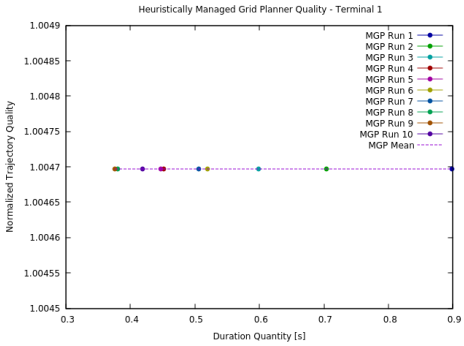
Table 11.23: Terminal 1 Heuristic Tunings

(a) Terminal 1 MGP Heuristic Tuning

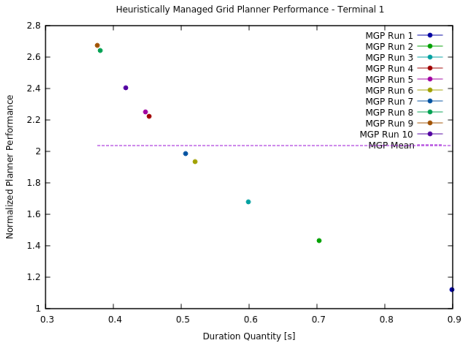
Property	Value
`costPolicy`	`AVERAGE`
`riskPolicy`	`SAFETY`
`minimumQuality`	9.0
`maximumQuality`	2.25
`qualityImprovement`	1.35
`significantChange`	0.5

(b) Terminal 1 MTP Heuristic Tuning

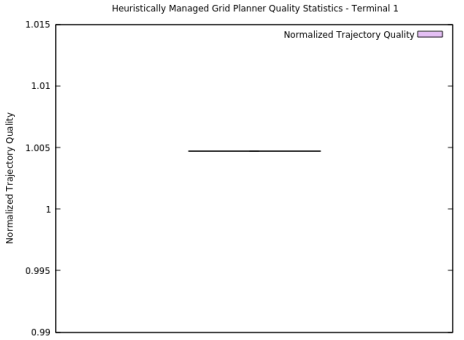
Property	Value
`costPolicy`	`AVERAGE`
`riskPolicy`	`SAFETY`
`maximumIterations`	1500
`bias`	3
`epsilon`	30.3412…
`neighborLimit`	4
`goalThreshold`	2.5
`extension`	`FEASIBLE`
`strategy`	`EXTEND`
`sampling`	`UNIFORM`
`minimumQuality`	0.0
`maximumQuality`	0.75
`qualityImprovement`	0.15
`significantChange`	0.5



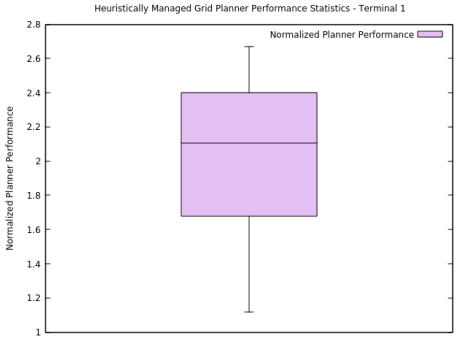
(a) H-AM MGP Quality



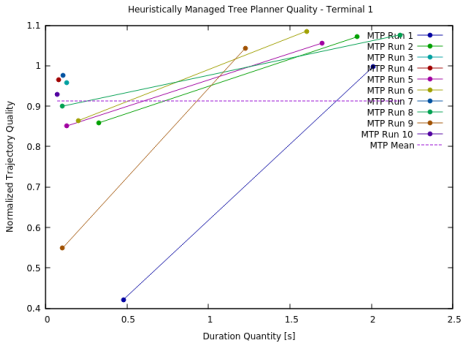
(b) H-AM MGP Performance



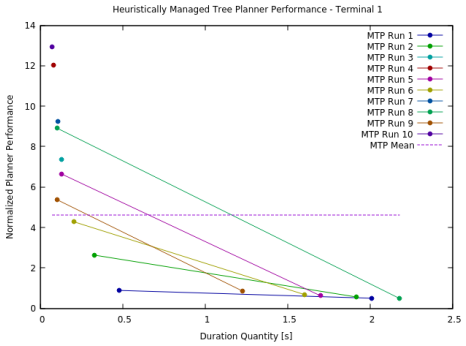
(c) H-AM MGP Quality Statistics



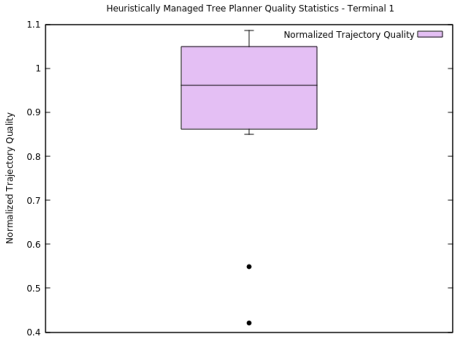
(d) H-AM MGP Performance Statistics



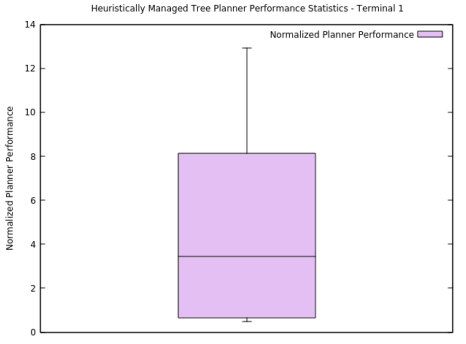
(e) H-AM MTP Quality



(f) H-AM MTP Performance



(g) H-AM MTP Quality Statistics



(h) H-AM MTP Performance Statistics

Figure 11.26: H-AM Quality and Performance: Terminal 1

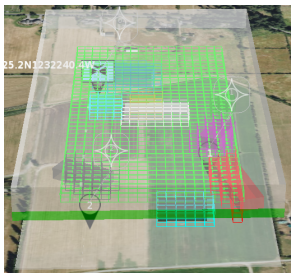
11.5.2 Terminal Scenario 2

Terminal Scenario 2 consists of an Environment featuring many stationary smaller SWIM Obstacles which penetrate the POI bounding box, including one non-RiskPolicy-exceeding Obstacle within the Aircraft safety sphere. The Scenario consists of an intermediate POI at an altitude of 60 meters ASL that has to be visited when traveling from the start to the goal POI, each at an altitude of 50 meters ASL, and is shown in Figures 11.27 from different directions for the PlanningGrid and PlanningContinuum, respectively.

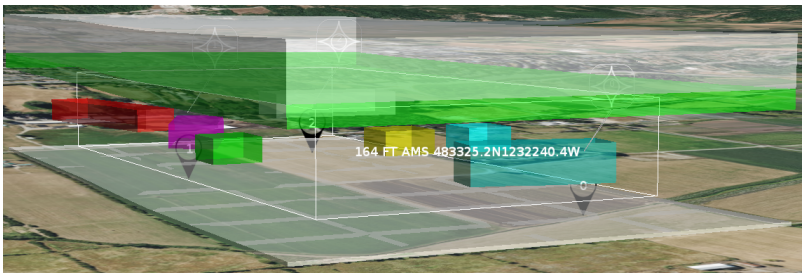
With many small scattered stationary SWIM Obstacles of varying costs across the spectrum, including one of cost 13 (cyan) penetrating the Aircraft safety sphere within the Features horizon of this Scenario, its Difficulty reaches the highest level *critical* ($0.1\overline{6}$) while its Severity is considered *severe*, resulting in the *terminal* FlightPhase association shown in Tables 11.24. Since distinct SWIM Obstacles contaminate the Aircraft safety sphere, POI bounding box, and enclosing Environment, respectively, different RiskPolicy cost assessments result.

The HeuristicAutonomicManager applies the HeuristicPlannerTunings shown in Tables 11.25. The Obstacle-contaminated POI bounding box leads to a tuning approach that considers more Features of the planning Environment, as explained in Section 9.3. As for Terminal Scenario 1 of Section 11.5.1, the heuristic inflation factor of the ManagedGridPlanner is adjusted to reflect the SWIM Obstacle costs within the contaminated POI bounding box and deflate this factor incrementally to achieve a 75% tighter sub-optimality bound. Similarly, the ManagedTreePlanner is tuned to be non-deterministic using a maximum of 1500 iterations to compute a feasible Trajectory while proceeding from a 100% to a 25% distance bias (75% cost bias). Its goal bias remains at a lower 3% and the sampling neighborhood is now increased to 5 due to the *critical* Difficulty of this Scenario. As before, the Sampling approach is UNIFORM using an EXTEND growth Strategy to favor exploration within the PlanningContinuum.

Given these two HeuristicPlannerTunings, we can confirm the deterministic normalized TrajectoryQuality of the ManagedGridPlanner as well as the non-deterministic normalized TrajectoryQuality of the ManagedTreePlanner in Figures 11.28. Having one intermediate POI leads to the computation of one partial followed by the complete Trajectory with possibly additional solutions for the incrementally increasing normalized TrajectoryQuality within each part.



(a) Terminal 2 Grid



(b) Terminal 2 Continuum

Figure 11.27: Terminal 2 Scenario

Table 11.24: Terminal 2 Feature Categories

(a) Terminal 2 Severity

Cost Assessment	ASS	POI	ENV
Maximum	13.0	70.0	80.0
Minimum	13.0	3.0	3.0
Risk Policy	13.0	∞	∞

(b) Terminal 2 Criticality

Criticality	Level
Difficulty	critical (0.1666…)
Severity	severe
Flight Phase	terminal

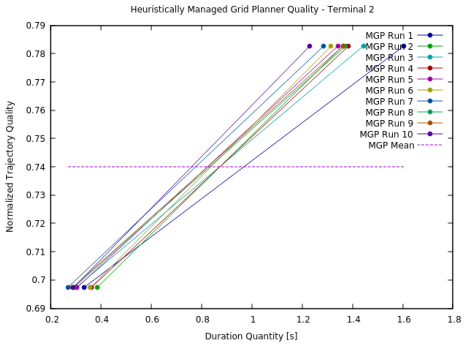
Table 11.25: Terminal 2 Heuristic Tunings

(a) Terminal 2 MGP Heuristic Tuning

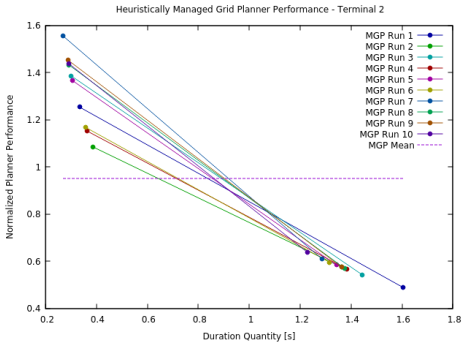
Property	Value
`costPolicy`	`AVERAGE`
`riskPolicy`	`SAFETY`
`minimumQuality`	$23.8\overline{3}$
`maximumQuality`	$5.958\overline{3}$
`qualityImprovement`	3.575
`significantChange`	0.5

(b) Terminal 2 MTP Heuristic Tuning

Property	Value
`costPolicy`	`AVERAGE`
`riskPolicy`	`SAFETY`
`maximumIterations`	1500
`bias`	3
`epsilon`	19.6360…
`neighborLimit`	5
`goalThreshold`	2.5
`extension`	`FEASIBLE`
`strategy`	`EXTEND`
`sampling`	`UNIFORM`
`minimumQuality`	0.0
`maximumQuality`	0.75
`qualityImprovement`	0.15
`significantChange`	0.5



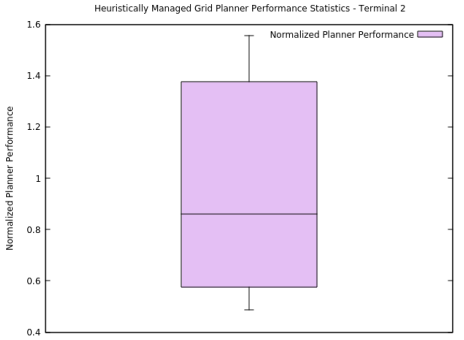
(a) H-AM MGP Quality



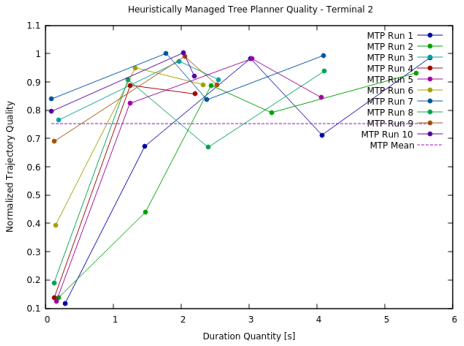
(b) H-AM MGP Performance



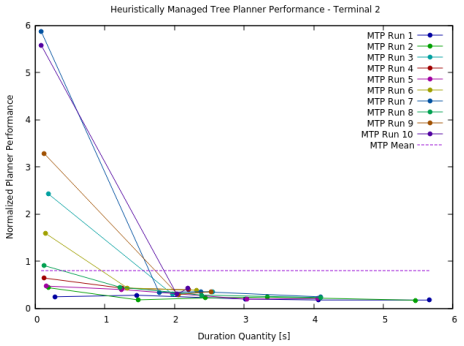
(c) H-AM MGP Quality Statistics



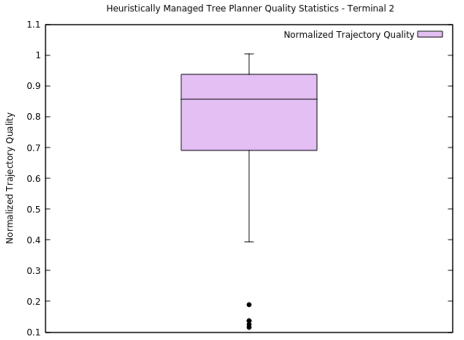
(d) H-AM MGP Performance Statistics



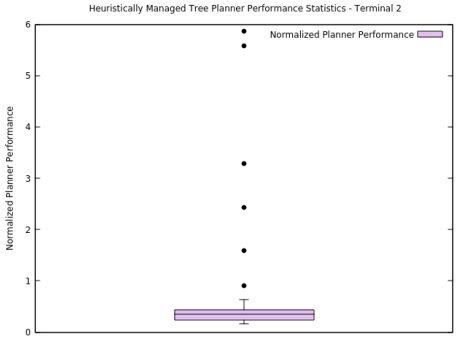
(e) H-AM MTP Quality



(f) H-AM MTP Performance



(g) H-AM MTP Quality Statistics



(h) H-AM MTP Performance Statistics

Figure 11.28: H-AM Quality and Performance: Terminal 2

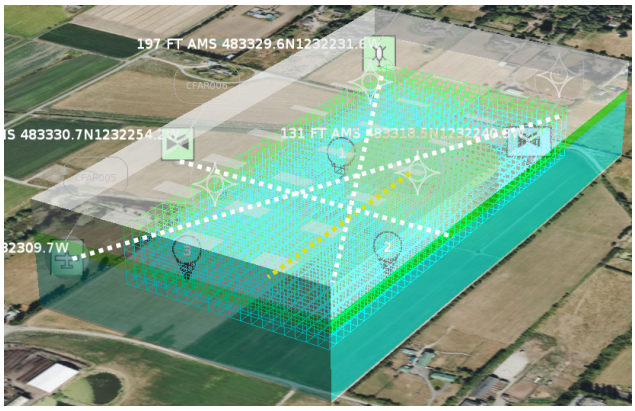
11.5.3 Terminal Scenario 3

Terminal Scenario 3 consists of an Environment featuring large all-enclosing stationary SWIM Obstacles which do not exceed the operational RiskPolicy and cannot be avoided. In addition, several moving SWIM Obstacles representing other aircraft as well as one non-RiskPolicy-exceeding weather system are included. The Scenario consists of two intermediate POIs, at an altitude of 50 meters ASL each, that have to be visited in sequence when traveling from the start to the goal POI, each at an altitude of 40 meters ASL, and is shown with the projected Obstacle movements in Figures 11.29 from different directions for the PlanningGrid and PlanningContinuum, respectively.

With several scattered moving SWIM Obstacles of costs 100 (white aircraft) and 40 (yellow weather system) in addition to the large all-enclosing Obstacles of cost 5 (light cyan) and 20 (light green), respectively, within the Features horizon of this Scenario, its Difficulty reaches the highest level *critical* (0.0340...) while its Severity is considered *severe*, resulting in the *terminal* FlightPhase association shown in Tables 11.26. While the Aircraft safety sphere is penetrated by the operationally feasible light cyan Obstacle, the RiskPolicy within the POI bounding box and enclosing Environment is clearly exceeded.

The HeuristicAutonomicManager applies the HeuristicPlannerTunings shown in Tables 11.27. The Obstacle-contaminated POI bounding box leads to a tuning approach that considers more Features of the planning Environment, as explained in Section 9.3. Since Terminal Scenarios 2 and 3 of Sections 11.5.2 and 11.5.3, respectively, feature the same Difficulty and Severity, their HeuristicPlannerTunings are equivalent and the same comments apply.

Given these two HeuristicPlannerTunings, we can confirm the deterministic normalized TrajectoryQuality of the ManagedGridPlanner as well as the non-deterministic normalized TrajectoryQuality of the ManagedTreePlanner in Figures 11.30. Having two intermediate POIs leads to the computation of two partial followed by the complete Trajectory with possibly additional solutions for the incrementally increasing normalized TrajectoryQuality within each part. As mentioned with similar Scenarios of all-enclosing non-RiskPolicy-exceeding SWIM Obstacles, the underestimating heuristics, if not properly inflated, as well as the higher maximum iterations may increase the DurationQuantity substantially. On the other hand, not deflating or increasing the cost bias sufficiently may lead to more costly, but still feasible and possibly shorter, solutions.



(a) Terminal 3 Grid



(b) Terminal 3 Continuum

Figure 11.29: Terminal 3 Scenario

Table 11.26: Terminal 3 Feature Categories

(a) Terminal 3 Severity

Cost Assessment	ASS	POI	ENV
Maximum	5.0	100.0	100.0
Minimum	5.0	5.0	5.0
Risk Policy	5.0	∞	∞

(b) Terminal 3 Criticality

Criticality	Level
Difficulty	critical (0.0340…)
Severity	severe
Flight Phase	terminal

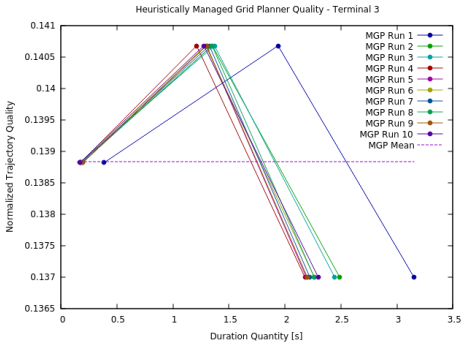
Table 11.27: Terminal 3 Heuristic Tunings

(a) Terminal 3 MGP Heuristic Tuning

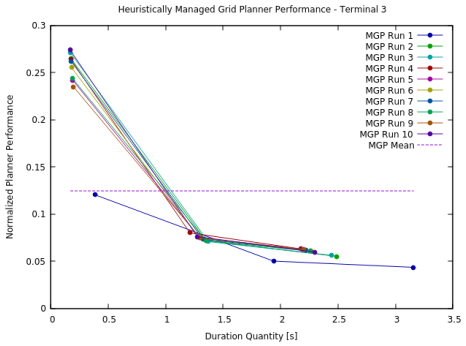
Property	Value
`costPolicy`	`AVERAGE`
`riskPolicy`	`SAFETY`
`minimumQuality`	62.25
`maximumQuality`	15.5625
`qualityImprovement`	9.3375
`significantChange`	0.5

(b) Terminal 3 MTP Heuristic Tuning

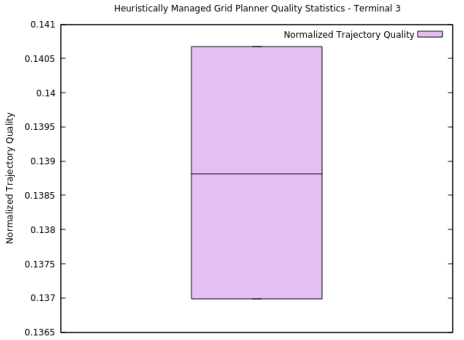
Property	Value
`costPolicy`	`AVERAGE`
`riskPolicy`	`SAFETY`
`maximumIterations`	1500
`bias`	3
`epsilon`	13.5917…
`neighborLimit`	5
`goalThreshold`	2.5
`extension`	`FEASIBLE`
`strategy`	`EXTEND`
`sampling`	`UNIFORM`
`minimumQuality`	0.0
`maximumQuality`	0.75
`qualityImprovement`	0.15
`significantChange`	0.5



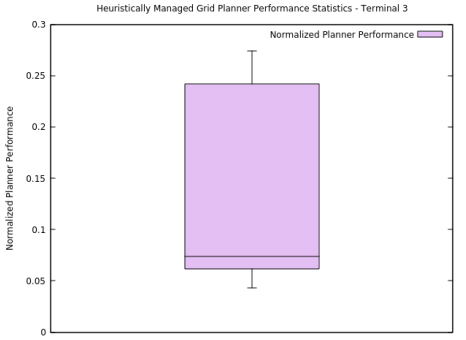
(a) H-AM MGP Quality



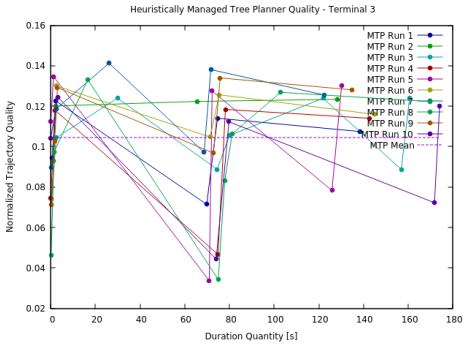
(b) H-AM MGP Performance



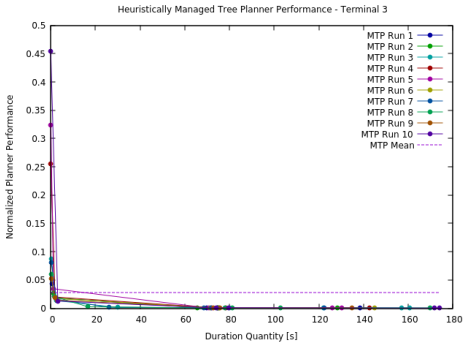
(c) H-AM MGP Quality Statistics



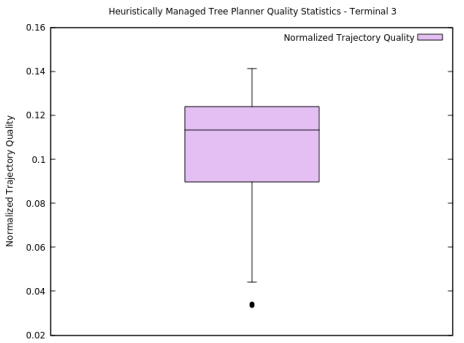
(d) H-AM MGP Performance Statistics



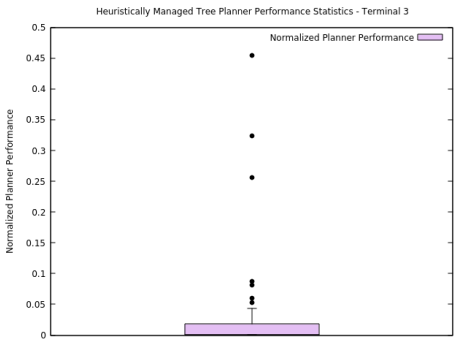
(e) H-AM MTP Quality



(f) H-AM MTP Performance



(g) H-AM MTP Quality Statistics



(h) H-AM MTP Performance Statistics

Figure 11.30: H-AM Quality and Performance: Terminal 3

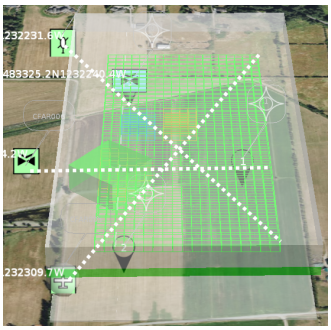
11.5.4 Terminal Scenario 4

Terminal Scenario 4 consists of an Environment featuring one large high-level and three smaller stationary non-RiskPolicy-exceeding SWIM Obstacles that can be avoided. Additionally, several moving SWIM Obstacles representing other aircraft are included. The Scenario consists of one intermediate POI, at an altitude of 60 meters ASL, that has to be visited when traveling from the start to the goal POI, each at an altitude of 50 meters ASL, and is shown with the projected Obstacle movements in Figures 11.31 from different directions for the PlanningGrid and PlanningContinuum, respectively.

With several scattered stationary and moving SWIM Obstacles of costs 10 (cyan), 20 (light green), 40 (yellow), and 100 (white aircraft), respectively, within the Features horizon of this Scenario, its Difficulty becomes *severe* (0.2320…) as well as its Severity, resulting in the *terminal* FlightPhase association shown in Tables 11.28. While the Aircraft safety sphere is not penetrated by any Obstacles, the RiskPolicy within the POI bounding box and enclosing Environment is clearly exceeded.

The HeuristicAutonomicManager applies the HeuristicPlannerTunings shown in Tables 11.29. The Obstacle-contaminated POI bounding box leads to a tuning approach that considers more Features of the planning Environment, as explained in Section 9.3. Featuring only a reduced Difficulty compared to Terminal Scenario 3 of Section 11.5.3, their HeuristicPlannerTunings are similar except for the reduced sampling neighborhood of 4.

Given these two HeuristicPlannerTunings, we can confirm the deterministic normalized TrajectoryQuality of the ManagedGridPlanner as well as the non-deterministic normalized TrajectoryQuality of the ManagedTreePlanner in Figures 11.32. Having one intermediate POI leads to the computation of one partial followed by the complete Trajectory with possibly additional solutions for the incrementally increasing normalized TrajectoryQuality within each part. The achieved DurationQuantity is very good for both ManagedPlanners, and the lower TrajectoryQuality in case of the ManagedGridPlanner is again the result of the constraining PlanningGrid.



(a) Terminal 4 Grid



(b) Terminal 4 Continuum

Figure 11.31: Terminal 4 Scenario

Table 11.28: Terminal 4 Feature Categories

(a) Terminal 4 Severity

Cost Assessment	ASS	POI	ENV
Maximum	0.0	100.0	100.0
Minimum	0.0	10.0	10.0
Risk Policy	0.0	∞	∞

(b) Terminal 4 Criticality

Criticality	Level
Difficulty	severe (0.2320…)
Severity	severe
Flight Phase	terminal

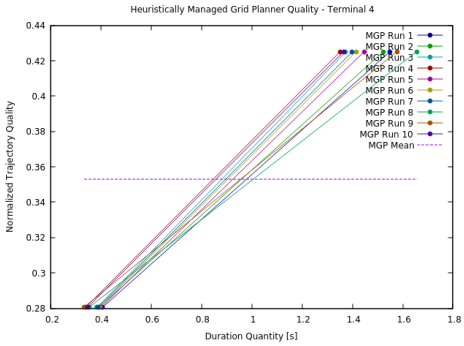
Table 11.29: Terminal 4 Heuristic Tunings

(a) Terminal 4 MGP Heuristic Tuning

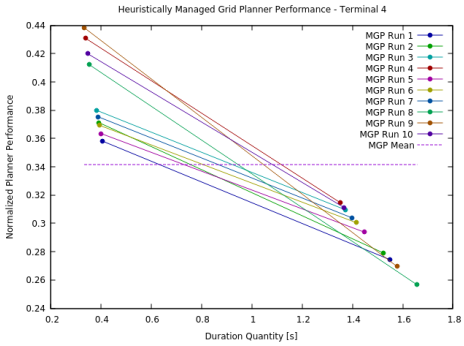
Property	Value
`costPolicy`	`AVERAGE`
`riskPolicy`	`SAFETY`
`minimumQuality`	71.0
`maximumQuality`	17.75
`qualityImprovement`	10.65
`significantChange`	0.5

(b) Terminal 4 MTP Heuristic Tuning

Property	Value
`costPolicy`	`AVERAGE`
`riskPolicy`	`SAFETY`
`maximumIterations`	1500
`bias`	3
`epsilon`	16.8343…
`neighborLimit`	4
`goalThreshold`	2.5
`extension`	`FEASIBLE`
`strategy`	`EXTEND`
`sampling`	`UNIFORM`
`minimumQuality`	0.0
`maximumQuality`	0.75
`qualityImprovement`	0.15
`significantChange`	0.5



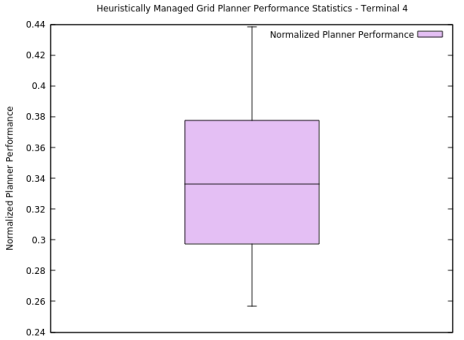
(a) H-AM MGP Quality



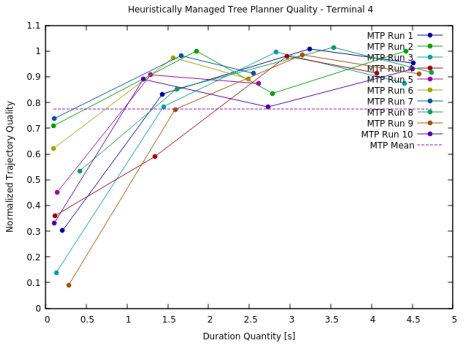
(b) H-AM MGP Performance



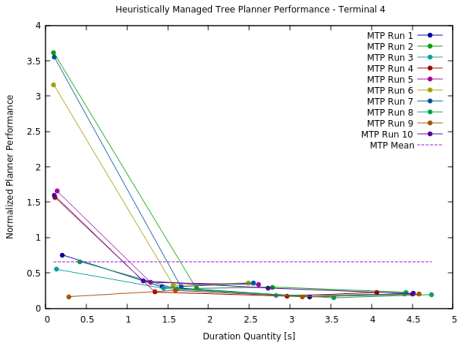
(c) H-AM MGP Quality Statistics



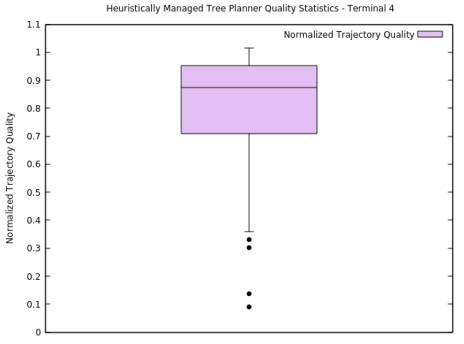
(d) H-AM MGP Performance Statistics



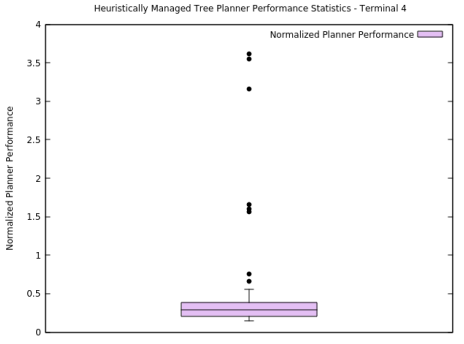
(e) H-AM MTP Quality



(f) H-AM MTP Performance



(g) H-AM MTP Quality Statistics



(h) H-AM MTP Performance Statistics

Figure 11.32: H-AM Quality and Performance: Terminal 4

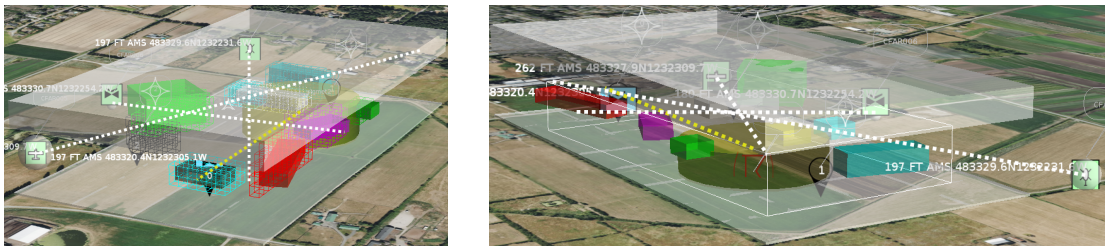
11.5.5 Terminal Scenario 5

Terminal Scenario 5 consists of an Environment featuring many stationary and moving SWIM Obstacles, including other aircraft and adverse weather. The Scenario consists of one intermediate POI, at an altitude of 40 meters ASL, that has to be visited when traveling from the start to the goal POI, at altitudes of 60 and 50 meters ASL, respectively, and is shown with the projected Obstacle movements in Figures 11.33 from different directions for both the PlanningGrid and PlanningContinuum.

With many scattered stationary and moving SWIM Obstacles of varying costs within the Features horizon of this Scenario, its Difficulty reaches the highest level *critical* (0.1166...) and a *severe* Severity, resulting in the *terminal* FlightPhase association shown in Tables 11.30. While the Aircraft safety sphere is only penetrated by a non-RiskPolicy-exceeding SWIM Obstacle of cost 4 (gray with light cyan grid due to a base cost of 1), the RiskPolicy within the POI bounding box and enclosing Environment is clearly exceeded.

The HeuristicAutonomicManager applies the HeuristicPlannerTunings shown in Tables 11.31. The Obstacle-contaminated POI bounding box leads to a tuning approach that considers more Features of the planning Environment, as explained in Section 9.3. With the same Difficulty and Severity assessments as Terminal Scenarios 2 and 3 of Sections 11.5.2 and 11.5.3, respectively, the HeuristicPlannerTuning is equivalent and the previously made statements apply again.

Given these two HeuristicPlannerTunings, we can confirm the deterministic normalized TrajectoryQuality of the ManagedGridPlanner as well as the non-deterministic normalized TrajectoryQuality of the ManagedTreePlanner in Figures 11.34. Having one intermediate POI leads to the computation of one partial followed by the complete Trajectory with possibly additional solutions for the incrementally increasing normalized TrajectoryQuality within each part. This time, the Trajectory computed by the ManagedGridPlanner fails to avoid the non-RiskPolicy-exceeding adverse yellow weather system due to the high final inflation factor, whereas the ManagedTreePlanner avoids the latter and achieves a significantly higher normalized TrajectoryQuality within only a moderately higher DurationQuantity. Therefore, the tuned final heuristic inflation might generally be too high to avoid lower cost Obstacles reliably and should be reviewed.



(a) Terminal 5 Grid **(b)** Terminal 5 Continuum

Figure 11.33: Terminal 5 Scenario

Table 11.30: Terminal 5 Feature Categories

(a) Terminal 5 Severity

Cost Assessment	ASS	POI	ENV
Maximum	4.0	100.0	100.0
Minimum	4.0	3.0	3.0
Risk Policy	4.0	∞	∞

(b) Terminal 5 Criticality

Criticality	Level
Difficulty	critical (0.1166…)
Severity	severe
Flight Phase	terminal

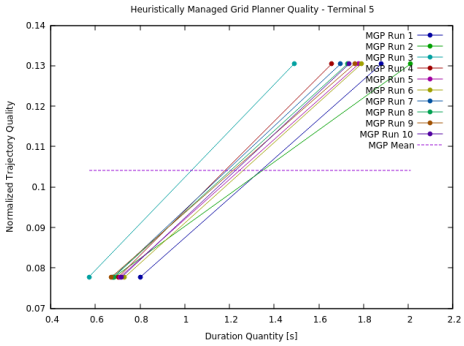
Table 11.31: Terminal 5 Heuristic Tunings

(a) Terminal 5 MGP Heuristic Tuning

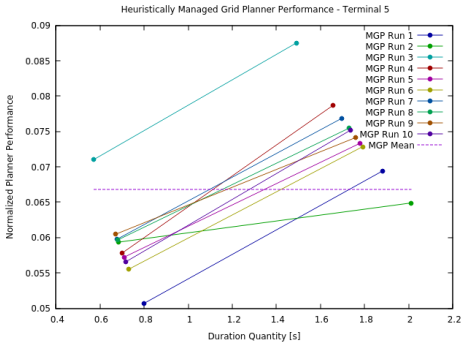
Property	Value
`costPolicy`	`AVERAGE`
`riskPolicy`	`SAFETY`
`minimumQuality`	$46.\overline{2}$
`maximumQuality`	$11.\overline{5}$
`qualityImprovement`	$6.9\overline{3}$
`significantChange`	0.5

(b) Terminal 5 MTP Heuristic Tuning

Property	Value
`costPolicy`	`AVERAGE`
`riskPolicy`	`SAFETY`
`maximumIterations`	1500
`bias`	3
`epsilon`	24.7876…
`neighborLimit`	5
`goalThreshold`	2.5
`extension`	`FEASIBLE`
`strategy`	`EXTEND`
`sampling`	`UNIFORM`
`minimumQuality`	0.0
`maximumQuality`	0.75
`qualityImprovement`	0.15
`significantChange`	0.5



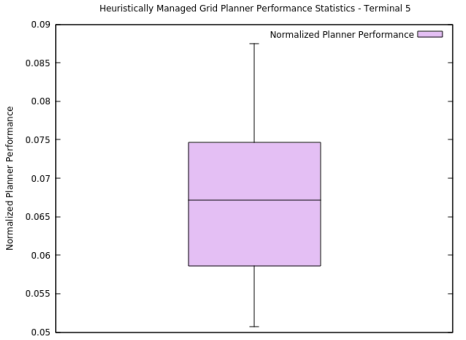
(a) H-AM MGP Quality



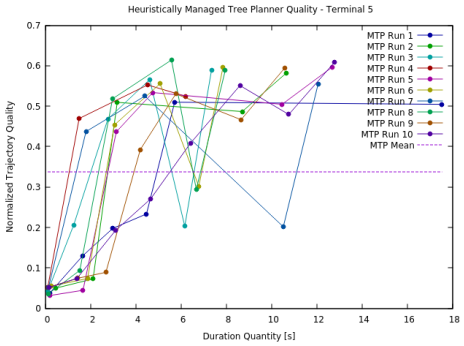
(b) H-AM MGP Performance



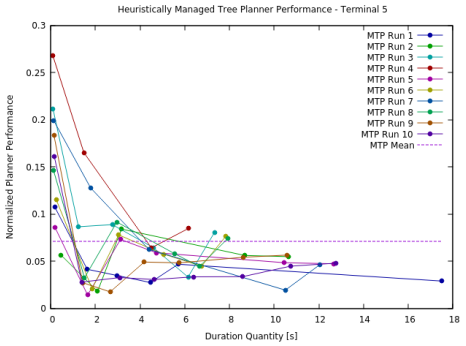
(c) H-AM MGP Quality Statistics



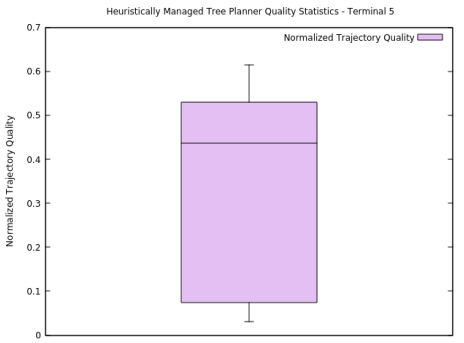
(d) H-AM MGP Performance Statistics



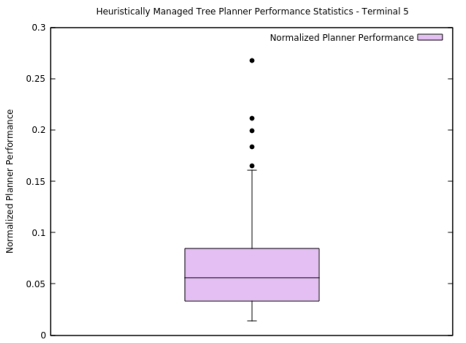
(e) H-AM MTP Quality



(f) H-AM MTP Performance



(g) H-AM MTP Quality Statistics



(h) H-AM MTP Performance Statistics

Figure 11.34: H-AM Quality and Performance: Terminal 5

11.6 Urgency

The *urgency* `FlightPhase` evaluation and training is performed with five associated `Scenarios`. Recall from Section 9.2 that this `FlightPhase` covers the case of a *critical* `Severity`. In this case, the `Aircraft` safety sphere is penetrated by `RiskPolicy`-exhausting SWIM `Obstacles`. Furthermore, the `RiskPolicy` within both the POI bounding box and enclosing `Environment` are exceeded. Any further increase in costs near the `Aircraft` would, consequently, violate the operational `RiskPolicy` and a possible `Trajectory` needs to avoid the threatening `Obstacles` in the surrounding space and time. The `Difficulty` of the *urgency* `FlightPhase` is arbitrary and covers the entire spectrum from *low* to *critical*. Since a *critical* `Severity` is also *severe* by definition, the *urgency* `FlightPhase` is additionally covered by one of the *transition* or *terminal* `FlightPhases` at the same time. The `HeuristicPlannerTuning` currently applies the approach for the respective secondary `FlightPhase` and, therefore, mainly considers the difference in `Difficulty` without providing any special handling. Figure 11.35 shows a `ManagedGridPlanner` for a typical *urgency* `FlightPhase` with the `Aircraft` exposed to a `RiskPolicy`-exhausting SWIM `Obstacle` (red) and its affected voxels as well as `RiskPolicy`-exceeding `Obstacles` within the POI bounding box. The high-cost space is ideally left immediately using the shortest feasible `Trajectory`.

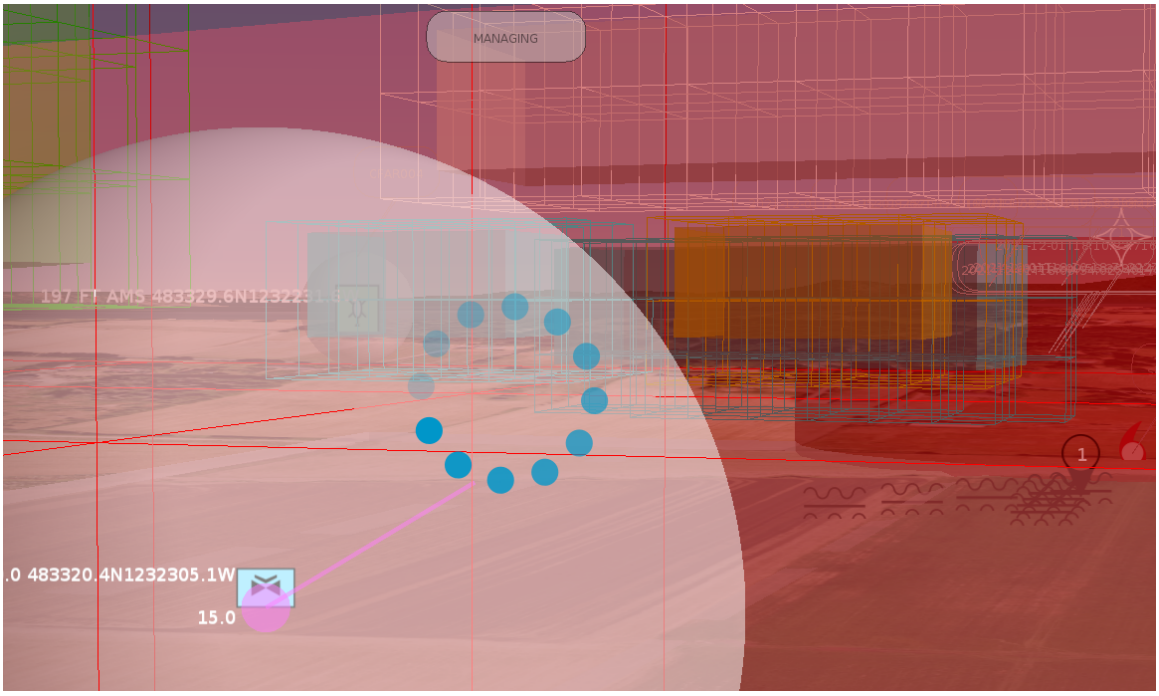


Figure 11.35: Managed Urgency Scenario

11.6.1 Urgency Scenario 1

Urgency Scenario 1 consists of an Environment featuring two stationary SWIM Obstacles. One of them exhausts the operational RiskPolicy and penetrates the Aircraft safety sphere while the other one is RiskPolicy-exceeding and located within the POI bounding box. The Scenario consists of only the start and goal POIs, each at an altitude of 50 meters ASL, and is shown in Figures 11.36 from different directions for the PlanningGrid and PlanningContinuum, respectively.

With the two stationary SWIM Obstacles of cost 50 (red) enclosing the Aircraft safety sphere, and 70 (magenta) within the POI bounding box, respectively, the Difficulty of this Scenario is *substantial* (0.5) with a Severity of *severe* to *critical*, resulting in the *transition* and *urgency* FlightPhase associations shown in Tables 11.32.

The HeuristicAutonomicManager applies the HeuristicPlannerTunings shown in Tables 11.33. The Obstacle-contaminated POI bounding box leads to a tuning approach that considers more Features of the planning Environment, as explained in Section 9.3. Since both the *transition* and *urgency* FlightPhases cover the Features of this Scenario within the Features horizon, a HeuristicPlannerTuning of the former is applied considering the associated Difficulty. Thus, the statements made for Transition Scenario 2 of Section 11.4.2 equally apply here.

Given these two HeuristicPlannerTunings, we can confirm the deterministic normalized TrajectoryQuality of the ManagedGridPlanner as well as the non-deterministic normalized TrajectoryQuality of the ManagedTreePlanner in Figures 11.37. Having no intermediate POIs immediately leads to the computation of the complete Trajectory, which in the case of the ManagedTreePlanner takes one or multiple non-deterministic steps with increasing normalized TrajectoryQuality. Whereas the ManagedGridPlanner does not leave the RiskPolicy-exhausting red Obstacle on the shortest possible Trajectory and is, furthermore, constrained by the discrete PlanningGrid when avoiding the RiskPolicy-exceeding magenta Obstacle, the ManagedTreePlanner achieves a significantly higher normalized TrajectoryQuality within a reasonable DurationQuantity. This behavior is again an indicator for a likely too high final heuristic inflation in case of the ManagedGridPlanner. Nonetheless, both solutions are feasible and obey the operational RiskPolicy.



(a) Urgency 1 Grid **(b)** Urgency 1 Continuum

Figure 11.36: Urgency 1 Scenario

Table 11.32: Urgency 1 Feature Categories

(a) Urgency 1 Severity

Cost Assessment	ASS	POI	ENV
Maximum	50.0	70.0	70.0
Minimum	50.0	50.0	50.0
Risk Policy	50.0	∞	∞

(b) Urgency 1 Criticality

Criticality	Level
Difficulty	substantial (0.5)
Severity	severe, critical
Flight Phase	transition, urgency

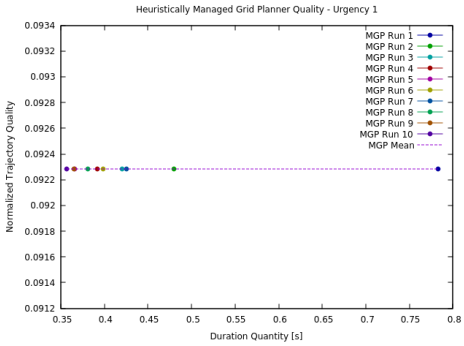
Table 11.33: Urgency 1 Heuristic Tunings

(a) Urgency 1 MGP Heuristic Tuning

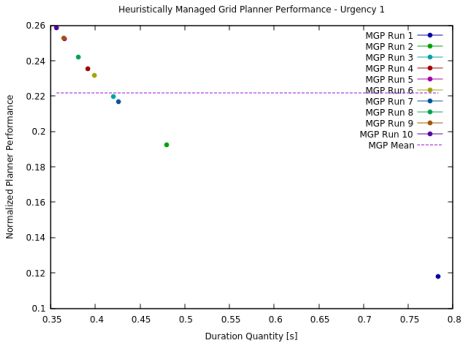
Property	Value
`costPolicy`	`AVERAGE`
`riskPolicy`	`SAFETY`
`minimumQuality`	61.0
`maximumQuality`	30.5
`qualityImprovement`	6.1
`significantChange`	0.5

(b) Urgency 1 MTP Heuristic Tuning

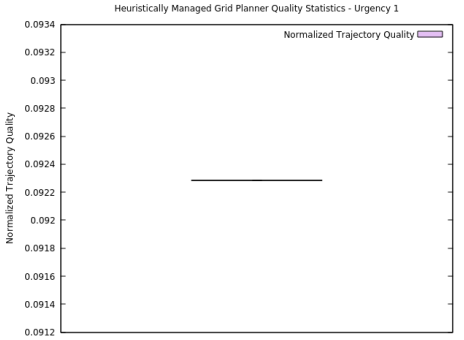
Property	Value
`costPolicy`	`AVERAGE`
`riskPolicy`	`SAFETY`
`maximumIterations`	1000
`bias`	5
`epsilon`	28.0579…
`neighborLimit`	3
`goalThreshold`	2.5
`extension`	`FEASIBLE`
`strategy`	`EXTEND`
`sampling`	`ELLIPSOIDAL`
`minimumQuality`	0.0
`maximumQuality`	0.5
`qualityImprovement`	0.1
`significantChange`	0.5



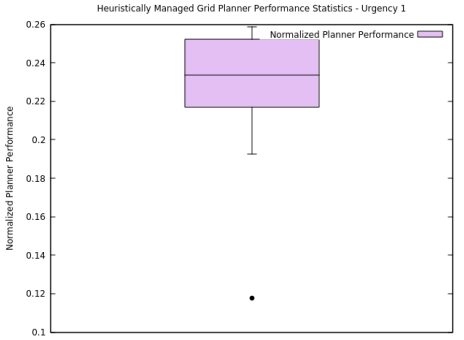
(a) H-AM MGP Quality



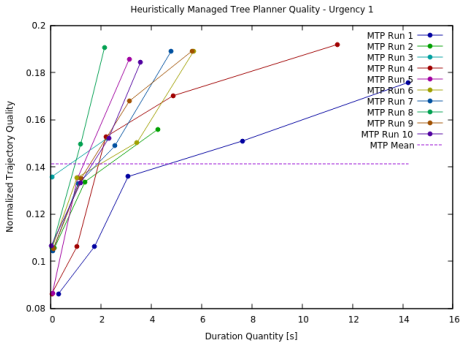
(b) H-AM MGP Performance



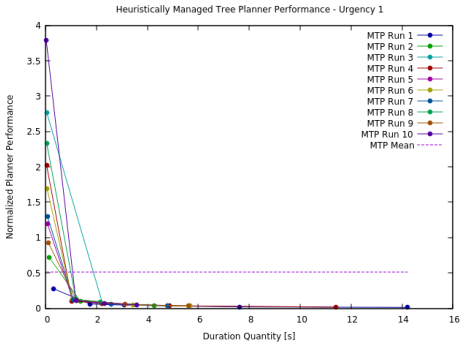
(c) H-AM MGP Quality Statistics



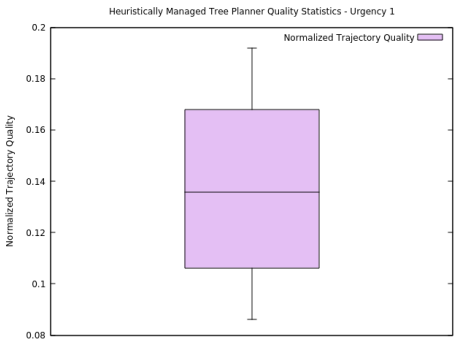
(d) H-AM MGP Performance Statistics



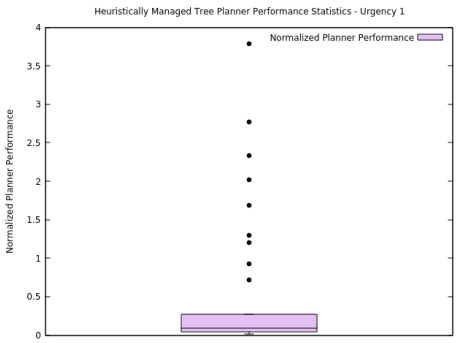
(e) H-AM MTP Quality



(f) H-AM MTP Performance



(g) H-AM MTP Quality Statistics



(h) H-AM MTP Performance Statistics

Figure 11.37: H-AM Quality and Performance: Urgency 1

11.6.2 Urgency Scenario 2

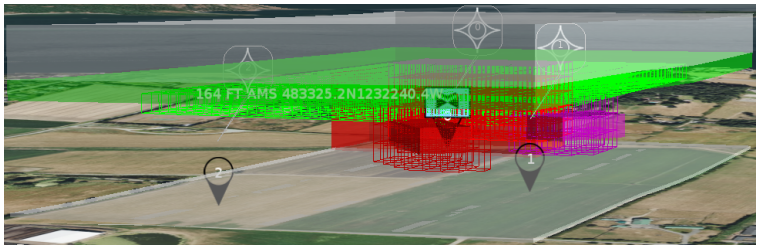
Urgency Scenario 2 consists of an Environment featuring four stationary SWIM Obstacles. One of them exhausts the operational RiskPolicy and penetrates the Aircraft safety sphere while two others are RiskPolicy-exceeding and located within the POI bounding box. The fourth one is a large non-RiskPolicy-exceeding SWIM Obstacle that only affects the remaining higher-level Environment. The Scenario consists of one intermediate POI, at an altitude of 60 meters ASL, that has to be visited when traveling from the start to the goal POI, each at an altitude of 50 meters ASL, and is shown in Figures 11.38 from different directions for the PlanningGrid and PlanningContinuum, respectively.

With the four stationary SWIM Obstacles of cost 50 (red) enclosing the Aircraft safety sphere, 60 (dark red) and 70 (magenta), respectively, within the POI bounding box, and 20 (light green) within the remaining Environment, the Difficulty of this Scenario remains *substantial* (0.5). This is due to the compensating larger high-level light green Obstacle. The additional Severity of *severe* to *critical* results in the *transition* and *urgency* FlightPhase associations shown in Tables 11.34.

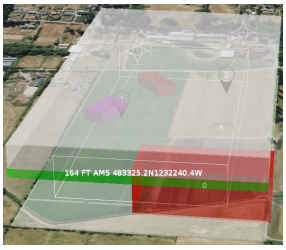
The HeuristicAutonomicManager applies the HeuristicPlannerTunings shown in Tables 11.35. The Obstacle-contaminated POI bounding box leads to a tuning approach that considers more Features of the planning Environment, as explained in Section 9.3. Since the Criticality assessment of this Scenario equals the one for the Urgency 1 Scenario of Section 11.6.1, their HeuristicPlannerTunings are equivalent and the previously made statements equally apply here.

Given these two HeuristicPlannerTunings, we can confirm the deterministic normalized TrajectoryQuality of the ManagedGridPlanner as well as the non-deterministic normalized TrajectoryQuality of the ManagedTreePlanner in Figures 11.39. Having one intermediate POI leads to the computation of one partial followed by the complete Trajectory, which in the case of the ManagedTreePlanner takes one or multiple non-deterministic steps with increasing normalized TrajectoryQuality for a single part. As for Urgency Scenario 1 of Section 11.6.1, the normalized TrajectoryQuality only improves along the DurationQuantity since the first Trajectory part has to leave the RiskPolicy-exhausting Obstacle with a significant penalty before proceeding with the avoidance of Obstacles within the POI bounding box for the second part. Whereas the ManagedGridPlanner is again constrained by the discrete PlanningGrid when avoiding the RiskPolicy-exceeding dark-red and

magenta `Obstacles`, the`ManagedTreePlanner` achieves a slightly higher normalized `TrajectoryQuality` within a reasonable `DurationQuantity`.



(a) Urgency 2 Grid



(b) Urgency 2 Continuum

Figure 11.38: Urgency 2 Scenario

Table 11.34: Urgency 2 Feature Categories

(a) Urgency 2 Severity

Cost Assessment	ASS	POI	ENV
Maximum	50.0	70.0	70.0
Minimum	50.0	50.0	20.0
Risk Policy	50.0	∞	∞

(b) Urgency 2 Criticality

Criticality	Level
Difficulty	substantial (0.5)
Severity	severe, critical
Flight Phase	transition, urgency

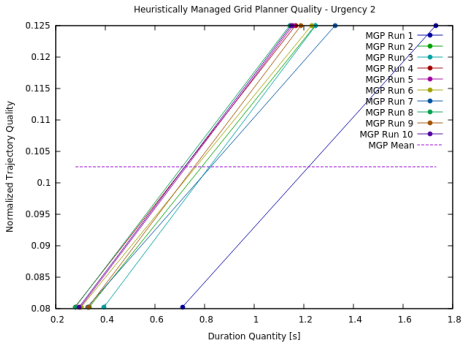
Table 11.35: Urgency 2 Heuristic Tunings

(a) Urgency 2 MGP Heuristic Tuning

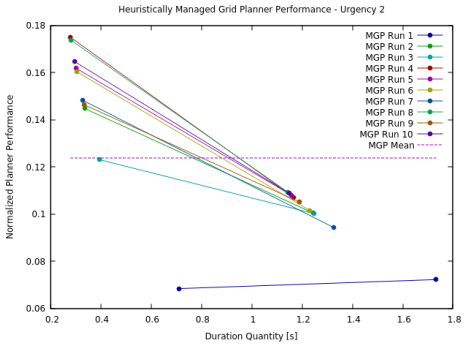
Property	Value
`costPolicy`	`AVERAGE`
`riskPolicy`	`SAFETY`
`minimumQuality`	61.0
`maximumQuality`	30.5
`qualityImprovement`	6.1
`significantChange`	0.5

(b) Urgency 2 MTP Heuristic Tuning

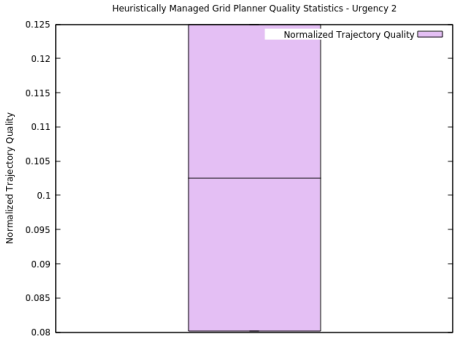
Property	Value
`costPolicy`	`AVERAGE`
`riskPolicy`	`SAFETY`
`maximumIterations`	1000
`bias`	5
`epsilon`	19.9225…
`neighborLimit`	3
`goalThreshold`	2.5
`extension`	`FEASIBLE`
`strategy`	`EXTEND`
`sampling`	`ELLIPSOIDAL`
`minimumQuality`	0.0
`maximumQuality`	0.5
`qualityImprovement`	0.1
`significantChange`	0.5



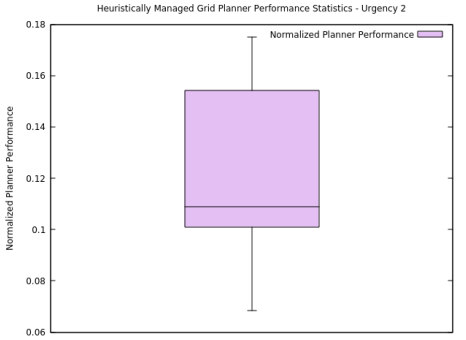
(a) H-AM MGP Quality



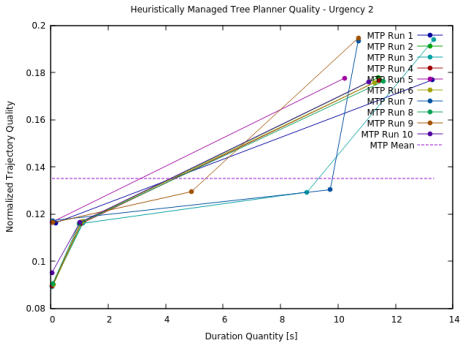
(b) H-AM MGP Performance



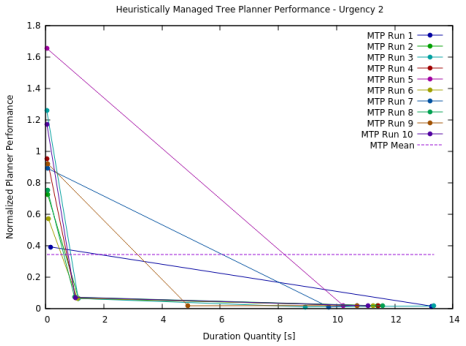
(c) H-AM MGP Quality Statistics



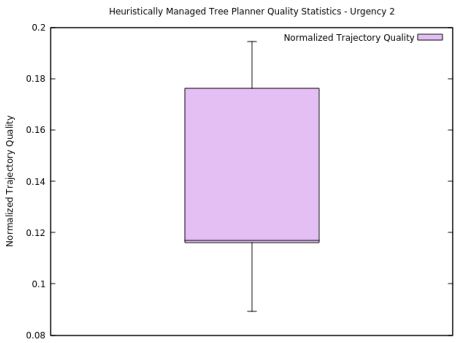
(d) H-AM MGP Performance Statistics



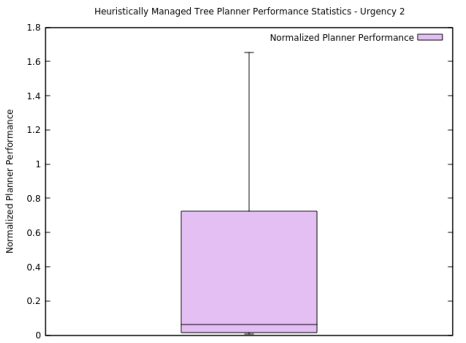
(e) H-AM MTP Quality



(f) H-AM MTP Performance



(g) H-AM MTP Quality Statistics



(h) H-AM MTP Performance Statistics

Figure 11.39: H-AM Quality and Performance: Urgency 2

11.6.3 Urgency Scenario 3

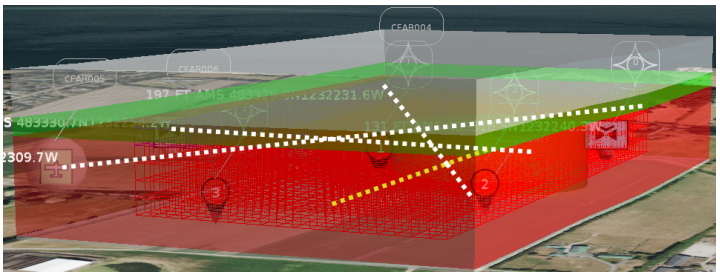
Urgency Scenario 3 consists of an Environment featuring an all-enclosing stationary SWIM Obstacle which exhausts the operational RiskPolicy and cannot be avoided. In addition, several moving SWIM Obstacles representing other aircraft as well as one non-RiskPolicy-exceeding weather system are included. The Scenario consists of two intermediate POIs, at an altitude of 50 meters ASL each, that have to be visited in sequence when traveling from the start to the goal POI, each at an altitude of 40 meters ASL, and is shown with the projected Obstacle movements in Figures 11.40 from different directions for the PlanningGrid and PlanningContinuum, respectively.

With several scattered moving SWIM Obstacles of costs 100 (white aircraft) and 40 (yellow weather system) in addition to the large all-enclosing Obstacles of cost 20 (light green) and 50 (red), respectively, within the Features horizon of this Scenario, its Difficulty reaches the highest level *critical* (0.0340...) while its Severity is considered *severe* to *critical*, resulting in the *terminal* and *urgency* FlightPhase association shown in Tables 11.36. While the RiskPolicy cost assessment for the Aircraft safety sphere is exhausted, it is clearly exceeded for the POI bounding box and remaining Environment.

The HeuristicAutonomicManager applies the HeuristicPlannerTunings shown in Tables 11.37. The Obstacle-contaminated POI bounding box leads to a tuning approach that considers more Features of the planning Environment, as explained in Section 9.3. Since both the *terminal* and *urgency* FlightPhases cover the Features of this Scenario within the Features horizon, a HeuristicPlannerTuning of the former is applied considering the associated Difficulty. Thus, the statements made for Terminal Scenario 3 of Section 11.5.3 equally apply here.

Given these two HeuristicPlannerTunings, we can confirm the deterministic normalized TrajectoryQuality of the ManagedGridPlanner as well as the non-deterministic normalized TrajectoryQuality of the ManagedTreePlanner in Figures 11.41. Having two intermediate POIs leads to the computation of two partial followed by the complete Trajectory with possibly additional solutions for the incrementally increasing normalized TrajectoryQuality within each part. As mentioned with similar Scenarios of all-enclosing non-RiskPolicy-exceeding SWIM Obstacles, the underestimating heuristics, if not properly inflated, as well as the higher maximum iterations may increase the DurationQuantity substantially. On the other hand, not deflating or increasing the cost bias sufficiently may lead to more costly, but still feasible and pos-

sibly shorter, solutions. The normalized `TrajectoryQuality` of this `Scenario` cannot be high because of the omnipresent high-cost penalty. The embedded `Obstacles` are adding to the `Difficulty` and `Severity` of this `Scenario` making it one of the most challenging ones in the set, resulting in a high `DurationQuantity`.



(a) Urgency 3 Grid



(b) Urgency 3 Continuum

Figure 11.40: Urgency 3 Scenario

Table 11.36: Urgency 3 Feature Categories

(a) Urgency 3 Severity

Cost Assessment	ASS	POI	ENV
Maximum	50.0	100.0	100.0
Minimum	50.0	40.0	20.0
Risk Policy	50.0	∞	∞

(b) Urgency 3 Criticality

Criticality	Level
Difficulty	critical (0.0340…)
Severity	severe, critical
Flight Phase	terminal, urgency

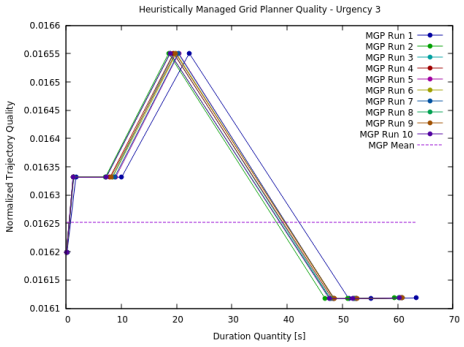
Table 11.37: Urgency 3 Heuristic Tunings

(a) Urgency 3 MGP Heuristic Tuning

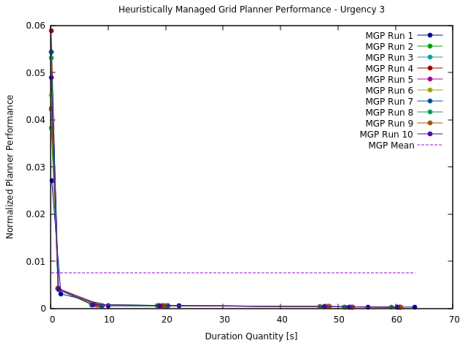
Property	Value
`costPolicy`	`AVERAGE`
`riskPolicy`	`SAFETY`
`minimumQuality`	73.5
`maximumQuality`	18.375
`qualityImprovement`	11.025
`significantChange`	0.5

(b) Urgency 3 MTP Heuristic Tuning

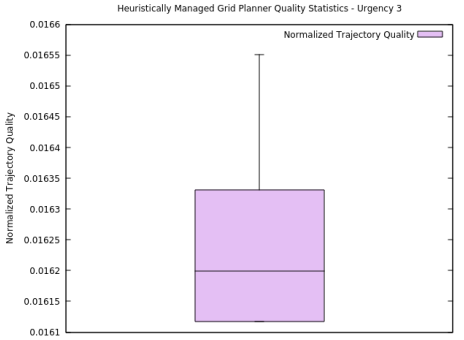
Property	Value
`costPolicy`	`AVERAGE`
`riskPolicy`	`SAFETY`
`maximumIterations`	1500
`bias`	3
`epsilon`	13.5917…
`neighborLimit`	5
`goalThreshold`	2.5
`extension`	`FEASIBLE`
`strategy`	`EXTEND`
`sampling`	`UNIFORM`
`minimumQuality`	0.0
`maximumQuality`	0.75
`qualityImprovement`	0.15
`significantChange`	0.5



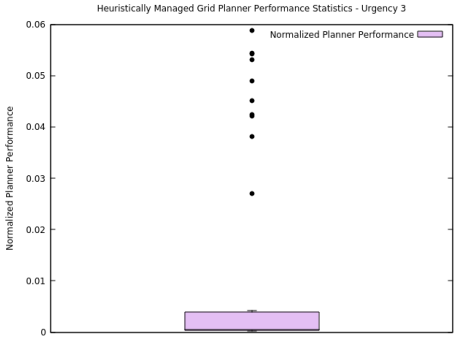
(a) H-AM MGP Quality



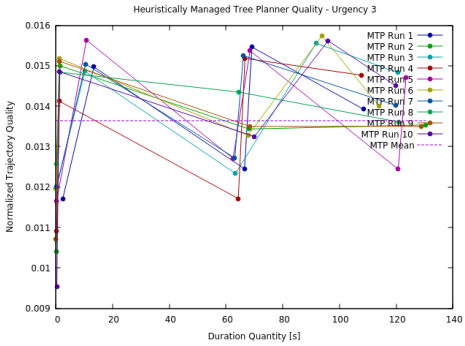
(b) H-AM MGP Performance



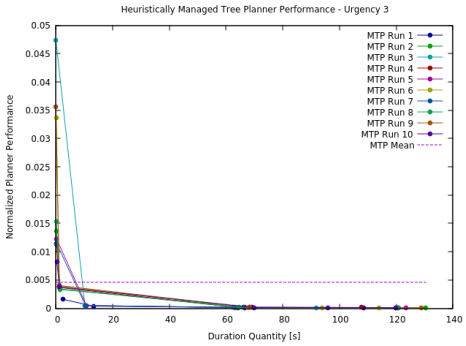
(c) H-AM MGP Quality Statistics



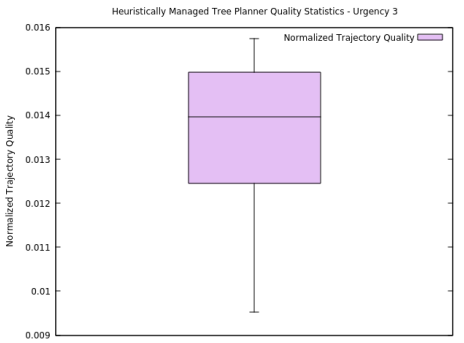
(d) H-AM MGP Performance Statistics



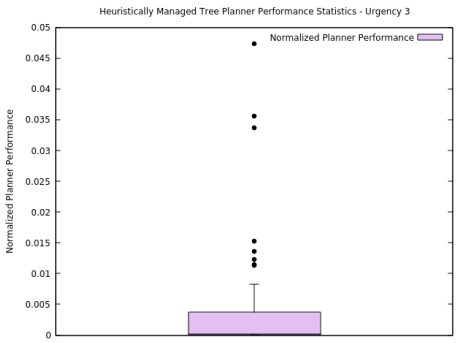
(e) H-AM MTP Quality



(f) H-AM MTP Performance



(g) H-AM MTP Quality Statistics



(h) H-AM MTP Performance Statistics

Figure 11.41: H-AM Quality and Performance: Urgency 3

11.6.4 Urgency Scenario 4

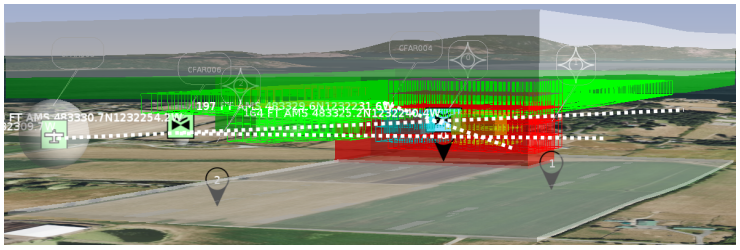
Urgency Scenario 4 consists of an Environment featuring one large high-level and four smaller stationary non-RiskPolicy-exceeding SWIM Obstacles. One of these smaller Obstacles is RiskPolicy-exhausting and encloses the Aircraft safety sphere. Additionally, several moving SWIM Obstacles representing other aircraft are included. The Scenario consists of one intermediate POI, at an altitude of 60 meters ASL, that has to be visited when traveling from the start to the goal POI, each at an altitude of 50 meters ASL, and is shown with the projected Obstacle movements in Figures 11.42 from different directions for the PlanningGrid and PlanningContinuum, respectively.

With several scattered stationary and moving SWIM Obstacles of costs 10 (cyan), 20 (light green), 40 (yellow), 50 (red), and 100 (white aircraft), respectively, within the Features horizon of this Scenario, its Difficulty reaches the highest level *critical* (0.0363…) with a *severe* to *critical* Severity, resulting in the *terminal* and *urgency* FlightPhase associations shown in Tables 11.38. While the Aircraft safety sphere is penetrated by the RiskPolicy-exhausting red Obstacle, the RiskPolicy within the POI bounding box and enclosing Environment is clearly exceeded by the presence of the moving white aircraft.

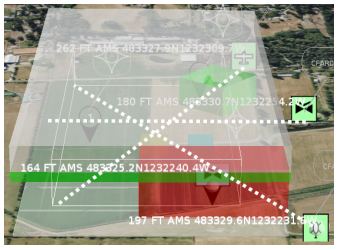
The HeuristicAutonomicManager applies the HeuristicPlannerTunings shown in Tables 11.39. The Obstacle-contaminated POI bounding box leads to a tuning approach that considers more Features of the planning Environment, as explained in Section 9.3. Since both the *terminal* and *urgency* FlightPhases cover the Features of this Scenario within the Features horizon, a HeuristicPlannerTuning of the former is applied considering the associated Difficulty. Thus, the statements made for Terminal Scenario 3 of Section 11.5.3 equally apply here.

Given these two HeuristicPlannerTunings, we can confirm the deterministic normalized TrajectoryQuality of the ManagedGridPlanner as well as the non-deterministic normalized TrajectoryQuality of the ManagedTreePlanner in Figures 11.43. Having one intermediate POI leads to the computation of one partial followed by the complete Trajectory with possibly additional solutions for the incrementally increasing normalized TrajectoryQuality within each part. Similar to Urgency Scenario 2 of Section 11.6.2 and due to the RiskPolicy-exhausting red Obstacle surrounding the Aircraft safety sphere and causing a significant inital penalty, the normalized TrajectoryQuality improves considerably after leaving the affected space.

The lower normalized `TrajectoryQuality` in case of the `ManagedGridPlanner` is again the result of the constraining `PlanningGrid`.



(a) Urgency 4 Grid



(b) Urgency 4 Continuum

Figure 11.42: Urgency 4 Scenario

Table 11.38: Urgency 4 Feature Categories

(a) Urgency 4 Severity

Cost Assessment	ASS	POI	ENV
Maximum	50.0	100.0	100.0
Minimum	50.0	10.0	10.0
Risk Policy	50.0	∞	∞

(b) Urgency 4 Criticality

Criticality	Level
Difficulty	critical (0.0363…)
Severity	severe, critical
Flight Phase	terminal, urgency

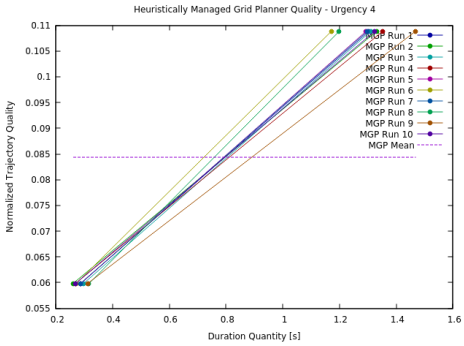
Table 11.39: Urgency 4 Heuristic Tunings

(a) Urgency 4 MGP Heuristic Tuning

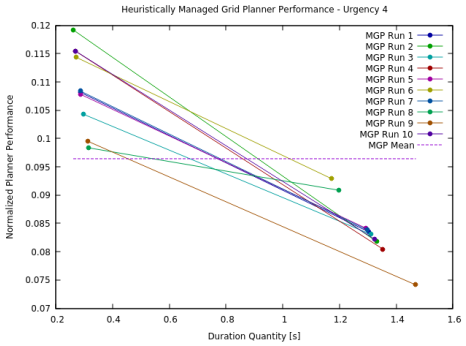
Property	Value
`costPolicy`	`AVERAGE`
`riskPolicy`	`SAFETY`
`minimumQuality`	$67.\overline{6}$
`maximumQuality`	$16.91\overline{6}$
`qualityImprovement`	10.15
`significantChange`	0.5

(b) Urgency 4 MTP Heuristic Tuning

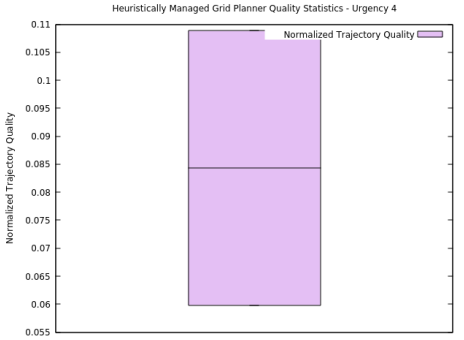
Property	Value
`costPolicy`	`AVERAGE`
`riskPolicy`	`SAFETY`
`maximumIterations`	1500
`bias`	3
`epsilon`	16.8343…
`neighborLimit`	5
`goalThreshold`	2.5
`extension`	`FEASIBLE`
`strategy`	`EXTEND`
`sampling`	`UNIFORM`
`minimumQuality`	0.0
`maximumQuality`	0.75
`qualityImprovement`	0.15
`significantChange`	0.5



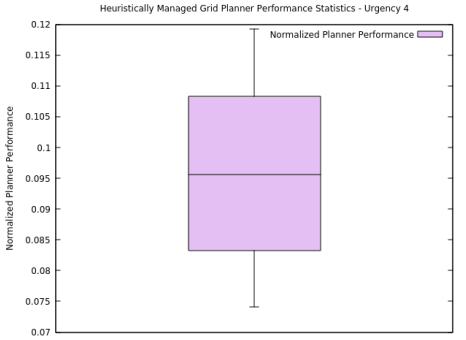
(a) H-AM MGP Quality



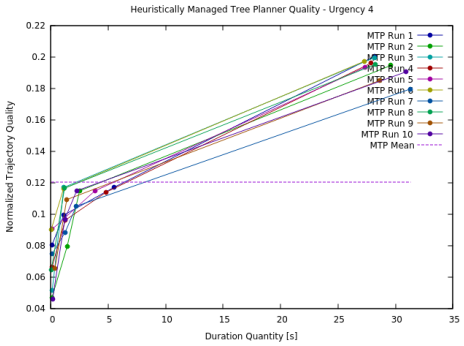
(b) H-AM MGP Performance



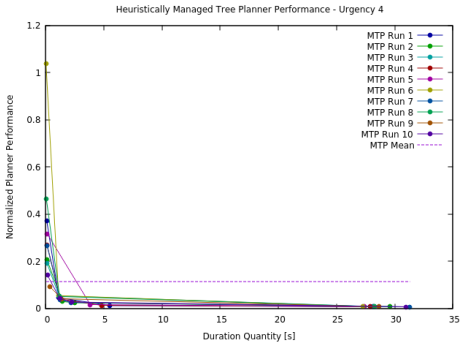
(c) H-AM MGP Quality Statistics



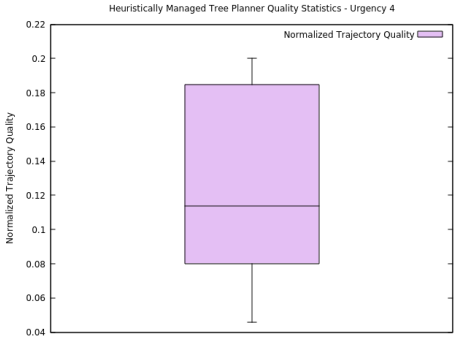
(d) H-AM MGP Performance Statistics



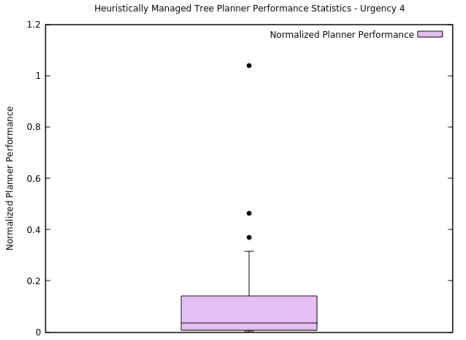
(e) H-AM MTP Quality



(f) H-AM MTP Performance



(g) H-AM MTP Quality Statistics



(h) H-AM MTP Performance Statistics

Figure 11.43: H-AM Quality and Performance: Urgency 4

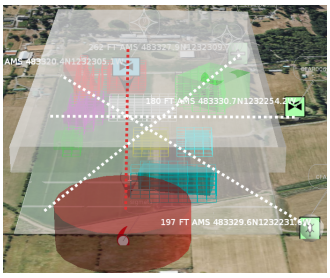
11.6.5 Urgency Scenario 5

Urgency Scenario 5 consists of an Environment featuring many stationary and moving SWIM Obstacles, including other aircraft and adverse weather. The Scenario comprises one intermediate POI, at an altitude of 40 meters ASL, that has to be visited when traveling from the start to the goal POI, at altitudes of 60 and 50 meters ASL, respectively, and is shown with the projected Obstacle movements in Figures 11.44 from different directions for both the PlanningGrid and PlanningContinuum.

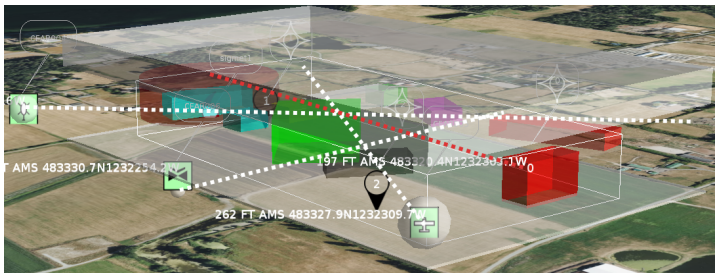
With many scattered stationary and moving SWIM Obstacles of varying costs within the Features horizon of this Scenario, its Difficulty reaches the highest level *critical* (0.0932…) and a *severe* to *critical* Severity, resulting in the *terminal* and *urgency* FlightPhase associations shown in Tables 11.40. While the Aircraft safety sphere is penetrated by the RiskPolicy-exhausting red Obstacle, the RiskPolicy within the POI bounding box and enclosing Environment is clearly exceeded by the presence of the moving dark red adverse weather and white aircraft in addition to the stationary white and magenta SWIM Obstacles.

The HeuristicAutonomicManager applies the HeuristicPlannerTunings shown in Tables 11.41. The Obstacle-contaminated POI bounding box leads to a tuning approach that considers more Features of the planning Environment, as explained in Section 9.3. Since both the *terminal* and *urgency* FlightPhases cover the Features of this Scenario within the Features horizon, a HeuristicPlannerTuning of the former is applied considering the associated Difficulty. Thus, the statements made for Terminal Scenario 5 of Section 11.5.5 equally apply here.

Given these two HeuristicPlannerTunings, we can confirm the deterministic normalized TrajectoryQuality of the ManagedGridPlanner as well as the non-deterministic normalized TrajectoryQuality of the ManagedTreePlanner in Figures 11.45. Having one intermediate POI leads to the computation of one partial followed by the complete Trajectory with possibly additional solutions for the incrementally increasing normalized TrajectoryQuality within each part. Similar to Urgency Scenario 4 of Section 11.6.4 and due to the RiskPolicy-exhausting red Obstacle surrounding the Aircraft safety sphere and causing a significant inital penalty, the normalized TrajectoryQuality improves after leaving the affected space but is limited to due the required detours within the Environment. The lower normalized TrajectoryQuality in case of the ManagedGridPlanner is again the result of the constraining PlanningGrid.



(a) Urgency 5 Grid



(b) Urgency 5 Continuum

Figure 11.44: Urgency 5 Scenario

Table 11.40: Urgency 5 Feature Categories

(a) Urgency 5 Severity

Cost Assessment	ASS	POI	ENV
Maximum	50.0	100.0	100.0
Minimum	50.0	3.0	3.0
Risk Policy	50.0	∞	∞

(b) Urgency 5 Criticality

Criticality	Level
Difficulty	critical (0.0932…)
Severity	severe, critical
Flight Phase	terminal, urgency

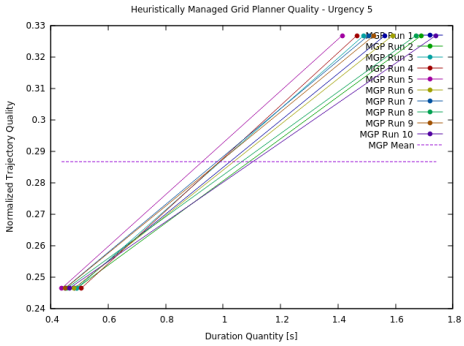
Table 11.41: Urgency 5 Heuristic Tunings

(a) Urgency 5 MGP Heuristic Tuning

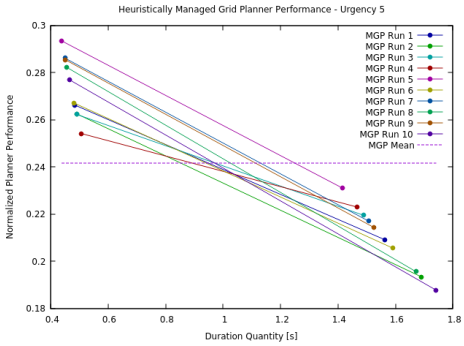
Property	Value
`costPolicy`	`AVERAGE`
`riskPolicy`	`SAFETY`
`minimumQuality`	$53.\overline{5}$
`maximumQuality`	$13.3\overline{8}$
`qualityImprovement`	$8.0\overline{3}$
`significantChange`	0.5

(b) Urgency 5 MTP Heuristic Tuning

Property	Value
`costPolicy`	`AVERAGE`
`riskPolicy`	`SAFETY`
`maximumIterations`	1500
`bias`	3
`epsilon`	24.7876…
`neighborLimit`	5
`goalThreshold`	2.5
`extension`	`FEASIBLE`
`strategy`	`EXTEND`
`sampling`	`UNIFORM`
`minimumQuality`	0.0
`maximumQuality`	0.75
`qualityImprovement`	0.15
`significantChange`	0.5



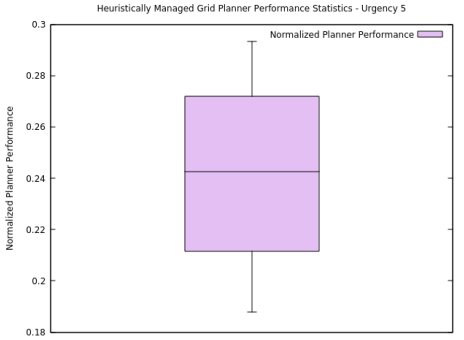
(a) H-AM MGP Quality



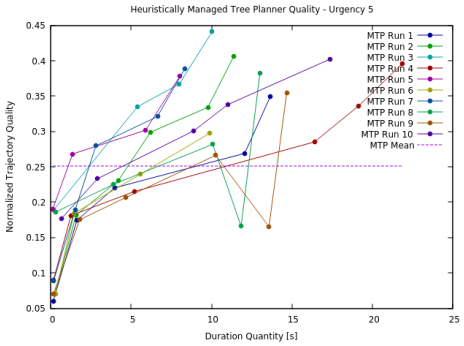
(b) H-AM MGP Performance



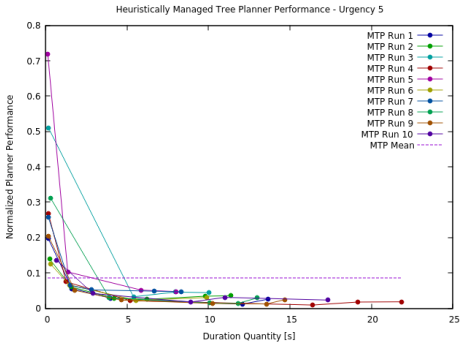
(c) H-AM MGP Quality Statistics



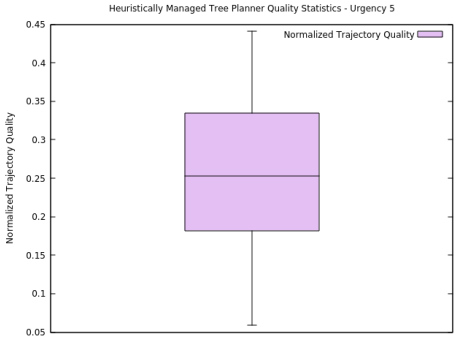
(d) H-AM MGP Performance Statistics



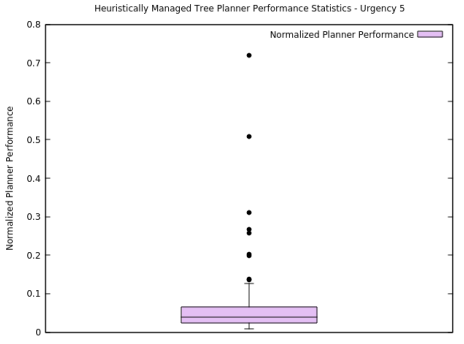
(e) H-AM MTP Quality



(f) H-AM MTP Performance



(g) H-AM MTP Quality Statistics



(h) H-AM MTP Performance Statistics

Figure 11.45: H-AM Quality and Performance: Urgency 5

11.7 Emergency

Recall from Sections 6.5 and 7.8 that a sudden and unexpected increase in cost at the position of the airborne datalink source due to the injection of dynamic SWIM `Obstacles` may always exceed the operational `RiskPolicy`. The airborne agent must, however, not be abandoned in this case. To mitigate this issue, the `RiskPolicy` is temporarily boosted to satisfy the unplanned situation and escape the existing threat. It should be emphasized that this behavior only applies for an *airborne* agent with the discussion naturally leading to the topic of contingencies and alternate destinations revisited in Chapter 15. If the agent is not airborne but exposed to `RiskPolicy`-exceeding SWIM `Obstacles`, then simply no feasible `Trajectory` can be computed. The intended implementation, therefore, attempts to mimic acceptable PDM and avoid a *pilot get-homitis* violating established operational policies.

Since for an airborne agent with a `RiskPolicy`-exceeding *emergency* `FlightPhase`, the applicable `RiskPolicy` is temporarily boosted to satisfy the existing situation, the `Scenario` becomes equivalent to the *urgency* `FlightPhase` and does not necessarily require a separate evaluation. The `HeuristicAutonomicManager` does currently not apply a special handling in this case. However, both *urgency* and *emergency* `FlightPhases` are recognized from the extracted `Features` within their horizon and could be tuned for differently.

11.8 Interpretation

Examining the diagrams for the normalized `TrajectoryQuality` when `ManagedPlanners` are tuned by the `HeuristicAutonomicManager`, it appears that our approach is viable for a wide range of different `Scenarios`. A first and feasible `Trajectory` is available quickly and usually within a few seconds, which results in a high initial normalized `PlannerPerformance`. During the elaboration of this initial feasible `Trajectory`, possibly along a series of intermediate POIs, the achieved normalized `TrajectoryQuality` may increase further improving the already feasible solution while the `PlannerPerformance` decreases rapidly due to generally low gains in `TractoryQuality` compared to the required additional `DurationQuantity`. Since each feasible `Trajectory` is uploaded to the datalink target, the airborne agent receives a first response to a dynamic change fairly quickly, followed by subsequent revisions and improvements while underway. Furthermore, it is apparent that some

implemented heuristics of the HeuristicPlannerTuning, that is employed by the HeuristicAutonomicManager, should be reviewed and can potentially be improved.

For example, the HeuristicAutonomicManager applies an initial heuristic inflation to the ManagedGridPlanner that considers both the base cost of the PlanningGrid as well as the average Obstacle cost within the POI bounding box. This approach nicely addresses an otherwise potentially underestimating heuristic and an unacceptably high resulting DurationQuantity, especially with all-enclosing RiskPolicy-exhausting Obstacles. However, without the existence of additional costs in the Environment, as in the case of Cruise Scenario 1 evaluated in Section 11.3.1, the ManagedGridPlanner is reduced to a basic A*, by using the base cost of its PlanningGrid as the heuristic inflation factor, leading to a longer search in an Obstacle-free Environment. A relatively high resolution PlanningGrid of division 40 is obviously not required in this Scenario. Therefore, tuning the Environment in addition to the ManagedPlanner based on the extracted Features would significantly improve the resulting PlannerPerformance and be equivalent to the adaptability when tuning epsilon in case of the ManagedTreePlanner. Additionally, supporting *any-angle* ManagedGridPlanners in the SAFCS core would partially remove the TrajectoryQuality restrictions imposed by the PlanningGrid. The applied final heuristic inflation sometimes appears too high and, as in the case of Terminal Scenario 5 evaluated in Section 11.5.5, may lead to less optimal solutions but within a shorter DurationQuantity.

In general, we are not suggesting a particular set of heuristics for the HeuristicAutonomicManager but rather do encourage the viable approach of extracting Scenario Features within a horizon and tuning ManagedPlanners to plan for a feasible initial and improving TrajectoryQuality within a reasonable DurationQuantity. Organizing the extracted Features into FeatureCategories of varying Criticality, in our case Difficulty, Severity and FlightPhase, helped to break down the complexities of different Scenarios.

The assessment of the Scenario based on a Features horizon may, however, lead to several problems not shown in our evaluation. The Features horizon is based on the ETD of the Aircraft and may, therefore, nicely capture the nearby Features in space and time. However, as explained in Section 9.2, extending the horizon for too long may include irrelevant Features that are only temporarily present. Specifying a horizon that is too short may, on the other hand, omit important Features. In the worst case, the Features of the Scenario may change dramatically from one horizon to another. Be-

sides the currently existing instrument error related to intersections of volumes in space and time discussed in Section 9.2, an unfortunate horizon choice may lead to an inappropriate PlannerTuning. For example, Cruise Scenario 1 of Section 11.3.1 leads to a very optimistic HeuristicPlannerTuning for the ManagedTreePlanner. Obviously, a goal bias of 100% using a full extension to the goal within a single iteration can only be feasible if there are indeed no obstructing RiskPolicy-exceeding SWIM Obstacles. However, such Obstacles may very well exist just beyond the specified Features horizon. Therefore, the HeuristicAutonomicManager should never tune too optimistically but rather acknowledge the possible weaknesses of the Features horizon.

Clearly, without any adaptive PlannerTuning an unmanaged Planner using a fix set of Specification Properties would not have performed in all of the presented Scenarios and may not have delivered any solution at all. For example, an insufficient number of iterations for small tree extensions and high cost bias in case of the ManagedTreePlanner may never find a solution. Therefore, we consider our existing HeuristicAutonomicManager as a starting point toward improved and more flexible PlannerTunings.

11.9 Sequential Model-Based Algorithm Configuration

To evaluate the SmacAutonomicManager we have performed several evaluation runs in TRAINING mode using the SmacPlannerEvaluator[4] mentioned in Section 9.7 and shown in Figure 9.12. Instead of optimizing the normalized PlannerPerformance, which would encourage fast but only feasible solutions of lower normalized TrajectoryQuality, we decided to optimize for normalized TrajectoryQuality using a relatively tight cut-off of 20 seconds. The cut-off has been chosen to be generous enough to encourage a higher TrajectoryQuality for most of our evaluation Scenarios while being tight for challenging Scenarios featuring multiple POIs and under-estimating heuristics (e.g., Scenarios of type 3). In all cases, the entire (multi-part) Trajectory, that is, all individual parts, had to be available at the cut-off time limit to consider the SmacPlannerTuning satisfactory. 1000 training runs have been performed for each of the FlightPhase sets evaluated in Sections 11.3, 11.4, 11.5, and 11.6, respectively.

[4] https://github.com/stephanheinemann/worldwind/blob/develop/src/main/java/com/cfar/swim/worldwind/managers/smac/SmacPlannerEvaluator.java

```
# name type range default
minimumQuality real [1.0, 100.0] [100.0]
maximumQuality real [1.0, 100.0] [1.0]
qualityImprovement real [0.1, 100.0] [20.0]
# forbidden clauses
{(minimumQuality < maximumQuality) ||
((qualityImprovement - 0.1) > (minimumQuality - maximumQuality))}
```

Listing 11.1: SAFCS AM: MGP SMAC Parameter Configuration Space

```
# name type range default
sampling categorical {ELLIPSOIDAL, UNIFORM, GAUSSIAN} [UNIFORM]
strategy categorical {EXTEND, CONNECT} [EXTEND]
#extension categorical {LINEAR, FEASIBLE} [FEASIBLE]
maxIterations integer [1, 2000] [1000]
epsilon real [1, 1000] [500]
bias integer [0, 100] [5]
#goalThreshold real [0.0, 500] [50]
neighborLimit integer [1, 10] [5]
initialCostBias real [0.0, 1.0] [0.0]
finalCostBias real [0.0, 1.0] [1.0]
improvementFactor real [0.001, 1.0] [0.2]

# forbidden clauses
#{epsilon < goalThreshold}
{(initialCostBias > finalCostBias) ||
((improvementFactor - 0.001) > (finalCostBias - initialCostBias))}
```

Listing 11.2: SAFCS AM: MTP SMAC Parameter Configuration Space

The applied parameter configuration spaces are shown in Listings 11.1 and 11.2 for the `ManagedGridPlanner` and `ManagedTreePlanner`, respectively.

For both `ManagedPlanners` we are interested in better parameter values for the minimum, maximum, and quality improvement granularity. Since, the chosen cut-off encourages feasible satisfactory solutions while allowing for consecutive potential improvements, we would like to know how much initial inflation (stronger distance bias) is necessary and how much final inflation (stronger cost bias) is allowable at what improvement granularity. We decided to keep single- and multi-part `Trajectory Scenarios` in the same `FlightPhase` set, although the optimization has to be completed for each individual part before proceeding to the next. Subsequent POIs are generally closer to each other, resulting in a shorter planning time, but this decision could also encourage a lower overall `TrajectoryQuality` to complete all parts and remain satisfactory. For the `ManagedTreePlanner` we are additionally interested in more appropriate values for the `Sampling` distribution, the connection `Strategy`, the maximum extension distance `epsilon`, the maximum number of iterations, the goal bias, and the size of the sampling neighborhood. As can be seen in Listings 11.1 and 11.2,

we have already constrained the parameter configuration space to reasonable values within our flight test area (e.g., epsilon) and according to results obtained from previous evaluations (e.g., maximum iterations and sampling neighborhood). Notice that the parameter configuration space could have been further constrained to speed up the automatic evaluation (e.g., integer instead of real values for the `ManagedGridPlanner` quality improvement).

Several training runs of 1000 evaluations for each `FlightPhase` surprisingly resulted in very different SMAC parameter configurations. Since SMAC combines random sampling with local searches within the neighborhood of promising configurations, slightly different result were expected (especially for parameters of type real and for non-deterministic algorithms). However, some training runs for the same `FlightPhase` resulted in very different incumbent configurations. One example for each `ManagedPlanner` and `FlightPhase` combination is shown in Tables 11.42, 11.43, 11.44, and 11.45. First of all, and without looking at the resulting configurations, we noticed that the best-on-average configuration did not consider the challenging `Scenarios` of type 3 (multi-part and underestimating heuristic) in all cases. In fact, an unsatisfactory result and a normalized `TrajectoryQuality` of 0 are predicted for `Scenarios` of type 3 for some of the incumbent configurations and, therefore, in favor of the less challenging `Scenarios` allowing for longer improvements. During the training runs, which can be monitored in our SAFCS UI, we have observed that configurations were tested that produced satisfactory results in these challenging `Scenarios` as well as in the less challenging ones (e.g. using a reduced final quality, higher goal bias, or smaller maximum number of iterations). However, these configurations were apparently disregarded and lost the challenge against the established or new incumbents. Presently, we cannot explain why this has happened and we have not yet investigated if different SMAC versions (2 versus 3) and implementations (Java versus Python) deliver the same results.

Nevertheless, we will attempt an interpretation of the final incumbent configurations. For the `ManagedGridPlanner` it can be seen that the minimum and maximum quality are much closer spaced for the *cruise* and *transition* `FlightPhases` compared to the *terminal* and *urgency* `FlightPhases`. Although the *urgency* `FlightPhase` covers all levels of `Difficulty` at a higher `Severity`, the *cruise* and *transition* `FlightPhases` are less difficult. Less difficult `FlightPhases` generally would require less deliberation and may already deliver optimal results based on an inflated heuristic. 1 to 10 improvement steps appear to be a reasonable choice in this case. An initially high inflation in more

Table 11.42: Cruise MTP SMAC Tunings

(a) Cruise MGP SMAC Tuning

Property	Value
costPolicy	AVERAGE
riskPolicy	SAFETY
minimumQuality	10.1266…
maximumQuality	5.2350…
qualityImprovement	0.8225…
significantChange	0.5

(b) Cruise MTP SMAC Tuning

Property	Value
costPolicy	AVERAGE
riskPolicy	SAFETY
maximumIterations	192
bias	63
epsilon	559.4535…
neighborLimit	1
goalThreshold	2.5
extension	FEASIBLE
strategy	CONNECT
sampling	GAUSSIAN
minimumQuality	0.1390…
maximumQuality	0.9326…
qualityImprovement	0.0585…
significantChange	0.5

Table 11.43: Transition MTP SMAC Tunings

(a) Transition MGP SMAC Tuning

Property	Value
costPolicy	AVERAGE
riskPolicy	SAFETY
minimumQuality	7.1246…
maximumQuality	5.5154…
qualityImprovement	1.5729…
significantChange	0.5

(b) Transition MTP SMAC Tuning

Property	Value
costPolicy	AVERAGE
riskPolicy	SAFETY
maximumIterations	88
bias	91
epsilon	417.7651…
neighborLimit	9
goalThreshold	2.5
extension	FEASIBLE
strategy	CONNECT
sampling	ELLIPSOIDAL
minimumQuality	0.6966…
maximumQuality	0.8794…
qualityImprovement	0.0018…
significantChange	0.5

difficult FlightPhases would make sense to find a sub-optimal solution more quickly and become satisfactory regardless of the achieved optimality. The much lower inflation (close to optimal) in both the *terminal* and *urgency* FlightPhases then may be required to achieve the best possible solution within the cut-off limit. However, the improvement granularity is very different (i.e., 75.6324… versus 2.8306…) and confusing

Table 11.44: Terminal MTP SMAC Tunings

(a) Terminal MGP SMAC Tuning

Property	Value
costPolicy	AVERAGE
riskPolicy	SAFETY
minimumQuality	77.3827…
maximumQuality	1.1033…
qualityImprovement	75.6324…
significantChange	0.5

(b) Terminal MTP SMAC Tuning

Property	Value
costPolicy	AVERAGE
riskPolicy	SAFETY
maximumIterations	68
bias	21
epsilon	456.2318…
neighborLimit	5
goalThreshold	2.5
extension	FEASIBLE
strategy	EXTEND
sampling	UNIFORM
minimumQuality	0.2533…
maximumQuality	0.6918…
qualityImprovement	0.0224…
significantChange	0.5

Table 11.45: Urgency MTP SMAC Tunings

(a) Urgency MGP SMAC Tuning

Property	Value
costPolicy	AVERAGE
riskPolicy	SAFETY
minimumQuality	98.3378…
maximumQuality	1.1143…
qualityImprovement	2.8306…
significantChange	0.5

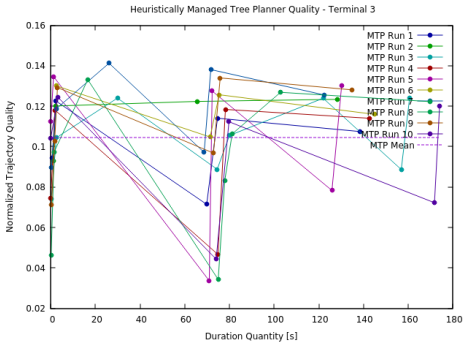
(b) Urgency MTP SMAC Tuning

Property	Value
costPolicy	AVERAGE
riskPolicy	SAFETY
maximumIterations	127
bias	16
epsilon	734.7491…
neighborLimit	7
goalThreshold	2.5
extension	FEASIBLE
strategy	CONNECT
sampling	UNIFORM
minimumQuality	0.1503…
maximumQuality	0.9356…
qualityImprovement	0.1370…
significantChange	0.5

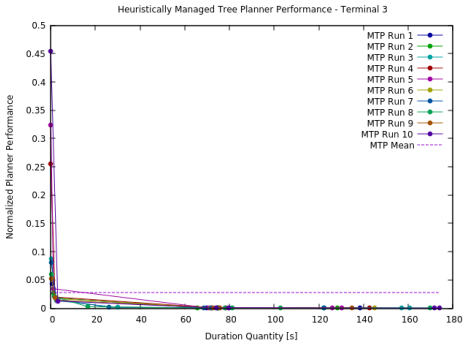
in these cases. A more aggressive improvement (75.6324…) would make more sense to find the more optimal solutions more quickly than a cautious one (2.8306…). We are presently unable to explain this difference. It is notably these two incumbent configurations which result in a predicted normalized `TrajectoryQuality` of 0 for their respective evaluation `Scenario` 3 and obviously due to their high final optimality requirement.

The most notable differences between the HeuristicPlannerTuning and the SmacPlannerTuning for the ManagedTreePlanner in all FlightPhases are the maximum number of iterations and the maximum extension distance epsilon. Both values indicate that a high normalized TrajectoryQuality can already be achieved using more distant samples within a much shorter DurationQuantity. If true, the HeuristicPlannerTuning could be improved significantly. Presently, a 5% POI distance is selected for epsilon together with a maximum number of iterations that ranges from 500 to 2000 depending on the determined FlightPhase. As a result, epsilon could potentially be increased to 50..100% POI distance while using only several hundred iterations instead of several thousand. The goal bias of the SmacPlannerTuning is also much higher compared to the HeuristicPlannerTuning. It is very high for the less difficult *cruise* (63%) and *transition* (91%) FlightPhases compared to the more difficult *terminal* (21%) and more severe *urgency* (16%) FlightPhases. However, we cannot explain the reason for the generally very high goal bias and the apparently arbitrary values as other evaluations (LaValle [LaV06], Rosa [dCR19]) have shown the requirement for much lower biases. Similar statements have to be made about the neighborhood size and the quality improvement parameters. A more difficult Scenario would intuitively require a larger neighborhood and increased cost bias compared to a less difficult one. Again, we are unable to explain the apparently arbitrary values at the moment.

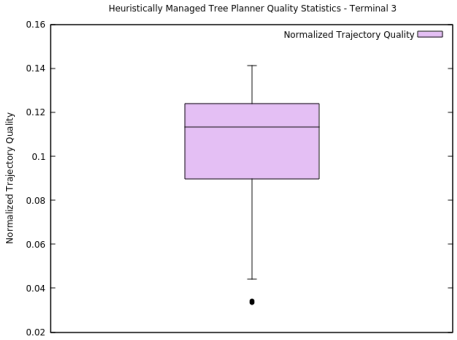
Comparing the HeuristicPlannerTuning with the SmacPlannerTuning for the challenging Terminal Scenario 3 of Section 11.5.3, one can clearly see in Figure 11.46 that a comparable final normalized TrajectoryQuality can be available within the 20 seconds cut-off time limit. The mean and median normalized TrajectoryQuality for 10 runs are very similar but achieved within a much shorter DurationQuantity for the SmacPlannerTuning. However, the overall variance becomes somewhat larger. After an inspection of the generated Reputation in the KnowledgeBase we expect to see similar results for the less challenging evaluation Scenarios that have been considered for the final incumbent. For the Scenarios of type 3, which were apparently not considered for the best-on-average configuration, we do not expect the same normalized TrajetoryQuality within the cut-off time limit, if a Trajectory can be computed at all.



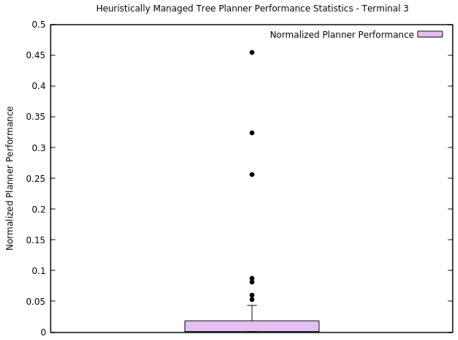
(a) H-AM MTP Quality



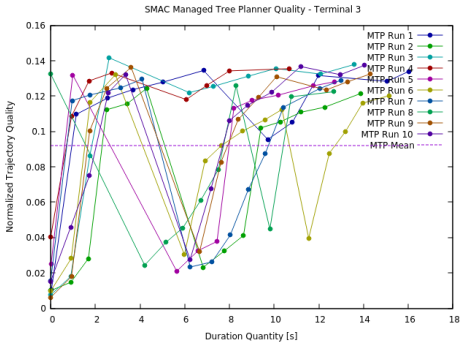
(b) H-AM MTP Performance



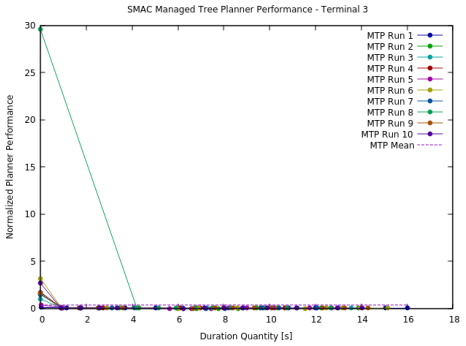
(c) H-AM MTP Quality Statistics



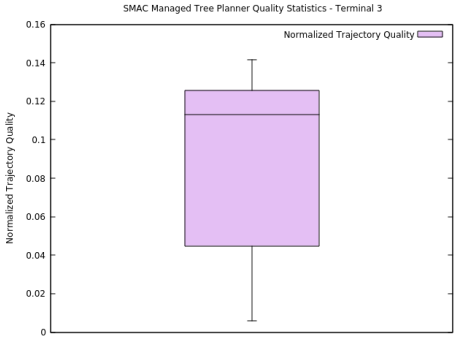
(d) H-AM MTP Performance Statistics



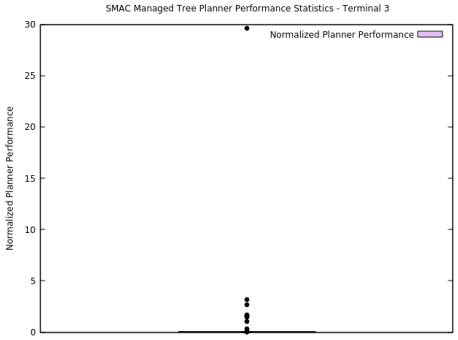
(e) SMAC-AM MTP Quality



(f) SMAC-AM MTP Performance



(g) SMAC-AM MTP Quality Statistics



(h) SMAC-AM MTP Performance Statistics

Figure 11.46: SMAC-AM Quality and Performance: Terminal 3

Therefore, the evaluation of the SmacAutonomicManager remains inconclusive, in particular, considering the predicted normalized TrajectoryQuality of 0 for challenging Scenarios (problem instances) of a common FlightPhase (problem set) as well as the apparently arbitrary values for other configuration parameters. Although we can derive potential improvements for the HeuristicAutonomicManager in terms of maximum iterations and extension distances, we conclude that different versions and implementations of SMAC should be compared first to confirm similar results.

11.10 Chapter Summary

In this chapter we evaluated the two currently supported AutonomicManagers of the SAFCS AM component, that is, the HeuristicAutonomicManager and the SmacAutonomicManager. We showed that a HeuristicPlannerTuning may create suitable Planner Specifications that result in feasible Trajectory solutions with a high TrajectoryQuality in Scenarios of very different Features. Without such an adaptation, feasible solutions may never be found in time as Features change. We, furthermore, performed an initial evaluation and interpretation of the obtained SmacPlannerTuning. However, the results remain inconclusive as we cannot explain why certain challenging Scenarios are not being considered to establish the final incumbent configuration. Additionally, some configuration parameters appear rather arbitrary and without any noticeable trend. We instead conclude that a more in-depth investigation would be required to confirm our results using different versions and implementations of SMAC.

Chapter 12

Vehicle Integration

12.1 Hardware

The UAV selected for our flight experiments is called *Tarot* and was designed, assembled and tested at the UVic CfAR. *Tarot* is a small quadrotor aircraft and is shown with its Pixhawk 4[1] flight controller in Figure 12.1.



(a) Tarot Unmanned Aerial Vehicle



(b) Pixhawk 4 Autopilot

Figure 12.1: Tarot UAV and Pixhawk 4 Autopilot

Pixhawk 4 is an open system flight controller and its architecture is well documented.[2] It can interface motors, sensors, and various payloads. Communication with manual controllers, GCSs, or smarter high-level systems, such as our SAFCS, can be established via a radio link. Alternatively, an airborne mission computer communicates with the Pixhawk 4 via a serial link and processes the radio link communications itself.

[1] https://docs.px4.io/master/en/flight_controller/pixhawk4.html
[2] https://docs.px4.io/master/en/concept/px4_systems_architecture.html

The Mavlink[3] protocol is used in both setups to receive flight controller telemetry as well as to send commands to affect its state and access its mission items. Several dedicated microservices facilitate topics such as missions and path planning. In fact, a path planning microservice[4], that is, trajectory interface, allows a companion mission computer to compute trajectories for obstacle avoidance. Furthermore, ADS-B, Flight Alarm (FLARM) and UAS Traffic Management (UTM) interfaces may enable air traffic avoidance.[5] In our case, and as discussed in Chapters 3 and 5, the SAFCS is envisioned to reside inside the SWIM cloud and uplink computed trajectories instead. Therefore, conceptually there is no difference in communication between a traditional GCS and our SAFCS for the intended experiments. Generally, an airborne companion system may, however, offer more flexible connectivity options, such as cellular or satellite networking abstracted by a Remote Procedure Call (RPC) API.

12.2 Software

The Pixhawk 4 flight controller facilitates the SIL simulation of customizable flight models using its simulator in combination with Gazebo[6]. We have simulated our UAS using the setup shown in Figures 12.2 and 12.3. The illustrations show the three main components of our setup, that is, the Pixhawk 4 flight controller software simulator being configured with a customized Gazebo flight model of our *Tarot* UAV, our SAFCS, and a monitoring QGroundControl GCS. The inclusion of a second monitoring and mission-proven GCS is intentional and shall serve as a safety backup for our early SAFCS prototype.

Not including the available *Tarot* airborne companion mission computer (Raspberry Pi[7]) used for other CfAR projects, leaves the communication particularly simple and reduces the number of potentially erroneous components for our initial experiments. In order to share the Mavlink traffic with the monitoring GCS, the simple Mavlink router MavP2P[8] is configured to forward all traffic as a User Datagram Protocol (UDP) client to QGroundControl and establish a Transmission Control Protocol (TCP) service for our SAFCS communications, respectively. Therefore, QGroundControl receives the entire

[3] https://mavlink.io/en/
[4] https://mavlink.io/en/services/trajectory.html
[5] https://docs.px4.io/master/en/advanced_features/
[6] https://docs.px4.io/master/en/simulation/gazebo.html
[7] https://www.raspberrypi.org/
[8] https://github.com/aler9/mavp2p

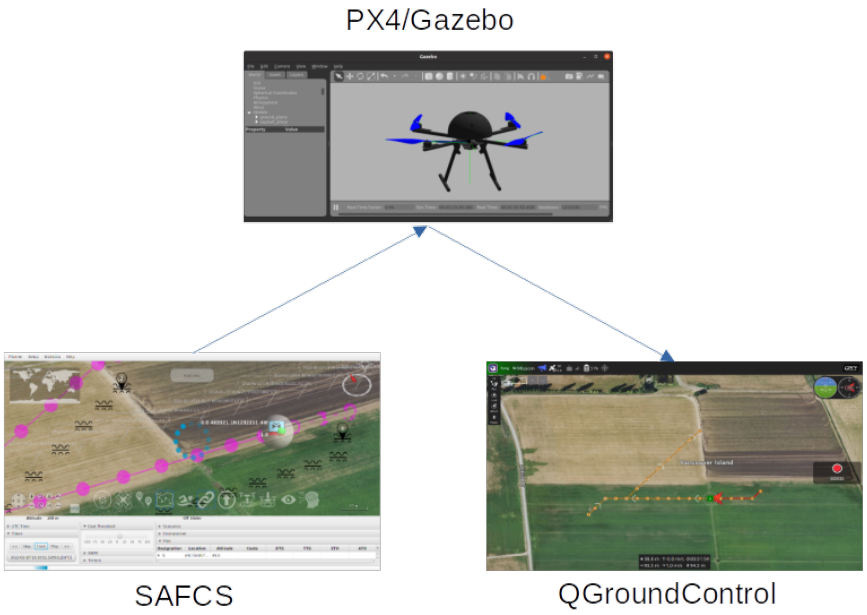


Figure 12.2: UAS Illustration

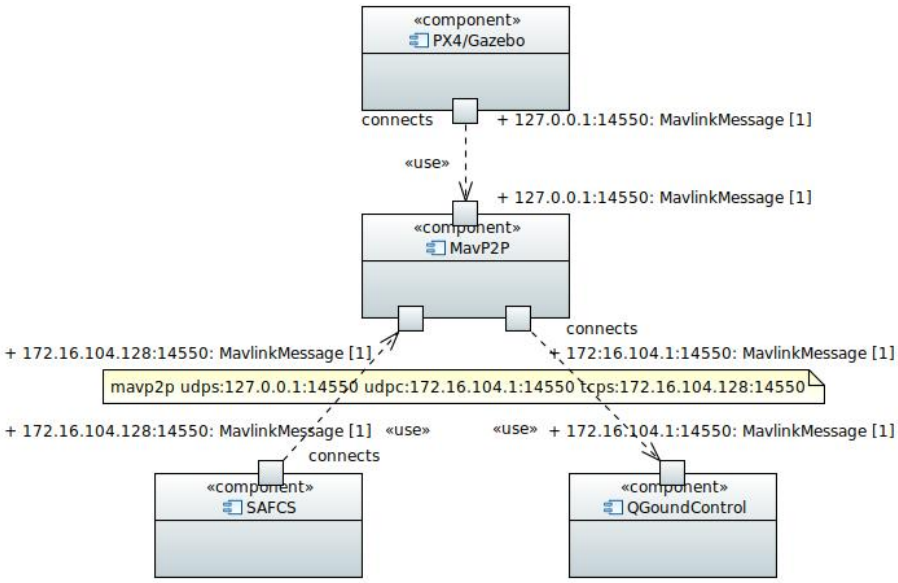


Figure 12.3: UAS Communication

downlink telemetry and becomes aware of any flight mode changes that are initiated by our SAFCS. Unfortunately, QGroundControl may only initiate but does not respond passively to the Mavlink mission microservice. Therefore, trajectory revisions, which are

uploaded by our SAFCS, are not automatically updated in QGroundControl. Instead, missions currently have to be downloaded manually by the GCS operator.

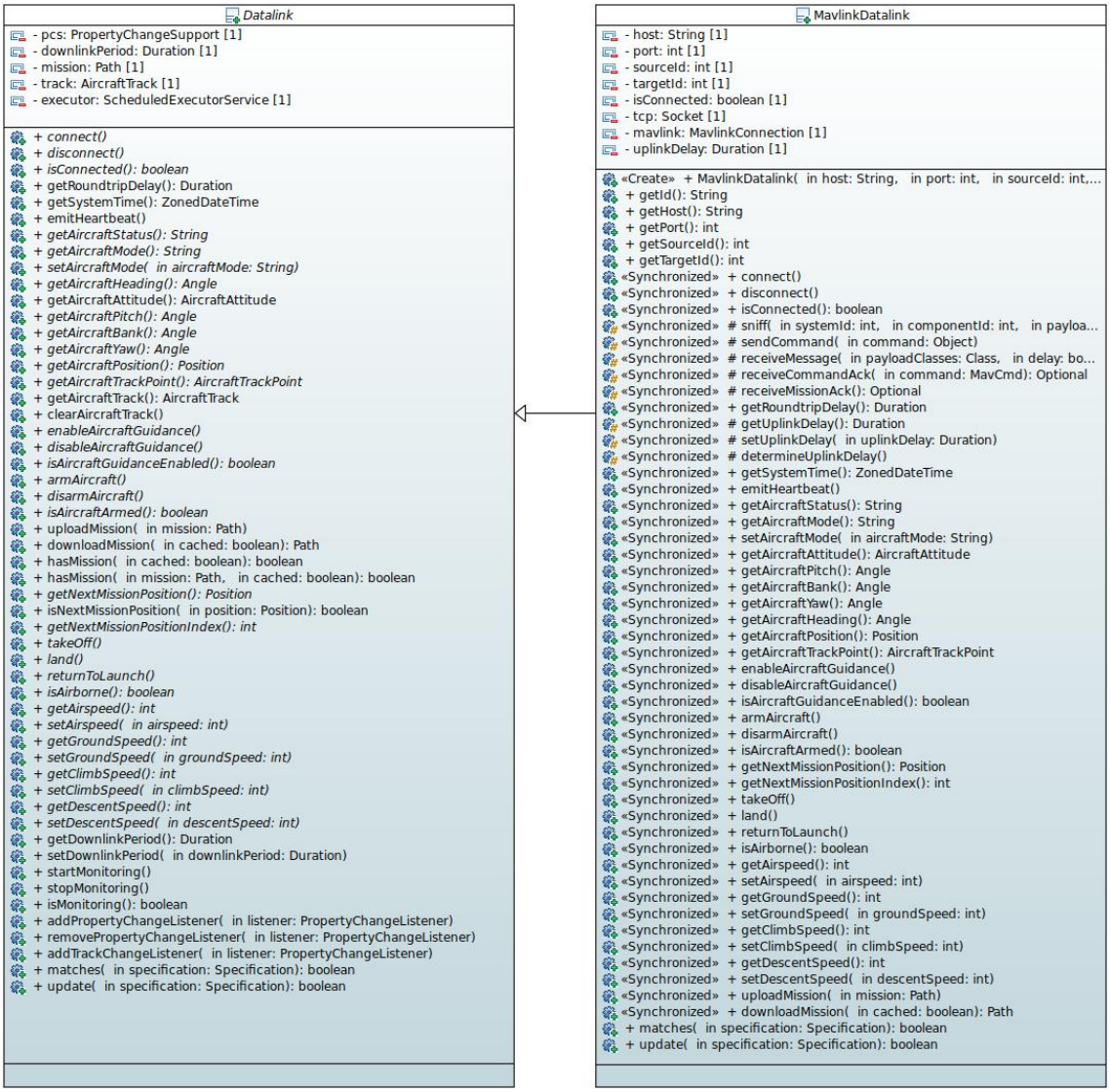


Figure 12.4: Mavlink Datalink

For the communication via the actual radio link, the MavP2P route is changed accordingly. For example, if the radio transceiver is available via the serial device node `/dev/ttyUSB0`, then the MavP2P route has to be changed from

```
mavp2p udps:127.0.0.1:14550 udpc:172.16.104.1:14550 tcps:172.16.104.128:14550
```

(which includes the locally forwarded UDP service) to

```
mavp2p serial:/dev/ttyUSB0:57600 udpc:172.16.104.1:14550 tcps:172.16.104.128:14550
```

Notice that this setup allows our SAFCS to connect to the forwarding Mavlink service from any host that can establish a route to the TCP service endpoint. However, the relatively slow radio link represents the bottleneck of this communication setup. As mentioned before, an airborne companion board may be equipped with higher-bandwidth communication devices. The `MavlinkDatalink` class and method signature shown in Figure 12.4 extends the abstract `Datalink` mentioned in Section 5.2.4 and wraps the required Mavlink communications for our experiments. More details about the `MavlinkDatalink`[9] can be found on our SAFCS core software repository.

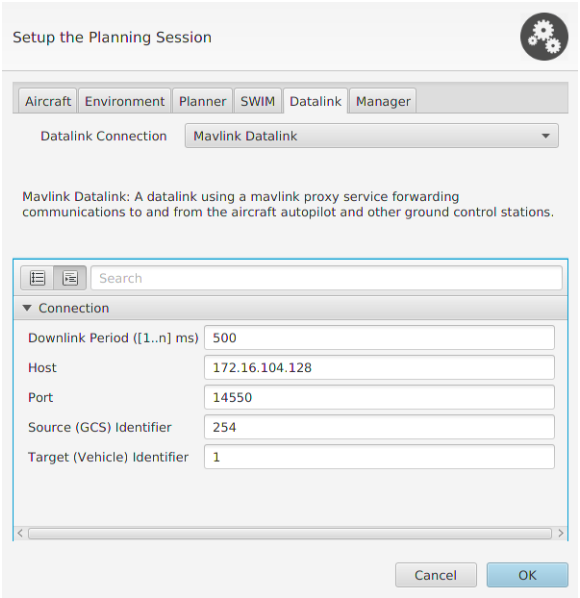


Figure 12.5: Mavlink Datalink Setup

The `MavlinkDatalink` setup of our SAFCS UI is shown in Figure 12.5 and lists the relevant `Specification Properties`, such as the TCP Mavlink service endpoint and communication identifiers.

[9] https://github.com/stephanheinemann/worldwind/blob/develop/src/main/java/com/cfar/swim/worldwind/connections/MavlinkDatalink.java

Chapter 13

Experiments

Due to scheduling and staffing difficulties, our first flight trial had to be postponed initially. Additionally, we experienced unexpected adverse weather on the day of the flight trial, soon after the first test flights were completed. Therefore, we were forced to take a longer than planned break in between the basic and more sophisticated SAFCS tests. Consequently, not all scheduled tests could be completed during this trial. Nevertheless, we validated the viability and stability of our SAFCS in terms of communication and planning. All performed tests were completed successfully and shall be described in this chapter.

13.1 Flight Test Crew

Following the established CfAR SOPs, we assembled a flight test crew, briefed the individual test flights and weather forecast, packed and installed the required equipment, surveyed the flight test area, and prepared operational checklists and test cards. The flight test crew consisted of

- the SAFCS operator and flight director,
- the backup GCS operator,
- the safety flight crew of two UAV pilots,
- the range safety officer, and
- the media observer crew.

The mentioned roles and responsibilities were fulfilled by the individuals shown in Figure 13.1.



(a) Safety-Pilots and Media Observers



(b) SAFCS and GCS Operators



(c) SAFCS Station and Operator



(d) Flight Test Van and Range Safety Officer

Figure 13.1: Flight Test Crew

It should be noted, that we established a control hierarchy and associated communication protocol for safety and redundancy reasons. This control hierarchy and SOPs would ensure a defined handover of control in case of an SAFCS, a backup GCS, or any higher AFCS mode failures. More precisely, the backup GCS would take control if the SAFCS in control fails. Furthermore, the safety flight crew would take control if the backup GCS in control fails or in case of any higher AFCS mode failures.

13.2 Flight Test Scenarios

Due to the limited time available and the practicality in terms of flight test van and initial aircraft take-off positions, we decided to perform the first SAFCS flight trial within scenarios derived from the one shown in Figure 13.2a with scenario-specific POIs. The flight test van was located inside the blue circle north and outside of the planning environment while the aircraft launchpad and safety flight crew were positioned within the

red circle. The flight test scenarios featured the same dimensions as explained in Section 11.2 where individual stationary or moving SWIM obstacles could be dynamically activated or deactivated as desired.



(a) Flight Test Scenario **(b)** Flight Test Aircraft

Figure 13.2: Flight Scenario and Aircraft

13.2.1 Setup and Limitations

In order to simulate changes in the environment on a larger scale as well as to reduce the involved risks in testing a new prototype, we decided to restrict all guided movement rates for the first flight trial to 1 meter per second. This restriction was applied for both lateral and vertical movements. The relatively slow progress had been used successfully in previous simulations and would ensure sufficient reaction time for our safety flight crew in case of system failures.

The minimum and default initial take-off altitudes were set to 35 and 40 meters ASL, respectively. For SAFCS online missions, the maximum allowable cruise cross-track error was set to 10 meters with a corresponding cruise timing error of 10 seconds (at 1 meter per second). If the aircraft were to leave these margins during the cruise portion, the active SAFCS online or managed planner would initiate re-planning and revise the 4D trajectory in flight. The take-off and landing windows were set to be within 5 meters laterally and 60 seconds to provide the SAFCS operator with sufficient time to confirm or deny take-off or landing clearance requests issued by the system. Additionally, all online mission ETDs were set to be more than 2 minutes into the future for the same reason.

13.2.2 Datalink

After having set up the flight test van, including the SAFCS and backup GCS, as well as having assured the aircraft serviceability, the first test confirmed the stability of the

datalink connection and an acceptable latency while on the ground. With the datalink enabled, the safety flight crew transported the aircraft to the take-off position. During this transport, both SAFCS and GCS operators monitored the aircraft downlink telemetry. In addition, the SAFCS operator tested the available aircraft view modes, including the top-down, follow and FPV (synthetic vision) modes. Once positioned on the launchpad at the take-off position, the SAFCS operator tested the uplink latency for several representative trajectories. The SAFCS and GCS operators, furthermore, practiced the established communication SOPs to synchronize the backup GCS with the uploaded SAFCS trajectories. All initial ground-based tests were satisfactory, and the trial proceeded with the airborne tests.

13.2.3 Take-Off, Landing, In-Flight Uplink and Returning

Flights 1 to 5 covered the important aspects of manual control, control hierarchy and handover SOPs as well as offline SAFCS missions with and without trajectory revisions while airborne. The first flight tested the serviceability of the aircraft while being controlled by the safety flight crew. Besides from integrating the safety flight crew from the very beginning and assuring readiness in case of failures, the first flight confirmed suitable datalink latencies via the deployed Mavlink router for both the SAFCS and GCS while airborne. It also served to confirm the correctness of the SAFCS elevation model using the FPV mode.

Flights 2 to 4 then focused on guided flights being controlled by either the SAFCS, the backup GCS or both practicing an in-flight handover. The associated flight trajectory is shown in Figure 13.3a and comprised a take-off, a short flight period to gain altitude, and a landing (return to launch) on the launchpad. The backup GCS controlled flight 2 entirely with the SAFCS monitoring. The SAFCS controlled flight 3 entirely with the backup GCS monitoring. Flight 4 was initiated by the SAFCS and control was handed over to the backup GCS practicing the established SOPs that would be used in case of an SAFCS failure.

Finally, flight 5, with an initial trajectory shown in Figure 13.3b, tested an SAFCS offline flight with an in-flight trajectory revision. This flight was the first to replace an existing mission for a guided aircraft in flight and served to validate the required basic functionality for online and managed flight operations with airborne trajectory revisions. The trajectory was manually revised half-way from the northern POI 0 to the



(a) Flights 2 to 4 **(b)** Flight 5



(c) Take-Off **(d)** Low-Level Flight

Figure 13.3: Basic Flight Tests

southern POI 1 and caused the aircraft to first return and then follow the new trajectory that would avoid a moving SWIM weather element during a past period of time.

All basic airborne tests were satisfactory and, following a forced adverse weather break (low cloud and rain showers), the trial proceeded with SAFCS online missions.

13.2.4 Online Planning and Dynamic SWIM Obstacles

Flights 6 and 7 covered SAFCS online planning with and without dynamic stationary SWIM obstacles. Besides revising trajectories as necessary, following the dynamic activation or deactivation of SWIM obstacles, the SAFCS online planners ensure the aircraft remains on track in both space and time.

Following a successful online mission without SWIM obstacles that confirmed planning accuracy, progress and tracking, the same POIs then served as the basis for an online flight with dynamic stationary SWIM obstacles, as shown in Figure 13.4.



(a) Online SWIM Flight (1)

(b) Online SWIM Flight (2)

(c) Online SWIM Flight (3)

(d) Online SWIM Flight (4)

Figure 13.4: Online SWIM Flight

As with all SAFCS online missions, the green datalink trackpoint ideally remains within the gray safety sphere, representing an online progress according to the plan, as shown in Figure 13.4b. If the aircraft leaves the planned track in space or time, then re-planning has to occur. Furthermore, if the environment changes, the existing trajectory is repaired as necessary based on the next active waypoint. The safety sphere is positioned at the first POI in this case and rejoins the aircraft trackpoint once it reaches the pick-up waypoint; Figures 13.4a and 13.4d show this situation following a dynamic SWIM obstacle update initiated as per Figure 13.4c.

Both test flights 6 and 7 were concluded satisfactorily and confirmed the previous simulations before the flight trial ended. However, the set turn anticipation configured for the aircraft autopilot led to slight differences between the planned and actual aircraft positions in time, particularly in tighter turns. Since the applied aircraft capabilities model is fairly simple at this point, these deviations were expected.

13.3 Future Flight Tests

Unfortunately, weather challenges and staff scheduling for other research activities prevented us from completing all planned flight tests of this otherwise very successful first flight trial. Therefore, the following flight tests shall be performed during a second flight trial given suitable weather conditions and staff availability.

13.3.1 Managed Planning and Dynamic SWIM Obstacles

Managed flight, being the highest level of SAFCS automation, shall be the focus of the future flight tests. Furthermore, dynamic stationary and moving obstacles shall be tested during both online and managed SAFCS missions with a stronger emphasis on grid planners and roadmap planners. These different topics can be covered in at least four additional test flights. We do, however, not expect results that differ significantly from the performed simulations as the first flight trial confirmed a stable and low latency datalink via the deployed Mavlink router with a backup GCS in the loop.

13.4 Lessons Learned

Although the time for the first flight trial was limited, some lessons were learned regarding the applied operational procedures as well as with respect to the usability of the SAFCS UI.

Operationally, each test flight was described to all crew members during the preflight preparations as well as on-site, referencing test cards that summarized the goals of the respective test and the tasks of the involved crew members. However, since all SWIM obstacles existed only in the virtual planning environment, the safety flight crew did not acquire the same level of situational awareness as the SAFCS and GCS operators. Therefore, expected attitude and altitude changes had to be communicated via radio to confirm the expected and normal operation of the SAFCS UAS.

This disadvantage could be addressed by streaming the SAFCS world view contents to an Augmented Reality (AR) system, such as the glasses depicted in Figure 13.5a. Pilots, or outside crew members in general, could wear these glasses to augment their reality with the presently computed SAFCS trajectories and the active SWIM obstacles,

[1] https://www.microsoft.com/en-us/hololens/hardware
[2] https://glass.aero



(a) Augmented Reality Glasses[1] **(b)** Augmented Reality Scenario[2]

Figure 13.5: Augmented Reality

and acquire an improved situational awareness similar to the one shown in Figure 13.5b. Notice that such a system could visually record each flight test for later assessments and demonstrations. Currently, the visual recordings of the aircraft in flight are separate from any logged data of the system. Although the SAFCS can be switched to an FPV mode, a visual recording with the actual aircraft would clearly be beneficial.

With respect to the usability of the SAFCS UI, some required interactions turned out to be cumbersome or unintuitive. Examples include the activity of datalink connections for individual scenarios and the timer for online and managed missions.

Each scenario is intentionally equipped with its own datalink instance. The idea is to eventually be able to guide several aircraft in both related or unrelated planning scenarios. However, only one active scenario can currently be controlled via the SAFCS UI while the non-active scenarios are kept in the background. If a non-active scenario is connected via its datalink, this situation is not immediately visible to the SAFCS operator. Consequently, several active datalink connections may unintentionally communicate with the same aircraft if not checked carefully. This situation has led to confusion at least once during the flight test.

Furthermore, the SAFCS timer currently has to be started manually while in online or managed flight operations. Since the ETD is usually set to a time instance in the near future, the timer cannot track the current time while setting up the online or managed mission. Once the SAFCS has started planning or managing a mission based on the selected ETD, the operator has to explicitly track the current time by setting the timer into the corresponding mode. During the flight tests, this manual action was skipped once, leading to confusion immediately after take-off. Since the world view was not tracking the present time, the displayed planned aircraft progress was not aligned with

the one of the actual aircraft. This situation has lead to the early abortion of one flight with the initial false assumption that the SAFCS was not working properly. Therefore, an improved SAFCS UI should ideally track real-time as a default once the aircraft has been cleared for take-off.

Part IV

Conclusions

Chapter 14

Summary

In this chapter we revisit and address all previously established research questions. In addition, we identify some of the challenges we had to tackle along the way and find solutions for. Finally, we summarize the most important contributions of our research.

14.1 Research Questions and Answers

> *RQ1*. Does a self-adaptive approach to decision making and trajectory planning improve the overall performance compared to a single (best on average) self-static approach using the quantifiable dependent variables mentioned in Section 1.3?

Without properly tuning the planning algorithms that have already been integrated into our SAFCS core, they may only achieve an unsatisfactory performance. In fact, an improper tuning may cause a planning algorithm to not deliver a solution at all. As a result, and if a solution can be computed, its deliberation time or trajectory costs are generally inferior to what could have been achieved otherwise. In particular, for dynamic environments with changing features during an online mission, a static approach may not cover all ensuing situations. Important features include the distance of POIs and their distribution within the planning environment, the density, location and movement of costly spaces within it and in time, and the operational policies that are being applied by OPS to assess the latter. As we have discussed in Chapter 9, an extensible set of features can be extracted from planning scenarios that leads to planner-dependent tunings. We have shown that tuning planning algorithms generally trades off optimality and the required deliberation time for ideally bounded sub-optimal solutions in more diffi-

cult scenarios. This approach is usually required in situations where applied heuristics are otherwise under-estimating significantly and the search space is not being reduced sufficiently. In general, planning algorithms may be configured to explore or exploit their environment differently depending on the extracted features. Performing this tuning adaptively, and based on the extracted features during an online mission, allows for timely and high-quality solutions in very different planning scenarios, as we have shown in Chapter 11. Furthermore, by encouraging competitions between the extensible algorithmic families realizing different planning paradigms in the SAFCS core, the disadvantages of a particular approach in certain scenarios may be filtered out successfully. Lastly, our set of autonomic managers follows a common workflow, but the level of sophistication of an individual instance may range from the application of basic heuristics to the collection and careful analysis of the applied tunings and resulting planner performances in certain scenarios. Therefore, current and future tools in the growing AI toolbox may further extend the capabilities of our SAFCS AM.

> *RQ2.* How do existing any-angle, anytime, dynamic, online algorithms have to be extended or modified to be included in a Smart Autoflight Control System?

We have discussed the most important aspects of the required algorithm modifications and extensions in Chapters 6, 7, and 8. The required extensions to all currently supported algorithms mainly relate to the environment they operate in. All SAFCS core planners operate in a 4D environment with spatially and temporally embedded SWIM obstacles, that is, in a 4D non-uniform cost environment. Furthermore, the assessment of the embedded costs depends on the operational cost and risk policies defined by OPS. This assessment may be very different for the different kinds of OPS, such as civil air transport, Search and Rescue (SAR), and military. We have decided to represent costs in space and time as discrete cost intervals that incur a certain advantage or penalty on the airborne agent when travelling a basic distance, that is, a step in the planning environment. This step can, for example, be the voxel passage in a planning grid, or a single extension or edge of a tree or roadmap in a planning continuum. More precisely, we associate a binary interval tree of cost intervals with each basic structure representing a step in our structured environments. The ETO at the start of such a step together with the ETE to complete the step transition allows us to efficiently look up the associated cost intervals. These cost intervals are then combined by our modified planners according to their operational cost and risk policies. To determine ETOs and ETEs for a planned

trajectory, our SAFCS core planners consider agent capabilities that may eventually be modelled to be dynamic and depend on the very environment the agent operates in.

With respect to the increasing functionality, we have implemented a class hierarchy within each of the supported planner families. Beyond the extensions described in the classic literature, such as any-angle, anytime, dynamic, and online, we have introduced managed planners in Chapter 9. Managed planners may be orchestrated by autonomic managers. They feature functional signatures to support tuning, team work, competition, performance measurement and reporting. Thus, managed planners are an essential extension to enable the MAPE-K cycle as they represent the manageable elements of the AE.

> *RQ3.* Which algorithms and tuning parameters are suited best for particular flight situations?

The SAFCS core presently contains the three planner families CGS, RRTs, and PRMs. The most sophisticated algorithm extensions within each of these families support any-angle, anytime, dynamic, and online trajectories. Clearly, since the SAFCS adaptively tunes managed planners during an online airborne mission, the supported planners have to possess the online capability and be able to deal with a moving agent. Our environments, furthermore, model partial or incomplete knowledge at the ETD. Hence, all managed planners should ideally incorporate dynamic changes efficiently without necessarily computing a trajectory from scratch. As real-time changes in the environment may limit the available deliberation times, a satisficing strategy is also desirable. Anytime planners sacrifice trajectory quality to satisfy the available duration quantity and, therefore, trade off optimality and unacceptable high deliberation times for bounded sub-optimality and acceptable deliberation times. If the available deliberation times are unknown in advance, then the process of finding sub-optimal solutions with a tighter sub-optimality bound simply continues as long as possible. This is similar to writing a draft document instead of the perfect final version, and continuously improving it until the deadline eventually prevents any further improvements. Lastly, any-angle algorithms, in particular those that consider the kinodynamic constraints of an agent that result from its capabilities in changing environments, including equipment failures, are very beneficial and can ensure the feasibility of the computed trajectory. The more of these aspects are considered by potential core planner candidates, the more realistic their trajectory solutions are likely to be. That being said, we have only implemented, modified, and extended a small number of the possible candidates. Other techniques

and paradigms, such as bio-inspired algorithms, should be studied carefully for their suitability.

In Chapter 9 we have explained our choices to categorize features of a planning scenario into flight phases. This approach is only one of many possible options and a simplified one at the same time. We have basically named different combinations of severity and difficulty to determine a suitable tuning using our domain knowledge about the available planners. Future core planners could operate differently, possess different tuning parameters and, therefore, may require entirely different feature categories. We should also keep in mind that in reality a flight phase can feature all levels of criticality and our naming conventions may be misleading. However, when operating under normal conditions, the complexities in the terminal, transition, or cruise flight phases are very different.

> *RQ4.* How can temporal SWIM data be integrated into a Smart Autoflight Control System to represent the associated costs in time while considering operational constraints?

In Chapter 5 we have introduced our SAFCS obstacle reporting framework. We integrate a SWIM connection to subscribe data from and publish data to the connected aviation cloud similar to other connections that we envision for our SAFCS concept. These connections eventually comprise of ATC and OPS feeds to, for example, modify the costs of embedded obstacles via clearances or change their assessment via modified operational cost and risk policies, respectively.

SWIM data features both a spatial and temporal component and can, therefore, be readily represented as cost intervals and collected in balanced binary cost interval trees. These trees are associated with spatial geometric extents, that are embedded into our planning environments via the mentioned obstacle reporting framework. As planners penetrate these spatial extents while performing basic steps (e.g., passing voxels, extensions, or edges), their existing or generated environment structures pick up the relevant costs of temporally overlapping cost intervals and consider them according to the applicable operational cost and risk policies.

We envision that the embedded costs and their assessment may also be modified following communications via the ATC and OPS connections. For example, AIXM airspaces may be associated with a dedicated responsible authority. This information could be extracted by a planner to request ATC clearances and, in turn, lower the cost of a potential trajectory.

> *RQ5.* How could the existing communication infrastructure, such as CPDLC and ACARS, be integrated into a Smart Autoflight Control System design as an effective means to realize a paradigm of *communication as sensing* to allowing a rational agent to interact with OPS and ATC to obtain policies, instructions or clearances and, hence, more options for automated planning?

Our SAFCS communication infrastructure is explained in Chapter 5. Although being an integral part of our SAFCS concept, ATC and OPS connections (or their simulations) are not yet implemented in the current SAFCS prototype. The communication is envisioned to be available to online planners and similar to the datalink communications with the airborne agent. Hence, online planners could send and receive CPDLC (or similar) messages to modify the costs in their planning environments. Similarly, ACARS messages could modify the specification of an online planner in terms of operational policies. We have explained how capable factories produce factory products, such as planners, according to specifications. These factory products can be updated with modified specifications. The operational cost and risk policies are part of these planner specifications and change how the planner assesses its environment. The same mechanism is already used by our SAFCS AM to tune managed planners following the extraction of scenario features. Therefore, these connections would blur any difference between machine or human rational actors. They will eventually allow us to study and better understand the *human in the loop* concept of our SAFCS.

> *RQ6.* What architectural design is suitable to support the dynamic extension and evolution of the Smart Autoflight Control System?

There are certainly many possible approaches to answer this question. Right from the beginning of our research we have envisioned the SAFCS to be located in a data center rather than aboard an aircraft. Realizing a smarter GCS, the concept focuses on the mission performance and considers the relevant stakeholders via its connections—the aircraft datalink being only one of them. Whether or not these connections communicate with human or machine rational actors, should not matter. For example, in our experiments we have actively guided an unmanned aircraft or its simulation. However, the computed trajectories could be made available to the crew of a manned aircraft and followed or disregarded very similar to the TASAR concept. We, furthermore, consider our SAFCS to be a meta-guidance system that is not necessarily required to communicate with the systems aboard an aircraft directly. However, such a communication

could be beneficial and instrument the sensors and avionics of the aircraft to feed more data into the SWIM cloud. For example, air data, such as temperatures, pressures and wind speeds, could be extracted. System data, such as engine status and performance, could be monitored and collected. Advantageous or critical situations could possibly be recognized and analyzed in the cloud and FIXM data amended accordingly. The connected stakeholders represented by human or machine rational actors could be informed if necessary and act as required. The future potential of such an infrastructure is undoubtedly very exciting.

Although we have equipped our SAFCS with the heavy-weight UI component described in Section 10, the functional signature of an SAFCS session can be made available via web services. Suitable web applications could address the different requirements of the connected human operators if required. Furthermore, as discussed in Chapter 5, the SAFCS core builds on factory products that are instantiated by available factories according to product specifications. Specifications are collected in registries and are managed by the SAFCS session. This approach allows us to quickly integrate new environments, aircraft, planners, connections, and autonomic managers, without changing the UI components. We imagine that available specifications and factory products are managed remotely in the future. This would most likely require a dedicated connection and interface to its stakeholder.

14.2 Challenges and Solutions

During our research for the development of our SAFCS prototype we have encountered many challenges that have led to modifications, trade-offs and simplifications of our original ideas. Some of these challenges and solutions shall be summarized here.

Many optimal planners, such as A* and RRT*, assume that local greediness leads to global optimality. While this assumption is correct in 3D given the required admissibility of applied heuristics, it does not necessarily hold in 4D. An earlier or later ETO may affect the costs in the environment significantly. An existing position could potentially be revisited (resulting in different waypoints at the same position), an orbit could delay the progress of a mission, or the aircraft could be slowed down or sped up to avoid costly spaces in time. To simplify these complexities we have elected to prevent our planners from revisiting positions at different times in a single-part trajectory. Currently, we do not support planned modifications in aircraft speed either, which includes holding or orbiting. As a result, our planners assume that the aircraft is in constant motion us-

ing a set speed according to its capabilities. While revisiting positions or delaying the progress of a mission is diametrically opposed to reducing operational costs, it may offer the only possible solution for a particular problem considering the operational cost and risk policies. While we argue that the ETD could be delayed in the first place to avoid an en-route delay, this does not hold for dynamic changes and incomplete knowledge about the environment.

The topic relates to the option of planning based on an ETA instead of an ETD. Some types of missions may require the agent to compute a trajectory and corresponding ETD to arrive at a certain POI at a particular ETA. While this backward planning is not more challenging in an environment featuring 3D obstacles and the preferred choice for agent-centered online planners, it does not translate easily into a 4D obstacle space-time. Planning from a goal POI toward the agent cannot be guaranteed to meet the moving agent in space and time. Again, modifying the planned speed of the agent or introducing POI slot times or, more generally, waypoint slots, could be a solution to this problem.

Waypoint slots or speed modifications could also address the consistency requirements during the repair and improvement of multi-part trajectories. Re-establishing consistency for the efficient repair of partial trajectories, particularly in 4D non-regular continuous environments, is very unlikely. Both the previous cost and ETO of an intermediate waypoint have to be confirmed before the remaining partial trajectory can be accepted. Otherwise, the propagation of under- followed by over-consistencies may become too costly compared to a trivial re-planning. POI ETOs without slots have, furthermore, forced us to implement a multi-part trajectory improvement strategy that improves the individual parts of a trajectory consecutively. Each part has to be improved to its maximum desired quality, to establish a corresponding fix POI ETO, before proceeding to the next. However, we can argue that the part closer to the agent is more time-critical for online planners and that cost modifications in 3D closer to start POIs are subject to the same problem.

The introduction of cost and risk policies has allowed us to acknowledge operators that assess the operational environment very differently. We are convinced that such policies should be established by any operation before a mission is planned. In reality these policies may be much more complex than just general threshold values; they may have to be defined by the operator itself. The policies relate to the common practice of clearly defining operational or personal minima beyond the set of rules that govern, for example, VFR and IFR operations. Consequently, for online planners operating in

environments with partial knowledge, agents may unexpectedly be exposed to unacceptable costs. We have already indicated that as a result trajectories may not be found without rising the risk policy temporarily and incorporating the concepts of alternates and unplanned landings. However, currently alternates and unplanned landings are not supported by our prototype.

Extracting features from a planning scenario in order to tune our managed planners accordingly, turned out to be one of the challenges to solve. Again, this is the result of a 4D planning environment with partial knowledge about the embedded costs. As obstacles are reported and embedded in our planning environments, features are re-extracted as a reaction. However, 4D obstacles may appear, disappear, overlap, change in size, and move in space over a period of time. This non-trivial problem had to be simplified significantly, and our features horizon model is, consequently, very limited. In fact, it can become a misleading sensor in situations of high volatility, very large or very small overlaps in space and time. However, we found our simplified features horizon model sufficient to perform initial evaluations and experiments. In the long-term, a more sophisticated sensor model would clearly be desirable though.

Lastly, our communication infrastructure including SWIM, ATC, and OPS connections has only been partially realized. Currently, a simulated SWIM connection can inject AIXM, IWXXM, and FIXM data files according to a specified frequency, quantity and probability. These data sets are, however, very limited in terms of supported elements and attributes. In general, supporting the SWIM protocols entirely would have been a massive undertaking by itself. Furthermore, as the access to the SWIM cloud was recently modified by the FAA, our SAFCS prototype can currently not connect to the live SWIM feed. With respect to our evaluation and experiments, this situation was not an issue. In fact, live data would most likely have to be scaled in space and time for this purpose. However, in the long-term, a live SWIM connection should be established again to perform simulations in larger planning environments.

Similarly, and although and integral part of our SAFCS concept, communication protocols such as CPDLC and ACARS have yet to be studied for their suitability. We envision simulated ATC and OPS stations to experiment and evaluate the integration of human rational actors in the loop with our SAFCS.

14.3 Contributions and Significance

Our research contributes to the ongoing efforts in understanding and developing smarter air transportation systems and autonomous trajectory-planning systems in particular. Our Smart Autoflight Control System infrastructure and similar systems aim at supporting both rational human and machine actors to gain operational, economical, strategic, and environmental advantages. They acknowledge and address the significance of using our resources more efficiently while minimizing the negative impact on our environment. The carbon footprint of the air transportation sector has become a significant challenge that needs to be addressed urgently. Additionally, it is a sector that is constantly growing and pushing the capacity of the available airspaces in terms of utilization, existing procedures, and human limitations.

Ubiquitously available data, modern software engineering paradigms, the available communication and cloud infrastructure, and the expanding AI toolbox have in recent years provided the necessary foundations to enable the required leaps and accelerate the development of systems and infrastructures such as our SAFCS. Our concept leverages these technological advances and relieves human actors while achieving the advantages mentioned before. Furthermore, it enables the research community to pursue similar research goals by introducing and evaluating additional planners, managers and connections.

(i) **Mission-Centered Smart Autoflight Control System Infrastructure**

Our presented SAFCS architecture describes a mission-centered infrastructure to address the significant challenges mentioned before and represents the main contribution of this dissertation. Our SAFCS encourages a considerable shift in how airborne agents are being guided. The aircraft itself is not considered the central element of our infrastructure—it is rather the efficient and safe execution of an airborne mission. As such, the aircraft and its crew, if any, are considered as relevant as OPS and ATC. Each of these actors is connected with and may communicate within our presented infrastructure concept.

(ii) **System-Wide Information Management Integration and Temporality Concept**

The SWIM cloud, which enables the sharing of data via its open aviation services and standardized protocols, is a major driver toward smarter applications, decision-support and autonomous systems in particular. It represents one of the

foundations of this dissertation and was not yet available a decade ago. We provided the SWIM community with XML bindings in order to facilitate processing SWIM data with Java software. We showed how temporal SWIM data can be embedded into our SAFCS by modifying existing dynamic planners and their environments while acknowledging the uncertainties during trajectory planning and subsequent revisions with respect to applicable costs and deliberation times enroute.

(iii) **Operational Policies and Constraints**

Ubiquitously available raw data, such as available via SWIM, is crucial to derive the necessary interpreted information for feeding smarter systems. However, the interpretation of raw data clearly depends on the operation that intends to exploit it. Civil air transport, SAR, and military operations clearly assess the same data differently. Therefore, we incorporated operational policies into the modified planners of our SAFCS core to support different OPS assessments. More precisely, these policies address how pertinent costs are taken into account and what risk is acceptable toward the conclusion of an airborne mission.

(iv) **Managed Planners**

Managed planners are another important contribution of this dissertation. We introduced managed planners, that can be tuned, are able to work in teams, compete with each other, and report their performance to their autonomic managers. Thus, managed planners are a necessary and important extension to the existing any-angle, anytime, dynamic, and online planners. The SAFCS core can be extended by managed planners that operate according to different paradigms. Currently, our SAFCS prototype supports managed planners of the CGS, RRT, and PRM families of algorithms. These families are still being extended, and the associated research by the AI community remains very active.

(v) **Feature Extraction, Planner Adaptation and Performance Reputation**

We created the SAFCS as an extensible self-adaptive infrastructure whose autonomic managers are able to tackle planning scenarios of varying features considering the applicable operational constraints. To achieve these goals, we introduced domain-specific sensors and actuators. We described a simplified features horizon model as well as the categorization of extracted features according to their criticality. Although the applied model or categories may be defined differently,

we are convinced that our approach to tune and select managed planners represents a necessary and important contribution. Collecting reputations of managed planners, that associate planner scenarios and applied tunings with the achieved performance, may serve as a long-term sensor for more sophisticated autonomic managers.

Our research may ideally motivate others and provide a platform to study technologies that help realizing the vision of self-separating and self-governed flight within the context of the Autonomous Flight Rules. Our SAFCS prototype can immediately be used for academic research projects. This research could address some of the future work items discussed in Chapter 15.

14.4 Chapter Summary

In this chapter we revisited our original research questions and answered them according to our findings. During our studies and implementation efforts we encountered numerous challenges, which forced us to find alternate solutions, take detours, or make concessions. We discussed some of the more prominent challenges that may have to be addressed in our future work. Finally, we highlighted our technological contributions and their significance to our society.

Chapter 15

Outlook and Future Work

15.1 Outlook

The current SAFCS UI, core and AM components have allowed us not just to answer some of our initial research questions but to provide an infrastructure that can be beneficial for future research addressing a variety of topics. These topics include the integration of existing and development of new `ManagedPlanners` as well as machine learning tools and techniques to realize future concrete SAFCS `AutonomicManagers`. The developed platform, furthermore, allows us to explore the disrupting impact and potential of *connected aviation* when communicating with rational actors—both human and machine—via the integrated `Connections` and `Communications`. It can hopefully serve as a research platform to address many questions still to be answered within the larger context of autonomous self-managed flight operations. The following sections summarize some of the potential next steps in the development of our SAFCS and may guide the way ahead.

15.2 Future Work

The aim of this section is to review and elaborate on some of the options and possible improvements that have been identified while developing our SAFCS and lead to a number of future work items and issues to be addressed more carefully. The majority of these options increase the level of sophistication of the applied planning models and would increase the flexibility during real-world airborne missions. They shall be kept in mind

for future extensions and particularly addressed when introducing planning algorithms of other families into the SAFCS core.

15.2.1 Environment Base Cost and Favorable Regions

Increasing the base cost of an `Environment` to allow for the embedding of negative SWIM `Obstacle` costs, as explained in Section 5.2.3, and, therefore, favorable regions has a significant impact on the `Performance` of `Planners`. Due to the generally higher base cost, that is, a cost factor of more than 1 for a step within the `Environment`, admissible and consistent heuristics based on the Euclidean distance generally underestimate the cost of the unexplored space. As a result, the search space may not be reduced significantly, and the search for an optimal solution may have to explore it in its entirety. The search may, therefore, take considerably longer, especially if the heuristic cannot remain inflated or the cost bias has to be increased, due to a higher `Difficulty` of the `Environment`. The same situation arises if larger spaces of a `Scenario` are affected by non-`RiskPolicy`-exceeding costs, such as for the `Scenarios` 3 in our evaluation of Chapter 11. However, since favorable regions (e.g., representing preferred wind conditions or airways) shall be considered our SAFCS, this topic should be reviewed and studied in greater detail.

15.2.2 Environment Tuning

As discussed in Chapter 9, `Environments` of the SAFCS core may be tuned by `AutonomicManagers` similar to `ManagedPlanners`. The flexible adaptation of hierarchy and resolution in `HierarchicalEnvironments` and `MultiResolutionEnvironments`, respectively, may significantly improve the `Performance` of the employed `Planner` for different levels of `Severity` and `Difficulty`. Additionally, the competition between `ManagedPlanners`, that are based on a discrete constraining `Environment` and other unconstrained ones, would be fairer. Hence, the `Environments` of `ManagedPlanners` should be managed accordingly and part of the actuated `Tuning`. Following a careful requirements analysis, similar interfaces, such as `ManagedEnvironment` and `EnvironmentTuning`, could potentially extend the existing architecture. The `Performance` of `EnvironmentTunings` would have to be combined with the associated `PlannerTuning` and persisted in the `Reputation` of the `KnowledgeBase`.

15.2.3 Environment Elevation Models

Although not explicitly covered in this dissertation, the SAFCS UI and core components have already been extended to support local elevation models. In fact, local high-resolution terrain was required to complete the evaluation of Chapter 11 using `Scenarios` within the UVic flight test area. Currently, high-resolution LIDAR-based elevation tiles, that have been made available by the Government of Canada,[1] can be imported into the SAFCS. Although loading these large files (e.g. several 100 Mb depending on the terrain) currently takes a significant amount of time while caching is already implemented, there is the option of automatically streaming higher-resolution terrain elevations on-demand from the corresponding WMS service.[2] The configuration of the SAFCS should be modified to use the relevant WMS service and the resulting performance of this streaming option should be evaluated.

15.2.4 Environment Air Data Intervals

Sections 5.2.3 and 6.1 described how costs within the SAFCS core `Environment` are integrated as binary interval trees—an efficient data structure to store and lookup relevant costs in time for a certain space voxel. These costs are accumulated according to an operational `CostPolicy` and `RiskPolicy` within the `computeCost` and `getStepCost` methods introduced in Listings 6.1 and 6.2, respectively.

A very similar approach could be taken to incorporate air data intervals into an `Environment` of the SAFCS core. These could include temperatures, pressures, humidity and wind data to allow similar calculations as supported by modern EFBs. Again binary interval trees associated with planning voxels, `Edges`, or other internal structures would be used to store these air data intervals and then looked up to enhance the currently static `Aircraft Capabilities`, as explained in Section 15.2.5, in order to determine more realistic passage durations and ETOs. The determined ETOs in line 5 of Listing 6.1 would then be closer to the performance data provided in the respective POHs of the real aircraft.

[1] https://open.canada.ca/data/en/dataset/957782bf-847c-4644-a757-e383c0057995

[2] https://www.nrcan.gc.ca/science-and-data/science-and-research/earth-sciences/geography/topographic-information/web-services/17216#g1

15.2.5 Dynamic Aircraft Capabilities

Section 5.2.2 explained how Aircraft are created as FactoryProducts by a capable AircraftFactory according to a Specification describing static Capabilities. The Capabilities can be configured ahead of the planning session but are static during the session giving constant values for performance data, such as cruise speed, climb and descent rates, and climb and descent angles.

In order to improve this model, Capabilities could contain performance functions instead of values. These performance functions would depend on the air data, that is, air density and wind, of embedded air data intervals, as explained in Section 15.2.4. The resulting performance would be closer to the Capabilities of the real aircraft operating within a certain Environment and the resulting Trajectory more realistic.

Another extension would be dynamic Capabilities based on received telemetry over a Datalink that could be incorporated during online planning. Actual aircraft performance and potential systems degradation or recovery could be employed to revise and improve a generated Trajectory. Aircraft system failures may, furthermore, influence the Obstacle assessment within an Environment. For example, a failed anti- or de-icing system would increase the cost in icing conditions. Similarly, a failed transponder would increase the cost of a transponder-mandatory airspace.

15.2.6 Aircraft Safety Space and Intersections

Currently, an Aircraft safety sphere whose radius is specified by OPS as part of the associated Capabilities is used to determine the Severity of planning Scenarios, as described in Section 9.2. However, the safety sphere model could be generalized to a safety ellipsoid in order to more accurately capture different longitudinal, lateral, and vertical speeds and rates. In combination with dynamic Aircraft Capabilities as mentioned in Section 15.2.5, the size and shape of the safety ellipsoid could change accordingly. Furthermore, the Aircraft safety ellipsoid could be employed for other relevant intersection tests, including terrain clearance and SWIM Obstacle intersections. Thus, in the online dynamic case, the required safety distances and corresponding Trajectory would have to be revised whenever the Capabilities of the Aircraft change.

15.2.7 Cost and Risk Policies

Section 6.1 detailed the cost computation for planning steps within an SAFCS `Environment`. When planning the passage of such an `Environment` space, the applicable costs in time are looked up in aggregated `CostIntervals` stored in binary interval trees and accumulated according to an operationally defined `CostPolicy` and `RiskPolicy`.

The applied accumulation function for costs is only one of many possible options. Currently, all time-relevant cost intervals during an `Environment` passage are considered even if only the smallest portion of a `CostInterval` is valid within the space to be traversed. This approach might be too conservative in some situations. If more than one `Environment` space affect the taken planning steps, a `CostPolicy` helps to adjust the accumulated costs for a more or less conservative assessment by minimizing, averaging, or maximizing the participating costs. Lastly, the applicable `RiskPolicy` limits the maximum acceptable local step cost through the `Environment`. However, the `RiskPolicy` does not encourage higher local costs in favor of earlier ETOs but merely adds flexibility if higher cost spaces need to be traversed. Therefore, ETOs are currently only used as tie-breakers if expansion costs are equal, leading to the lowest cost `Trajectory` primarily but featuring the earliest ETA secondarily.

Whether or not the `RiskPolicy` should be interpreted as a risk-seeking policy, without exceeding the operational `RiskPolicy` but in favor of earlier ETAs, could be an option and made available via the OPS interface. As a result, applied heuristics may remain inflated, or the search more distance-biased as necessary.

The `RiskPolicy` thresholds themselves are currently associated with fix enumeration literals. However, OPS might be interested in defining their own `RiskPolicy` literals and thresholds. This added flexibility could be supported via the SAFCS UI or core configuration files.

Finally, the SAFCS UI presently displays the most conservative maximizing `CostPolicy` for spaces in time that are being affected by several `CostIntervals`. This approach might be confusing for OPS monitoring a `Planner` operating under a different `CostPolicy`. Therefore, the applicable `CostPolicy` should be selectable to display an `Environment` in the SAFCS UI accordingly. Alternatively, the operational `CostPolicy` could be selected automatically in line with the current `Planner` or `AutonomicManager` `Specification`.

15.2.8 Planning Continuum: Shape, Resolution and Hierarchy

The currently supported `PlanningContinuum` is derived from a geometric box. Recall from Section 5.2.3 that vertices and `Edges` represent the internal structure of this `StructuredEnvironment`. However, a `PlanningContinuum` does not realize a `HierarchicalEnvironment`. `Edges` could be used to define children of their parent `PlanningContinuum`. For example, the vertices of an `Edge` could represent the focii of an ellipsoid, or the top and bottom centers of a capped cylinder. This approach to a hierarchical `PlanningContinuum`, in turn, raises the question about the shape of the (parent) `PlanningContinuum` in general. A geometric box created from a geographical `Sector` is obviously only one of the many options for the geometric extents of a `PlanningContinuum`.

The resolution of a `PlanningContinuum` as a `MultiResolutionEnvironment` is presently used to determine intersections of `Edges` with high-resolution terrain. It is not used to discretize the continuum. Combining the available sampling distributions with different sampling resolutions could be an interesting option to consider.

15.2.9 Planning Roadmap: IFR/VFR Air-Routes and Airways

`PlanningRoadmaps` and PRM `Planners` have been implemented[3] and evaluated (Ferreira [BF19]) but still need to be refactored and integrated into the current SAFCS core development branch. An interesting option in addition to the supported FADPRM with desired regions would be the import of the existing IFR and VFR air-route and airway network—possibly via AIXM. This network could form the basis of an otherwise dynamically created roadmap. Since air-route and airways offer terrain clearance and Air Traffic Services (ATSs), they could be considered preferable for AFR operations as well.

15.2.10 Flexible Waypoint Slots

Section 6.4 explained the consistency conditions for dynamic repairs of an established `Trajectory` in the SAFCS core. In addition to cost consistency, the ETO consistency requirement is essential for repaired `ADStarWaypoints` in a valid `Trajectory`. This ETO consistency is necessary for 4D planning since any dependent `Waypoint` costs have been established based on both space and time. However, this strong consistency con-

[3] https://github.com/stephanheinemann/worldwind/tree/feature/continuum-planner/src/main/java/com/cfar/swim/worldwind/ai/prm

dition leads to a number of drawbacks. First, consistency is even harder to establish, particularly in irregular SAFCS Environments, and may not justify the effort of a repair attempt. Second, it prevents backward online planning since, even if ETAs at goal Waypoints served as the basis for 4D planning, reaching the moving agent at a certain ETO and position is highly unlikely. Lastly, 4D Obstacle avoidance is restricted since revisiting Positions in a single-part Trajectory is disallowed to limit computational complexity, save resources (e.g., in terms of fuel, battery, and airspace congestion), and reduce the overall ETE. As a consequence, the agent may not be able to avoid Obstacles efficiently without changing cruising speeds or holding / loitering over a Position in a more challenging Environment.

Flexible Waypoint slots could potentially address the mentioned issues and increase flexibility during real-world missions. A flexible Waypoint slot would essentially be an operationally constrained ETO tolerance for consistency. For example, the operator could specify that ETOs have to be met within −10 minutes. With such a tolerance, a repair would be complete even if a dependent leg of a single- or multi-part Trajectory was established later than the new early ETO. It would cause the agent to hold or loiter at the flexible Waypoint before proceeding. The tolerance could also be specified in terms of available contingency fuel or endurance for this procedure. Hence, a slightly ETO over-consistent (early arrival) Waypoint would not be problematic. However, an ETO under-consistent (late arrival) Waypoint would not meet the planned conditions of the remaining original Trajectory.

Another option for the DynamicPlanner would be to adjust the cruising speed of the Aircraft within its Capabilities to meet a flexible or non-flexible Waypoint within the operationally constrained ETO tolerance, if any. The connecting step (leg) to or from this Waypoint could be used to speed-up or slow-down the Aircraft as required. Again, operationally constrained speed modifications would have to be specified and potentially requested via an established AtcConnection.

A very different approach to achieve Waypoint consistency could be the dynamic refinement of the planning Environment. As discussed in Section 5.2.3, the resolution of an Environment can be refined, possibly avoiding dynamic changes all together. The DynamicPlanner could apply a refinement policy depending on the size of the dynamic change. This, in turn, could be a tuning opportunity for the SAFCS AM.

POIs, being special Waypoints, may also be associated with deliberate actions and associated action-delays. These actions could be part of the airborne mission and may have to be completed before proceeding on the computed Trajectory. For example,

an aerial work mission may require the dispersal of fertilizer, or a surveillance mission the recording of images and video. Action- or special-purpose POIs may also identify in-flight alternates or suitable landing areas, as described in Section 15.2.17. Action-delays are, therefore, very similar to `Waypoint` slots and may be implemented as such.

15.2.11 Significant Change Criteria and Policies

Section 6.4 introduced details of the SAFCS core implementation of the AD* algorithm—one of the supported lifelong, anytime, dynamic planning algorithms that is extended with online and managed functionalities to become a `ManagedGridPlanner` at the disposal of the SAFCS AM. This algorithm attempts to repair invalid legs of an established `Trajectory` at a lower computational cost than planning from scratch. As indicated, the repair approach relies on `Waypoint` consistency and is more likely to succeed in regular `Environments`, and is particularly beneficial if `Trajectory` parts closer to the goal in a longer `Trajectory` are affected by dynamic changes. Listing 6.13 shows the currently implemented criteria to determine the likelihood of an efficient repair as well as the strategy for the modification of the inflation factor. Similarly, Section 7.7 presented the same idea for the ADRRT algorithm of the non-deterministic RRT family of `Planners`.

The assessment of whether or not a significant change, that does not justify a repair attempt, has occurred is based on a threshold percentage of affected `Waypoints` in an established `Trajectory`. It is therefore only a quantitative measure and not necessarily the most effective one. A more effective measure could be the proximity of the change to the start position. Furthermore, the fact that changes may occur that do not affect the established `Trajectory` but would lead to improvements due to the removal of previously existing costs, is not reflected in the established condition.

15.2.12 Adaptive Deflation

Section 6.3 mentions the incremental improvements and subsequent revisions of ϵ-sub-optimal solutions of an SAFCS `AnytimePlanner`, such as the `ARAStarPlanner` and `ADStarPlanner`. The deflation takes place as a tunable but fix decrease of the inflation factor, as shown in lines 53 to 59 of Listing 6.13, until a final deflation, that is, the maximum quality, is reached.

However, as mentioned by Likhachev, Ferguson, Gordon, Stentz, and Thrun [LFG+05, LFG+08], an adaptive approach could be taken that deflates the inflation factor ϵ to the actually achieved sub-optimality bound of the last found solution. There-

with, subsequent increments would be bounded to better solutions than the one already found. This minor modification would be independent of the tuned deflation constant and could lead to an accelerated progress toward the desired quality of a computed `Trajectory`.

15.2.13 Priority Queue Pruning

Sections 6.3 and 6.4 mentioned the use of backups of `Planner` contexts in between incremental improvement steps. When restoring backups, all inconsistent states are merged into the expandable priority queue. In case of the `ARAStarPlanner`, these can only be over-consistent states, whereas the more sophisticated `ADStarPlanner` allows for both over- and under-consistencies.

The SAFCS core implementation is currently not concerned with whether or not any of these restored inconsistent states cannot be part of an improved `Trajectory` based on their already established costs. However, as indicated by Likhachev, Ferguson, Gordon, Stentz, and Thrun [LFG+05, LFG+08], the priority queue of expandable states could potentially be pruned. States, that already feature a higher deflated cost estimate than the actual cost of the goal, that is,

$$f(s) > g(s_{goal})$$

cannot be part of an improved `Trajectory` when applying a consistent heuristic.

15.2.14 Backups versus Planner Instances

Section 6.4 comments on the fact that backups to store `Planner` contexts in between incremental `Trajectory` improvements might not be the best choice. A backup contains all relevant states and queues and, in particular for larger state spaces, might be computationally expensive to create and restore.

Another approach could potentially use separate new `Planner` instances for each incremental step by simply passing the relevant modified and filtered (e.g., adaptive deflation and pruning) states to the new instance. Since references to previously created `Planner` instances are not required, these can be garbage-collected immediately after their use.

15.2.15 Sampling Attempts and Sampling Sub-Space

Section 7.6 explains the strategy of progressively shifting from a distance to a cost bias in order to improve the quality of a computed `Trajectory`. Listing 7.14 shows the currently implemented sampling approach of the `ARRTreePlanner` that serves as the basis for its more sophisticated `ManagedTreePlanner`. Recall that samples are taken according to the selected `Sampling` distribution, that is, one of `UNIFORM`, `GAUSSIAN`, or `ELLIPSOIDAL`. However, since the applied composite heuristic, when shifting toward a stronger cost bias, essentially restricts the space of valid samples, an improved variant could only return samples that do not violate the currently established cost bound. A solution with this intention in mind has already been implemented on the corresponding feature branch[4] but still needs to be reviewed carefully.

Another improvement could increase the number of samples taken from the restricted sub-space. As the valid sampling space is reduced progressively, the likelihood of finding feasible non-`RiskPolicy`-exceeding `Edges` decreases in `Scenarios` featuring higher levels of `Difficulty` and `Severity`. Hence, a higher number of samples could lead to more successful searches while increasing the required runtime.

15.2.16 Nearest Neighbors

Sections 7.4 and 7.6 explained how the nearest neighbors of a new RRT sample are considered for the extension of the existing tree. The nearest metric is currently based on the normalized distance within the `PlanningContinuum` but could also consider the `Capabilities` of the planned `Aircraft`. For example, neighbors that require climbs or descents could be associated with a farther nearest metric compared to near level neighbors. Different metrics have already been implemented in the `PlanningContinuum` and can easily substitute the existing metric supplied to a higher-order function.

15.2.17 In-Flight Alternates and Precautionary Landings

For `OnlinePlanners` in a `DynamicEnvironment` an airborne `Datalink` source may be facing threats that exceed the previously established operational `RiskPolicy`. If these threats are perceived at the present in-flight position, then there is no other option but to boost the `RiskPolicy` temporarily in order to escape the threat.

[4] https://github.com/stephanheinemann/worldwind/tree/feature/continuum-planner/src/main/java/com/cfar/swim/worldwind/ai/rrt

However, in some situations the airborne agent may not be directly exposed to higher costs, but the `OnlinePlanner` may nevertheless not be able to establish a `Trajectory` that respects the operational `RiskPolicy`. This might be, for example, due to adverse weather at the destination or a closed destination aerodrome or airspace. In these cases, the mission goal has to be abandoned. Instead, the agent has to be directed to proceed to one of possible several prioritized pre-planned in-flight alternates. If these in-flight alternates cannot be reached for similar reasons, then an unplanned precautionary landing should be initiated.

Regardless of the concrete situation, the operator should be informed about any unplanned adjustments to the `RiskPolicy` and alterations regarding the mission goal. The SAFCS UI should, therefore, issue the corresponding in-flight dialogs to keep OPS in the (human) loop. A supported (simulated) `AtcConnection` could be employed to amend flight plans and obtain the necessary clearances, if required. Furthermore, the SAFCS UI should support the creation of special-purpose POIs, such as in-flight alternate and action POIs, as mentioned in Section 15.2.10.

15.2.18 Core Planners

The SAFCS core `Planners` represent an extensible set of planning families that can follow very different paradigms. We consider this diversity and the option of selecting from and tuning these different `Planners` as `ManagedPlanners` one of the major advantages of our SAFCS. Extensions allow us to study the suitability and performance of many different approaches and make the SAFCS more capable over time. Possible future extensions to consider include 3D FD* (Carsten, Ferguson and Stentz [CFS06]), Anya (Harabor [HG13]), Anytime Multi-Resolution Multi-Heuristic A* (AMRA*) (Saxena, Kusnur and Likhachev [SKL21]), lattice-based planners (Pivtoraiko and Kelly [PK08]), bio-inspired algorithms, such as the Genetic Algorthim (GA) described by Rathbun, Kragelund and Pongpunwattana [RDKP02], as well as the various available RRT extensions. Research in the area of motion planners is very active and hard to keep up with. We try to maintain a list of interesting paper references on our GitHub project site[5] under the *Ideas* column for future investigation.

[5] `https://github.com/stephanheinemann/worldwind/projects/1`

15.2.19 Features Horizon Model

Section 9.2 describes the Features horizon model serving as the short-term *sensor* model of our SAFCS AM in order to assess the Criticality of a managed planning Scenario in terms of its Difficulty and Severity. We mentioned the existing deficiencies of this model, that is, the existing *instrument error*, due to possible intersections of Obstacles in space and time. We, furthermore, emphasized the limitations of this model itself, in particular, for highly volatile Scenarios. In these cases, which can be easily constructed, the Features horizon may not capture the relevant Features to tune the ManagedPlanners successfully. Therefore, the Features horizon model should be revisited and potential improvements could include the more expensive computation of sensed Obstacle intersections.

Another question relates to the extracted Features of a Scenario themselves. The number of extracted Features and their level of abstraction may influence any algorithms using them. For example, Features may include fairly low-level values, such as distances and volumes. More abstract Features can then be derived without necessarily being directly visible to the mentioned algorithms. These more abstract Features may include ratios or arbitrarily complex functional derivations of the extracted low-level Features. Therefore, the effect of extracting high-level Features transparently versus deriving them in a more opaque fashion should be studied more carefully. These considerations, in turn, have an influence on the number of extracted Features and consequently the performance of any consuming algorithm.

15.2.20 Autonomic Managers

Similar to the ManagedPlanners of the SAFCS core, the set of concrete AutonomicManagers of the SAFCS AM and their applied PlannerTunings is extensible. The research community covering the areas of hyper-planning and automatic configuration is a very active one, and different approaches have been applied to not just configure but also select suitable algorithms from a portfolio. Therefore, the existing algorithms and frameworks should be studied more extensively. Possible candidates include the ones discussed in Section 3.3 and mentioned on the AutoML[6] website.

[6] https://www.automl.org/automl/

15.2.21 Aircraft Sensors and SWIM Publication

As explained in Section 3.1, the SWIM infrastructure allows for both publishing of and subscribing to aviation data using the open AIXM, FIXM, WXXM, AMXM and MMIXM standards. The SAFCS currently simulates only a very limited connection and static XML files without the option of publishing, that is, sharing, data. However, local aircraft sensors could sense, collect, and distribute air-data, such as temperature, pressure, humidity as well as upper winds (WXXM), and detected traffic (e.g., via means of visual, radar or ADS-B). Even the own revised `Trajectory` could be published as FIXM during the airborne mission. The collected data could be translated into the respective SWIM formats and extend the already existing data. This option would be particularly interesting in relation to multi-aircraft missions and leads to the broader idea of the SAFCS as an AE instance—one of many in the SAFCS cloud.

15.2.22 Distributed Multi-Agent SAFCS

The SAFCS AE presented in this dissertation should be understood as a single instance within the broader concept of a potentially large SAFCS cloud. Collaboration between the individual SAFCS instances could be realized via the `Connection` and `Communication` framework previously covered in Section 5.2.4 and include data sharing via SWIM. Therefore, the SAFCS could serve as a research platform to pursue a variety of research goals, including sensor-fusion and connected aviation.

15.2.23 Live and Simulated Connections

Presently, and following an FAA interface change, all SAFCS `Connections` are simulated and simplified as required to only trigger the associated functionality in the core. Although large scale SWIM data is not really usable for local flight tests unless downscaled and replayed at a faster pace, the real `SwimConnection` should be established again[7] and the available data formats supported more completely to simulate flights with a `SimulatedDatalink`. Similarly, with respect to ACARS and CPDLC, only simulated and simplified `OpsConnections` and `AtcConnections`, respectively, make any sense. These `Connections` and associated `Communications` could help to study the *human in the loop* concept and realize a *communication as sensing* paradigm to modify the `Features` of a planning `Scenario` through human interaction. For example, AIXM

[7] https://www.faa.gov/air_traffic/technology/swim/products/get_connected/

airspaces may require clearances for a computed Trajectory. The costs associated with these airspaces could be lowered following a Communication via the established AtcConnection. The airsapce, which is represented by an embedded Obstacle in the Environment, would aggregate the data required by an OnlinePlanner to establish the corresponding AtcConnection with the responsible ATC unit.

15.2.24 Physics Engine and Models

Due to the limitations of the NASA WorldWind SDK with respect to collision checking, that is, precise intersection tests of rigid bodies in a Scenario, we decided to integrate the JBullet[8] physics engine into our SAFCS. Currently, only the collision detector of JBullet is invoked to test for intersections of differently shaped Obstacles with the internal structures (e.g., grid voxels and Edges) of our planning Environments. However, the potential of having combined a full-featured physics engine with our SAFCS has not been fully explored. There might be actual use cases for rigid body dynamics, in particular, for simulated data sources.

15.2.25 Statistics and Knowledge Base Inspection

Presently, the KnowledgeBase and its contained Reputation can only be inspected programmatically. In fact, while performing the evaluation of our SAFCS, as described in Chapter 11, JUnit[9] tests were used to extract data and transform the collected TrajectoryQuality and PlannerPerformance into gnuplot[10] scripts. It would be very beneficial to be able to inspect, filter and analyze the currently available KnowledgeBase via the SAFCS UI itself. The JavaFX chart package[11] may provide all necessary elements for visualization of related statistics.

15.3 Chapter Summary

In this chapter we explained some of the possible future work items addressing both improvements and extensions to our existing SAFCS. There is no particular order or priority required when pursuing the remaining goals, but it should be emphasized that

[8] http://jbullet.advel.cz
[9] https://junit.org
[10] http://www.gnuplot.info/
[11] https://openjfx.io/javadoc/13/javafx.controls/javafx/scene/chart/package-summary.html

some solutions may have a considerable impact on others. For example, in-flight alternates and action POIs are an important extension that affect all existing and future `ManagedPlanners`. Therefore, to minimize future refactoring efforts, some topics should probably be addressed rather earlier than later.

References[12]

[BBS95] A. G. Barto, S. J. Bradtke, and S. P. Singh. Learning to act using real-time dynamic programming. *Artif. Intell.*, 72(1-2):81–138, 1995. 42

[BBvEB17] R. Burger, M. Bharatheesha, M. van Eert, and R. Babuska. Automated tuning and configuration of path planning algorithms. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, may 2017. 69

[BCKO08] M. d. Berg, O. Cheong, M. v. Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag TELOS, Santa Clara, CA, USA, 3rd ed. edition, 2008. 13

[BF19] H. M. Botelho Ferreira. Probabilistic roadmaps for aircraft 4d path planning. Master's thesis, Técnico Lisboa, University of Victoria, Av. Rovisco Pais 1, 1049-001 Lisboa, Portugal, 2019. 234, 257, 372

[BK00] R. Bohlin and L. E. Kavraki. Path planning using lazy PRM. In *Proceedings of the 2000 IEEE International Conference on Robotics and Automation, ICRA 2000, April 24-28, 2000, San Francisco, CA, USA* [IEE00], pages 521–528. 31

[BKH13] K. Belghith, F. Kabanza, and L. Hartman. Randomized path planning with preferences in highly complex dynamic environments. *Robotica*, 31(8):1195–1208, 2013. 33

[BKHN06] K. Belghith, F. Kabanza, L. Hartman, and R. Nkambou. Anytime dynamic path-planning with flexible probabilistic roadmaps. In *Proceedings of the 2006 IEEE International Conference on Robotics and Automation, ICRA 2006, May 15-19, 2006, Orlando, Florida, USA* [IEE06b], pages 2372–2377. 33

[12] The numbers at the end of each bibliography item are the backward references to the pages where it was cited.

[BW12] M. Ballin and D. Wing. Traffic aware strategic aircrew requests (TASAR). In *12th AIAA Aviation Technology, Integration, and Operations (ATIO) Conference and 14th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*. American Institute of Aeronautics and Astronautics, sep 2012. 58

[CFS06] J. Carsten, D. Ferguson, and A. Stentz. 3D field D: improved path planning and replanning in three dimensions. In *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2006, October 9-15, 2006, Beijing, China* [IEE06a], pages 3381–3386. 23, 121, 122, 377

[CGKM12] R. Calinescu, C. Ghezzi, M. Z. Kwiatkowska, and R. Mirandola. Self-adaptive software needs quantitative verification at runtime. *Commun. ACM*, 55(9):69–77, 2012. 50

[CLRS09] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009. 28

[Cre13] J. Creswell. *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches*. SAGE Publications, 2013. 6, 9

[CYB+18] J. Cano, Y. Yang, B. Bodin, V. Nagarajan, and M. O'Boyle. Automatic parameter tuning of motion planning algorithms. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, oct 2018. 70

[dCR19] M. D. da Cost Rosa. UAV 4D path planning using rapidly-exploring random trees. Master's thesis, Técnico Lisboa, University of Victoria, Av. Rovisco Pais 1, 1049-001 Lisboa, Portugal, 2019. 234, 257, 338

[dFGQ12] L. de Filippis, G. Guglieri, and F. Quagliotti. Path planning strategies for UAVS in 3D environments. *Journal of Intelligent and Robotic Systems*, 65(1-4):247–264, 2012. 48

[DM12] N. Dadkhah and B. Mettler. Survey of motion planning literature in the presence of uncertainty: Considerations for UAV guidance. *Journal of Intelligent and Robotic Systems*, 65(1-4):233–246, 2012. 45

[DO11] S. L. Devadoss and J. O'Rourke. *Discrete and Computational Geometry*. Princeton University Press, 2011. 13

[FKS06] D. Ferguson, N. Kalra, and A. Stentz. Replanning with RRTs. In *Proceedings of the 2006 IEEE International Conference on Robotics and Automation, ICRA 2006, May 15-19, 2006, Orlando, Florida, USA* [IEE06b], pages 1243–1248. 35, 36, 38, 159

[FLS05] D. Ferguson, M. Likhachev, and A. Stentz. A guide to heuristic-based path planning. *Proceedings of the international workshop on planning under uncertainty for autonomous systems, international conference on automated planning and scheduling (ICAPS)*, pages 9–18, 2005. 17, 18, 20

[FS05a] D. Ferguson and A. Stentz. The field D* algorithm for improved path planning and replanning in uniform and non-uniform cost environments. *Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, Tech. Rep. CMU-RI-TR-05-19*, 2005. 23, 24, 26

[FS05b] D. Ferguson and A. Stentz. Field D*: An interpolation-based path planner and replanner. In S. Thrun, R. A. Brooks, and H. F. Durrant-Whyte, editors, *Robotics Research: Results of the 12th International Symposium, ISRR 2005, October 12-15, 2005, San Francisco, CA, USA*, volume 28 of *Springer Tracts in Advanced Robotics*, pages 239–253. Springer, 2005. 23, 121

[FS06a] D. Ferguson and A. Stentz. Anytime RRTs. In *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2006, October 9-15, 2006, Beijing, China* [IEE06a], pages 5369–5375. 37, 39, 160

[FS06b] D. I. Ferguson and A. Stentz. Multi-resolution field D. In T. Arai, R. Pfeifer, T. R. Balch, and H. Yokoi, editors, *Intelligent Autonomous Systems 9 - IAS-9, Proceedings of the 9th International Conference on Intelligent Autonomous Systems, University of Tokyo, Tokyo, Japan, March 7-9, 2006*, pages 65–74. IOS Press, 2006. 24, 25, 122

[FS07] D. Ferguson and A. Stentz. Anytime, dynamic planning in high-dimensional search spaces. In *2007 IEEE International Conference on Robotics and Automation, ICRA 2007, 10-14 April 2007, Roma, Italy*, pages 1310–1315. IEEE, 2007. 40, 160

[GKM10] C. Goerzen, Z. Kong, and B. Mettler. A survey of motion planning algorithms from the perspective of autonomous UAV guidance. *Journal of Intelligent and Robotic Systems*, 57(1-4):65–100, 2010. 44

[GS02] G. Gigerenzer and R. Selten. Bounded rationality: The adaptive toolbox, 2002. 55, 120

[GVO13] M. Garcia, A. Viguria, and A. Ollero. Dynamic graph-search algorithm for global path planning in presence of hazardous weather. *Journal of Intelligent and Robotic Systems*, 69(1-4):285–295, 2013. 48

[HG13] D. D. Harabor and A. Grastien. An optimal any-angle pathfinding algorithm. In D. Borrajo, S. Kambhampati, A. Oddi, and S. Fratini, editors, *Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling, ICAPS 2013, Rome, Italy, June 10-14, 2013*. AAAI, 2013. 28, 121, 122, 377

[HHLB11] F. Hutter, H. H. Hoos, and K. Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Lecture Notes in Computer Science*, pages 507–523. Springer Berlin Heidelberg, 2011. 70, 72, 236, 413, 416

[HKLR02] D. Hsu, R. Kindel, J. Latombe, and S. M. Rock. Randomized kinodynamic motion planning with moving obstacles. *I. J. Robotic Res.*, 21(3):233–256, 2002. 45

[HMS14] S. Heinemann, H. A. Müller, and A. Suleman. Smart autoflight control systems. In *CASCON 2014*, page 400. IBM Canada Ltd., 2014. 4

[HMS18] S. Heinemann, H. A. Müller, and A. Suleman. Toward smarter autoflight control system infrastructure. *Journal of Aerospace Information Systems*, 15(6):353–365, jun 2018. 58, 82

[HZ01] E. A. Hansen and S. Zilberstein. LAO^*: A heuristic search algorithm that finds solutions with loops. *Artif. Intell.*, 129(1-2):35–62, 2001. 43

[IEE00] IEEE. *Proceedings of the 2000 IEEE International Conference on Robotics and Automation, ICRA 2000, April 24-28, 2000, San Francisco, CA, USA*. IEEE, 2000. 382, 386

[IEE06a] IEEE. *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2006, October 9-15, 2006, Beijing, China*. IEEE, 2006. 383, 384

[IEE06b] IEEE. *Proceedings of the 2006 IEEE International Conference on Robotics and Automation, ICRA 2006, May 15-19, 2006, Orlando, Florida, USA*. IEEE, 2006. 382, 384, 391

[JHvdB+11] L. Jaillet, J. Hoffman, J. van den Berg, P. Abbeel, J. M. Porta, and K. Y. Goldberg. EG-RRT: environment-guided random trees for kinodynamic motion planning with uncertainty and obstacles. In *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2011, San Francisco, CA, USA, September 25-30, 2011*, pages 2646–2652. IEEE, 2011. 45

[JL00] J. J. K. Jr. and S. M. LaValle. RRT-connect: An efficient approach to single-query path planning. In *Proceedings of the 2000 IEEE International Conference on Robotics and Automation, ICRA 2000, April 24-28, 2000, San Francisco, CA, USA* [IEE00], pages 995–1001. 34

[KC03] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, 2003. 49, 68, 82

[KF10] S. Karaman and E. Frazzoli. Incremental sampling-based algorithms for optimal motion planning. In *Robotics: Science and Systems VI*. Robotics: Science and Systems Foundation, jun 2010. 159

[KL02] S. Koenig and M. Likhachev. D* lite. In R. Dechter and R. S. Sutton, editors, *Proceedings of the Eighteenth National Conference on Artificial Intelligence and Fourteenth Conference on Innovative Applications of Artificial Intelligence, July 28 - August 1, 2002, Edmonton, Alberta, Canada.*, pages 476–483. AAAI Press / The MIT Press, 2002. 19, 120, 138

[KLC95] L. P. Kaelbling, M. L. Littman, and A. R. Cassandra. Partially observable markov decision processes for artificial intelligence. In I. Wachsmuth, C. Rollinger, and W. Brauer, editors, *KI-95: Advances in Artificial Intelligence, 19th Annual German Conference on Artificial Intelligence, Bielefeld, Germany, September 11-13, 1995, Proceedings*, volume 981 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 1995. 41

[KLC98] L. P. Kaelbling, M. L. Littman, and A. R. Cassandra. Planning and acting in partially observable stochastic domains. *Artif. Intell.*, 101(1-2):99–134, 1998. 41

[KLF04] S. Koenig, M. Likhachev, and D. Furcy. Lifelong planning A. *Artif. Intell.*, 155(1-2):93–146, 2004. 21, 138

[KSLO96] L. E. Kavraki, P. Svestka, J. Latombe, and M. H. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE T. Robotics and Automation*, 12(4):566–580, 1996. 29, 30

[KSPP10] S. Kopriva, D. Sislák, D. Pavlícek, and M. Pechoucek. Iterative accelerated A* path planning. In *Proceedings of the 49th IEEE Conference on Decision and Control, CDC 2010, December 15-17, 2010, Atlanta, Georgia, USA*, pages 1201–1206. IEEE, 2010. 28

[KVR+12] D. Karr, R. Vivona, D. Roscoe, S. Depascale, and D. Wing. Autonomous operations planner: A flexible platform for research in flight-deck support for airborne self-separation. In *12th AIAA Aviation Technology, Integration, and Operations (ATIO) Conference and 14th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*. American Institute of Aeronautics and Astronautics, sep 2012. 398

[Lad06] R. Laddaga. Self adaptive software problems and projects. *Software Evolvability, IEEE International Workshop on*, 0:3–10, 2006. 49

[Lav98] S. M. Lavalle. Rapidly-exploring random trees: A new tool for path planning. Technical report, Iowa State University, 1998. 159

[LaV06] S. M. LaValle. *Planning algorithms*. Cambridge University Press, 2006. 13, 338

[LBL04] S. M. LaValle, M. S. Branicky, and S. R. Lindemann. On the relationship between classical grid search and probabilistic roadmaps. *I. J. Robotic Res.*, 23(7-8):673–692, 2004. 17

[LFG+05] M. Likhachev, D. I. Ferguson, G. J. Gordon, A. Stentz, and S. Thrun. Anytime dynamic A*: An anytime, replanning algorithm. In S. Biundo, K. L. Myers, and K. Rajan, editors, *Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling (ICAPS 2005), June 5-10 2005, Monterey, California, USA*, pages 262–271. AAAI, 2005. 21, 22, 120, 133, 136, 138, 374, 375

[LFG+08] M. Likhachev, D. Ferguson, G. J. Gordon, A. Stentz, and S. Thrun. Anytime search in dynamic graphs. *Artif. Intell.*, 172(14):1613–1643, 2008. 21, 120, 138, 141, 374, 375

[LGT03] M. Likhachev, G. J. Gordon, and S. Thrun. ARA*: Anytime A* with provable bounds on sub-optimality. In S. Thrun, L. K. Saul, and B. Schölkopf, editors, *Advances in Neural Information Processing Systems 16 [Neural Information Processing Systems, NIPS 2003, December 8-13, 2003, Vancouver and Whistler, British Columbia, Canada]*, pages 767–774. MIT Press, 2003. 21, 120

[LK99] S. LaValle and J. Kuffner. Randomized kinodynamic planning. In *Proceedings 1999 IEEE International Conference on Robotics and Automation (Cat. No.99CH36288C)*. IEEE, 1999. 159

[LKJ01] S. M. LaValle and J. J. Kuffner Jr. Rapidly-exploring random trees: Progress and prospects. In K. Donald, B.; Lynch and e. Rus, D., editors, *Algorithmic and Computational Robotics: New Directions*, pages 293–308, 2001. 33, 34

[LS09] M. Likhachev and A. Stentz. Probabilistic planning with clear preferences on missing information. *Artif. Intell.*, 173(5-6):696–721, 2009. 42

[MCKK21] M. Moll, C. Chamzas, Z. Kingston, and L. E. Kavraki. HyperPlan: A framework for motion planning algorithm selection and parameter optimization. In *IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems*, 2021. To appear. 70

[MHW14] J. M. Maris, M. A. Haynes, and D. J. Wing. Traffic aware planner (TAP) flight evaluation. In *14th AIAA Aviation Technology, Integration, and Operations Conference*. American Institute of Aeronautics and Astronautics, jun 2014. 418

[MK12] Mausam and A. Kolobov. *Planning with Markov Decision Processes: An AI Perspective*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012. 41

[NK13] A. Nash and S. Koenig. Any-angle path planning. *AI Magazine*, 34(4):9, 2013. 25, 27

[NKL09] A. Nash, S. Koenig, and M. Likhachev. Incremental Phi*: Incremental any-angle path planning on grids. In C. Boutilier, editor, *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, pages 1824–1830, 2009. 27, 121

[NKT10] A. Nash, S. Koenig, and C. A. Tovey. Lazy Theta*: Any-angle path planning and path length analysis in 3D. In M. Fox and D. Poole, editors, *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*. AAAI Press, 2010. 25, 121

[OMT98] P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-based runtime software evolution. In K. Torii, K. Futatsugi, and R. A. Kemmerer, editors, *Forging New Links, Proceedings of the 1998 International Conference on Software Engineering, ICSE 98, Kyoto, Japan, April 19-25, 1998.*, pages 177–186. IEEE Computer Society, 1998. 51, 52

[Pea84] J. Pearl. *Heuristics - intelligent search strategies for computer problem solving*. Addison-Wesley series in artificial intelligence. Addison-Wesley, 1984. 19

[PK08] M. Pivtoraiko and A. Kelly. Differentially constrained motion replanning using state lattices with graduated fidelity. In *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, sep 2008. 377, 408

[RDKP02] D. Rathbun, P. D, S. Kragelund, and A. Pongpunwattana. An evolution based path planning algorithm for autonomous motion of a UAV through uncertain environments. In *Algebra Universalis*, pages 551–607, 2002. 45, 46, 68, 377

[RHS19] M. Rosa, S. Heinemann, and A. Suleman. Improvements for heuristically biasing RRT growth. unpublished, 2019. 176

[RN10] S. J. Russell and P. Norvig. *Artificial Intelligence - A Modern Approach (3. internat. ed.)*. Pearson Education, 2010. 13, 41

[RPPC08] S. Ross, J. Pineau, S. Paquet, and B. Chaib-draa. Online planning algorithms for POMDPs. *J. Artif. Intell. Res. (JAIR)*, 32:663–704, 2008. 42

[RS20] V. Rospotniuk and R. Small. Optimal any-angle pathfinding on a sphere. *arXiv preprint arXiv:2004.12781*, 2020. 29, 121, 122

[Seb14] Y. B. Sebbane. *Planning and Decision Making for Aerial Robots*. Springer, 2014. 45

[Sim82] H. A. Simon. *Models of bounded rationality: Empirically grounded economic reason*, volume 3. MIT press, 1982. 55, 120

[SKL21] D. M. Saxena, T. Kusnur, and M. Likhachev. AMRA*: Anytime multi-resolution multi-heuristic A*, 2021. 377, 397

[SKY08] X. Sun, S. Koenig, and W. Yeoh. Generalized adaptive A*. In L. Padgham, D. C. Parkes, J. P. Müller, and S. Parsons, editors, *7th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2008), Estoril, Portugal, May 12-16, 2008, Volume 1*, pages 469–476. IFAAMAS, 2008. 33

[SLB09] Y. Shoham and K. Leyton-Brown. *Multiagent Systems - Algorithmic, Game-Theoretic, and Logical Foundations*. Cambridge University Press, 2009. 52

[ST09] M. Salehie and L. Tahvildari. Self-adaptive software: Landscape and research challenges. *TAAS*, 4(2), 2009. 49, 50

[Ste93] A. Stentz. Optimal and efficient path planning for unknown and dynamic environments. *International Journal of Robotics and Automation*, 10:89–100, 1993. 138

[SVP09a] D. Sislák, P. Volf, and M. Pechoucek. Accelerated A* path planning. In C. Sierra, C. Castelfranchi, K. S. Decker, and J. S. Sichman, editors, *8th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2009), Budapest, Hungary, May 10-15, 2009, Volume 2*, pages 1133–1134. IFAAMAS, 2009. 28, 121

[ŠVP09b] D. Šišlák, P. Volf, and M. Pechoucek. Accelerated A* trajectory planning: Grid-based path planning comparison. *Proceedings of the 19th International Conference on Automated Planning & Scheduling (ICAPS)*, pages 74–81, 2009. 28

[TBF05] S. Thrun, W. Burgard, and D. Fox. *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press, 2005. 41

[UK15] T. Uras and S. Koenig. An empirical comparison of any-angle path-planning algorithms. In L. Lelis and R. Stern, editors, *Proceedings of the Eighth Annual Symposium on Combinatorial Search, SOCS 2015, 11-13 June 2015, Ein Gedi, the Dead Sea, Israel.*, pages 206–211. AAAI Press, 2015. 28

[US03] C. Urmson and R. Simmons. Approaches for heuristically biasing RRT growth. In *Proceedings 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003) (Cat. No.03CH37453)*. IEEE, 2003. 159, 175

[vdBFK06] J. P. van den Berg, D. Ferguson, and J. Kuffner. Anytime path planning and replanning in dynamic environments. In *Proceedings of the 2006 IEEE International Conference on Robotics and Automation, ICRA 2006, May 15-19, 2006, Orlando, Florida, USA* [IEE06b], pages 2366–2371. 32, 160

[vdBO05] J. P. van den Berg and M. H. Overmars. Roadmap-based motion planning in dynamic environments. *IEEE Transactions on Robotics*, 21(5):885–897, 2005. 31

[VKR06] R. Vivona, D. Karr, and D. Roscoe. Pattern-based genetic algorithm for airborne conflict resolution. In *AIAA Guidance, Navigation, and Control Conference and Exhibit*. American Institute of Aeronautics and Astronautics, jun 2006. 63

[WBB+19] D. J. Wing, K. A. Burke, K. Ballard, J. Henderson, and J. Woodward. Initial TASAR operations onboard Alaska Airlines. In *AIAA Aviation 2019 Forum*. American Institute of Aeronautics and Astronautics, jun 2019. 58, 61

[WC11] D. Wing and W. Cotton. For spacious skies: Self-separation with "autonomous flight rules" in US domestic airspace. In *11th AIAA Aviation Technology, Integration, and Operations (ATIO) Conference*. American Institute of Aeronautics and Astronautics, jun 2011. 61, 397

[WCM09] P. P. Wu, D. A. Campbell, and T. Merz. On-board multi-objective mission planning for unmanned aerial vehicles. In *IEEE Aerospace Conference*, Big Sky, Montana, April 2009. IEEE. 47

[WCM11] P. P. Wu, D. A. Campbell, and T. Merz. Multi-objective four-dimensional vehicle motion planning in large dynamic environments. *IEEE Transactions on Systems, Man, and Cybernetics, Part B*, 41(3):621–634, 2011. 48

[WCR+06] M. Wzorek, G. Conte, P. Rudol, T. Merz, S. Duranti, and P. Doherty. From Motion Planning to Control - A Navigation Framework for an Autonomous Unmanned Aerial Vehicle. In *Proceedings of the 21st Bristol UAV Systems Conference (UAVS)*, 2006. 46

[WD06] M. Wzorek and P. Doherty. Reconfigurable path planning for an autonomous unmanned aerial vehicle. In D. Long, S. F. Smith, D. Borrajo, and L. McCluskey, editors, *Proceedings of the Sixteenth International Conference on Automated Planning and Scheduling, ICAPS 2006, Cumbria, UK, June 6-10, 2006*, pages 438–441. AAAI, 2006. 47

[Win19] D. J. Wing. The TASAR project: Launching aviation on an optimized route toward aircraft autonomy. Technical Publication (TP) NASA/TP-2019-220432, NF1676L-34474, NASA Langley Research Center Hampton, VA, United States, 1 Nasa Dr, Hampton, VA 23666, United States, December 2019. Work of the US Gov. Public Use Permitted. 58, 59, 62, 63, 66

[WKD10] M. Wzorek, J. Kvarnström, and P. Doherty. Choosing path replanning strategies for unmanned aircraft systems. In R. I. Brafman, H. Geffner, J. Hoffmann, and H. A. Kautz, editors, *Proceedings of the 20th International Conference on Automated Planning and Scheduling, ICAPS 2010, Toronto, Ontario, Canada, May 12-16, 2010*, pages 193–200. AAAI, 2010. 47

[WLBW16] C. Wilkinson, J. Lynch, R. Bharadwaj, and K. Woodham. Verification of adaptive systems. Final Report DOT/FAA/TC-16/4, United States Federal Aviation Administration, FAA National Headquarters, 950 L'Enfant Plaza North, S.W., 950 L'Enfant Plaza, Washington, DC 20024, April 2016. The Federal Aviation Administration William J. Hughes Technical Center Aviation Research Division COR was Charles Kilgore. 74, 75, 76

[WvdB13] D. J. Webb and J. van den Berg. Kinodynamic RRT*: Asymptotically optimal motion planning for robots with linear dynamics. In *2013 IEEE International Conference on Robotics and Automation, Karlsruhe, Germany, May 6-10, 2013*, pages 5054–5061. IEEE, 2013. 45

[YBHS11] P. Yap, N. Burch, R. C. Holte, and J. Schaeffer. Block A*: Database-driven search with applications in any-angle path-planning. In W. Burgard and D. Roth, editors, *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2011, San Francisco, California, USA, August 7-11, 2011*. AAAI Press, 2011. 27, 121

[Zil96] S. Zilberstein. Using anytime algorithms in intelligent systems. *AI Magazine*, 17(3):73–83, 1996. 56

Part V

Appendices

Appendix A

Glossary

A.1 Acronyms

A* A* Algorithm

In computer science, A* is a computer algorithm that is widely used in path-finding and graph traversal, the process of plotting an efficiently traversable path between multiple points, called nodes. [1]

A*PS A* with Post Smoothing

A* with post smoothing applies a smoothing step after a grid-based path has been found with A*. [2]

AA* Accelerated A*

Accelerated A* is a fast any-angle variant of A*. [3]

AC Autonomic Context

An Autonomic Context interacts with an Autonomic Element operating within it.

ACARS Aircraft Communications Addressing and Reporting System

The Aircraft Communications Addressing and Reporting System is a digital datalink system for transmission of short messages between aircraft and ground stations via airband radio or satellite. [4]

[1] https://en.wikipedia.org/wiki/A*_search_algorithm
[2] https://en.wikipedia.org/wiki/A*_search_algorithm
[3] https://en.wikipedia.org/wiki/Any-angle_path_planning
[4] https://en.wikipedia.org/wiki/ACARS

AD* Anytime D*

Anytime D* is an anytime, dynamic variant of A*. [5]

ADC Air Data Computer

An air data computer is an essential avionics component found in modern aircraft. This computer, rather than individual instruments, can determine the calibrated airspeed, Mach number, altitude, and altitude trend data from an aircraft's pitot-static system. [6]

ADRRT Anytime Dynamic RRT

Anytime Dynamic RRT combines ARRT and DRRT creating an anytime, dynamic variant of RRT. [7]

ADS-B Automatic Dependent Surveillance-Broadcast

Automatic dependent surveillance-broadcast is a cooperative surveillance technology in which an aircraft determines its position via satellite navigation and periodically broadcasts it, enabling it to be tracked. The information can be received by air traffic control ground stations as a replacement for secondary radar. It can also be received by other aircraft to provide situational awareness and allow self separation. [8]

AGL Above Ground Level

In aviation, atmospheric sciences and broadcasting, a height above ground level is a height measured with respect to the underlying ground surface. [9]

AE Autonomic Element

An Autonomic Element comprises an Autonomic Manager and a Managed Element and interacts with its Autonomic Context.

AFCS Automatic Flight Control System

An autopilot is a system used to control the trajectory of a vehicle without constant 'hands-on' control by a human operator being required. [10]

[5] https://en.wikipedia.org/wiki/Anytime_algorithm
[6] https://en.wikipedia.org/wiki/Air_data_computer
[7] https://en.wikipedia.org/wiki/Rapidly_exploring_random_tree
[8] https://en.wikipedia.org/wiki/Automatic_Dependent_Surveillance-Broadcast
[9] https://en.wikipedia.org/wiki/Height_above_ground_level
[10] http://en.wikipedia.org/wiki/Autopilot

AFR Autonomous Flight Rules

Autonomous Flight Rules are proposed as a new set of operating regulations in which aircraft navigate on tracks of their choice while self-separating from traffic and weather. AFR would exist alongside IFR and VFR as one of three available flight options for any appropriately trained and qualified operator with the necessary certified equipment. (Wing and Cotton [WC11]

AI Artificial Intelligence

Artificial intelligence is the intelligence exhibited by machines or software. It is also the name of the academic field of study which studies how to create computers and computer software that are capable of intelligent behavior. [11]

AIRMET Airmen's Meteorological Information

Airmen's Meteorological Information, is a concise description of weather phenomena that are occurring or may occur along an air route that may affect aircraft safety. Compared to SIGMETs, AIRMETs cover less severe weather: moderate turbulence and icing, sustained surface winds of 30 knots or more, or widespread restricted visibility. [12]

AIXM Aeronautical Information Exchange Model

The Aeronautical Information Exchange Model is designed to enable the management and distribution of Aeronautical Information Services (AIS) data in digital format. AIXM is based on GML and is one of the GML Application Schemas which is applicable for the Aeronautical domain. It was developed by the United States FAA, the United States National Geospatial Intelligence Agency (NGA) and the European Organization for the Safety of Air Navigation (EUROCONTROL). The current version is AIXM 5.1.1. [13]

AM Autonomic Manager

An Autonomic Manager is part of an Autonomic Element and manages its Managed Element.

AMRA* Anytime Multi-Resolution Multi-Heuristic A*

The anytime multi-resolution multi-heuristic variant of A* is described by Saxena, Kusnur and Likhachev [SKL21].

[11] https://en.wikipedia.org/wiki/Artificial_intelligence
[12] https://en.wikipedia.org/wiki/AIRMET
[13] https://en.wikipedia.org/wiki/AIXM

AMXM Aerodrome Mapping Exchange Model

The Aerodrome Mapping Exchange Model is an open technical resource owned by EUROCAE / RTCA. The AMXM is provided in support of AMDB data exchange. Multiple user groups benefit from using AMDBs, such as pilots, controllers, aerodrome managers, and aerodrome emergency/security personnel. [14]

Anya Anya

Anya is the first and only known optimal any-angle grid-based path planning algorithm to date. [15]

AO* A* for Acyclic Graphs

AO* is an A* variant for acyclic graphs, such as and-or trees. [16]

AOP Autonomous Operations Planner

The Autonomous Operations Planner, developed by NASA, is a flexible and powerful prototype of a flight-deck automation system to support self-separation of aircraft. The AOP incorporates a variety of algorithms to detect and resolve conflicts between the trajectories of its own aircraft and traffic aircraft while meeting route constraints, such as required times of arrival, and avoiding airspace hazards, such as convective weather and restricted airspace. (Karr et al. [KVR+12]). TAP is an AOP derivative. [17]

API Application Programming Interface

An application programming interface is a connection between computers or between computer programs. It is a type of software interface, offering a service to other pieces of software. A document or standard that describes how to build such a connection or interface is called an API specification. A computer system that meets this standard is said to implement or expose an API. The term API may refer either to the specification or to the implementation. [18]

ARA* Anytime Repairing A*

Anytime Repairing A* is an anytime (unfocused) repairing variant of A*. [19]

[14] https://www.amxm.aero/
[15] https://en.wikipedia.org/wiki/Any-angle_path_planning
[16] https://en.wikipedia.org/wiki/And-or_tree
[17] https://ntrs.nasa.gov/citations/20120014583
[18] https://en.wikipedia.org/wiki/API
[19] https://en.wikipedia.org/wiki/Anytime_algorithm

AR Augmented Reality

Augmented reality is an interactive experience of a real-world environment where the objects that reside in the real world are enhanced by computer-generated perceptual information, sometimes across multiple sensory modalities, including visual, auditory, haptic, somatosensory and olfactory. [20]

ARINC Aeronautical Radio Incorporated

Aeronautical Radio Incorporated, established in 1929, was a major provider of transport communications and systems engineering solutions for eight industries: aviation, airports, defense, government, healthcare, networks, security, and transportation. ARINC had installed computer data networks in police cars and railroad cars and also maintains the standards for line-replaceable units. [21]

ARRT Anytime RRT

Anytime RRT is an anytime variant of RRT. [22]

ASL Above Mean Sea Level

Height above mean sea level is a measure of the vertical distance (height, elevation or altitude) of a location in reference to a historic mean sea level taken as a vertical datum. [23]

ATC Air Traffic Control

Air traffic control is a service provided by ground-based controllers who direct aircraft on the ground and through controlled airspace, and can provide advisory services to aircraft in non-controlled airspace. [24]

ATFM Air Traffic Flow Management

Air traffic flow management is the regulation of air traffic in order to avoid exceeding airport or air traffic control capacity in handling traffic (hence the alternative name of Air Traffic Flow and Capacity Management - ATFCM), and to ensure that available capacity is used efficiently. [25]

ATO Actual Time Over

The actual time over records the time a waypoint is reached by an airborne agent.

[20] https://en.wikipedia.org/wiki/Augmented_reality
[21] https://en.wikipedia.org/wiki/ARINC
[22] https://en.wikipedia.org/wiki/Anytime_algorithm
[23] https://en.wikipedia.org/wiki/Height_above_sea_level
[24] https://en.wikipedia.org/wiki/Air_traffic_control
[25] https://en.wikipedia.org/wiki/Air_traffic_flow_management

ATS Air Traffic Service

In aviation, an air traffic service is a service which regulates and assists aircraft in real-time to ensure their safe operations. [26]

AWR Airborne Weather Radar

Weather radar, also called weather surveillance radar and Doppler weather radar, is a type of radar used to locate precipitation, calculate its motion, and estimate its type (rain, snow, hail etc.). [27]

Block A* Block A*

Block A* is a string pulling any-angle variant of A*. [28]

CBT Computer-Based Training

Computer-based training is an educational technology (commonly abbreviated as EduTech, or EdTech) which is the combined use of computer hardware, software, and educational theory and practice to facilitate learning. [29]

CfAR Center for Aerospace Research

The UVic Center for Aerospace Research is an organization devoted to the promotion of Unmanned Aerial Vehicles for commercial and research applications. [30]

CPDLC Controller-Pilot Data Link Communications

Controller-pilot data link communications, also referred to as controller pilot data link, is a method by which air traffic controllers can communicate with pilots over a datalink system. [31]

CPS Cyber-Physical System

A cyber-physical system is a system of collaborating computational elements controlling physical entities. [32]

CGS Classical Grid Search

Grid-based approaches overlay a grid on the configuration space, and assume

[26] https://en.wikipedia.org/wiki/Air_traffic_service
[27] https://en.wikipedia.org/wiki/Weather_radar
[28] https://en.wikipedia.org/wiki/Any-angle_path_planning
[29] https://en.wikipedia.org/wiki/Educational_technology
[30] https://cfar.engr.uvic.ca/
[31] https://en.wikipedia.org/wiki/Controller-pilot_data_link_communications
[32] https://en.wikipedia.org/wiki/Cyber-physical_system

each configuration is identified with a grid point. At each grid point, the robot is allowed to move to adjacent grid points as long as the line between them is completely contained within the free configuration space. [33]

CI Continuous Integration

Continuous integration is the practice, in software engineering, of merging all developer working copies to a shared mainline several times a day. [34]

CRM Crew Resource Mangement

Crew resource management or cockpit resource management is a set of training procedures for use in environments where human error can have devastating effects. Used primarily for improving aviation safety, CRM focuses on interpersonal communication, leadership, and decision making in the cockpit of an airliner. [35]

GCS Ground Control Station

A ground control station is a land- or sea-based control center that provides the facilities for human control of unmanned vehicles in the air or in space. A ground control station could be used to control UAVs or rockets within or above the atmosphere. [36]

GFA Graphical Forecast Area

Graphical Forecast Area is a weather map issued by Nav Canada to pilots that includes high and low pressure systems, fronts, precipitation, winds, icing, and turbulence. [37]

GUI Graphical User Interface

The graphical user interface is a form of user interface that allows users to interact with electronic devices through graphical icons and audio indicator such as primary notation, instead of text-based user interfaces, typed command labels or text navigation. [38]

D* Dynamic A*

Dynamic A* is an incremental repairing variant of A*. [39]

[33] https://en.wikipedia.org/wiki/Motion_planning#Grid-Based_Search
[34] https://en.wikipedia.org/wiki/Continuous_integration
[35] https://en.wikipedia.org/wiki/Crew_resource_management
[36] https://en.wikipedia.org/wiki/UAV_ground_control_station
[37] https://en.wikipedia.org/wiki/Prognostic_chart
[38] https://en.wikipedia.org/wiki/Graphical_user_interface
[39] https://en.wikipedia.org/wiki/D*

D* Lite D* Lite

D* Lite is a focused variant of D*. [40]

DAL Design Assurance Level

The design assurance level is determined from the safety assessment process and hazard analysis by examining the effects of a failure condition in the system. The failure conditions are categorized by their effects on the aircraft, crew, and passengers. [41]

DP Dynamic Programming

In mathematics, management science, economics, computer science, and bioinformatics, dynamic programming is a method for solving a complex problem by breaking it down into a collection of simpler sub-problems, solving each of those sub-problems just once, and storing their solutions – ideally, using a memory-based data structure. [42]

DoF Degrees of Freedom

In many scientific fields, the degrees of freedom of a system is the number of parameters of the system that may vary independently. For example, a point in the plane has two degrees of freedom for translation: its two coordinates; a non-infinitesimal object on the plane might have additional degrees of freedoms related to its orientation. [43]

DRRT Dynamic RRT

Dynamic RRT is a dynamically repairing variant of RRT. [44]

DTG Distance To Go

The distance to go for an airborne agent refers to the distance from a presently visited to another designated (the next or final) waypoint.

EFB Electronic Flight Bag

An electronic flight bag is an electronic information management device that helps flight crews perform flight management tasks more easily and efficiently

[40] https://en.wikipedia.org/wiki/D*#D.2A_Lite
[41] https://en.wikipedia.org/wiki/DO-178C
[42] https://en.wikipedia.org/wiki/Dynamic_programming
[43] https://en.wikipedia.org/wiki/Degrees_of_freedom
[44] https://en.wikipedia.org/wiki/Rapidly_exploring_random_tree

with less paper providing the reference material often found in the pilot's carry-on flight bag, including the flight-crew operating manual, navigational charts, etc. [45]

ERRT Extended RRT

Extended RRT is an online (interleaved) variant of RRT. [46]

ETA Estimated Time of Arrival

The estimated time of arrival for an airborne agent is the time at which the agent is expected to arrive at its destination.

ETD Estimated Time of Departure

The estimated time of departure for an airborne agent is the time at which the agent is expected to depart at its origin.

ETE Estimated Time Enroute

The estimated time enroute for an airborne agent is the expected duration required to travel between two designated waypoints.

ETO Estimated Time Over

The estimated time over estimates the time a waypoint is expected to be reached by an airborne agent.

FAA Federal Aviation Administration

The Federal Aviation Administration is the largest transportation agency of the United States government and regulates all aspects of civil aviation in the country as well as over surrounding international waters. [47]

FADPRM Flexible Anytime Dynamic PRM

Flexible, anytime, dynamic PRM is an anytime, dynamic variant of PRM implementing the concept of desirable regions. [48]

FD Winds and Temperatures Aloft Forecast

Winds aloft, officially known as the winds and temperatures aloft forecast is a forecast of specific atmospheric conditions in terms of wind and temperature at cer-

[45] https://en.wikipedia.org/wiki/Electronic_flight_bag
[46] https://en.wikipedia.org/wiki/Rapidly_exploring_random_tree
[47] https://en.wikipedia.org/wiki/Federal_Aviation_Administration
[48] https://en.wikipedia.org/wiki/Probabilistic_roadmap

tain altitudes, typically measured in feet above mean sea level. The forecast is specifically used for aviation purposes. [49]

FD* Field D*

Field D* is an interpolation-based any-angle variant of D*. [50]

FDR Flight Data Recorder

A flight data recorder preserves the recent history of the flight through the recording of dozens of parameters collected several times per second. [51]

FIC Flight Information Center

Flight Information Center are centres employing flight service specialists responsible for the management and dissemination of flight safety related information. [52]

FIXM Flight Information Exchange Model

The Flight Information Exchange Model is a global exchange standard capturing Flight and Flow information. FIXM is implemented in UML and XML and fully supports the data exchange requirements for the Flight and Flow Information for a Collaborative Environment (FF-ICE) concept, as defined by the ICAO-ATMRPP. [53]

FLARM Flight Alarm

FLARM is proprietary electronic system used to selectively alert pilots to potential collisions between aircraft. [54]

FMS Flight Management System

A flight management system is a fundamental component of a modern airliner's avionics. An FMS is a specialized computer system that automates a wide variety of in-flight tasks, reducing the workload on the flight crew to the point that modern civilian aircraft no longer carry flight engineers or navigators. [55]

[49] https://en.wikipedia.org/wiki/Winds_aloft
[50] https://en.wikipedia.org/wiki/Any-angle_path_planning
[51] https://en.wikipedia.org/wiki/Flight_recorder
[52] https://en.wikipedia.org/wiki/Flight_Information_Centre
[53] https://www.fixm.aero/
[54] https://en.wikipedia.org/wiki/FLARM
[55] https://en.wikipedia.org/wiki/Flight_management_system

FPV First Person View

First-person view, or simply video piloting, is a method used to control a radio-controlled vehicle from the driver or pilot's view point. [56]

FXML EFF-ects eXtended Markup Language FXML is an XML-based user interface markup language created by Oracle Corporation for defining the user interface of a JavaFX application. FXML presents an alternative to designing user interfaces using procedural code, and allows for abstracting program design from program logic. [57]

GA Genetic Algorthim

In computer science and operations research, a genetic algorithm is a metaheuristic inspired by the process of natural selection that belongs to the larger class of evolutionary algorithms. Genetic algorithms are commonly used to generate high-quality solutions to optimization and search problems by relying on biologically inspired operators such as mutation, crossover and selection. [58]

GAA* Generalized Adaptive A*

Generalized Adaptive A* is an adaptive variant of A*. [59]

GAMA General Aviation Manufacturers Association

The General Aviation Manufacturer's Association is an aviation industry trade association. It is the industry trade association representing general aviation (non-military and non-airliner) aircraft manufacturers and related enterprises, chiefly in the United States. It is headquartered in Washington, D.C., with an office in Brussels, Belgium. [60]

GIS Geo-Information System

A geographic information system or geographical information system is a system designed to capture, store, manipulate, analyze, manage, and present all types of spatial or geographical data. [61]

GML Geography Markup Language

The Geography Markup Language is the XML grammar defined by the Open

[56] https://en.wikipedia.org/wiki/First-person_view_(radio_control)
[57] https://en.wikipedia.org/wiki/FXML
[58] https://en.wikipedia.org/wiki/Genetic_algorithm
[59] https://en.wikipedia.org/wiki/Adaptive_behavior
[60] https://en.wikipedia.org/wiki/General_Aviation_Manufacturers_Association
[61] https://en.wikipedia.org/wiki/Geographic_information_system

Geospatial Consortium (OGC) to express geographical features. GML serves as a modeling language for geographic systems as well as an open interchange format for geographic transactions on the Internet. Key to GML's utility is its ability to integrate all forms of geographic information, including not only conventional "vector" or discrete objects, but coverages and sensor data. [62]

GNSS Global Navigation Satellite System

A satellite navigation or satnav system is a system that uses satellites to provide autonomous geo-spatial positioning. It allows small electronic receivers to determine their location (longitude, latitude, and altitude/elevation) to a high precision (within a few centimeters to metres) using time signals transmitted along a line of sight by radio from satellites. [63]

HCI Human-Computer Interface

A human-computer interface is a HMI where the machine in question is a computer. [64]

HIL Hardware in the Loop

Hardware in the loop simulation is a technique that is used in the development and test of complex real-time embedded systems. It provides an effective platform by adding the complexity of the plant under control to the test platform. [65]

HMI Human-Machine Interface

A human-machine interface interfaces machines with physical input hardware such as keyboards, mice, or game pads, and output hardware such as computer monitors, speakers, and printers. [66]

HRRT Heuristic RRT

Heuristic RRT extends the basic RRT with a heuristic in order to perform an informed selection of samples for the tree extensions. [67]

ICAO International Civil Aviation Organization

The International Civil Aviation Organization is a specialized and funding agency

[62] https://en.wikipedia.org/wiki/Geography_Markup_Language
[63] https://en.wikipedia.org/wiki/Satellite_navigation
[64] https://en.wikipedia.org/wiki/User_interface
[65] https://en.wikipedia.org/wiki/Hardware-in-the-loop_simulation
[66] https://en.wikipedia.org/wiki/User_interface
[67] https://en.wikipedia.org/wiki/Rapidly_exploring_random_tree

of the United Nations. It changes the principles and techniques of international air navigation and fosters the planning and development of international air transport to ensure safe and orderly growth. Its headquarters is located in the Quartier International of Montreal, Quebec, Canada. [68]

IDAL Item Development Assurance Level

The item development assurance level is determined from the safety assessment process and hazard analysis by examining the effects of a failure condition in the system. The failure conditions are categorized by their effects on the aircraft, crew, and passengers. [69]

IFR Instrument Flight Rules

Rules and regulations established to govern flight under conditions in which flight by outside visual reference is not safe. IFR flight depends upon flying by reference to instruments in the flight deck, and navigation is accomplished by reference to electronic signals. [70]

IMC Instrument Meteorological Conditions

Instrument meteorological conditions is an aviation flight category that describes weather conditions that require pilots to fly primarily by reference to instruments, and therefore under IFR, rather than by outside visual references under VFR. [71]

IP* Incremental Φ^*

Incremental Φ^* is an incremental variant of Θ^*. [72]

ITWS Integrated Terminal Weather System

The Integrated Terminal Weather System provides improved integration of weather data into timely, accurate aviation weather information. This is an essential component in reducing delays and improving NAS capacity use while enhancing aviation safety. [73]

IWXXM ICAO Meteorological Information Exchange Model

ICAO Meteorological Information Exchange Model is a format for reporting

[68] https://en.wikipedia.org/wiki/International_Civil_Aviation_Organization
[69] https://en.wikipedia.org/wiki/DO-178C
[70] https://en.wikipedia.org/wiki/Instrument_flight_rules
[71] https://en.wikipedia.org/wiki/Instrument_meteorological_conditions
[72] https://en.wikipedia.org/wiki/Incremental_heuristic_search
[73] https://www.faa.gov/air_traffic/technology/itws/

weather information in XML. IWXXM includes XML-based representations for products standardized in International Civil Aviation Organization Annex *III* and World Meteorological Organization No. 49, Vol *II*, such as METAR, SPECI, TAF, and SIGMET. IWXXM products are used for operational exchanges of meteorological information for use in aviation. [74]

LAO* Loop AO*

Loop AO* combines A* with DP to solve MDPs more efficiently. [75]

LIDAR Light Detection and Ranging

Light Detection and Ranging is a method for determining ranges (variable distance) by targeting an object with a laser and measuring the time for the reflected light to return to the receiver. [76]

LRM Lattice Roadmap

Lattice-based motion planners combine the strengths of probabilistic approaches such as RRTs with CGS algorithms. (Pivtoraiko and Kelly [PK08])

LPRM Lazy PRM

Lazy PRM is a modified variant of PRM that defers collision checking to the query phase. [77]

LT* Lazy Θ^*

Lazy Θ^* is a reduced visibility checking variant of Θ^*. [78]

LPA* Lifelong Planning A*

Lifelong Planning A* is a continuously planning variant of A*. [79]

MMI Man-Machine Interface

A man-machine interface is a different term for HMI.

MAPE-K Monitor, Analyze, Plan, Execute - Knowledge

An Autonomic Element is managed by an Autonomic Manager that monitors its Managed Element, analyzes the monitored data and plans actions to be executed based on knowledge about the element and its Autonomic Context.

[74] https://en.wikipedia.org/wiki/IWXXM
[75] https://en.wikipedia.org/wiki/Markov_decision_process
[76] https://en.wikipedia.org/wiki/Lidar
[77] https://en.wikipedia.org/wiki/Probabilistic_roadmap
[78] https://en.wikipedia.org/wiki/Any-angle_path_planning
[79] https://en.wikipedia.org/wiki/Incremental_heuristic_search

MAUT Multi-Attribute Utility Theory

Multi-Attribute Utility Theory is a structured methodology designed to handle the trade-offs among multiple objectives. [80]

ME Managed Element

A Managed Element is part of an Autonomic Element and managed by its Autonomic Manager.

METAR Aviation Routine Weather Report

METAR is a format for reporting weather information. A METAR weather report is predominantly used by pilots in fulfilment of a part of a pre-flight weather briefing, and by meteorologists, who use aggregated METAR information to assist in weather forecasting. [81]

MCDM Multi-Criteria Decision Making

Multiple-criteria decision-making or multiple-criteria decision analysis is a subdiscipline of operations research that explicitly considers multiple criteria in decision-making environments. [82]

MDP Markov Decision Process

Markov decision processes provide a mathematical framework for modelling decision making in situations where outcomes are partly random and partly under the control of a decision maker. [83]

MMIXM Maintenance Management Information Exchange Model

Maintenance Management Information Exchange Model is an FAA data standard which is aimed to support the exchange of Operations and Maintenance (O&M) information between NAS systems and services. [84]

MVP Model-View-Presenter

Model-view-presenter is a derivation of the model-view-controller architectural pattern, and is used mostly for building user interfaces. In MVP, the presenter assumes the functionality of the "middle-man". In MVP, all presentation logic is pushed to the presenter. [85]

[80] https://en.wikipedia.org/wiki/Multi-attribute_utility
[81] https://en.wikipedia.org/wiki/METAR
[82] https://en.wikipedia.org/wiki/Multiple-criteria_decision_analysis
[83] https://en.wikipedia.org/wiki/Markov_decision_process
[84] https://mmixm.aero/
[85] https://en.wikipedia.org/wiki/Model-view-presenter

NAS National Airspace System

The National Airspace System is the airspace, navigation facilities and airports of the United States along with their associated information, services, rules, regulations, policies, procedures, personnel and equipment. It includes components shared jointly with the military. It is one of the most complex aviation systems in the world, and services air travel in the United States and over large portions of the world's oceans. [86]

NASA National Aeronautics and Space Administration

The National Aeronautics and Space Administration is an independent agency of the United States federal government responsible for the civilian space program as well as aeronautics and space research. [87]

NextGen Next Generation Air Transportation System

The Next Generation Air Transportation System is an ongoing modernization project of the United States NAS. The United States FAA began work on NextGen improvements in 2007 and plans to have all major components in place by 2025. The goals of the modernization include using new technologies and procedures to increase the safety, efficiency, capacity, access, flexibility, predictability, and resilience of the NAS while reducing the environmental impact of aviation. [88]

NOTAM Notice to Airmen

A Notice to Airmen is a notice filed with an aviation authority to alert aircraft pilots of potential hazards along a flight route or at a location that could affect the safety of the flight. NOTAMs are unclassified notices or advisories distributed by means of telecommunication that contain information concerning the establishment, conditions or change in any aeronautical facility, service, procedure or hazard, the timely knowledge of which is essential to personnel and systems concerned with flight operations. [89]

NP Non-Deterministic Polynomial Time

In computational complexity theory, NP is one of the most fundamental complexity classes. The abbreviation NP refers to non-deterministic polynomial time. [90]

[86] https://en.wikipedia.org/wiki/National_Airspace_System
[87] https://en.wikipedia.org/wiki/NASA
[88] https://en.wikipedia.org/wiki/Next_Generation_Air_Transportation_System
[89] https://en.wikipedia.org/wiki/NOTAM
[90] https://en.wikipedia.org/wiki/NP_(complexity)

OAD* Online AD*

An AD* variant that monitors the progress of a planned agent and revises plans accordingly if necessary.

OADRRT Online ADRRT

An ADRRT variant that monitors the progress of a planned agent and revises plans accordingly if necessary.

OPS Command and Control, Operations or Dispatch

Command and control is a set of organizational and technical attributes and processes that employs human, physical, and information resources to solve problems and accomplish missions to achieve the goals of an organization or enterprise. [91]

PAO* Propagating AO*

Propagating AO* combines AO* with fast propagation of hidden state information.

PBGA Pattern-Based Genetic Algortihm

A pattern-based genetic algorithm applies established patterns to create modifications in new offspring generations of a GA. [92]

PDM Pilot Decision Making

Pilot decision making is a process that aviators perform to effectively handle troublesome situations that are encountered. Pilot decision-making is applied in almost every stage of the flight as it considers weather, airspaces, airport conditions, ETA and so forth. [93]

PF Pilot Flying

In commercial aviation with a two-person flight crew, the Pilot Flying is the pilot operating the flight controls of the aircraft. The other pilot is referred to as the PM or PNF. Before a flight departs, the PIC must decide who the PF and PM will be for the remainder of the flight, or for any specific phase of flight (e.g., take-off, approach or landing). Typically, the flight crew swap roles on the return leg. [94]

[91] https://en.wikipedia.org/wiki/Command_and_control
[92] https://en.wikipedia.org/wiki/Genetic_algorithm
[93] https://en.wikipedia.org/wiki/Pilot_decision_making
[94] https://en.wikipedia.org/wiki/Pilot_flying

PIC Pilot In Command

The pilot in command of an aircraft is the person aboard the aircraft who is ultimately responsible for its operation and safety during flight. [95]

PIREP Pilot Report

A pilot report is a report of actual weather conditions encountered by an aircraft in flight. This information is usually relayed by radio to the nearest ground station. The message would then be encoded and relayed to other weather offices and air traffic service units. [96]

PM Pilot Monitoring

A Pilot Monitoring shares the cockpit workload with the PF.

PNF Pilot Not Flying

A Pilot Not Flying is another term for PM.

PPCP Probabilistic Planning with Clear Preferences

Probabilistic Planning with Clear Preferences is a planning approach that combines POMDP solution with clear preferences on hidden variables. [97]

POH Pilot Operating Handbook

A pilot operating handbook serves as a manual and reference publication and is available for all aircraft types.

POI Point of Interest

A point of interest in a multi-part trajectory separates two independent trajectory parts that connect at the same ETO. A point of interest may also be associated with specific actions at the designated location and associated delays.

POJO Plain-Old Java Object

In software engineering, a plain old Java object is an ordinary Java object, not bound by any special restriction. The term was coined by Martin Fowler, Rebecca Parsons and Josh MacKenzie in September 2000. [98]

POMDP Partially Observable Markov Decision Process

A partially observable Markov decision process is a generalization of an MDP. A

[95] https://en.wikipedia.org/wiki/Pilot_in_command
[96] https://en.wikipedia.org/wiki/Pilot_report
[97] https://en.wikipedia.org/wiki/Partially_observable_Markov_decision_process
[98] https://en.wikipedia.org/wiki/Plain_old_Java_object

POMDP models an agent decision process in which it is assumed that the system dynamics are determined by an MDP, but the agent cannot directly observe the underlying state. Instead, it must maintain a probability distribution over the set of possible states, based on a set of observations and observation probabilities, and the underlying MDP. [99]

PRM Probabilistic Roadmap

The probabilistic roadmap planner is a motion planning algorithm in robotics, which solves the problem of determining a path between a starting configuration of the robot and a goal configuration while avoiding collisions. An example of a probabilistic random map algorithm exploring feasible paths around a number of polygonal obstacles. The basic idea behind PRM is to take random samples from the configuration space of the robot, testing them for whether they are in the free space, and use a local planner to attempt to connect these configurations to other nearby configurations. [100]

QRM Quasi-Random Roadmap

RNP Required Navigation Performance

Required navigation performance is a type of performance-based navigation that allows an aircraft to fly a specific path between two 3D-defined points in space. [101]

ROAR Random Online Adaptive Racing

Random Online Adaptive Racing is a model-free variant of SMBO that applies the same intensification proecedure as SMAC but only randomly selects new configuration parameter candidates without fitting and accessing a regression model. (Hutter, Hoos, Leyton-Brown [HHLB11])

RPC Remote Procedure Call

In computer science, a Remote Procedure Call is an inter-process communication that allows a computer program to cause a subroutine or procedure to execute in another address space (commonly on another computer on a shared network) without the programmer explicitly coding the details for this remote interaction. [102]

[99] https://en.wikipedia.org/wiki/Partially_observable_Markov_decision_process
[100] https://en.wikipedia.org/wiki/Probabilistic_roadmap
[101] https://en.wikipedia.org/wiki/Required_navigation_performance
[102] https://en.wikipedia.org/wiki/Remote_procedure_call

RRT Rapidly-Exploring Random Tree

A rapidly-exploring random tree is an algorithm designed to efficiently search non-convex, high-dimensional spaces by randomly building a space-filling tree. The tree is constructed incrementally from samples drawn randomly from the search space and is inherently biased to grow toward large unexplored areas of the problem. [103]

RRT* Optimal RRT

A RRT* algorithm is an optimal variant of RRT in the probabilistic sense. [104]

RTCA Radio Technical Commission for Aeronautics

The RTCA Inc. is a United States non-profit organization that develops technical guidance for use by government regulatory authorities and by industry. [105]

RTDP Real-Time DP

Real-Time DP is an anytime variant of VI that interleaves policy computation and execution. [106]

RVSM Reduced Vertical Separation Minima

Reduced vertical separation minima is the reduction, from 2,000 feet to 1,000 feet, of the standard vertical separation required between aircraft flying between flight level 290 (29,000 ft) and flight level 410 (41,000 ft). [107]

SA Situational Awareness

Situational awareness is the perception of environmental elements with respect to time or space, the comprehension of their meaning, and the projection of their status after some variable has changed, such as time, or some other variable, such as a predetermined event. [108]

SAFCS Smart Autoflight Control System

The Smart Autoflight Control System is a proposed self-adaptive architecture for aircraft trajectory planning and decision support consisting of a core and a managing component. The core component provides the relevant infrastructure for

[103] https://en.wikipedia.org/wiki/Rapidly_exploring_random_tree
[104] https://en.wikipedia.org/wiki/Rapidly_exploring_random_tree
[105] https://en.wikipedia.org/wiki/Radio_Technical_Commission_for_Aeronautics
[106] https://en.wikipedia.org/wiki/Markov_decision_process#Value_iteration
[107] https://en.wikipedia.org/wiki/Reduced_vertical_separation_minima
[108] https://en.wikipedia.org/wiki/Situation_awareness

planning and the planning context, including environments, aircraft, planners, and connections. The managing component is responsible for setting up planning sessions, and thereafter, continuously monitoring, evaluating, and adjusting its configuration to improve the achievable planning performance.

SAR Search and Rescue

Search and rescue is the search for and provision of aid to people who are in distress or imminent danger. The general field of search and rescue includes many specialty sub-fields, typically determined by the type of terrain the search is conducted over. [109]

SARC Search and Rescue Center

Search and rescue is the search for and provision of aid to people who are in distress or imminent danger. The general field of search and rescue includes many specialty sub-fields, typically determined by the type of terrain the search is conducted over. These include mountain rescue; ground search and rescue, including the use of search and rescue dogs; urban search and rescue in cities; combat search and rescue on the battlefield and air-sea rescue over water. [110]

SDK Software Development Kit

A software development kit is a collection of software development tools in one installable package. They facilitate the creation of applications by having a compiler, debugger and perhaps a software framework. [111]

SFOC Special Flight Operations Certificate

A Special Flight Operations Certificate gives one permission to operate a UAV outside the rules for a specific purpose. [112]

SGS Subsampled Grid Search

SIGMET Significant Meteorological Information

Significant Meteorological Information, is a weather advisory that contains meteorological information concerning the safety of all aircraft. There are two types

[109] https://en.wikipedia.org/wiki/Search_and_rescue
[110] https://en.wikipedia.org/wiki/Search_and_rescue
[111] https://en.wikipedia.org/wiki/Software_development_kit
[112] https://tc.canada.ca/en/aviation/drone-safety/drone-pilot-licensing/get-permission-special-drone-operations

of SIGMET - convective and non-convective. The criteria for a non-convective SIGMET to be issued are severe or greater turbulence over a 3,000-square-mile ($7,800 km^2$) area, severe or greater icing over a 3,000-square-mile ($7,800 km^2$) area or IMC over a 3,000-square-mile ($7,800 km^2$) area due to dust, sand, or volcanic ash. [113]

SIL Software in the Loop

The software in the loop development simulation replaces the required hardware components of an embedded system by software components that model and simulate their relevant aspects.

SLAM Simultaneous Localization and Mapping

In robotic mapping, simultaneous localization and mapping is the computational problem of constructing or updating a map of an unknown environment while simultaneously keeping track of an agent's location within it. [114]

SMAC Sequential Model-Based Algorithm Configuration

Sequential model-based algorithm configuration is a versatile tool for optimizing algorithm parameters (or the parameters of some other process we can run automatically, or a function we can evaluate, such as a simulation). [115]

SMBO Sequential Model-Based Optimization

Model-based optimization methods construct a regression model (often called a response surface model) that predicts performance and then uses this model for optimization. Sequential model-based optimization iterates between fitting a model and gathering additional data based on this model. (Hutter, Hoos, Leyton-Brown [HHLB11])

SOP Standard Operating Procedure

A standard operating procedure is a set of step-by-step instructions compiled by an organization to help workers carry out routine operations. SOPs aim to achieve efficiency, quality output and uniformity of performance, while reducing miscommunication and failure to comply with industry regulations. [116]

[113] https://en.wikipedia.org/wiki/SIGMET
[114] https://en.wikipedia.org/wiki/Simultaneous_localization_and_mapping
[115] https://www.automl.org/automated-algorithm-design/algorithm-configuration/smac/
[116] https://en.wikipedia.org/wiki/Standard_operating_procedure

SPECI Special METAR

METARs typically come from airports or permanent weather observation stations. Reports are generated once an hour or half-hour, but if conditions change significantly, a report known as a *special* may be issued. [117]

SUA Special Use Airspace

Special use airspace is an area designated for operations of a nature such that limitations may be imposed on aircraft not participating in those operations. Often these operations are of a military nature. The designation of SUAs identifies for other users the areas where such activity occurs, provides for segregation of that activity from other users, and allows charting to keep airspace users informed of potential hazards. [118]

SVM Support Vector Machines

In machine learning, support vector machines are supervised learning models with associated learning algorithms that analyze data used for classification and regression analysis. [119]

SWIM System-Wide Information Management

System-Wide Information Management is a global Air Traffic Management industry initiative to harmonize the exchange of Aeronautical, Weather and Flight information for all Airspace Users and Stakeholders. [120]

T* Θ^*

Theta* is a string pulling any-angle variant of A* [121]

TAF Terminal Aerodrome Forecast

In meteorology and aviation, terminal aerodrome forecast or terminal area forecast is a format for reporting weather forecast information, particularly as it relates to aviation. TAFs are issued every 6 hours: 0000, 0600, 1200, 1800 UTC, and generally apply to a 24 or 30-hour period, and an area within approximately five statute miles from the center of an airport runway complex. [122]

[117] https://en.wikipedia.org/wiki/SPECI
[118] https://en.wikipedia.org/wiki/Special_use_airspace
[119] https://en.wikipedia.org/wiki/Support_vector_machine
[120] https://en.wikipedia.org/wiki/System_Wide_Information_Management
[121] https://en.wikipedia.org/wiki/Any-angle_path_planning
[122] https://en.wikipedia.org/wiki/Terminal_aerodrome_forecast

TAP Traffic Aware Planner

NASA's Traffic Aware Planner is a cockpit decision support tool that has the potential to achieve significant fuel and time savings when it is embedded in the data-rich NextGen airspace. (Maris, Haynes and Wing [MHW14]) [123]

TASAR Traffic Aware Strategic Aircrew Requests

Traffic Aware Strategic Aircrew Requests is a software system to provide pilots with traffic aware optimized flight trajectories to increase ATC approval of beneficial change requests. [124]

TC Transport Canada

Transport Canada is the department within the Government of Canada responsible for developing regulations, policies and services of road, rail, marine and air transportation in Canada. [125]

TCP Transmission Control Protocol

The Transmission Control Protocol is one of the main protocols of the Internet protocol suite. It originated in the initial network implementation in which it complemented the Internet Protocol. [126]

TRL Technology Readiness Level

Technology readiness levels are a method for estimating the maturity of technologies during the acquisition phase of a program, developed at NASA during the 1970s. The use of TRLs enables consistent, uniform discussions of technical maturity across different types of technology. [127]

TTG Time To Go

The time to go for an airborne agent refers to the duration from a presently visited to another designated (the next or final) waypoint.

UAS Unmanned Aircraft System

Unmanned aircraft system emphasises the importance of other elements beyond an aircraft itself. It includes elements such as the ground control stations, data links and other related support equipment. [128]

[123] https://ntrs.nasa.gov/citations/20140011404
[124] https://technology.nasa.gov/patent/LAR-TOPS-148
[125] https://en.wikipedia.org/wiki/Transport_Canada
[126] https://en.wikipedia.org/wiki/Transmission_Control_Protocol
[127] https://en.wikipedia.org/wiki/Technology_readiness_level
[128] https://en.wikipedia.org/wiki/Unmanned_aerial_vehicle

UAV Unmanned Aerial Vehicle

An unmanned aerial vehicle, commonly known as a drone and also referred to by several other names, is an aircraft without a human pilot aboard. The flight of UAVs may be controlled either autonomously by on-board computers or by the remote control of a pilot on the ground or in another vehicle. [129]

UDP User Datagram Protocol

In computer networking, the User Datagram Protocol is one of the core members of the Internet protocol suite. With UDP, computer applications can send messages, in this case referred to as datagrams, to other hosts on an Internet Protocol network. [130]

UI User Interface

The interface between the software and its user. The user interface can take the form of a GUI and is sometimes referred to as the HMI or MMI. [131]

UTC Coordinated Universal Time

Coordinated Universal Time is the primary time standard by which the world regulates clocks and time. It is within about one second of mean solar time at zero degrees longitude, and is not adjusted for daylight saving time. [132]

UTM UAS Traffic Management

UAS Traffic Management is an air traffic management ecosystem under development for autonomously controlled operations of UAS by the FAA, NASA, other federal partner agencies, and industry. [133]

UVic University of Victoria

The University of Victoria is a public research university located in the municipalities of Oak Bay and Saanich, British Columbia, Canada. The university traces its roots to Victoria College, the first post-secondary institution established in the province of British Columbia in 1903. It was reincorporated as a university in 1963. [134]

[129] https://en.wikipedia.org/wiki/Unmanned_aerial_vehicle
[130] https://en.wikipedia.org/wiki/User_Datagram_Protocol
[131] https://en.wikipedia.org/wiki/User_interface
[132] https://en.wikipedia.org/wiki/Coordinated_Universal_Time
[133] https://en.wikipedia.org/wiki/Unmanned_aircraft_system_traffic_management
[134] https://en.wikipedia.org/wiki/University_of_Victoria

VFR Visual Flight Rules

Visual flight rules are a set of regulations under which a pilot operates an aircraft in weather conditions generally clear enough to allow the pilot to see where the aircraft is going. [135]

VI Value Iteration

Value iteration, which is also called backward induction, is a DP programming technique. [136]

VMC Visual Meteorological Conditions

In aviation, visual meteorological conditions is an aviation flight category in which VFR flight is permitted – that is, conditions in which pilots have sufficient visibility to fly the aircraft maintaining visual separation from terrain and other aircraft. [137]

WA* Weighted A*

Weighted A* is a weighted variant of A* where current and estimated costs are multiplied by weight factors. [138]

WLT* Weighted Lazy Θ^*

Weighted Lazy Θ^* is a weighted variant of Lazy Θ^* where current and estimated costs are multiplied by weight factors. [139]

WMS Web Map Service

A Web Map Service is a standard protocol developed by the Open Geospatial Consortium in 1999 for serving georeferenced map images over the Internet. [140]

WXXM Weather Information Exchange Model

The Weather Information Exchange Model is designed to enable the management and distribution of weather data in digital format (XML). WXXM version 2.0, set to be finalized in 2014, is based on GML and is one of the GML Application Schemas. It is being developed by the United States FAA and the European Organization for the Safety of Air Navigation (EUROCONTROL). [141]

[135] https://en.wikipedia.org/wiki/Visual_flight_rules
[136] https://en.wikipedia.org/wiki/Markov_decision_process#Value_iteration
[137] https://en.wikipedia.org/wiki/Visual_meteorological_conditions
[138] https://en.wikipedia.org/wiki/A*_search_algorithm
[139] https://en.wikipedia.org/wiki/Any-angle_path_planning
[140] https://en.wikipedia.org/wiki/Web_Map_Service
[141] https://en.wikipedia.org/wiki/WXXM_(data_model)

XML Extensible Markup Language

Extensible Markup Language is a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable. The World Wide Web Consortium's XML 1.0 Specification of 1998 and several other related specifications – all of them free open standards – define XML. [142]

[142] https://en.wikipedia.org/wiki/XML

Appendix B

Project Statistics

B.1 Project Statistics

This annex lists the contents and development branch statistics of the most relevant SAFCS software projects. The currently open issues and other project statistics can be found on the respective project web-sites.

- SAFCS core and management components[1]
- SAFCS UI[2]
- SAFCS dronekit datalink[3]

The corresponing released software artifacts are available in the Maven Central Repository[4].

B.1.1 com.uvic-cfar.swim:worldwind:1.0.0-alpha4-SNAPSHOT

Package Name	File Name	Type	Blank	JavaDoc	Comment	Code	Total
com.binarydreamers.trees	BinaryTree.java	src	93	124	78	830	1125
com.binarydreamers.trees	IntegerInterval.java	src	6	0	15	32	53
com.binarydreamers.trees	Interval.java	src	1	0	15	5	21
com.binarydreamers.trees	IntervalTree.java	src	105	143	86	904	1238
com.binarydreamers.trees	LongInterval.java	src	7	0	15	32	54
com.binarydreamers.trees	package-info.java	src	0	0	0	1	1
com.cfar.swim.worldwind.aircraft	A320.java	src	26	61	4	73	164
com.cfar.swim.worldwind.aircraft	Aircraft.java	src	21	93	2	102	218
com.cfar.swim.worldwind.aircraft	Capabilities.java	src	78	270	109	264	721
com.cfar.swim.worldwind.aircraft	CapabilitiesException.java	src	4	41	0	8	53
com.cfar.swim.worldwind.aircraft	CivilAircraft.java	src	11	51	1	36	99
com.cfar.swim.worldwind.aircraft	CombatIdentification.java	src	6	46	0	16	68
com.cfar.swim.worldwind.aircraft	FixedWingCivilAircraft.java	src	11	51	1	33	96
com.cfar.swim.worldwind.aircraft	H135.java	src	26	61	4	73	164
com.cfar.swim.worldwind.aircraft	Iris.java	src	27	61	4	73	165

[1] https://github.com/stephanheinemann/worldwind/projects/1
[2] https://github.com/stephanheinemann/worldwind-ui/projects/1
[3] https://github.com/stephanheinemann/droneconnect/projects/1
[4] https://search.maven.org/search?q=g:com.uvic-cfar.swim

com.cfar.swim.worldwind.aircraft	Quadcopter.java	src	11	51	1	36	99
com.cfar.swim.worldwind.aircraft	RotaryWingCivilAircraft.java	src	10	51	0	33	94
com.cfar.swim.worldwind.aircraft	package-info.java	src	0	29	0	1	30
com.cfar.swim.worldwind.connections	AtcConnection.java	src	7	29	3	13	52
com.cfar.swim.worldwind.connections	Communication.java	src	12	64	0	30	106
com.cfar.swim.worldwind.connections	Connection.java	src	8	52	0	9	69
com.cfar.swim.worldwind.connections	Datalink.java	src	86	327	8	215	636
com.cfar.swim.worldwind.connections	DatalinkCommunicator.java	src	18	126	2	18	164
com.cfar.swim.worldwind.connections	DatalinkTracker.java	src	16	114	1	18	149
com.cfar.swim.worldwind.connections	DronekitDatalink.java	src	95	325	8	427	855
com.cfar.swim.worldwind.connections	LiveSwimConnection.java	src	9	42	0	28	79
com.cfar.swim.worldwind.connections	MavlinkDatalink.java	src	179	423	49	1082	1733
com.cfar.swim.worldwind.connections	OpsConnection.java	src	7	29	3	13	52
com.cfar.swim.worldwind.connections	SimulatedDatalink.java	src	92	339	19	357	807
com.cfar.swim.worldwind.connections	SimulatedSwimConnection.java	src	36	121	4	209	370
com.cfar.swim.worldwind.connections	SwimConnection.java	src	25	117	0	86	228
com.cfar.swim.worldwind.connections	package-info.java	src	0	29	0	1	30
com.cfar.swim.worldwind.data.aixm	AixmLoader.java	src	36	79	15	148	278
com.cfar.swim.worldwind.data.aixm	package-info.java	src	0	29	0	1	30
com.cfar.swim.worldwind.data.fixm	FixmLoader.java	src	42	86	10	194	332
com.cfar.swim.worldwind.data.fixm	package-info.java	src	0	29	0	1	30
com.cfar.swim.worldwind.data.iwxxm	IwxxmLoader.java	src	40	58	22	140	260
com.cfar.swim.worldwind.data.iwxxm	IwxxmPositionFilter.java	src	6	29	3	9	47
com.cfar.swim.worldwind.data.iwxxm	IwxxmSigmetReference.java	src	25	116	0	85	226
com.cfar.swim.worldwind.data.iwxxm	IwxxmUpdater.java	src	55	29	39	257	380
com.cfar.swim.worldwind.data.iwxxm	package-info.java	src	0	29	0	1	30
com.cfar.swim.worldwind.data	AixmData.java	src	24	29	9	102	164
com.cfar.swim.worldwind.data	Conversions.java	src	13	29	5	58	105
com.cfar.swim.worldwind.data	DataActivationListener.java	src	6	50	0	7	63
com.cfar.swim.worldwind.data	GmlData.java	src	67	29	12	186	294
com.cfar.swim.worldwind.data	IwxxmData.java	src	29	29	1	143	202
com.cfar.swim.worldwind.data	ObstacleLoader.java	src	5	42	0	6	53
com.cfar.swim.worldwind.data	OmData.java	src	34	29	4	124	191
com.cfar.swim.worldwind.data	SamData.java	src	2	29	0	3	34
com.cfar.swim.worldwind.data	SwimLoader.java	src	10	47	1	34	92
com.cfar.swim.worldwind.data	SwimProtocol.java	src	1	35	0	8	44
com.cfar.swim.worldwind.data	SwimResource.java	src	11	67	0	38	116
com.cfar.swim.worldwind.data	package-info.java	src	0	29	0	1	30
com.cfar.swim.worldwind.environments	DirectedEdge.java	src	19	81	2	51	153
com.cfar.swim.worldwind.environments	DynamicEnvironment.java	src	12	82	0	13	107
com.cfar.swim.worldwind.environments	DynamicHierarchicalEnvironment.java	src	5	44	0	6	55
com.cfar.swim.worldwind.environments	Edge.java	src	75	263	22	323	683
com.cfar.swim.worldwind.environments	Environment.java	src	39	288	5	57	389
com.cfar.swim.worldwind.environments	HierarchicalEnvironment.java	src	12	95	0	18	125
com.cfar.swim.worldwind.environments	MultiResolutionEnvironment.java	src	7	61	0	8	76
com.cfar.swim.worldwind.environments	PlanningContinuum.java	src	176	789	50	837	1852
com.cfar.swim.worldwind.environments	PlanningGrid.java	src	179	803	45	847	1874
com.cfar.swim.worldwind.environments	PlanningRoadmap.java	src	5	36	5	7	53
com.cfar.swim.worldwind.environments	SamplingEnvironment.java	src	12	113	0	19	144
com.cfar.swim.worldwind.environments	StructuralChangeListener.java	src	4	39	0	5	48
com.cfar.swim.worldwind.environments	StructuredEnvironment.java	src	5	49	0	6	60
com.cfar.swim.worldwind.environments	package-info.java	src	0	29	0	1	30
com.cfar.swim.worldwind.flight	FlightPhase.java	src	34	138	8	103	283
com.cfar.swim.worldwind.flight	package-info.java	src	0	29	0	1	30
com.cfar.swim.worldwind.geom.precision	Precision.java	src	23	51	0	42	116
com.cfar.swim.worldwind.geom.precision	PrecisionDouble.java	src	13	99	3	39	154
com.cfar.swim.worldwind.geom.precision	PrecisionPosition.java	src	15	122	1	59	197
com.cfar.swim.worldwind.geom.precision	PrecisionVec4.java	src	13	109	1	47	170
com.cfar.swim.worldwind.geom.precision	package-info.java	src	0	29	0	1	30
com.cfar.swim.worldwind.geom	Box.java	src	80	302	17	379	778
com.cfar.swim.worldwind.geom	Collisions.java	src	74	176	29	270	549
com.cfar.swim.worldwind.geom	Cube.java	src	5	48	0	10	63
com.cfar.swim.worldwind.geom	CubicGrid.java	src	59	284	12	269	624
com.cfar.swim.worldwind.geom	LineSegment.java	src	51	218	20	218	507
com.cfar.swim.worldwind.geom	Neighborhood.java	src	1	44	0	6	51
com.cfar.swim.worldwind.geom	RegularGrid.java	src	85	431	15	397	928
com.cfar.swim.worldwind.geom	package-info.java	src	0	29	0	1	30
com.cfar.swim.worldwind.javafx	PlanningTimePicker.java	src	19	73	1	83	176
com.cfar.swim.worldwind.javafx	SwimDataListView.java	src	26	79	2	76	183
com.cfar.swim.worldwind.javafx	ThresholdCostSlider.java	src	12	60	0	39	111
com.cfar.swim.worldwind.javafx	TrajectoryStylist.java	src	6	40	0	27	73
com.cfar.swim.worldwind.javafx	package-info.java	src	0	29	0	1	30
com.cfar.swim.worldwind.managers.heuristic	HeuristicAutonomicManager.java	src	5	53	0	17	75
com.cfar.swim.worldwind.managers.heuristic	HeuristicPlannerTuning.java	src	21	70	25	144	260
com.cfar.swim.worldwind.managers.heuristic	package-info.java	src	0	29	0	1	30
com.cfar.swim.worldwind.managers.smac	SmacAutonomicManager.java	src	69	201	44	406	720
com.cfar.swim.worldwind.managers.smac	SmacManagerMode.java	src	3	42	0	5	50
com.cfar.swim.worldwind.managers.smac	SmacPlannerEvaluator.java	src	33	107	27	201	368
com.cfar.swim.worldwind.managers.smac	SmacPlannerEvaluatorObserver.java	src	5	41	0	6	52
com.cfar.swim.worldwind.managers.smac	SmacPlannerTuning.java	src	16	84	19	109	228
com.cfar.swim.worldwind.managers.smac	package-info.java	src	0	29	0	1	30
com.cfar.swim.worldwind.managers	AbstractAutonomicManager.java	src	143	558	38	621	1360
com.cfar.swim.worldwind.managers	AutonomicManager.java	src	24	147	0	37	208
com.cfar.swim.worldwind.managers	package-info.java	src	0	29	0	1	30
com.cfar.swim.worldwind.managing	AbstractPerformance.java	src	21	128	0	85	234
com.cfar.swim.worldwind.managing	AbstractTuning.java	src	13	78	0	38	129
com.cfar.swim.worldwind.managing	Criticality.java	src	12	65	0	41	118
com.cfar.swim.worldwind.managing	Difficulty.java	src	37	139	31	100	307
com.cfar.swim.worldwind.managing	DurationQuantity.java	src	17	95	1	51	164
com.cfar.swim.worldwind.managing	FeatureCategory.java	src	3	43	0	4	50
com.cfar.swim.worldwind.managing	FeatureTuning.java	src	15	97	0	45	157
com.cfar.swim.worldwind.managing	Features.java	src	84	141	32	1160	1417
com.cfar.swim.worldwind.managing	KnowledgeBase.java	src	15	68	2	58	143
com.cfar.swim.worldwind.managing	NumericPerformance.java	src	6	47	0	15	68
com.cfar.swim.worldwind.managing	NumericQuality.java	src	17	95	1	51	164
com.cfar.swim.worldwind.managing	NumericQuantity.java	src	16	95	1	49	161
com.cfar.swim.worldwind.managing	Performance.java	src	9	67	0	11	87
com.cfar.swim.worldwind.managing	PerformanceContext.java	src	36	103	2	125	266
com.cfar.swim.worldwind.managing	PlannerPerformance.java	src	6	47	0	16	69
com.cfar.swim.worldwind.managing	PlannerTuning.java	src	9	74	0	25	108
com.cfar.swim.worldwind.managing	Quality.java	src	5	44	0	6	55
com.cfar.swim.worldwind.managing	Quantity.java	src	5	44	0	6	55
com.cfar.swim.worldwind.managing	Reputation.java	src	35	157	1	130	323
com.cfar.swim.worldwind.managing	Severity.java	src	68	233	65	185	551
com.cfar.swim.worldwind.managing	TrajectoryQuality.java	src	20	95	5	56	176
com.cfar.swim.worldwind.managing	Tuning.java	src	8	58	0	11	77
com.cfar.swim.worldwind.managing	package-info.java	src	0	29	0	1	30
com.cfar.swim.worldwind.planners.cgs.adstar	ADStarPlanner.java	src	107	394	53	491	1045
com.cfar.swim.worldwind.planners.cgs.adstar	ADStarWaypoint.java	src	28	165	11	101	305
com.cfar.swim.worldwind.planners.cgs.adstar	package-info.java	src	0	29	0	1	30

com.cfar.swim.worldwind.planners.cgs.arastar	ARAStarPlanner.java	src	93	393	26	391	903
com.cfar.swim.worldwind.planners.cgs.arastar	ARAStarWaypoint.java	src	11	84	0	32	127
com.cfar.swim.worldwind.planners.cgs.arastar	package-info.java	src	0	29	0	1	30
com.cfar.swim.worldwind.planners.cgs.astar	AStarWaypoint.java	src	24	121	2	84	231
com.cfar.swim.worldwind.planners.cgs.astar	ForwardAStarPlanner.java	src	95	342	52	338	827
com.cfar.swim.worldwind.planners.cgs.astar	package-info.java	src	0	29	0	1	30
com.cfar.swim.worldwind.planners.cgs.fielddstar	package-info.java	src	0	29	0	1	30
com.cfar.swim.worldwind.planners.cgs.oadstar	OADStarPlanner.java	src	153	489	48	756	1446
com.cfar.swim.worldwind.planners.cgs.oadstar	package-info.java	src	0	29	0	1	30
com.cfar.swim.worldwind.planners.cgs.thetastar	ThetaStarPlanner.java	src	15	62	5	62	144
com.cfar.swim.worldwind.planners.cgs.thetastar	package-info.java	src	1	29	7	1	38
com.cfar.swim.worldwind.planners.managed	ManagedGoals.java	src	18	98	0	62	178
com.cfar.swim.worldwind.planners.managed	ManagedGridPlanner.java	src	52	232	4	264	552
com.cfar.swim.worldwind.planners.managed	ManagedPlanner.java	src	12	75	0	21	108
com.cfar.swim.worldwind.planners.managed	ManagedTreePlanner.java	src	51	226	4	258	539
com.cfar.swim.worldwind.planners.managed	package-info.java	src	0	29	0	1	30
com.cfar.swim.worldwind.planners.prm	package-info.java	src	0	29	0	1	30
com.cfar.swim.worldwind.planners.rrt.adrrt	ADRRTreePlanner.java	src	89	330	34	392	845
com.cfar.swim.worldwind.planners.rrt.adrrt	ADRRTreeWaypoint.java	src	10	64	0	23	97
com.cfar.swim.worldwind.planners.rrt.adrrt	package-info.java	src	0	29	0	1	30
com.cfar.swim.worldwind.planners.rrt.arrt	ARRTreePlanner.java	src	122	439	19	485	1065
com.cfar.swim.worldwind.planners.rrt.arrt	ARRTreeWaypoint.java	src	9	64	0	23	96
com.cfar.swim.worldwind.planners.rrt.arrt	package-info.java	src	0	29	0	1	30
com.cfar.swim.worldwind.planners.rrt.brrt	RRTreePlanner.java	src	116	381	51	451	999
com.cfar.swim.worldwind.planners.rrt.brrt	RRTreeWaypoint.java	src	17	84	0	41	142
com.cfar.swim.worldwind.planners.rrt.brrt	package-info.java	src	0	29	0	1	30
com.cfar.swim.worldwind.planners.rrt.drrt	DRRTreePlanner.java	src	95	346	29	431	901
com.cfar.swim.worldwind.planners.rrt.drrt	DRRTreeWaypoint.java	src	10	65	0	23	98
com.cfar.swim.worldwind.planners.rrt.drrt	package-info.java	src	0	29	0	1	30
com.cfar.swim.worldwind.planners.rrt.hrrt	HRRTreeAlgorithm.java	src	5	51	0	6	62
com.cfar.swim.worldwind.planners.rrt.hrrt	HRRTreePlanner.java	src	74	236	5	275	590
com.cfar.swim.worldwind.planners.rrt.hrrt	HRRTreeQuality.java	src	24	119	5	74	222
com.cfar.swim.worldwind.planners.rrt.hrrt	HRRTreeQualityVariant.java	src	3	44	0	5	52
com.cfar.swim.worldwind.planners.rrt.hrrt	HRRTreeWaypoint.java	src	16	100	2	58	176
com.cfar.swim.worldwind.planners.rrt.hrrt	package-info.java	src	0	29	0	1	30
com.cfar.swim.worldwind.planners.rrt.oadrrt	OADRRTreePlanner.java	src	155	494	48	776	1473
com.cfar.swim.worldwind.planners.rrt.oadrrt	package-info.java	src	0	29	0	1	30
com.cfar.swim.worldwind.planners.rrt.rrtstar	RRTreeStarPlanner.java	src	28	94	28	147	297
com.cfar.swim.worldwind.planners.rrt.rrtstar	package-info.java	src	0	29	0	1	30
com.cfar.swim.worldwind.planners.rrt	Extension.java	src	4	45	0	5	54
com.cfar.swim.worldwind.planners.rrt	Sampling.java	src	5	49	0	6	60
com.cfar.swim.worldwind.planners.rrt	Status.java	src	5	46	0	6	57
com.cfar.swim.worldwind.planners.rrt	Strategy.java	src	4	43	0	5	52
com.cfar.swim.worldwind.planners.rrt	package-info.java	src	0	29	0	1	30
com.cfar.swim.worldwind.planners	AbstractPlanner.java	src	46	207	1	183	437
com.cfar.swim.worldwind.planners	AnytimePlanner.java	src	9	87	0	10	106
com.cfar.swim.worldwind.planners	DynamicObstacleListener.java	src	9	61	0	10	80
com.cfar.swim.worldwind.planners	DynamicPlanner.java	src	4	48	0	5	57
com.cfar.swim.worldwind.planners	LifelongPlanner.java	src	5	47	0	6	58
com.cfar.swim.worldwind.planners	MissionLoader.java	src	14	64	4	58	140
com.cfar.swim.worldwind.planners	OnlinePlanner.java	src	14	88	0	16	118
com.cfar.swim.worldwind.planners	PlanRevisionListener.java	src	5	41	0	6	52
com.cfar.swim.worldwind.planners	Planner.java	src	20	135	11	27	193
com.cfar.swim.worldwind.planners	package-info.java	src	0	29	0	1	30
com.cfar.swim.worldwind.planning	CostInterval.java	src	31	154	0	105	290
com.cfar.swim.worldwind.planning	CostMap.java	src	5	38	0	9	52
com.cfar.swim.worldwind.planning	CostPolicy.java	src	5	54	0	14	73
com.cfar.swim.worldwind.planning	RiskPolicy.java	src	15	78	0	37	130
com.cfar.swim.worldwind.planning	TimeInterval.java	src	26	128	1	97	252
com.cfar.swim.worldwind.planning	Trajectory.java	src	45	184	0	166	395
com.cfar.swim.worldwind.planning	Waypoint.java	src	62	209	17	176	464
com.cfar.swim.worldwind.planning	WeightedCostInterval.java	src	18	129	0	51	198
com.cfar.swim.worldwind.planning	package-info.java	src	0	29	0	1	30
com.cfar.swim.worldwind.registries.aircraft	A320Properties.java	src	5	38	0	26	69
com.cfar.swim.worldwind.registries.aircraft	AircraftFactory.java	src	12	60	0	47	119
com.cfar.swim.worldwind.registries.aircraft	AircraftProperties.java	src	66	236	0	253	555
com.cfar.swim.worldwind.registries.aircraft	AircraftPropertiesBeanInfo.java	src	10	47	0	112	169
com.cfar.swim.worldwind.registries.aircraft	H135Properties.java	src	5	38	0	26	69
com.cfar.swim.worldwind.registries.aircraft	IrisProperties.java	src	5	38	0	26	69
com.cfar.swim.worldwind.registries.aircraft	package-info.java	src	0	29	0	1	30
com.cfar.swim.worldwind.registries.connections	DatalinkFactory.java	src	11	60	0	45	116
com.cfar.swim.worldwind.registries.connections	DatalinkProperties.java	src	15	73	0	44	132
com.cfar.swim.worldwind.registries.connections	DatalinkPropertiesBeanInfo.java	src	11	52	0	29	92
com.cfar.swim.worldwind.registries.connections	DronekitDatalinkProperties.java	src	18	80	0	51	149
com.cfar.swim.worldwind.registries.connections	DronekitDatalinkPropertiesBeanInfo.java	src	9	47	0	34	90
com.cfar.swim.worldwind.registries.connections	LiveSwimConnectionProperties.java	src	3	35	0	5	43
com.cfar.swim.worldwind.registries.connections	MavlinkDatalinkProperties.java	src	28	100	0	83	211
com.cfar.swim.worldwind.registries.connections	MavlinkDatalinkPropertiesBeanInfo.java	src	9	47	0	44	100
com.cfar.swim.worldwind.registries.connections	SimulatedDatalinkProperties.java	src	22	108	1	71	202
com.cfar.swim.worldwind.registries.connections	SimulatedDatalinkPropertiesBeanInfo.java	src	10	47	0	47	104
com.cfar.swim.worldwind.registries.connections	SimulatedSwimConnectionProperties.java	src	26	101	0	87	214
com.cfar.swim.worldwind.registries.connections	SimulatedSwimConnectionPropertiesBeanInfo.java	src	10	47	0	47	104
com.cfar.swim.worldwind.registries.connections	SwimConnectionFactory.java	src	9	61	1	54	125
com.cfar.swim.worldwind.registries.connections	SwimConnectionProperties.java	src	25	114	0	82	221
com.cfar.swim.worldwind.registries.connections	SwimConnectionPropertiesBeanInfo.java	src	11	52	0	53	116
com.cfar.swim.worldwind.registries.connections	package-info.java	src	0	29	0	1	30
com.cfar.swim.worldwind.registries.environments	EnvironmentBoundaryValidator.java	src	4	45	0	10	59
com.cfar.swim.worldwind.registries.environments	EnvironmentFactory.java	src	9	65	1	52	127
com.cfar.swim.worldwind.registries.environments	EnvironmentProperties.java	src	18	95	0	58	171
com.cfar.swim.worldwind.registries.environments	EnvironmentPropertiesBeanInfo.java	src	11	52	0	35	98
com.cfar.swim.worldwind.registries.environments	EnvironmentValidBoundaries.java	src	8	35	0	21	64
com.cfar.swim.worldwind.registries.environments	PlanningContinuumProperties.java	src	12	69	0	31	112
com.cfar.swim.worldwind.registries.environments	PlanningContinuumPropertiesBeanInfo.java	src	8	47	0	29	84
com.cfar.swim.worldwind.registries.environments	PlanningGridProperties.java	src	14	80	0	37	131
com.cfar.swim.worldwind.registries.environments	PlanningGridPropertiesBeanInfo.java	src	9	47	0	29	85
com.cfar.swim.worldwind.registries.environments	PlanningRoadmapProperties.java	src	3	35	0	5	43
com.cfar.swim.worldwind.registries.environments	package-info.java	src	0	29	0	1	30
com.cfar.swim.worldwind.registries.managers	AbstractManagerDeliberationValidator.java	src	4	45	0	11	60
com.cfar.swim.worldwind.registries.managers	AbstractManagerProperties.java	src	50	278	0	211	539
com.cfar.swim.worldwind.registries.managers	AbstractManagerPropertiesBeanInfo.java	src	11	52	0	95	158
com.cfar.swim.worldwind.registries.managers	AbstractManagerValidDeliberation.java	src	8	36	0	21	65
com.cfar.swim.worldwind.registries.managers	HeuristicManagerProperties.java	src	3	35	0	5	43
com.cfar.swim.worldwind.registries.managers	ManagerFactory.java	src	9	65	0	86	160
com.cfar.swim.worldwind.registries.managers	ManagerProperties.java	src	28	194	0	33	255

com.cfar.swim.worldwind.registries.managers	SmacManagerProperties.java	src	25	100	0	77	202
com.cfar.swim.worldwind.registries.managers	SmacManagerPropertiesBeanInfo.java	src	10	47	0	47	104
com.cfar.swim.worldwind.registries.managers	package-info.java	src	0	29	0	1	30
com.cfar.swim.worldwind.registries.planners.cgs	ADStarProperties.java	src	19	115	0	70	204
com.cfar.swim.worldwind.registries.planners.cgs	ADStarPropertiesBeanInfo.java	src	10	52	0	32	94
com.cfar.swim.worldwind.registries.planners.cgs	ARAStarProperties.java	src	22	138	0	89	249
com.cfar.swim.worldwind.registries.planners.cgs	ARAStarPropertiesBeanInfo.java	src	10	52	0	45	107
com.cfar.swim.worldwind.registries.planners.cgs	ARAStarQualityValidator.java	src	4	45	0	11	60
com.cfar.swim.worldwind.registries.planners.cgs	ARAStarValidQuality.java	src	8	35	0	21	64
com.cfar.swim.worldwind.registries.planners.cgs	ForwardAStarProperties.java	src	6	45	0	14	65
com.cfar.swim.worldwind.registries.planners.cgs	OADStarDeliberationValidator.java	src	4	45	0	11	60
com.cfar.swim.worldwind.registries.planners.cgs	OADStarProperties.java	src	37	218	0	151	406
com.cfar.swim.worldwind.registries.planners.cgs	OADStarPropertiesBeanInfo.java	src	9	47	0	71	127
com.cfar.swim.worldwind.registries.planners.cgs	OADStarValidDeliberation.java	src	8	36	0	21	65
com.cfar.swim.worldwind.registries.planners.cgs	ThetaStarProperties.java	src	6	45	0	13	64
com.cfar.swim.worldwind.registries.planners.cgs	package-info.java	src	0	29	0	1	30
com.cfar.swim.worldwind.registries.planners.prm	package-info.java	src	0	29	0	1	30
com.cfar.swim.worldwind.registries.planners.rrt	ADRRTreeProperties.java	src	16	91	0	49	156
com.cfar.swim.worldwind.registries.planners.rrt	ADRRTreePropertiesBeanInfo.java	src	9	52	0	31	92
com.cfar.swim.worldwind.registries.planners.rrt	ARRTreeProperties.java	src	26	141	0	99	266
com.cfar.swim.worldwind.registries.planners.rrt	ARRTreePropertiesBeanInfo.java	src	10	53	0	50	113
com.cfar.swim.worldwind.registries.planners.rrt	ARRTreeQualityValidator.java	src	4	45	0	11	60
com.cfar.swim.worldwind.registries.planners.rrt	ARRTreeValidQuality.java	src	8	35	0	21	64
com.cfar.swim.worldwind.registries.planners.rrt	DRRTreeProperties.java	src	16	92	0	49	157
com.cfar.swim.worldwind.registries.planners.rrt	DRRTreePropertiesBeanInfo.java	src	9	48	0	28	85
com.cfar.swim.worldwind.registries.planners.rrt	HRRTreeProperties.java	src	25	119	0	86	230
com.cfar.swim.worldwind.registries.planners.rrt	HRRTreePropertiesBeanInfo.java	src	10	53	0	50	113
com.cfar.swim.worldwind.registries.planners.rrt	OADRRTreeDeliberationValidator.java	src	4	45	0	11	60
com.cfar.swim.worldwind.registries.planners.rrt	OADRRTreeProperties.java	src	37	218	0	151	406
com.cfar.swim.worldwind.registries.planners.rrt	OADRRTreePropertiesBeanInfo.java	src	9	47	0	71	127
com.cfar.swim.worldwind.registries.planners.rrt	OADRRTreeValidDeliberation.java	src	8	36	0	21	65
com.cfar.swim.worldwind.registries.planners.rrt	RRTreeProperties.java	src	35	185	0	138	358
com.cfar.swim.worldwind.registries.planners.rrt	RRTreePropertiesBeanInfo.java	src	11	53	0	69	133
com.cfar.swim.worldwind.registries.planners.rrt	package-info.java	src	0	29	0	1	30
com.cfar.swim.worldwind.registries.planners	AbstractPlannerProperties.java	src	21	111	1	71	204
com.cfar.swim.worldwind.registries.planners	AbstractPlannerPropertiesBeanInfo.java	src	11	52	0	35	98
com.cfar.swim.worldwind.registries.planners	AnytimePlannerProperties.java	src	8	77	0	9	94
com.cfar.swim.worldwind.registries.planners	DynamicPlannerProperties.java	src	4	48	0	5	57
com.cfar.swim.worldwind.registries.planners	OnlinePlannerProperties.java	src	18	146	0	19	183
com.cfar.swim.worldwind.registries.planners	PlannerFactory.java	src	9	65	2	287	363
com.cfar.swim.worldwind.registries.planners	PlannerProperties.java	src	8	62	0	13	83
com.cfar.swim.worldwind.registries.planners	package-info.java	src	0	29	0	1	30
com.cfar.swim.worldwind.registries	AbstractFactory.java	src	19	108	0	52	179
com.cfar.swim.worldwind.registries	Factory.java	src	6	60	0	7	73
com.cfar.swim.worldwind.registries	FactoryProduct.java	src	6	53	0	6	65
com.cfar.swim.worldwind.registries	Properties.java	src	6	57	0	9	72
com.cfar.swim.worldwind.registries	PropertiesBeanInfo.java	src	12	60	0	33	105
com.cfar.swim.worldwind.registries	Registry.java	src	11	66	0	29	106
com.cfar.swim.worldwind.registries	Specification.java	src	71	120	0	220	411
com.cfar.swim.worldwind.registries	package-info.java	src	0	29	0	1	30
com.cfar.swim.worldwind.render.airspaces	ObstacleBox.java	src	59	240	24	252	575
com.cfar.swim.worldwind.render.airspaces	ObstacleCylinder.java	src	54	229	9	220	512
com.cfar.swim.worldwind.render.airspaces	ObstacleSphere.java	src	51	226	9	207	493
com.cfar.swim.worldwind.render.airspaces	package-info.java	src	0	29	0	1	30
com.cfar.swim.worldwind.render.annotations	ControlAnnotation.java	src	32	134	0	107	273
com.cfar.swim.worldwind.render.annotations	DepictionAnnotation.java	src	7	64	0	41	112
com.cfar.swim.worldwind.render.annotations	package-info.java	src	0	29	0	1	30
com.cfar.swim.worldwind.render	Obstacle.java	src	11	75	1	16	103
com.cfar.swim.worldwind.render	ObstacleColor.java	src	6	42	0	71	119
com.cfar.swim.worldwind.render	ObstacleCylinder.java	src	48	228	10	202	488
com.cfar.swim.worldwind.render	ObstaclePath.java	src	41	202	3	184	430
com.cfar.swim.worldwind.render	ThresholdRenderable.java	src	5	47	0	6	58
com.cfar.swim.worldwind.render	TimedRenderable.java	src	6	47	0	7	60
com.cfar.swim.worldwind.render	VerticalCylinder.java	src	16	96	0	57	169
com.cfar.swim.worldwind.render	package-info.java	src	0	29	0	1	30
com.cfar.swim.worldwind.session	ObstacleManager.java	src	18	109	0	20	147
com.cfar.swim.worldwind.session	ObstacleProvider.java	src	5	51	0	6	62
com.cfar.swim.worldwind.session	Scenario.java	src	204	637	34	1018	1893
com.cfar.swim.worldwind.session	Session.java	src	102	287	11	384	784
com.cfar.swim.worldwind.session	SessionManager.java	src	11	56	0	25	92
com.cfar.swim.worldwind.session	Setup.java	src	22	96	1	58	177
com.cfar.swim.worldwind.session	package-info.java	src	0	29	0	1	30
com.cfar.swim.worldwind.terrain	Terrain.java	src	27	81	2	110	220
com.cfar.swim.worldwind.terrain	package-info.java	src	0	29	0	1	30
com.cfar.swim.worldwind.tests	AircraftTest.java	test	13	38	6	41	98
com.cfar.swim.worldwind.tests	BoxTest.java	test	24	44	3	137	208
com.cfar.swim.worldwind.tests	CriticalityTest.java	test	27	35	48	170	280
com.cfar.swim.worldwind.tests	CubicGridTest.java	test	22	44	0	78	144
com.cfar.swim.worldwind.tests	DatalinkTest.java	test	22	45	43	73	183
com.cfar.swim.worldwind.tests	DepictionTest.java	test	9	41	0	27	77
com.cfar.swim.worldwind.tests	ObstaclesTest.java	test	14	41	1	54	110
com.cfar.swim.worldwind.tests	PhysicsTest.java	test	25	41	18	93	177
com.cfar.swim.worldwind.tests	PlannerTest.java	test	19	38	1	89	147
com.cfar.swim.worldwind.tests	PlanningGridTest.java	test	20	41	2	140	203
com.cfar.swim.worldwind.tests	PoliciesTest.java	test	18	41	6	95	160
com.cfar.swim.worldwind.tests	RegularGridTest.java	test	54	35	313	180	582
com.cfar.swim.worldwind.tests	SwimConnectionTest.java	test	6	38	1	16	61
com.cfar.swim.worldwind.tests	UpdaterTest.java	test	5	35	1	9	50
com.cfar.swim.worldwind.tests	WaypointTest.java	test	25	37	18	69	149
com.cfar.swim.worldwind.tests	package-info.java	test	0	29	0	1	30
com.cfar.swim.worldwind.tracks	AircraftAttitude.java	src	14	78	0	38	130
com.cfar.swim.worldwind.tracks	AircraftTrack.java	src	18	91	1	59	169
com.cfar.swim.worldwind.tracks	AircraftTrackError.java	src	23	106	0	78	207
com.cfar.swim.worldwind.tracks	AircraftTrackPoint.java	src	31	191	0	154	376
com.cfar.swim.worldwind.tracks	AircraftTrackPointError.java	src	20	94	0	69	183
com.cfar.swim.worldwind.tracks	AircraftTrackSegment.java	src	6	43	0	16	65
com.cfar.swim.worldwind.tracks	package-info.java	src	0	29	0	1	30
com.cfar.swim.worldwind.util	Customizable.java	src	5	46	0	6	57
com.cfar.swim.worldwind.util	Depictable.java	src	6	50	0	7	63
com.cfar.swim.worldwind.util	Depiction.java	src	30	125	6	132	293
com.cfar.swim.worldwind.util	Designatable.java	src	4	45	0	5	54
com.cfar.swim.worldwind.util	Enableable.java	src	5	46	0	6	57
com.cfar.swim.worldwind.util	Identifiable.java	src	3	39	0	4	46
com.cfar.swim.worldwind.util	ResourceBundleLoader.java	src	15	62	0	45	122
com.cfar.swim.worldwind.util	package-info.java	src	0	29	0	1	30
com.cfar.swim.worldwind.weather	TropicalCyclone.java	src	7	29	0	20	56
com.cfar.swim.worldwind.weather	WeatherSymbolMap.java	src	5	33	0	9	47
com.cfar.swim.worldwind.weather	package-info.java	src	0	29	0	1	30
gov.nasa.worldwind.geom	TransformationMatrix.java	src	17	94	1	65	177
gov.nasa.worldwind.geom	package-info.java	src	0	29	0	1	30

| 53 package(s) | 339 file(s) | java | 7456 | 32279 | 2131 | 31206 | 73072 |

B.1.2 com.uvic-cfar.swim:worldwind-ui:1.0.0-alpha3-SNAPSHOT

Package Name	File Name	Type	Blank	JavaDoc	Comment	Code	Total
com.cfar.swim.worldwind.ui.environment	EnvironmentPresenter.java	src	26	115	4	138	283
com.cfar.swim.worldwind.ui.environment	EnvironmentView.java	src	5	42	0	9	56
com.cfar.swim.worldwind.ui.environment	package-info.java	src	0	29	0	1	30
com.cfar.swim.worldwind.ui.plan.waypoint	WaypointDialog.java	src	32	85	0	127	244
com.cfar.swim.worldwind.ui.plan.waypoint	WaypointPresenter.java	src	8	43	0	20	71
com.cfar.swim.worldwind.ui.plan.waypoint	WaypointView.java	src	2	35	0	4	41
com.cfar.swim.worldwind.ui.plan.waypoint	package-info.java	src	0	29	0	1	30
com.cfar.swim.worldwind.ui.plan	PlanPresenter.java	src	70	185	3	291	549
com.cfar.swim.worldwind.ui.plan	PlanView.java	src	5	42	0	9	56
com.cfar.swim.worldwind.ui.plan	package-info.java	src	0	29	0	1	30
com.cfar.swim.worldwind.ui.planner	PlannerAlert.java	src	24	53	0	92	169
com.cfar.swim.worldwind.ui.planner	PlannerAlertResult.java	src	7	46	0	22	75
com.cfar.swim.worldwind.ui.planner	PlannerCountdownAlert.java	src	5	47	0	42	94
com.cfar.swim.worldwind.ui.planner	PlannerPresenter.java	src	30	64	0	131	225
com.cfar.swim.worldwind.ui.planner	PlannerView.java	src	2	35	0	4	41
com.cfar.swim.worldwind.ui.planner	package-info.java	src	0	29	0	1	30
com.cfar.swim.worldwind.ui.scenario	ScenarioPresenter.java	src	35	140	0	258	433
com.cfar.swim.worldwind.ui.scenario	ScenarioView.java	src	5	42	0	9	56
com.cfar.swim.worldwind.ui.scenario	package-info.java	src	0	29	0	1	30
com.cfar.swim.worldwind.ui.setup	SetupDialog.java	src	32	53	4	136	225
com.cfar.swim.worldwind.ui.setup	SetupModel.java	src	22	98	0	65	185
com.cfar.swim.worldwind.ui.setup	SetupPresenter.java	src	51	151	0	332	534
com.cfar.swim.worldwind.ui.setup	SetupView.java	src	11	72	0	34	117
com.cfar.swim.worldwind.ui.setup	package-info.java	src	0	29	0	1	30
com.cfar.swim.worldwind.ui.splash	SplashPresenter.java	src	10	44	0	28	82
com.cfar.swim.worldwind.ui.splash	SplashView.java	src	5	42	0	9	56
com.cfar.swim.worldwind.ui.splash	package-info.java	src	0	29	0	1	30
com.cfar.swim.worldwind.ui.swim	SwimPresenter.java	src	27	90	0	167	284
com.cfar.swim.worldwind.ui.swim	SwimView.java	src	5	42	0	9	56
com.cfar.swim.worldwind.ui.swim	package-info.java	src	0	29	0	1	30
com.cfar.swim.worldwind.ui.terrain	TerrainPresenter.java	src	25	84	0	126	235
com.cfar.swim.worldwind.ui.terrain	TerrainView.java	src	5	42	0	9	56
com.cfar.swim.worldwind.ui.terrain	package-info.java	src	0	29	0	1	30
com.cfar.swim.worldwind.ui.threshold	ThresholdPresenter.java	src	10	57	0	37	104
com.cfar.swim.worldwind.ui.threshold	ThresholdView.java	src	5	42	0	9	56
com.cfar.swim.worldwind.ui.threshold	package-info.java	src	0	29	0	1	30
com.cfar.swim.worldwind.ui.time	TimePresenter.java	src	25	102	4	126	257
com.cfar.swim.worldwind.ui.time	TimeView.java	src	5	42	0	9	56
com.cfar.swim.worldwind.ui.time	package-info.java	src	0	29	0	1	30
com.cfar.swim.worldwind.ui.timer	TimerPresenter.java	src	21	90	0	102	213
com.cfar.swim.worldwind.ui.timer	TimerView.java	src	6	42	0	9	57
com.cfar.swim.worldwind.ui.timer	package-info.java	src	0	29	0	1	30
com.cfar.swim.worldwind.ui.util	ResourceBundleLoader.java	src	5	40	0	10	55
com.cfar.swim.worldwind.ui.util	package-info.java	src	0	29	0	1	30
com.cfar.swim.worldwind.ui.world	ViewMode.java	src	1	35	0	11	47
com.cfar.swim.worldwind.ui.world	WorldMode.java	src	1	35	0	15	51
com.cfar.swim.worldwind.ui.world	WorldModel.java	src	39	225	0	192	456
com.cfar.swim.worldwind.ui.world	WorldPresenter.java	src	219	508	53	1425	2205
com.cfar.swim.worldwind.ui.world	WorldService.java	src	3	38	9	5	55
com.cfar.swim.worldwind.ui.world	WorldView.java	src	5	42	0	9	56
com.cfar.swim.worldwind.ui.world	package-info.java	src	0	29	0	1	30
com.cfar.swim.worldwind.ui	Main.java	src	3	40	1	7	51
com.cfar.swim.worldwind.ui	SplashScreenLoader.java	src	9	52	0	33	94
com.cfar.swim.worldwind.ui	WorldwindPlanner.java	src	7	51	11	38	107
com.cfar.swim.worldwind.ui	package-info.java	src	0	29	0	1	30
15 package(s)	55 file(s)	java	813	3568	89	4123	8593

B.1.3 com.uvic-cfar.swim:worlwdind-jaxb:1.0.0-alpha1-SNAPSHOT

Package Name	File Name	Type	Blank	JavaDoc	Comment	Code	Total
com.cfar.swim.worldwind.jaxb.adapters	AircraftAdapter.java	src	14	28	29	70	141
com.cfar.swim.worldwind.jaxb.adapters	BoxAdapter.java	src	9	28	29	31	97
com.cfar.swim.worldwind.jaxb.adapters	CapabilitiesAdapter.java	src	11	28	29	47	115
com.cfar.swim.worldwind.jaxb.adapters	CostIntervalAdapter.java	src	13	28	29	39	109
com.cfar.swim.worldwind.jaxb.adapters	CylinderAdapter.java	src	11	28	29	28	96
com.cfar.swim.worldwind.jaxb.adapters	DatalinkAdapter.java	src	12	28	31	36	107
com.cfar.swim.worldwind.jaxb.adapters	DatumAdapter.java	src	11	28	29	43	111
com.cfar.swim.worldwind.jaxb.adapters	DepictionAdapter.java	src	16	33	29	63	141
com.cfar.swim.worldwind.jaxb.adapters	DronekitDatalinkAdapter.java	src	9	28	31	20	88
com.cfar.swim.worldwind.jaxb.adapters	DurationAdapter.java	src	5	28	29	14	76
com.cfar.swim.worldwind.jaxb.adapters	EnvironmentAdapter.java	src	13	33	29	40	115
com.cfar.swim.worldwind.jaxb.adapters	GeometricBoxAdapter.java	src	11	33	29	36	109
com.cfar.swim.worldwind.jaxb.adapters	GlobeAdapter.java	src	8	28	29	25	90
com.cfar.swim.worldwind.jaxb.adapters	LiveSwimAdapter.java	src	6	28	31	14	79
com.cfar.swim.worldwind.jaxb.adapters	LocationAdapter.java	src	9	28	29	20	86
com.cfar.swim.worldwind.jaxb.adapters	ObstacleBoxAdapter.java	src	11	28	29	37	105
com.cfar.swim.worldwind.jaxb.adapters	ObstacleCylinderAdapter.java	src	13	28	29	41	111
com.cfar.swim.worldwind.jaxb.adapters	ObstacleSphereAdapter.java	src	13	28	29	36	106
com.cfar.swim.worldwind.jaxb.adapters	ObstaclesAdapter.java	src	11	28	29	41	109
com.cfar.swim.worldwind.jaxb.adapters	PlannerAdapter.java	src	27	110	29	280	446
com.cfar.swim.worldwind.jaxb.adapters	PlanningContinuumAdapter.java	src	11	33	29	34	107
com.cfar.swim.worldwind.jaxb.adapters	PlanningGridAdapter.java	src	14	33	29	45	121
com.cfar.swim.worldwind.jaxb.adapters	PositionAdapter.java	src	6	28	29	20	83
com.cfar.swim.worldwind.jaxb.adapters	ScenarioAdapter.java	src	12	28	29	66	135

com.cfar.swim.worldwind.jaxb.adapters	SectorAdapter.java	src	10	28	29	25	92
com.cfar.swim.worldwind.jaxb.adapters	SimulatedDatalinkAdapter.java	src	10	28	31	31	100
com.cfar.swim.worldwind.jaxb.adapters	SimulatedSwimAdapter.java	src	9	28	29	25	91
com.cfar.swim.worldwind.jaxb.adapters	SphereAdapter.java	src	11	28	29	27	95
com.cfar.swim.worldwind.jaxb.adapters	SwimAdapter.java	src	13	28	29	58	128
com.cfar.swim.worldwind.jaxb.adapters	TimeAdapter.java	src	6	28	29	19	82
com.cfar.swim.worldwind.jaxb.adapters	TimeIntervalAdapter.java	src	10	28	29	20	87
com.cfar.swim.worldwind.jaxb.adapters	TrackErrorAdapter.java	src	11	28	29	27	95
com.cfar.swim.worldwind.jaxb.adapters	TrackPointErrorAdapter.java	src	10	28	29	22	89
com.cfar.swim.worldwind.jaxb.adapters	TrajectoryAdapter.java	src	11	28	29	35	103
com.cfar.swim.worldwind.jaxb.adapters	WaypointAdapter.java	src	11	28	29	58	126
com.cfar.swim.worldwind.jaxb.adapters	WaypointsAdapter.java	src	11	28	29	24	92
com.cfar.swim.worldwind.jaxb.adapters	package-info.java	src	0	0	29	1	30
com.cfar.swim.worldwind.jaxb.tests	RoundtripTest.java	test	16	0	32	37	85
com.cfar.swim.worldwind.jaxb.tests	package-info.java	test	0	0	29	1	30
com.cfar.swim.worldwind.jaxb	ScenarioMarshaller.java	src	9	28	29	31	97
com.cfar.swim.worldwind.jaxb	ScenarioUnmarshaller.java	src	14	49	29	48	140
com.cfar.swim.worldwind.jaxb	package-info.java	src	0	0	29	1	30
3 package(s)	42 file(s)	java	438	1192	1229	1616	4475