NASA/TM–2016–219196

The Stratway Program for Strategic Conflict Resolution: User's Guide

George E. Hagen, Ricky W. Butler, Jeffrey M. Maddalon
Langley Research Center, Hampton, Virginia

May 2016

NASA/TM–2016–219196

The Stratway Program for Strategic Conflict Resolution: User's Guide

George E. Hagen, Ricky W. Butler, Jeffrey M. Maddalon
Langley Research Center, Hampton, Virginia

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

May 2016

Available from:

NASA STI Program / Mail Stop 148
NASA Langley Research Center
Hampton, VA 23681-2199
Fax: 757-864-6500

Abstract

Stratway is a strategic conflict detection and resolution program. It provides both intent-based conflict detection and conflict resolution for a single ownship in the presence of multiple traffic aircraft and weather cells defined by moving polygons. It relies on a set of heuristic search strategies to solve conflicts. These strategies are user configurable through multiple parameters. The program can be called from other programs through an application program interface (API) and can also be executed from a command line.

Contents

1 Introduction

Stratway is an aircraft-independent, strategic conflict resolution program. This program receives the planned routes or trajectories of nearby aircraft and makes adjustments to its own route to resolve conflicts with other aircraft or weather. Unlike a full-featured strategic resolver, such as the Autonomous Operations Planner [6], it does not seek to produce a fuel-efficient, nearly optimal resolution. Instead, Stratway resolves conflicts by moving points in the vicinity of a conflict. It presupposes that the input route is already nearly optimal and therefore an appropriate resolution is just a small deviation from the original path. The Stratway resolver relies exclusively on kinematic calculations and therefore does not depend on the dynamic performance of any specific aircraft. However, the output resolution can be controlled by user-specified acceleration and limit parameters.

An important goal of the Stratway project is to explore the verification and validation issues associated with strategic conflict resolution. Therefore, Stratway has been developed in a manner that makes it possible to establish that a generated resolution satisfies key safety properties. However, a proof that a resolution will always be produced cannot been achieved for this program. This is an unfortunate consequence of the fact that Stratway uses a set of heuristic strategies to search for a conflict-free route. Because of the inherent need for heuristic search[1], a full verification of a strategic solver is currently beyond the state of the art in verification. However, the formal verification of several key components, e.g., the conflict detection algorithm, is practical and is being pursued in the project.

The primary purposes of this paper are (1) to describe the resolution strategies (section 4), (2) to describe the application programming interface (API) of Stratway (section 6), and (3) to provide examples of its use. The resolution strategies are based on heuristic search techniques and serve as the intellectual core of Stratway. The API is valuable to programmers who wish to integrate Stratway into another program, though Stratway can also be run as a stand alone program, both as a command-line tool and as part of a sophisticated graphical visualization with tools to create and modify the routes of aircraft. These user-level tools are described in [5]. Stratway is part of the NASA Langley FormalATM software distribution, which also includes basic utility functions as well as implementations of the ACCoRD framework [9].

Version This document is current as of Stratway version v2.5.2.

2 Flight Plans and Trajectories

The planned route of an aircraft used in Stratway can either be a flight plan or a trajectory. The notion of a flight plan has several different meanings in the national airspace system today:

[1]Given the enormous state space for strategic resolution, it is necessary to perform some form of heuristic search. All practical strategic resolution approaches known to the authors use heuristic search.

1. a sequence of 2-D waypoints (horizontal positions only)
2. a sequence of 2-D waypoints augmented with constraints on altitude and ground speed—usually in the form of a required time of arrival (RTA)
3. a sequence of 3-D waypoints (horizontal and vertical position) with a nominal ground speed
4. a sequence of 4-D waypoints (3-D position and time)

The most common form of a flight plan today is described by the second item. However, this form of flight plan does not contain enough information to perform conflict detection. Conflict detection requires not only a path, but also an indication of when the aircraft is at a location. In addition, aircraft can be separated vertically, instead of horizontally, so aircraft altitude is also critical for determining the presence or absence of a conflict. For these reasons, Stratway requires that a flight plan be completely specified as a sequence of 4-D waypoints (the fourth item). An example flight plan is illustrated in Figure 1.

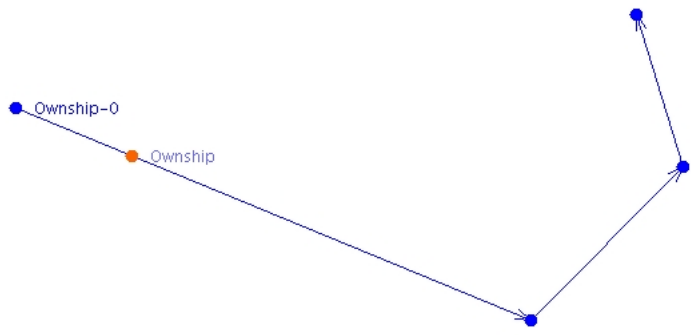

Figure 1. Sample Flight Plan

Stratway can also operate on a more accurate planned route that is referred to as a trajectory or a kinematic plan. In a kinematic plan, more information is included at each velocity change, particularly information about the aircraft's acceleration. This additional information is referred to as trajectory change points (TCPs) and approximates the information available in the standard for communicating aircraft intent [7]. A kinematic plan generated from the plan in Figure 1 is shown in Figure 2. The vertex points in the flight plan have been replaced with two new TCPs that are colored yellow in Figure 2. These points are a beginning of turn (BOT) point, and an end of turn (EOT) point.

2.1 Terminology

Stratway deals with 4-D points (constraints) and the lines that connect them. These 4-D points are called *points*, which can represent either traditional waypoints in a

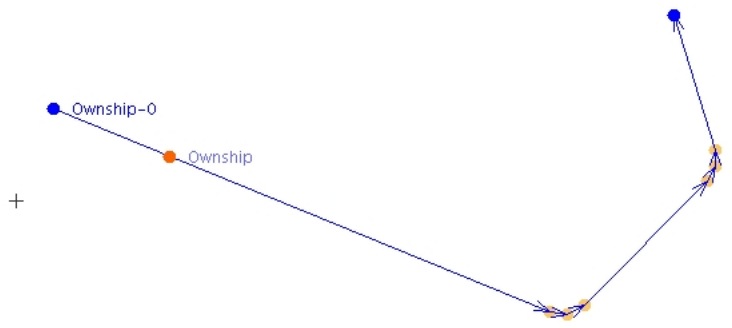

Figure 2. Sample Kinematic Plan

flight plan or the trajectory change points generated by a flight management system (FMS). Each line between two subsequent points is called a *segment* (occasionally these are referred to as *legs*). We call a sequence of these points (and their inferred segments) associated with a particular aircraft a *plan*, which abstracts both traditional flight plans and trajectories. Some points can be designated as *trajectory change points* (TCPs). If a plan contains such points, we refer to the plan as a *kinematic* plan (or *trajectory*) and if they do not contain any TCPs then we say that the plan is *linear*.

It is expected that plan information in actual or simulated craft will be communicated via Automatic Dependent Surveillance-Broadcast (ADS-B) or other similar means, that is based on Global Positioning System (GPS) data. Therefore, highly accurate position information will be available, from which accurate groundspeed can be calculated that is independent of airspeed. The objects representing plans (the `Plan` class) contain only 4-D point information, so changes in the specified time for a point results in a speed change on the segment leading up to that point. The Stratway visualization allows users to easily draw plans in a graphical manner.

2.2 Trajectory Change Points (TCPs)

The Stratway resolution strategies rely on an internal trajectory generator that can be configured to approximate the characteristics of different aircraft types [4]. We often refer to a Stratway generated trajectory as a *kinematic plan.* In a kinematic plan, the following types of TCPs are allowed, representing areas of acceleration:

`BOT`	beginning of turn
`EOT`	end of turn
`BGS`	beginning of ground speed acceleration
`EGS`	end of ground speed acceleration
`BVS`	beginning of vertical speed acceleration
`EVS`	end of vertical speed acceleration

Every BOT point must be followed by an EOT point and there can be no overlaps with other turns. Every BGS point must be followed by an EGS point and there can be no overlaps with other ground speed accelerations. Every BVS point must be followed by an EVS and there can be no overlaps with other vertical speed accelerations.

The region between a begin TCP and the corresponding end TCP are constant acceleration regions. The TCPs are annotated with the acceleration rate to enable simple calculation of the speed and velocity for all points within these regions. Stratway produces resolutions that include these TCPs but the resolution plans are also easily converted back to linear plans, if that is desired.

2.3 Construction of Plans

There are two ways to construct a plan:

- Develop each waypoint separately and deliver them to a Stratway object via API methods (Section 6),
- Create a comma separated values (CSV) text file and load one or more complete plans (Section 6.3).

The basic textual representation of plans, for example, in a CSV file, is simple, as shown in Figure 3. This text file defines two plans: one for aircraft `Ownship` and the other for aircraft `Traffic`. The columns provide latitudes and longitudes in units of degrees, altitudes in feet, and time in seconds.

```
NAME,      lat,      long,       altitude,    time
Ownship,   0.2994,   -0.6298,    5000.0,      0.00
Ownship,   0.2994,   -0.1579,    5000.0,      391.36
Ownship,   0.2034,    0.4806,    5000.0,      900.42
Ownship,  -0.0102,    1.1440,    5000.0,      1800.00
Ownship,   0.0663,    1.7125,    5000.0,      2535.49

Traffic,  -0.2526,   -0.6266,    5000.0,      0.00
Traffic,  -0.2364,   -0.1876,    5000.0,      364.29
Traffic,  -0.1040,    0.3438,    5000.0,      812.88
Traffic,   0.3402,    0.8284,    5000.0,      1303.99
Traffic,   0.9189,    1.2190,    5000.0,      1794.31
```

Figure 3: CSV formated plan example.

There is also a textual version of kinematic plans that can be used, but it is not recommended that these be constructed manually. It is very easy to construct inconsistent plans in this way because there are many fields in a kinematic plan that have a strict mathematical relationship to other fields. The API contains a set of functions that can automatically construct a kinematic plan from a linear plan. These are described in a separate report [4].

3 Approach to Strategic Conflict Resolution

A key concept present in many future air traffic management concepts (including NextGen) is Trajectory Based Operations (TBO), which is especially concerned with high-altitude cruise operations in en route airspace. The shift from today's clearance-based system to a trajectory-based system should enable aircraft to fly flight paths created by full performance-based navigation systems that take both operator preferences and optimal airspace system performance into consideration [3]. But this increased flexibility and performance comes at a price—airspace conflicts will frequently arise and must be safely resolved in a timely manner. Many advocates for TBO believe that this will require the exchange of trajectory information in real time [2]. The particular details about this information exchange is still very much a research issue. However, some progress has been made in this area by the RTCA in the development of the DO-242A minimum aviation system performance standards for Automatic Dependent Surveillance-Broadcast (ADS-B). In particular, Appendix A provides "Intent Guidance Material for Future ADS-B Intent Broadcast". Nevertheless, there are many competing ideas and there is no consensus about how this should be done.

Stratway is a strategic conflict detection and resolution program that has the following goals:

- Usable for any aircraft type, independent of any particular flight management system,
- Key functions are formally verified using the Prototype Verification System (PVS) theorem prover, so that a strong safety case can be constructed around its use,
- Operate on trajectories and flight plans that can be communicated in an efficient manner,
- implemented in both Java and C++.

In Figure 4 we see a complex traffic scenario where the ownship's plan is in conflict with two other aircraft; Figure 5 shows the Stratway resolution. The Stratway track solution removes both conflicts.

4 Conflict Resolution

The Stratway program consists of four basic strategies for resolving conflicts: `Track`, `Vertical`, `Speed`, and `SideStep`. The behavior of these strategies can be controlled by user-settable parameters.

4.1 Stratway Strategies

Each of the Stratway strategies uses iterative techniques to search for solutions. By default, they are applied in a predetermined order. Candidate solutions are

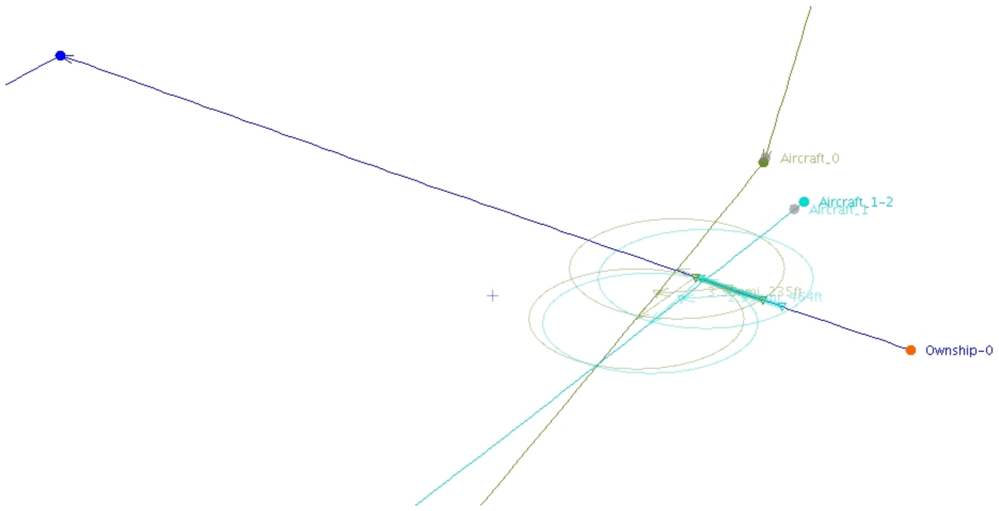

Figure 4: Complex Traffic Scenario With Double Conflict

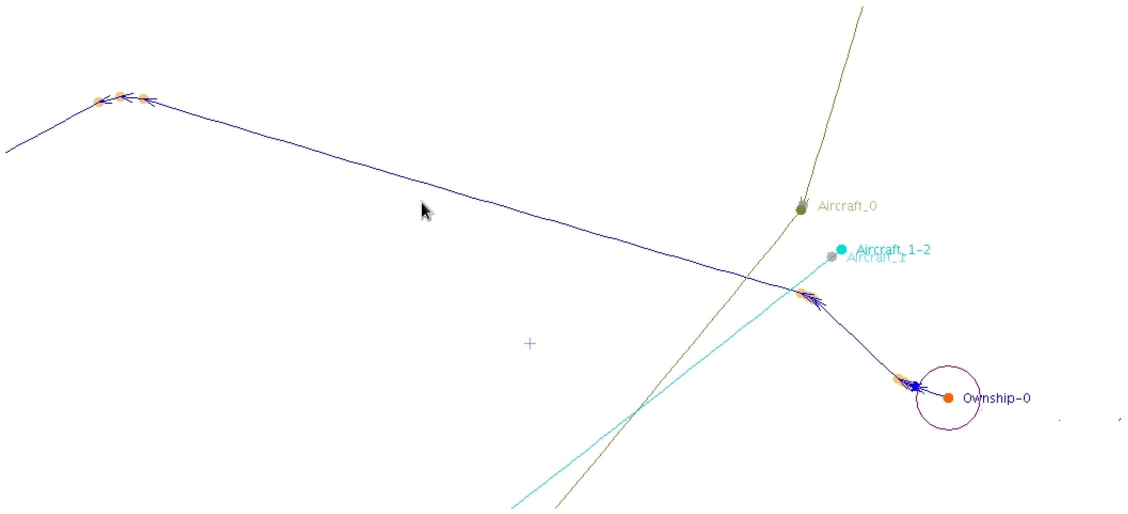

Figure 5: Complex Traffic Scenario With Double Conflict, Resolved

kinematic plans generated based on user-provided physical limits (acceptable acceleration values, limits on speeds, etc.). If the candidate solution is conflict free, no further solutions are sought, and `resolveConflicts()` returns this solution. There are three key properties that the Stratway strategies must achieve:

- The location in the plan where the ownship is currently located cannot move
- The velocity vector at the point where the ownship is currently located cannot change. However, an acceleration must start at or after this point.
- If the program provides a solution, the detected conflict will have been resolved and no new conflicts will have been introduced into earlier parts of the plan. Conflicts beyond the detection threshold may remain in the plan.

4.1.1 The Track Strategy

The `Track` strategy seeks to resolve the conflict through a sequence of turns without changing the current vertical speed of the aircraft. The track strategy begins by removing points in the vicinity of the conflict from the ownship plan. We call this *isolating the conflict*. This isolation is illustrated in Figures 6 and 7. This step is especially useful for a plan that has many closely-spaced points.

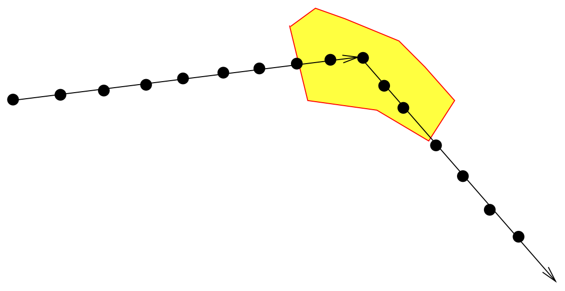

Figure 6: Before `isolateConflict()`

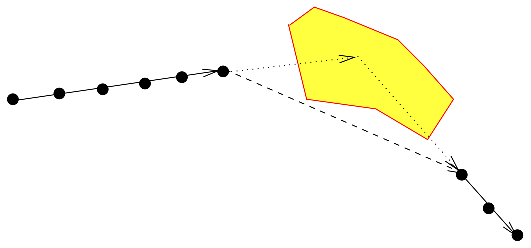

Figure 7: After `isolateConflict()`

The next step is to construct a linear plan that (1) starts at the first isolation point, (2) turns away from the conflict, (3) continues parallel to the line connecting the isolation points, and then (4) turns back to the final isolation point. The template for the track strategy is shown in Figure 8. In this figure isolation points are shown as hexagons. This template is varied by iterating over parameters that define its structure. Three aspects of this pattern are varied in nested for loops: the target track angle (`targetTrk`) from point 1 to point 2, the length of the segment from point 1 to point 2 (`LEG1`), and the length of the segment from point 2 to point three (`LEG2`).

The location of point 1 is calculated first. Using a kinematic turn function (that is provided the speed into the turn and an aircraft bank angle), the location where the velocity vector first points in the direction of the target track (i.e., `targetTrk`), is calculated, resulting in point 1. Then the vertex point 0 is computed as an intersection point. Point 2 is calculated as a linear projection from point 1 with length (`LEG1`). Next point 3 is computed from length (`LEG2`) and the direction of the line between the hexagonal isolation points. Point 4 is then calculated using the final isolation point: this is the beginning of a turn that results in the velocity vector correctly aligned with the final isolation point. Point 5 is computed as an intersection point of the line from point 3 to point 4 and the line out of the final isolation point.

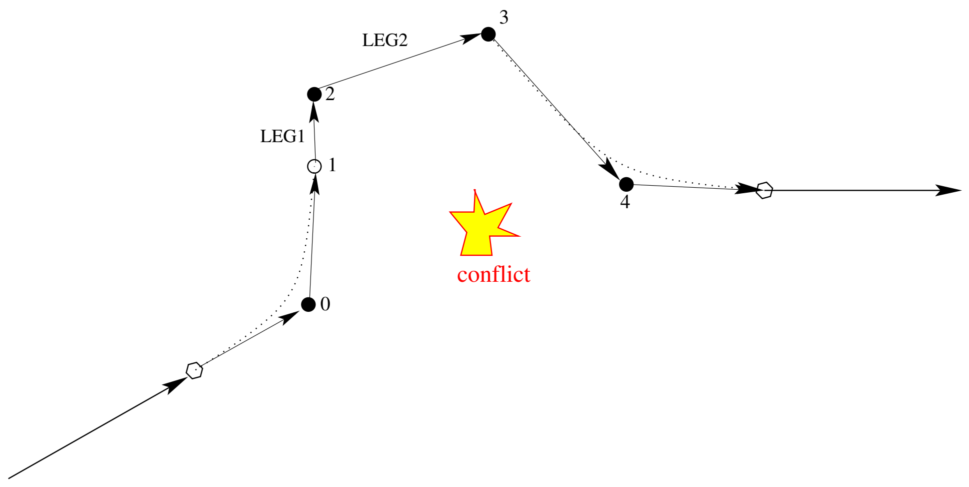

Figure 8: Template For Track Strategy

The parameters of the template are varied in a set of nested for loops, as seen in Figure 9. Note that the target evasive track (`targetTrk`) is determined by an angle `trkDelta` and a turn direction `dir` which is either +1 or -1. After the linear plan is constructed, a kinematic plan is generated from the linear plan, which creates four turn acceleration zones. This kinematic plan is inserted within the isolation points of the ownship plan and checked with a strategic conflict probe called `CDII`. If the proposed resolution is conflict free, then this resolution is returned by Stratway. Otherwise the iteration continues. If all four nested loops are exhausted, the `Track` strategy fails.

The ground speed remains constant throughout the maneuver. Because the distance is greater in the resolution as compared to the original plan, the time at the

```
for (trkDelta = trkStart; trkDelta < 70; trkDelta += trkStep) {
   for (dir = -1; dir < 2; dir += 2) {
      for (Seg1 = 0.0; Seg1 <= maxSeg1; Seg1 += step1) {
         for (Seg2 = 0.0; Seg2 <= maxSeg2; Seg2 += step2) {
            Generate Linear Plan
            Translate Into Kinematic Plan
            if conflict-free then return solution;
         }
      }
   }
}
return failure;
```

Figure 9: Track strategy algorithm

last isolation point must be increased. This time delta increase is then propagated throughout the remainder of the plan.

TrackEarly Variation The `Track` strategy has a preset variation called `TrackEarly`. `TrackEarly` expands the isolation zone to be closer to the ownship's current position, allowing for an earlier (and potentially less severe) maneuver. See Section 4.3.

4.1.2 The Vertical Strategy

The `Vertical` strategy seeks to resolve the conflict through a sequence of climbs and descents without changing the current track of the aircraft. This strategy begins by removing points in the vicinity of the conflict from the ownship plan using the same `isolateConflic()t` method as the track strategy. The template used for the vertical strategy is shown in Figure 10. This template is varied by iterating over parameters that define its structure. Three aspects of this pattern are varied in nested for loops: the vertical speed (`targetVs`) from point 1 to point 2, the length of the segment from point 1 to point 2 (`LEG1`), and the length of the segment from point 2 to point three (`LEG2`).

The location of point 1 is calculated first. Using a kinematic vertical acceleration function (that is provided the vertical speed into the first isolation point and a vertical acceleration value), the location where the target vertical speed is achieved is calculated, resulting in point 1. Given this point the vertex point 0 is computed as an intersection point. (Note that after point 0 is calculated, point 1 is redundant.)

These parameters are varied in nested for-loops, as seen in Figure 11. Note that target vertical speed (`targetVs`) is determined by a delta speed `vsDelta` and an up/down direction variable `dir` which is either +1 or -1. Similarly to the `Track` strategy, after the linear plan is constructed, a kinematic plan is generated from the linear plan, which creates four vertical acceleration zones. This kinematic plan is

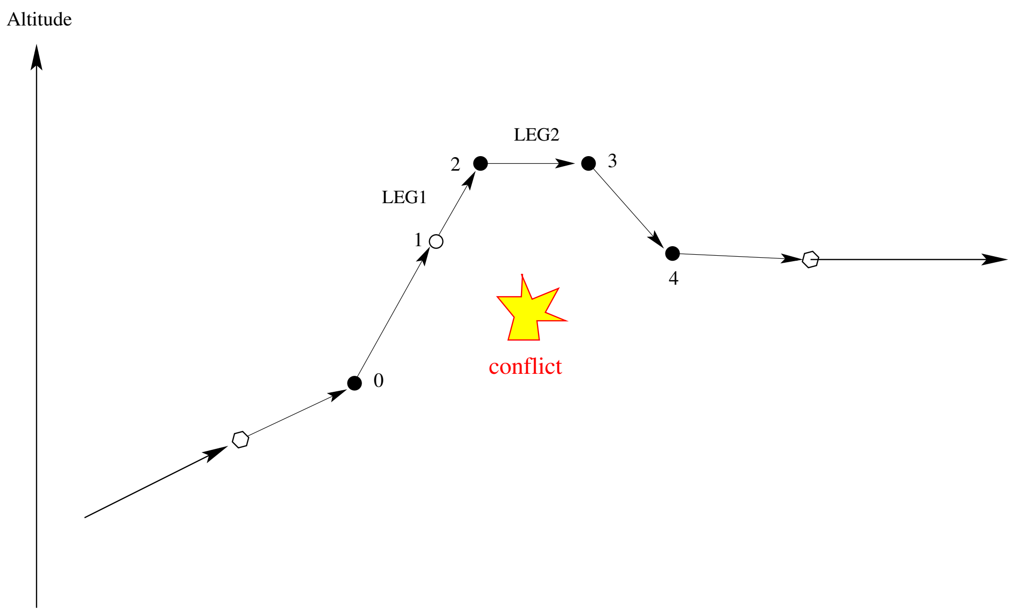

Figure 10: Template For Vertical Strategy

```
for (int vsDel = 500; vsDel <= 3000; vsDel += 500) {
   for (dir = -1; dir < 2; dir += 2) {
      for (Seg1 = 0.0; Seg1 <= maxSeg1; Seg1 += step1) {
         for (Seg2 = 0.0; Seg2 <= maxSeg2; Seg2 += step2) {
            Generate Linear Plan
            Translate Into Kinematic Plan
            if conflict-free then return solution;
         }
      }
   }
}
```

Figure 11: Vertical strategy algorithm

inserted within the isolation points of the ownship plan and checked with a strategic conflict probe called `CDII`. If the proposed resolution is conflict-free, then this resolution is returned by Stratway. Otherwise the iteration continues. If all four nested loops are exhausted, the `Vertical` strategy fails. The ground speed remains constant throughout the maneuver.

4.1.3 The Speed Strategy

The `Speed` strategy seeks to resolve the conflict through a sequence of ground speed changes without changing the current vertical speed of the aircraft or making any turns. Unlike the `Track` and `Vertical` strategies, the `Speed` strategy does not perform an `isolateConflict()` step. The strategy seeks to retain the locations of the points in the plan and only alter the times of those points. The concept for this strategy is illustrated in Figure 12. In this illustration, there is a conflict that

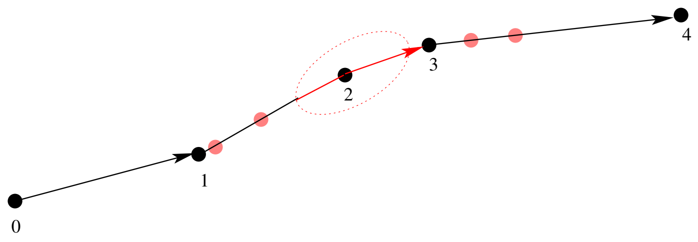

Figure 12: Concept of the Speed Strategy

starts on the segment between points 1 and 2 and ends on the segment between points 2 and 3. The strategy inserts acceleration zones before and after the conflict in order to increase or decrease the speed in the conflict region. The strategy seeks to introduce the speed change at an early time, because this results in a smaller ground speed change. The beginning and end points of the acceleration zones are marked by red colored dots.

4.1.4 The SideStep Strategy

The `SideStep` strategy seeks to resolve conflicts by only moving the waypoint immediately before a conflict. It is not as powerful as the `Track` resolution, but it can sometimes find a simpler, more efficient solution than the `Track` strategy. This strategy does not use the `isolateConflict()` function to remove points near a conflict, so it works most effectively for flight plans or trajectories with long segments. It inserts a lead-in point ahead of the current location of the aircraft if it is currently on the plan segment immediately before the conflict segment. This point is labeled blue in Figure 13.

4.1.5 Default Order of Strategies

The default order of the strategies is: `SideStep`, `TrackEarly`, `Vertical`, `Speed`, `Track`. Advanced users may wish to alter the order of the application of the

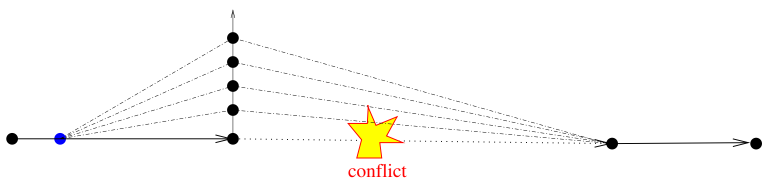

Figure 13: The SideStep Strategy

strategies or only use a subset of them. This is achieved by calling a method `setStrategies()` which takes a string argument that lists the strategies by name in the order desired. The current list of strategy names are `SideStep`, `TrackEarly`, `Vertical`, `Speed`, and `Track`. For example, the code skeleton in Figure 14 illustrates how to do this through the Java API.

```
    ArrayList<Plan> plans = new ArrayList<Plan>(10);
    plans = ....
    Stratway sw = new Stratway();
    for (Plan fp: plans) {
        sw.addPlan(fp);
    }
    Plan solution = new Plan;
    setStrategies("Track, Vertical");
    solution = sw.resolveConflicts();
```

Figure 14: Setting strategy order

4.2 Stratway Output

The `resolveConflicts()` method returns a new ownship plan. If `getResolutionStatus() == CONFLICT_FREE`, then the returned plan will be conflict free. The Java code in Figure 15 illustrates this. The returned plan file

```
     Stratway sw = new Stratway();
     Plan nPlan = sw.resolveConflicts();
     if (sw.getResolutionStatus()
         == Stratway.ResolutionStatusValue.CONFLICT_FREE) {
        ...
     }
```

Figure 15: Checking resolution status

`nPlan` will be a kinematic plan, that is, the repaired section will contain TCPs. This plan can be converted into a linear plan if that is desired as follows:

```
     Plan lPlan = TrajGen.makeLinearPlan(nPlan);
```

All of the acceleration zones will be removed and replaced with constant velocity segments. See Sections 6.10.5 and 6.10.4 for more details.

4.2.1 Utilities to Manipulate Plans

A `Plan` object is a sequence of `NavPoints`. The `NavPoint` class is discussed in Section 6.10.2. The `Plan` class is intended to serve as an abstraction for both aircraft flight plans and aircraft trajectories. The `Plan` class provides methods to construct plans, retrieve information from plans, and also to calculate relevant information about plans.

Individual `NavPoints` are retrieved from a `Plan` using the `point` method:

```
int i = myPlan.size()-1;
NavPoint p0 = myPlan.point(i);
```

In this code segment the last point (i.e., point `i`) in plan `myPlan` is retrieved. The following functions can be used to obtain the position and velocity that corresponds to a specific point of time `t` in a plan:

```
public Position position(double t)
public Velocity velocity(double t)
```

Plans are typically created from an empty plan by adding `NavPoints` using the `add` method, as shown in Figure 16. In this code segment, four 3-D `Positions` are created by specifying latitudes, longitudes, and altitudes. Then 4-D NavPoints (i.e. `np0`, `np1`, `np2`, `np3`) are created from this by associating times. Finally, these are added to the `newPlan` object using the `add` method. See section 6.10.3 for more details.

```
Plan newPlan = new Plan();
Position p0  = new Position(LatLonAlt.make( 0.0000, 0.0000, 23000));
Position p1  = new Position(LatLonAlt.make(-0.3870, 0.9358, 23000));
Position p2  = new Position(LatLonAlt.make(-0.3062, 1.6488, 23000));
Position p3  = new Position(LatLonAlt.make(-0.0785, 2.1500, 23000));
NavPoint np0 = new NavPoint(p0, 937.1);
NavPoint np1 = new NavPoint(p1, 1376.3);
NavPoint np2 = new NavPoint(p2, 1687.5);
NavPoint np3 = new NavPoint(p3, 1926.0);
newPlan.add(np0);
newPlan.add(np1);
newPlan.add(np2);
newPlan.add(np3);
```

Figure 16: Constructing a plan.

4.3 User Configuration of Strategies

The Stratway strategies can be tailored through user-definable parameters. For example, the direction of the maneuvers can be constrained or the size of the isolation region can be altered. There are also early/late options which control how near to the conflict the Stratway program starts the maneuver.

The direction of the solutions can be constrained using the following Stratway methods:

`setTurnDirectionLeftRight()`	Directs the Track or SideStep algorithm to only return solutions where the ownship turns left (-1) or turns right (+1).
`setTurnDirectionBehindFront()`	Directs the Track or SideStep algorithm to only return solutions where the ownship goes behind (-1) or in front (+1) of the traffic aircraft.
`setVerticalDirectionDownUp()`	Directs the Vertical solution algorithm to only return solutions where the ownship descends (-1) or climbs (+1).
`setSpeedDirectionSlowFast()`	Directs the Speed solution algorithm to only return solutions where the ownship turns slows down (-1) or speeds up (+1).
`setSpeedDirectionBehindFront()`	Directs the Speed solution algorithm to only return solutions where the ownship goes behind (-1) or in front (+1) of the traffic aircraft.

The Track strategy has an “early” mode. Setting early mode causes the track maneuver to attempt to start at an earlier time and to recapture the original plan at a later time, ideally resulting in a more gradual change.

4.4 Feasibility Checking of Potential Solutions

The `resolveConflicts()` method uses heuristic search methods (i.e., the strategies) to find solutions on a segment-by-segment basis. These methods start with the first conflict and then proceed to subsequent conflicts after a fairly local solution to the first conflict is found. When a potential solution has been found (i.e., it passes the conflict detector), the program next checks that the solution is physically reasonable. For example, it throws away potential solutions that require a ground speed that exceeds a specified maximum value. The feasibility checker is governed by parameters that are user specifiable. If a solution produced by a strategy fails a feasibility test, it is rejected and another strategy is tried. The checks are performed using the following constants:

`minGroundSpeed`	50.0 kn	minimum ground speed allowed
`maxGroundSpeed`	700.0 kn	maximum ground speed allowed
`maxVerticalSpeed`	5000.0 fpm	maximum vertical speed allowed
`minVerticalSpeed`	-5000.0 fpm	minimum vertical speed allowed
`maxAltitude`	40000.0 ft	maximum altitude allowed
`minAltitude`	0.0 ft	minimum altitude allowed
`latLonAccuracy`	0.10 nmi	max error for latlon calculations
`defaultClimbRate`	1500 fpm	rate of climb for stepClimb
`maxBankAngle`	30.0 deg	max acceptable bank angle
`maxGsAccel`	2.0 m/s^2	max acceptable horizontal acceleration
`maxVsAccel`	2.0 m/s^2	max acceptable vertical acceleration

These constant parameters may be specified in input files, in the Visualization Tool through the Parameter Panel or through the API mutator methods.

5 Parameters

The Stratway program is flexible and highly configurable by the user. Parameters can be set within any of the three main interfaces: command-line, API, and visualization. This configurability has advantages and disadvantages. A key advantage of this approach is that the program can be used in a wide range of applications. The major disadvantage is that this flexibility increases the learning curve associated with the program. Although the parameter defaults are carefully chosen, users should not assume that these defaults will be appropriate for their particular application.

Some of the more useful parameters are defined as follows:

backtrackSearch If set to true, Stratway will check additional combinations of strategies.

bufDdet sets the horizontal detection buffer distance. This is added to `D` for detection calculations, allowing for a given amount of uncertainty in data.

bufDres sets the horizontal resolution buffer distance. This is added to `D` for detection calculations, allowing for a given amount of uncertainty in data.

bufHdet sets the vertical detection buffer distance. This is added to `H` for detection calculations, allowing for a given amount of uncertainty in data.

bufHres sets the vertical resolution buffer distance. This is added to `H` for detection calculations, allowing for a given amount of uncertainty in data.

checkPerformance If set to true, will enforce performance feasibility constraints.

D sets the horizontal regulation separation distance.

H sets he vertical separation distance.

isolationLookahead Several strategies remove points that are within a conflict zone in order to allow for more reasonable resolutions. This value represents a time buffer before and after a conflict, between which all points are removed.

latLonAccuracy Approximate allowed error in calculating relative latitude and latitude positions.

maxAltitude sets the maximum altitude allowed.

maxBankAngle sets the max acceptable bank angle.

maxGsAccel sets the max acceptable horizontal acceleration rate.

maxVsAccel sets the max acceptable vertical acceleration rate.

maxGroundSpeed sets the maximum allowed ground speed.

maxVerticalSpeed sets the maximum vertical speed allowed.

minAltitude sets the minimum altitude allowed.

minGroundSpeed sets the minimum allowed ground speed..

minVerticalSpeed sets the minimum vertical speed allowed.

strategies A comma or whitespace-delimited list of the strategies to be tried in the search, and in which order. Only one resolution will be generated.

T sets both `T_d` and `T_r` to the same value [sec].

T_d sets the maximum conflict detection lookahead time, from the TimeOfCurrentPosition (usually the start of the ownship plan) [sec].

T_r sets the maximum resolution detection lookahead time, from the TimeOfCurrentPosition (usually the start of the ownship plan) [sec].

TimeOfCurrentPosition This indicates where the ownship aircraft is currently located relative to its plan. This is only needed if Stratway is given a plan that has ownship points in the "past".

See also the note on units in Section 6.1.

6 Application Programming Interface (API)

The API to Stratway allows another program to call Stratway functions, such as `resolveConflicts()`, that operate on a set of plans. The function `resolveConflicts()` returns a revised ownship plan with all conflicts and losses of separation removed. The API provides the tools needed to construct and manipulate plans. The expected use is as follows:

1. Create a Stratway object `S`.

2. For each aircraft (ownship first), create a `Plan` object `P` and call `S.addPlan(P)`.
3. Set parameter values using API functions (if default value is not desired).
4. `Plan rtn = S.resolveConflicts();`
5. If `S.hasError()` is true then Stratway has generated an error, otherwise `rtn` contains an updated plan.
6. If `S.getResolutionStatus()` returns the value `Stratway.ResolutionStatusValue.CONFLICT_FREE` then `rtn` contains a conflict-free plan.
7. Otherwise, conflicts remain—use `S.getDetector()` to examine the unresolved conflicts.
8. If there are no remaining conflicts, use the call `rtn.point(i)` to retrieve the new position for each waypoint `i`.
9. Call `S.clearPlans()`, and repeat as necessary.

See section 6.12 for code examples.

6.1 A Word on Units

Many times Stratway will need to be supplied with a particular value. Most API calls that require a value will typically fall into one of several patterns:

- `class.make(double` *value*, `String` *unit*`)`: The indicated *value* is to be interpreted in the user-specified *unit* for "make" methods. This pattern is typically used in factory methods for low-level objects.
- `class.make(double` *value*`)`: If the unit-specifying string is not included in the formal parameters to the "make" call, it is understood to follow usual unit conventions for air traffic in the United States. For example, a call to `LatLonAlt.make()` assumes degrees for latitude and longitude and feet for altitude.
- `method(double` *value*, `String` *unit*`)`: The indicated *value* is to be interpreted in the user-specified *unit*. This pattern is typically used for setters and getters.
- `class.method(double` *value*`)`: Methods without an explicit unit specifier assume standard SI [8] base or derived units (radians, meters, seconds, meters per second). This includes "mk"-style factory methods—as opposed to "make" factory methods described above.

Examples of acceptable unit strings include `kts`, `deg`, `nmi`, `ft`, `s`, `fpm`, and `m/s`. See the `Util.Units` class for more unit definitions. It is **strongly recommended** that users utilize method calls that explicitly specify the intended units in order to avoid confusion.

Parameters in text files generally default to usual air traffic unit conventions (angles in degrees, horizontal distances in nautical miles, altitudes in feet, ground speeds in knots, vertical speeds in feet per minute, time in seconds), though it is always possible to specify a given unit, for example:

```
D = 5.0 [nmi]
H = 1000.0 [ft]
T = 5.0 [min]
```

6.2 Building a Plan

The Stratway API provides convenient methods in the `Plan` class to construct plans. These methods construct plans from a sequence of `NavPoints`. The programmer first creates a `Plan` object as follows:

```
Plan own = new Plan()
```

`NavPoints` are added using the `add` method:

```
own.add(NavPoint.makeLatLonAlt(lat1,lon1,alt1,t1));
own.add(NavPoint.makeLatLonAlt(lat2,lon2,alt2,t2));
  .
  .
own.add(NavPoint.makeLatLonAlt(latN,lonN,altN,tN));
```

The Stratway object is created and then a plan is added as follows:

```
Stratway sw = new Stratway();
sw.addPlan(own);
```

By convention, the first plan added is considered to be the ownship. The traffic aircraft plans are created in the same way and then added

```
sw.addPlan(traf1);
sw.addPlan(traf2);
   .
   .
   .
sw.addPlan(trafN);
```

Additional flexibility is available through methods in the `NavPoint` class. See Appendix A for more detail.

6.2.1 Time Considerations

The Stratway program assumes that all points in a plan are distinct in time. In fact, two points in a plan cannot be closer in time than the internal plan parameter `minDt` $= 10^{-5}$ seconds. This is necessary to avoid numerical problems with certain operations and to keep point indexing straightforward. If one attempts to add a new point that is within `minDt` seconds of a point that is already in the plan, the new point will be dropped.

6.2.2 Key Parameters

A plan is interpreted and processed by user-defined parameters. Often the defaults are appropriate, but the Stratway user should exercise due diligence in setting these parameters appropriately. The following parameters should be given special consideration: `TimeOfCurrentPosition`, `T_d`, and `T_r`. See section 5 for a detailed description of available parameters.

6.3 Reading a Plan CSV File

There are several reader classes included in the FormalATM distribution that allow for the automatic reading of CSV plan files, including sets of parameters. The code in Figure 17 will load the plans described in `filename` into the Stratway object `sw`.

```
import gov.nasa.larcfm.Util.PlanReader;
.
.
.
PlanReader reader = new PlanReader();
reader.open(filename);
for (int i = 0; i < reader.size(); i++) {
   sw.addPlan(reader.getPlan(i));
}
```

Figure 17: Reading in plans

For files that also contain polygon information, the `PolyReader` class may be used instead.

6.4 Conflict Resolution

After the Stratway object is created and plans have been added using the `sw.addPlan` method, the `sw.resolveConflicts()` method can be called. This method applies strategies for each detected conflict in a predetermined order. This predetermined order can be the default order or a user-specified order. The output is a new plan for the ownship. For example:

```
Plan own = sw.resolveConflicts();
```

provides such a new plan.

6.5 Resolution Status

Stratway is not guaranteed to find a conflict-free resolution; the user must be sure to determine the status of what was returned. After a call to `resolveConflicts()`, the status can be found as follows:

```
Stratway.ResolutionStatusValue v = sw.getResolutionStatus();
```

The return type `ResolutionStatusValue` is an enumeration with the following values:

`UNKNOWN`	No resolution has been attempted or Stratway terminated in an unknown failure state.
`CONFLICT_FREE`	Stratway returned a conflict free ownship plan ("success").
`SEARCH_DONE`	Call to nextResolution() has failed with no more possible solutions.
`UNRESOLVED`	No conflicts were able to be resolved.
`LOS`	Ownship started in loss of separation; Stratway cannot continue. In this situation a state-based program such as `Chorus` can be used to get the aircraft out of loss of separation [1].

If the returned value is `SEARCH_DONE`, `UNRESOLVED`, or `LOS`, the `getDetector()` method described in Section 6.7 can be used to get information about remaining conflicts.

6.6 Messages, Errors, and Warnings

Stratway implements the `ErrorReporter` interface (Section 6.10.1), meaning it internally logs any errors and warnings. Collectively, these are referred to as *messages.* In general, an error means that the software encountered a situtation where there is no reasonable recovery. Any results should be ignored. *Warnings,* on the other hand, represent situations where the results could be correct, but more likely are in error. Errors should never be ignored, whereas warnings, depending on context could be ignored. If Stratway has encountered a serious error, the call `sw.hasErrors()` will return `true`. Errors and warnings produced by Stratway can be obtained as follows:

```
if (sw.hasMessage()) System.out.println(sw.getMessage());
```

In addition, Stratway logs internal strategy messages. These messages provide information regarding how the strategy was computed. This is purely informational and can be ignored. The detailed strategy messages after a call to `ResolveConflicts()` can be printed as follows

```
System.out.println(sw.getStratMessage());
```

A single-line abbreviated message listing the strategies used in `ResolveConflicts()` can be obtained by:

```
System.out.println(sw.getStratSummaryMessage());
```

Note that over extended periods of time, Stratway can generate a large amount of logged information, and it is highly recommended that the user periodically clear this data to free memory. Status messages are automatically cleared when the `sw.clearPlans()` or `sw.clearStratMessage()` are called. Error messages are cleared when they are read via `sw.getMessage()`.

6.7 The `getDetector()` Method

After a Stratway object (e.g., `sw`) has been constructed, details about all of the ownship conflicts can be obtained. To do this, the user first creates a `Detector` object as follows:

```
    Detector det = sw.getDetector();
```

The status is then available from the `conflict()` and `size()` methods:

```
     boolean conflictsFound = det.conflict();
     int numConflicts = det.size();
```

Next, methods can be called to obtain the information about the times associated with the conflict using the following `Detector` methods:

`double getTimeIn(int i)`	Returns the start time of the conflict [s]. This value is in absolute time.
`double getTimeOut(int i)`	Returns the end time of the conflict [s]. This value is in absolute time.
`double getTimeClosest(int i)`	Returns the time of closest approach for the conflict [s]. This value is in absolute time.

The code fragment in Figure 18 illustrates how to print out information about all of the conflicts.

```
Stratway sw = new Stratway();
Detector det = sw.getDetector();
for (int j = 0; j < det.size(); j++) {
   double tmIn = det.getTimeIn(j);
   double tmOut = det.getTimeIn(j);
   double tmClosest = det.getTimeIn(j);
   System.out.print("Conflict "+j+": "+tmIn+","+tmOut+","+tmClosest);
}
```

Figure 18: Printing conflict information

The `getDetector` method can be used before and after a call to `resolveConflicts()`. If `resolveConflicts()` is only partially successful (i.e. it was only able to resolve a subset of the conflicts), then information about the remaining conflicts will be returned. The following methods are also available:

`int getTrafficID(int i)`	Returns the traffic ID (string identifier) for the *i*-th conflict.
`int getTrafficIndex(int i)`	Returns the index number in the set of plans sent to the `Detector` (or `sw`) corresponding to the traffic aircraft involved in conflict *i*.
`void sortConflicts()`	Sorts all conflicts based on their start times.
`int getNextConflict(double tt)`	Returns the index of the next conflict which starts after time *tt*.

The `getDetector` function can also take a parameter that is a new ownship plan:

```
Detector det = sw.getDetector(ownshipPlan);
```

In this case, the detection status is reported with respect to this alternate ownship plan rather than the one currently in the Stratway object. For example, such a call can return a `Detector` object that can be used to get conflict information about the (uncorrected) original ownship plan after `resolveConflicts()` has been called. This method does not change the state of the Stratway object.

Note that conflicts returned in a `Detector` object are not in any particular order unless the user explicitly requests that they be sorted.

6.8 Backtracking Solutions

It is possible to retrieve multiple solutions from Stratway, all using the same search strategy list by using a backtracking search.

```
Stratway sw = new Stratway();
Plan p = sw.resolveConflict();
while (sw.getResolutionStatus() ==
   ResolutionStatusValue.CONFLICT_FREE) {
   p = sw.nextResolution();
}
```

Figure 19: Returning multiple solutions

The fragment in Figure 19 will loop through all resolutions that can be generated using different orderings of the strategies in the default strategy list, stopping when there are no more possible solutions. There are also variations that may be used with arbitrary strategy lists. If the resolution status returns `SEARCH_DONE`, then no more backtracking is possible using this strategy list. Setting the parameter `backtracksearch` to false will disable this function.

6.9 Weather Polygons

In addition to aircraft traffic, Stratway has been designed to solve conflicts while respecting polygonal regions to avoid (such as weather cells or special use airspace) or remain within (containment). Polygons are arbitrary two-dimensional shapes with an associated lower and upper altitude bound. These polygons may be moving or static.

Moving polygons are represented by a sequence of polygons, each with a time stamp, analogous to an aircraft's plan. Alternatively, a time sequence of polygons, each with a velocity vector, can be used to describe a weather system. The first approach allows the shape of the polygon to smoothly change between time anchor points (*morphing*), but the number of vertices must remain constant. The second approach allows a different number of vertices for the sequence of polygons, but there is a potential discontinuity at each time point—the shape of the moving polygon stays the same until the next polygon is processed, where it may change.

Stratway recognizes *simple polygons*. Polygons may be non-convex, but they may not contain any zero-length, overlapping, or crossing edges. Each vertex must connect to exactly two edges—there is an implicit edge between the last vertex and the first vertex—and a vertex may not intersect with an edge that it is not connected to. To ensure no edges overlap, vertices should be added in either a clockwise or counterclockwise ordering.

There are three ways to specify a moving polygon path:

- A single polygon with a velocity vector (static),
- A sequence of polygons with a time associated with each polygon (morphing or using average velocity),
- A sequence of polygons each with a velocity vector and time-stamp (using user-specified velocitied).

In the first case, the polygon never changes shape and continues forever with the specified velocity. If the specified velocity is zero, then the polygon is stationary. For variaions on the second and third cases, see Section 6.9.3, below.

Simplifying polygons by reducing the total number of edges can significantly increase Stratway's performance.

6.9.1 Specifying a Single Polygon With a Velocity Vector

A polygon is defined as a `SimplePoly`, which has a sequence of Positions, entered either clockwise or counterclockwise, along with a top and bottom altitude. The `SimplePoly` then has a velocity associated with it (as well as identifier) by being placed in a `PolyPath`. It may have a user-specified velocity associated with it, or it may derive its velocity from other polygons in its path (and have an implied zero velocity if it is a singleton). An example of creating a single moving polygon is in Figure 20.

6.9.2 Specifying a Sequence of Polygons

The following steps can be used to define a polygon with more complex movement in Stratway:

```
Stratway sw = new Stratway();

Position v0 = Position.makeLatLonAlt(  2.571, 17.684, 0.0);
Position v2 = Position.makeLatLonAlt(  6.313, 11.484, 0.0);
Position v4 = Position.makeLatLonAlt( 10.361, 17.226, 0.0);
Position v6 = Position.makeLatLonAlt( 13.454, 22.139, 0.0);
Position v8 = Position.makeLatLonAlt( 10.341, 22.368, 0.0);

SimplePoly sPoly1 = new SimplePoly(0,10000,"ft");

sPoly1.addVertex(v0);
sPoly1.addVertex(v2);
sPoly1.addVertex(v4);
sPoly1.addVertex(v6);
sPoly1.addVertex(v8);

Velocity v = Velocity.makeTrkGsVs(145,510,0.0);

PolyPath pp = new PolyPath("Storm_A");
pp.addPolygon(sPoly1, v, 0.0);

sw.addPolyPath(pp);
```

Figure 20: Creating a single moving polygon

- Define each polygon by adding vertices to a `SimplePoly` object.
- Create a `PolyPath` by adding each of the above polygons with a timestamp.

The code segment in Figure 21 illustrates the process in Java. The instruction `SimplePoly sPoly1 = new SimplePoly(0,10000,"ft");` creates a base 3-D object for the polygon's horizontal structure. The first two parameters define the bottom and top altitudes of the polygon. The `PolyPath` in this example has only two polygons with timestamps 0.0 and 54210.0.

It is also possible to create a simple 2-step `PolyPath` with the method `pathFromState()`. Given `sPoly1` from above, the following code will produce the same path (within the system's numeric precision):

```
Velocity v = Velocity.makeTrkGsVs(302.2438, 34.5235, 0.0000);
PolyPath pp = PolyPath.pathFromState("ppA",sPoly1,v,0.0,54210.0);
```

6.9.3 Polygon Path Modes

Each polygon path object has a parameter that defines how sequences of polygons are interpreted, its "path mode". This parameter can take one of four distinct values that determine how polygon velocities are treated between steps. These are `MORPHING`, `AVG_VEL`, `USER_VEL`, and `USER_VEL_FINITE`. Generally the mode is determined by the method used to add steps to the PolyPath, but it can be explicitly set via the `setPathMode()` method.

MORPHING A morphing path treats all vertices as having individual velocities. This allows the polygon's shape to alter between steps. These are calculated by comparing a given step with the subsequent step. The final step in such a path does not have a velocity associated with it, and signals the end of the path. When in `MORPHING` mode, each step must have the same number of vertices as all previous steps. This is the default mode set when polygons are added to the path with the `addPolygon(poly, time)` method.

A polygon on a morphing path smoothly transforms between steps even if the shape of each step is different, and has a distinct beginning and end of existence (except static polygons, below). If morphing is selected then the Stratway program uses interpolation between the polygons to determine intermediate states. This is illustrated in Figure 22.

AVG_VEL An `AVG_VEL` (average velocity) path moves the entire polygon with a constant velocity, preserving its shape. This velocity is calculated by comparing a polygon's "average point"[2] with the subsequent step's average point. The final step does not have a velocity associated with it and signals the end of the path. Average velocity paths treat each step as a distinct state change from the previous one and do

[2]This is the average of all vertices, which is distinct from the *centroid*, the (weighted) average of the polygon's included area, or "center of mass". For morphing polygons, the centroid's velocity between two steps is not necessarily consistent, while the average point's velocity is.

```
Stratway sw = new Stratway();

Position v0 = Position.makeLatLonAlt(  2.571, 17.684, 0.0);
Position v2 = Position.makeLatLonAlt(  6.313, 11.484, 0.0);
Position v4 = Position.makeLatLonAlt( 10.361, 17.226, 0.0);
Position v6 = Position.makeLatLonAlt( 13.454, 22.139, 0.0);
Position v8 = Position.makeLatLonAlt( 10.341, 22.368, 0.0);

SimplePoly sPoly1 = new SimplePoly(0,10000,"ft");

sPoly1.addVertex(v0);
sPoly1.addVertex(v2);
sPoly1.addVertex(v4);
sPoly1.addVertex(v6);
sPoly1.addVertex(v8);

SimplePoly sPoly2 = new SimplePoly(0,10000,"ft");

Position v10 = Position.makeLatLonAlt(  6.270,  1.227, 0.0);
Position v12 = Position.makeLatLonAlt( 10.012,  5.026, 0.0);
Position v14 = Position.makeLatLonAlt( 14.068, 10.763, 0.0);
Position v16 = Position.makeLatLonAlt( 17.154, 15.685, 0.0);
Position v18 = Position.makeLatLonAlt( 14.040, 15.914, 0.0);

sPoly2.addVertex(v10);
sPoly2.addVertex(v12);
sPoly2.addVertex(v14);
sPoly2.addVertex(v16);
sPoly2.addVertex(v18);

PolyPath pp = new PolyPath("pp");
pp.addPolygon(sPoly1,0.0);
pp.addPolygon(sPoly2,54210.0);

sw.addPolyPath(pp);
```

Figure 21: Creating a PolyPath

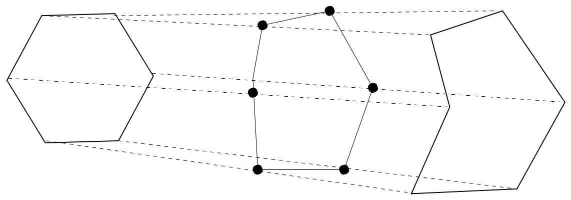

Figure 22: Morphing Path

not require individual steps to have the same number of vertices as previous steps. This is the default mode set when paths are generated via the `pathFromState()` methods.

A polygon on an average velocity path will have a constant shape between each step, but may abruptly change shape at each step. This sort of path has a distinct beginning and end of existence (excepting static polygons, below). This is illustrated in Figure 23.

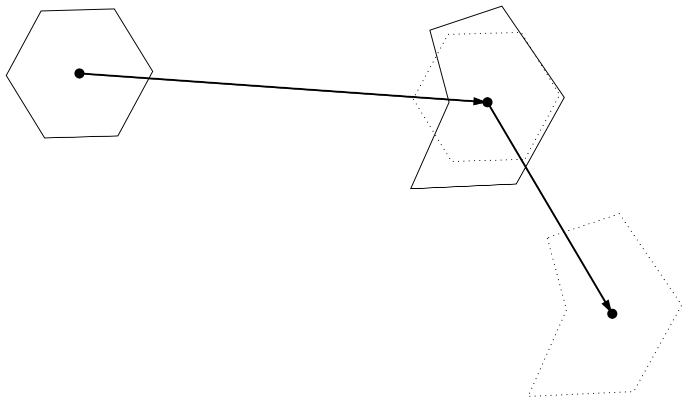

Figure 23: Average Velocity Path

USER_VEL An `USER_VEL` (user-specified velocity) path moves the entire polygon with a constant but arbitrary velocity, preserving its shape. At each step, the user must specify the polygon's velocity, and it need not have any relation with previous or successive steps. This path is a series of states that begin and end in sequence and have the same identifier, but are not necessarily continuous. User-specified velocity paths treat each step as a distinct state change from the previous and do not require individual steps to have the same number of vertices as previous steps. The final step does *not* signal the end of the path. The final state continues indefinitely with the specified velocity (which may be zero). These paths continue indefinitely. This is the default mode set when polygons are added to the path with the `addPolygon(poly, velocity, time)` method.

A polygon on a user-specified velocity path will have a constant shape between each step, but may abruptly change shape and position at each step—the direction of travel does not necessarily have to lead to the next step. This sort of path has a distinct beginning but no end. This is illustrated in Figure 24.

USER_VEL_FINITE A `USER_VEL_FINITE` path is the same as a `USER_VEL` path except that the last step signifies the termination time of the polygon[3].

Switching Modes It is generally recommended that the mode of a path be implicitly set via the type of constructor used. However, the mode can be altered after the

[3]The actual `SimplePoly` stored as the last step is not directly referenced, and may be empty.

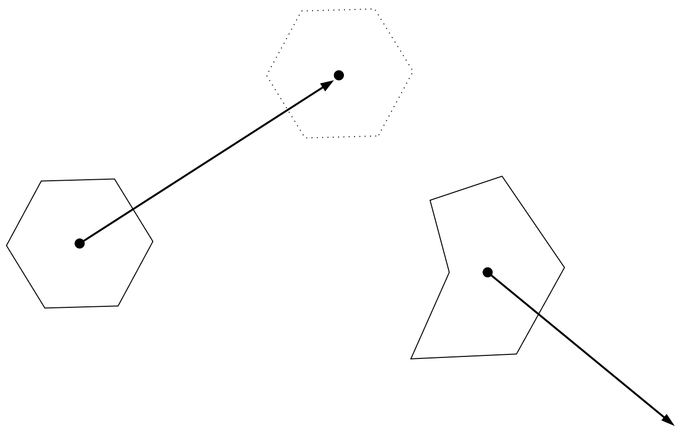
Figure 24: User-Specified Velocity Path

initial construction using the `setPathMode()` method. Switching between `MORPHING` and `AVG_VEL` modes only requires that the number of vertices be consistent. If `setPathMode()` method is used to switch *from* the `USER_VEL` or `USER_VEL_FINITE` mode, all user velocity information will be deleted. If this method is used to switch *to* the `USER_VEL` or `USER_VEL_FINITE` mode, it will assign average velocities to each step, and a zero velocity to the last step.

6.9.4 Static Polygons

A `PolyPath` with only one step in either the `MORPHING` or `AVG_VEL` mode (no matter what time is assigned) can be used to describe a perpetual, unmoving polygon. This may be used to represent something such as a special use airspace region that will never change. It is necessary to have at least a two-step `PolyPath` in order to represent a polygon that has a definite start and end time.

There are two predicates that indicate the end-status of a polypath:

`sw.isStatic()`	Returns true if this `PolyPath` is static.
`sw.isContinuing()`	Returns true if this `PolyPath` is continuing This applies to both static polygons, which are constant, and `USER_VEL` paths, which have no definite end point.

6.9.5 ReRouteWx (Java only)

The API function `reRouteWx` can be used to generate a new large-scale lateral route that avoids weather. This is intended to be a preprocessing step to a normal `resolveConflicts()` call, when localized heuristic strategies are unlikely to find a solution. It is different from `resolveConflicts()` in several respects:

- It does not resolve conflicts with traffic aircraft.
- It does not seek to retain as many original plan points as possible.

- Similar to the `Track` strategy, it operates in two dimensions, and will attempt to avoid polygons through horizontal maneuvers.
- Because it operates in two dimensions, polygon heights are ignored.

This function performs a heuristic search to find a new path from the current/first point of the plan to the last point of the plan. The method `reRouteWx` is invoked as follows:

```
Pair<Boolean,Plan> ret = sw.reRouteWx(gridSize, buffer, factor);
```

Note that `ret.first` is a boolean value that is true if the resulting plan happens to be conflict free and `ret.second` is the resulting plan. If `ret.second` is of size zero or contains errors, the rerouting attempt failed. If the returned plan has no errors and the boolean value is false, then the proposed plan still contains conflicts that may be able to be resolved with a subsequent call to `resolveConflicts()`. The `reRouteWx` function creates a very simple vertical profile. It creates a constant vertical speed from `timeOfCurrentPosition` to the final point in the plan.

The `reRouteWx` method finds a solution by creating a rectangular grid around the ownship plan (starting at `timeOfCurrentPosition`) and computing weights for each grid location. This weight is time dependent. If a weather polygon is present within this grid at a particular time, then the weight is infinite. The weight may also factor in a function of the distance of the grid from the current location of the search path. There are user-settable parameters which can be used to control the weightings and hence affect the heuristic search. These parameters include:

`gridSize`	Formal parameter of method that sets size of grid square (in meters).
`buffer`	Formal parameter of method that sets the size of the extension of the grid in meters outside of the rectangle around the ownship plan.
`factor`	Formal parameter of method that specifies the weighting of the closeness to the ownship path, if 0 then this feature is not in effect.
`unZigReroute`	Stratway parameter that controls whether the discovered route is smoothed before returning it to the user.
`reRouteLeadIn`	Stratway parameter that sets the amount of time at beginning and end of a plan that cannot be changed by the reRoute Method.
`reduceGridPath`	Stratway parameter that controls whether collinear midpoints are removed from the solution.
`fastPolygonReroute`	Stratway parameter that impacts the grid search wrt polygons (speed vs precision).

Other factors that influence the search are distance from the path end point (heading directly to the end is favored) and changes in the rerouting direction (more gradual turns are favored); these behaviors are currently hard-coded and not user-settable.

If the rerouting function fails, it is unlikely (but possible) that a normal resolution on the original plan will succeed. The given `gridSize` should generally be smaller than the diameter of most polygons (the visualization uses a default grid size of 10 nmi). Reducing the grid size will generally allow for more complicated paths, but will increase the search time.

The buffer should be large enough to allow a new plan to potentially reach around weather. The visualization, for example, uses a buffer that is the larger of 30 nmi or 20% of the ownship plan's length. The `reRouteLeadIn` parameter should be large enough to accomodate a potentially sharp turn. It is recommended that it be larger than the standard turn radius for the aircraft's current speed.

Currently `factor` values greater than 10 will generally cause the rerouting function to stay close to the original plan. Factor values of less than 3 will often take the most direct route to the end of the plan.

If the parameter `fastPolygonReroute` is set to true, then `ReRouteWx` uses a faster check for determining whether a polygon blocks a grid square. In this case, an overapproximation of the polygon is used. However, it may fail to find potential routes, especially if they would terminate near a polygon. If set to false, a more precise check is performed, but the search time is increased by the greater number of polygon edges that are processed. But, this enables `ReRouteWx` to find solutions that skirt closer to the weather polygons. Reducing the `gridSize` and turning off `fastPolygonReroute` will allow for more paths skirting between closely packed weather cells, though this may adversely affect reroute computation times. Reducing `gridSize` will generally have less effect on the result if `fastPolygonReroute` is on.

6.9.6 Dynamic Containment Constraint

The *dynamic containment constraint* is a constraint that restricts resolutions to remain within a predefined, though possibly evolving, volume. This *dynamic containment volume* is the union of the space defined by a set of polygons and their associated altitude zones. A Stratway strategy solution is forced to stay within the dynamic containment volume.

If a dynamic containment volume is defined (by adding one or more `PolyPath`s to Stratway via the `Stratway.addDynamicContainmentPolygon()` method), then strategies will only succeed if the ownship both avoids any losses of separation with traffic and weather, and also continuously remains within the union of the polygons making up the overall containment volume.

Polygons in the containment volume may be static or dynamic. Static polygons are generally used to designate a fixed geographic bound. Dynamic polygons can be used to prevent the ownship from straying too far (both spatially and temporally) from a defined path.

The dynamic containment constraint is not, by itself, a traditional geofence. A containment check is only performed on sections of the plan that are modified by the `resolveConflicts()` method. Therefore, the use of this constraint does not guarantee containment in regions of the ownship plan that are not near conflicts and, hence, will not correct containment violations in an input plan that is not originally conflict free. However, if the original ownship plan is not initially within the containment volume, the Stratway resolution call will return a warning message.

The `reRouteWx()` method will generally return a plan that remains fully within the specified dynamic containment region. Thus it can potentially be used to repair ownship plans that are outside of the containment volume. But neither `resolveConflicts()` nor `reRouteWx()` method will alter the initial point (defined by `timeOfCurrentPosition`) or the final point in the plan. The `resolveConflicts()` method can be forced to repair all containment violations by surrounding the containment region with normal polygons. This would make all original deviations from the containment region a conflict that must be repaired.

The method `dynamicContainmentExitTime()` can be used to check if the current ownship (or a potential alternative) will leave the containment volume. If this call returns a negative value, the ownship never leaves the volume. The method `dynamicContainmentEntryTime()` can be used to insure that the plan is not originally outside of the containment volume.

These algorithms are still experimental, and for time-dependent containment constraints, it is possible that a successful local resolution may cause the ownship to exit the dynamic containment volume at some region that is not near a conflict. If Stratway produces an otherwise successful resolution that exits the dynamic volume, a warning will be issued. The method `dynamicContainmentCheck()` can be used to determine if the ownship ever leaves the containment volume.

6.10 Java

The Java API is based on the `Plan`, `Stratway`, and `Detector` classes. See the code Javadoc output for the most current version. The following subsections discuss some of the main methods of interest.

6.10.1 ErrorReporter Interface

The `Plan`, `Stratway`, and `Detector` classes all implement the `ErrorReporter` interface. Errors and warnings are accumulated internally and must be explicitly checked. The generic name for both errors and warning is *messages.* In general, an error means that the software encountered a situtation where there is no reasonable recovery. Any results should be ignored. *Warnings,* on the other hand, represent situations where the results could be correct, but more likely are in error. Errors should never be ignored, whereas warnings, depending on context could be ignored.

`hasError()` Returns true if an error has been encountered. This indicates that there is inconsistent data in the object, and consequently, the results should be viewed with suspicion.

`hasMessage()` Returns true if an error or warning has been encountered. Warnings indicate something unusual (but possibly still correct).

`string getMessage()` Returns a string representation of any errors or warnings. Calling this method will clear any messages and as well as both the error and warning status.

`string getMessageNoClear()`: Return a string representation of any errors or warnings. Calling this method will *not* clear any messages or reset the error or warning status to none.

6.10.2 Position, Velocity, and NavPoint Classes

There are four position-related classes and one velocity class that can be used to define a 3-D or 4-D point in Stratway.

`Point` This class contains Cartesian (i.e., XYZ) coordinates.

`LatLonAlt` This class contains geodesic coordinates, i.e., latitude, longitude, and altitude values are stored.

`Position` A 3-D coordinate class that generalizes both Cartesian and geodesic coordinates. (Note that this does not automatically perform a *projection* between the two data types. The `Position` will be valid in only one of these interpretations, which is indicated by the Boolean `isLatLon()` method.)

`NavPoint` A 4-D point class, which augments `Position` with time. `NavPoints` can also include supplemental information, such as the point's name and type. `NavPoints` are generally created using a position and a time. Certain `NavPoints` contain metadata that designats them as TCPs. These `NavPoints` are generated as part of the trajectory generation process (Section 6.10.4).

`Velocity` A class describing a velocity. This can be defined using the x, y, z components or using track, ground speed, and vertical speed.

Certain operations may result in `INVALID` instances of these data types, generally indicating a failed computation or invalid input values.

The code segment in Figure 25 illustrates the creation of `Position` and `NavPoint` objects. The constructed `NavPoint np1` is located at latitude 1.8578 deg, longitude -8.1082 deg, and altitude 10,000 ft. It's specified time is 1000 secs.

```
double lat =  1.8578;  // degrees
double lon =  -8.1082; // degrees
double alt =  10000;   // feet
Position p1 = Position.makeLatLonAlt(lat,lon,alt);
NavPoint np1 = new NavPoint(p1, 1000.0);
```

Figure 25: NavPoint creation

The code segment in Figure 26 illustrates the creation of a `Velocity` object.

6.10.3 Plan Class

`Plans` contain an ordered list of `NavPoints`. A `Plan` is said to be *linear* if it contains no TCPs, and *kinematic* if it does. In general only a *consistent kinematic plan* (with

```
    double trk =  20.1; // degrees
    double gs =  555.5; // knots
    double vs =  399.9; // fpm
    Velocity v1 = Velocity.makeTrkGsVs(trk,gs,vs);
```

Figure 26: Velocity creation

proper TCPs) will have continuous (or nearly so) velocities throughout the plan. A linear plan will have significant discontinuities in the velocity vector at vertex points.

There are some occasions where it is desirable to ignore the acceleration zones in a kinematic `Plan` and treat it as just a sequence of linear segments. We call this a *linear interpretation* of the `Plan`. Some of the methods in the `Plan` class provide the ability to interpret and process a kinematic plan linearly. In the *linear interpretation*, velocities and positions are interpolated linearly (along great circles in the case of geodesic positions) between points in the plan—all accelerations are zero. In the *kinematic interpretation*, points marked as TCPs designate acceleration zones and positions and velocities between them are interpolated using kinematic equations. Because it contains no TCPs, a *linear plan* has no acceleration zones and therefore it will behave the same under both interpretations.

The following are the most important methods in the `Plan` class:

`Plan()` The constructor for a generic empty flightplan.

`Plan(String name)` Is the constructor for an empty plan for an aircraft labeled *name*.

`int add(NavPoint np)` Adds a `NavPoint` to the `Plan`. It returns the index of the point.

`int size()` Returns the size of a `Plan`. A 0-size `Plan` may be created if there is an error. Points are indexed by an integer $i : 0 \leq i <$ `size()`.

`Plan planFromState(String id, Position pos, Velocity, v, double startTime, double endTime)` Returns a new `Plan` that is simply a projection of the specified position and velocity. This new `Plan` begins at *startTime* and ends at *endTime*.

`double getTime(int i)` Returns the time attribute for point i in seconds.

`NavPoint point(int i)` Returns the 4-D `NavPoint` for i. The `NavPoint` object includes position and time data, as well as possible additional metadata.

`Position position(double tm)` Returns the (interpolated) 3-D position at time *tm*.

`Velocity velocity(double tm)` Returns the *instantaneous* velocity at time *tm*. For geodesic points, the track angle usually changes throughout a segment. As a result, the track angle of this `Velocity` object may be considerably different than track angles at either end of the segment. Similarly, the velocity will change in kinematic acceleration zones.

`Velocity initialVelocity(int i)` and `Velocity initialVelocity(double t)`: returns the *instantaneous* velocity at a segment's start (i) or at a given time t.

6.10.4 TrajGen Class

This class is concerned with the translation between linear (without TCPs) and kinematic (with TCPs) `Plans`[4].

`Plan makeKinematicPlan(Plan lpc, double bankAngle, double gsAccel, double vsAccel,...)` (several variants): Returns a kinematic version of the linear plan `lpc`, using the given acceleration values. The returned `Plan` will have errors logged if the translation failed (indicating the returned `Plan` is erroneous).

`Plan makeLinearPlan(Plan kpc)`: Returns a linear version of the plan `kpc`. The returned `Plan` will have errors logged if the translation failed (indicating the returned `Plan` is erroneous).

6.10.5 Stratway Class

The `Stratway` class is the primary interface to the strategic resolution capabilities. The most important methods are descibed as follows:

`Stratway()`: Constructor.

`void clearPlans()` Removes all plans and clears all strategy-related messages. It does not clear error/warning messages.

`void addPlan(Plan p)` Adds a `Plan` *p* to stratway. The first plan added must be the ownship's plan, all others are traffic aircraft. If the user retains a copy of the ownship plan, it can be compared to returned versions (via equality) to see what has changed.

`void addPolyPath(PolyPath pp)` Adds a `PolyPath` *pp* to stratway describing a weather or special use polygon to avoid.

`void addDynamicContainmentPolygon(PolyPath pp)` Adds a `PolyPath` *pp* to stratway describing a volume to remain within.

`Plan resolveConflicts()` Is the primary method for solving conflicts. It processes the internal set of `Plans` (added via `addPlan()`), and possibly `PolyPaths`, and returns a new ownship `Plan`. If successful, the new ownship `Plan` will contain no conflicts, and a call to `getResolutionStatus()` will return the value

[4]In the `Detector` class, a kinematic plan is always interpreted linearly. For this reason, the kinematic generator adds an extra point in the middle of a turn.

CONFLICT_FREE. If the resolution fails, getResolutionStatus() will return a value other than CONFLICT_FREE. In this case, the returned Plan may have some conflicts resolved, but not all. This method applies the default set of strategies in the default order. The user may change the default order of strategies using the setStrategies() method. It should be noted that the internal state of the ownship plan is the same as the returned value after a call to this method.

Plan resolveConflicts(int[] strategies) Given an internal set of Plans (added via addPlan()), and possibly PolyPaths, this method resolves the conflicts by applying the strategies in a user-specified order. It alters the internal ownship Plan and returns a copy of the new plan. If successful, this method returns a new ownship Plan with no conflicts and a call to getResolutionStatus() will return the value CONFLICT_FREE. The strategies should be an array (of size maxStrategies in C++, or arbitrary length in Java), with each element indicating a strategy to try (in order).

ResolutionStatusValue getResolutionStatus() Returns the status of the resolution generated by resolveConflicts(). Any value other than CONFLICT_FREE indicates that some conflicts may remain in the plan. This is identical to a call to getResolutionStatus(0).

void setOwnship(String s) Make the plan with name s the ownship. This is accomplished by moving this plan to index location 0.

Detector getDetector() This returns a Detector object that has been populated with detection information for the current set of plans at the current time. If this is called prior to a resolveConflicts() call, the returned object will include any detected conflicts on the original set of input Plans. If this is called after a call to resolveConflicts(), the returned object will include information on unresolved conflicts, if any. Aircraft in the Detector object are identified by the order in which Plans were added to the Stratway object (ownship is 0, first traffic is 1, etc.). The conflicts in the Detector are not necessarily sorted, so the conflict with index 0 is not necessarily the first the ownship will encounter. Stratway does not directly indicate if the input ownship plan is already conflict free. This information can be retrieved by accessing the corresponding Detector object.

Detector getDetector(Plan p) This returns a Detector object that is populated with the specified ownship Plan p and any traffic Plans stored in this Stratway object. This allows one to, for example, examine conflict information on the original (uncorrected) ownship plan, or a plan of interest returned from multipleResolutions(). It behaves otherwise similarly to getDetector().

Various parameters can be set using the provided setter and getter methods. See Section 5, or the StratwayParameters class for a current list of all parameters. Also note that each Plan and PolyPath must have a unique name.

6.10.6 Detector Class

The `Detector` class contains detailed information about all of the conflicts between the ownship and traffic aircraft and polygons. `Detector` objects are expected to be created via the `getDetector()` method in a `Stratway` object. The most important methods are:

`int size()`: Returns the number of conflicts in this detector.

`boolean conflict()`: Returns true if a conflict was detected. The same as the test `d.size() == 0`.

`double getTimeIn(int i)`: Returns the absolute start time of the i-th conflict.

`double getTimeOut(int i)`: Returns the absolute end time of the i-th conflict.

`double getTimeClosest(int i)`: Returns the absolute time of closest approach for the i-th conflict.

`getClosestVert(int i)`: Returns the vertical distance between two aircraft for conflict i, at the time of closest approach.

`getClosestHoriz(int i)`: Returns the horizontal distance between two aircraft for conflict i, at the time of closest approach.

`int getTrafficID(int i)`: Returns the traffic ID for the ith conflict. This is the index number of the `Plan` representing the aircraft in question. The ownship is always index 0.

`void sortConflicts()`: Sort all conflicts based on their start times. Sorting the conflicts will invalidate previous conflict index references. If the conflicts are *not* sorted, then conflict index 0 may not be the first conflict.

`int getNextConflict(double t)`: Returns the index of the next conflict that starts *after* time t. Returns -1 if there are no conflicts starting after t. Note that if the ownship is initially in loss of separation, that conflict "starts" at the ownship plan's start time.

`int getConflictWithTraffic(int ac, double t)`: Returns the index of the "first" ownship conflict with traffic aircraft or polygon ac, starting *at or after* time t. "First" here is based on the current ordering of conflicts, which may be unsorted. If there are no applicable conflicts with the traffic aircraft, this returns a negative value.

It is also possible to populate a `Detector` object with conflict information for an arbitrary set of plans via the `detection` method (and then access the results as listed above); however, this is not necessary for any `Detector` retrieved from a `Stratway` object.

6.11 C++

Method names and behaviors are nearly identical to the Java versions. See the `Stratway.h`, `Detector.h`, and `Plan.h` files for the most current information.

6.12 Example Use of the API

This section presents one example captured in both Java and C++. The example illustrates how to load geodesic points into a plan and how in invoke Stratway to resolve conflicts. Here, the Java example is described in detail. Companion figures for C++ are Figures 30, 31, 32, and 33.

This example illustrates the basic method for invoking the Stratway algorithm. The example is structured in three pieces: loading the data into the plan, loading the plans into Stratway, and finally checking the resolution that comes from Stratway. Figure 27 illustrates how to load `NavPoint` data into a plan using the `add()` method. `NavPoint`'s can be created in a number of ways, and this example uses the `makeLatLonAlt()` factory method. The four parameters to `makeLatLonAlt()` include the degrees of latitude, degrees of longitude (negative means west of the prime meridian), the altitude in feet, and the time in seconds the aircraft will be at that point. Alternatively, the plan data could be read from a file with the `PlanReader` object.

Figure 28 shows how plans are added to the Stratway object with the `addPlan()` method. The first plan added to Stratway is always the ownship plan, unless it is directly changed using the `setOwnship()` method. The resolutions are always a modification to the ownship plan. After the plans are added, Stratway generates a `Detector` object that is used to get the number and times of conflicts.

Finally, Figure 29 shows how to call the Stratway object to create a strategic resolution. Once created, this resolution is displayed if it is different than the ownship plan. Next, the resolution is evaluated (`getResolutionStatus()`) to determine if the resolution was able to resolve all conflicts. In this example `resolveConflicts()` is always called. Alternatively, one could first examine the `Detector` object from Figure 28 and then only call `resolveConflicts()` if there are conflicts.

7 Core Detection Algorithms

Stratway's detection algorithm is ultimately based on a Euclidean state detection class called `CDCylinder` that uses an algorithm called CD3D. The primary functions of this algorithm has been abstracted into an interface called `Detection3D` (in the ACCoRD libraries). Implementing classes must be able to provide a loss of separation (or equivalent) function and a conflict detection (or equivalent) function that sets values for time in, time out, and time of closest approach if a conflict occurs. It is possible to replace `CDCylinder` in Stratway's detection functions with any class that implements `Detection3D`. The `setCoreDetection(Detection3D d)` method allows this behavior modification.

Changing the core detection algorithm will directly affect Stratway's internal

```
// These importings are used...
//import gov.nasa.larcfm.Util.Position;
//import gov.nasa.larcfm.Util.LatLonAlt;
//import gov.nasa.larcfm.Util.NavPoint;
//import gov.nasa.larcfm.Util.Plan;
//import gov.nasa.larcfm.Stratway.Stratway;
//import gov.nasa.larcfm.Stratway.Detector;

Stratway sw = new Stratway();
Plan own = new Plan("Ownship");
Plan traffic = new Plan("Traffic");

own.add(NavPoint.makeLatLonAlt(33.01, -94.50, 35000.0, 1200.0));
own.add(NavPoint.makeLatLonAlt(33.05, -93.87, 35000.0, 1440.0));
own.add(NavPoint.makeLatLonAlt(32.39, -91.24, 35000.0, 2370.0));
own.add(NavPoint.makeLatLonAlt(32.60, -89.51, 35000.0, 3900.0));
own.add(NavPoint.makeLatLonAlt(32.45, -88.17, 35000.0, 4300.0));
own.add(NavPoint.makeLatLonAlt(33.10, -85.33, 18400.0, 5800.0));
own.add(NavPoint.makeLatLonAlt(33.43, -83.81, 18562.8, 7100.0));
own.add(NavPoint.makeLatLonAlt(33.57, -82.27, 18782.1, 7563.1));

traffic.add(NavPoint.makeLatLonAlt(36.02, -88.58, 35000.0, 1565.1));
traffic.add(NavPoint.makeLatLonAlt(34.46, -88.13, 33000.0, 2822.4));
traffic.add(NavPoint.makeLatLonAlt(33.24, -87.68, 31000.0, 3850.3));
traffic.add(NavPoint.makeLatLonAlt(32.63, -87.16, 29000.0, 4800.0));
traffic.add(NavPoint.makeLatLonAlt(30.74, -84.55, 25000.0, 6068.9));
traffic.add(NavPoint.makeLatLonAlt(30.40, -82.70, 23000.0, 7229.9));
```

Figure 27: Stratway Java Example: Load Plans

```
sw.addPlan(own); // ownship is always added first
System.out.println("Ownship plan is:");
System.out.println(own);

sw.addPlan(traffic);
System.out.println("Traffic plan is:");
System.out.println(traffic);

// Show conflict
Detector d = sw.getDetector();
for (int i = 0; i < d.size(); i++) {
  System.out.println("Ownship conflicts with aircraft "+
                    d.getTrafficID(i)+" at time "+d.getTimeIn(i));
}
```

Figure 28: Stratway Java Example: Add Plans to Stratway

detection functionality, as well as any `Detectors`, `KinematicBands`, or `KinematicIntentBands` extracted from that instance of Stratway. Resolution strategies may or may not be affected, depending on the specific strategy. `Bands` and `IntentBands` are hard-coded to use the CD3D algorithm and not affected.

Stratway uses distinct instances for detection and resolution. In the default case, detection uses an instance with D and H values, while resolution uses an instance that also includes buffers. These parameters need to be set individually. The algorithm used in resolutions is determined by the `SetResDetection()` method. Similarly, it is possible to specify the detection algorithm used for polygon detection and resolution using analogous methods.

The “set detection” methods make an internal copy of the Detection3D object, allowing the original to be freely deleted. The “get detection” methods return references to the internal instances, and can be directly modified (but should not be deleted).

7.1 Java Example

The Java example in Figure 34 assumes a new user-defined `User3D` class has been created that implements `Detection3D`. It initially uses the `User3D` algorithm for both detection and resolution, then resets Stratway to use the default `CDCylinder` for detection (but not resolution).

7.2 C++ Example

Figure 35 shows a C++ equivalent of the above example. There are pointer and reference versions of the calls, named to indicate the appropriate interface. This example uses pointers.

As always, the user must take care when dealing with pointers and references in

```
// Resolve conflicts
Plan solution = sw.resolveConflicts();

// See the resolution
if ( ! solution.equals(own)) {
    System.out.println("New plan is: ");
    System.out.println(solution);

    // Check if conflict(s) were resolved
    if ( sw.getResolutionStatus()
         == Stratway.ResolutionStatusValue.CONFLICT_FREE) {
      System.out.println("All conflicts resolved.");
    } else {
      System.out.println("Not all conflicts resolved.");
      d = sw.getDetector();
      d.sortConflicts();
      for (int i = 0; i < d.size(); i++) {
        System.out.println("Ownship conflicts with aircraft "+
          d.getTrafficID(i)+" at time "+d.getTimeIn(i));
      }
    }
} else {
  System.out.println("No conflicts in plan");
}

// Check for errors
if (sw.hasError()) {
  System.out.println(sw.getMessage());
}
```

Figure 29: Stratway Java Example: Resolution

```
#include "Plan.h"
#include <vector>
#include <string>
#include <iostream>

class Data {
  // holds our flightplan data
  public double lat, lon, alt, time; // position data
  public boolean isFixed;            // true if a fixed point
}

larcfm::Plan buildPlan(const std::vector<Data>& d,
                       const std::string& aircraftName) {
  larcfm::Plan p = larcfm::Plan(aircraftName);
 // add the other points
  for (int i = 1; i < d.size(); i++) {
      p.addLL(d[0].lat, d[0].lon, d[0].alt, d[0].time);
  }
  if (p.hasError()) {
    std:cout << "error in planbuilder:" << p.getMessage()
             << std::endl;
    exit(1);
  } else {
    return p;
  }
}
```

Figure 30: Stratway API C++ Example 1: buildPlan

```
#include "Plan.h";
#include "Stratway.h";
#include <vector>

larcfm::Plan fixPlan(larcfm::Stratway& sw,
                     const std::vector<larcfm::Plan>& plans) {
  sw.getMessage(); // clears any error messages
  sw.clearPlans(); // clears internal data

  for (int i = 0; i < plans.length; i++) {
    sw.addPlan(plans[i]);
  }

  return sw.resolveConflicts();
}
```

Figure 31: Stratway API C++ Example 1: fixPlan

C++. `Detection3D` objects internal to a Stratway instance will be destroyed with the Stratway instance or when replaced by a new "set" call. The C++ the user is responsible for cleaning up the original instances of any `Detection3D` objects passed into Stratway.

8 Concluding Remarks

The Stratway program is a strategic conflict resolution program that operates on intent information that is received in the form of plan objects. Plan objects are time sequences of 3-D points in either Euclidean or geodesic coordinates. The program produces a revised ownship plan object that avoids the conflict zones present in the original. The plan objects that are processed can be either linear plan objects or kinematic plan objects. The latter includes acceleration zones and thus provides the ability to define a more flyable, continuous velocity vector. The Stratway program has also been designed to process moving polygons that can be used to represent weather systems or other regions of space that must be avoided. The resolution algorithms produce solutions that remain outside of these moving polygons. The user can also specify a set of polygons that prescribe a time-dependent containment region. If this is done, the Stratway program returns solutions that remain within the union of these *dynamic containment* polygons.

The Stratway program also provides a weather reroute function that uses a grid-based A*-search algorithm. This function can be used to find a new route that avoids polygons, but it does not avoid traffic aircraft. It is different from the standard resolution strategies in that it does not seek to preserve existing points. The weather reroute function can be controlled through several parameters including one that indicates how much weight should be given to the current flight path. The

```
#include "Plan.h"
#include "LatLonAlt.h"
#include "Stratway.h"
#include <vector>
#include <string>
#include <iostream>

class AirCraftData {
  public std::vector<Data> data; // from buildPlan figure
  public std::string name;
}
...
  // inside main()
  larcfm::Stratway sw;
  // assume we have an array of AirCraftData from somewhere
...
  while(sceneriosLeftToExamine) {
    std::vector<larcfm::Plan> plans;
    plans.reserve(numAircraft);
    // populate aircraft data, with ownship data in aircraft[0]
    for(int i = 0; i < numAircraft; i++) {
      plans.push_back(buildPlan(aircraft[i].data, aircraft[i].name));
    }
    larcfm::Plan result = fixPlan(sw,plans);
    if (sw.hasError()) {
      std::cout << "Stratway encountered an error: "
                << sw.getMessage() << std::endl;
      // error handling ...
    } else if (sw.getResolutionStatus == CONFLICT_FREE) {
     if (result == plans[0]) {
        // there were no conflicts to start with! ...
     } else {
        // success!  re-integrate your data
        for (int i = 0; i < result.size(); i++) {
          double pointTime = result.getTime(i);
          LatLonAlt pointPosition = result.getPositionLL(i);
          // feed point data back into your system ...
        }
     }
    } else {
      // resolution failed, perform a tactical solution ...
    }
    ...
   } // end while loop
```

Figure 32: Stratway API C++ Example 1: main() fragment

```
#include "Plan.h"
#include "LatLonAlt.h"
#include "Stratway.h"
#include <iostream>


   ...
   if (sw.getResolutionStatus == larcfm::CONFLICT_FREE) {
     // success ...
   } else {
     // conflicts still remain
     larcfm::Detector d = sw.getDetector();
     // print out conflicts in order:
     d.sortConflicts();
     for (int i = 0; i < d.size(); i++) {
       std::cout << "Ownship conflicts with aircraft "
                 << d.getTrafficID(i) << " at time "
                 << d.getTimeIn(i) << std::endl;
     }
   }
   ...
```

Figure 33: Stratway API C++ Example 1: Detector access

```
import gov.nasa.larcfm.Stratway.Stratway;
import gov.nasa.arcfm.ACCoRD.CDCylinder;
  ...
  Stratway sw = new Stratway();
  Detection3D d = new User3D();
  sw.setCoreDetection(d); // internal copy
  sw.setResDetection(d);  // internal copy
  // Load Plans here
  ...
  Detector det = sw.getDetector();
  //process det here
  ...
  sw.setCoreDetection(new CDCylinder());
  // use CDCylinder
  ...
```

Figure 34: Java setCoreDetection()

```
#include "Stratway.h"
#include "CDCylinder.h"
#include "Detection3D.h"
  ...
  Stratway sw = new Stratway();
  Detection3D *d = new User3D();
  sw.setCoreDetectionPtr(d);
  sw.setResDetectionPtr(d);
  delete d; // be sure to free this object!
  // Load Plans here
  ...
 Detector det = sw.getDetector();
  //process det here
  ...
  d = new CDCylinder();
  sw.setCoreDetectionPtr(d);
  delete d; // be sure to free this object!
  // use CDCylinder
  ...
```

Figure 35: C++ setCoreDetection()

reroute function can be used in conjunction with the resolution strategies to produce trajectories that are conflict-free from other aircraft.

The Stratway program has been developed so that it can be easily executed from other programs. It has a simple application program interface (API) which is available in either Java or C++. The program is highly user configurable through use of parameters. The strategic detection algorithm used in Stratway (i.e., `CDII`) is also highly configurable, enabling different definitions of the protection zones around aircraft.

9 References

1. Ricky Butler, George Hagen, and Jeffrey Maddalon. The chorus conflict and loss of separation resolution algorithms. Technical Memorandum NASA/TM-2013-218030, NASA, Langley Research Center, Hampton VA 23681-2199, USA, Aug 2013.

2. INTENT Consortium. INTENT project: The transition towards global air and ground collaboration in traffic separation assurance. http://www.intentproject.org.

3. Federal Aviation Administration (FAA). Trajectory Based Operations. http://www.faa.gov/nextgen/portfolio/sol_sets/tbo/.

4. George Hagen and Ricky Butler. Towards a formal semantics of flight plans and trajectories. Technical Memorandum NASA/TM-2014-218662, NASA, Langley Research Center, Hampton VA 23681-2199, USA, Dec 2014.

5. George Hagen and Ricky Butler. Stratway visualization tool user's manual. DRAFT to be published, NASA, Langley Research Center, Hampton VA 23681-2199, USA, April 2016.

6. David A. Karr, Robert A. Vivona, Stephen M. DePascale, and David J. Wing. Autonomous operations planner: A flexible platform for research in flight-deck support for airborne self-separation autonomous operations planner: A flexible platform for research in flight-deck support for airborne self-separatio. In *12th AIAA Aviation Technology, Integration, and Operations (ATIO) Conference*, 2012.

7. RTCA SC-186. *Minimum Aviation System Performance Standards for Automatic Dependent Surveillence Broadcast (ADS-B)*. RTCA, 2002.

8. Barry N. Taylor and Ambler Thompson. The international system of units (SI). Technical Report NIST Special Publication 330, National Institute of Standards and Technology, Gaithersburg, MD, 2008.

9. NASA Langley Formal Methods Team. Airborne coordinated conflict resolution and detection (ACCoRD). http://shemesh.larc.nasa.gov/people/cam/ACCoRD/.

Appendix A

NavPoints, Positions, and Velocities

The `NavPoint` Class embellishes a `Position` with various attributes. These attributes fall into two basic categories:

- history of modification,
- acceleration features (i.e., TCPs).

These attributes are used by the Stratway program. They allow the Stratway user to constrain the solution space of the program.

The `Position` class provides a common interface for both Euclidean and geodesic positions. Figure A1 illustrates the different `Position` constructors. The method `isLatLon()` returns a boolean value indicating whether the position is geodesic or not. Factory methods are also available, as seen in Figure A2.

```
LatLonAlt lla = LatLonAlt.make(lat, lon, alt);
Position pos = new Position(lla);      // create with Lat/Lon data
double x,y,z;
Position pos2 = new Position(x, y, z); // create with Euclidan data
Vect3 v3;
Position pos3 = new Position(v3);      // create with Euclidan data
```

Figure A1: Position constructors

```
Position pos4 = Position.makeLatLonAlt(lat, lon, alt)
Position pos5 = Position.mkLatLonAlt(lat, lon, alt)
Position pos6 = Position.makeLatLonAlt(lat, lat_unit, lon, lon_unit,
                   alt, alt_unit);
Position pos7 = Position.makeXYZ(double x, double y, double z);
Position pos8 = Position.mkXYZ(double x, double y, double z);
Position pos9 = Position.makeXYZ(x, x_unit, y, y_unit, z, z_unit);
```

Figure A2: Position factory methods

Most of our programs allow either Euclidean or geodesic data, but they cannot be mixed. If the first data point entered is Euclidean then all of the rest of the data must also be Euclidean. If the position is Euclidean the accessors `x()`, `y()` and `z()` can be used to retrieve the components. If the position is geodesic, then the accessors `lat()`, `lon()`, `alt()` are used. These return the results in standard internal units (SI). If output is desired in `degrees` and `feet`, then the following accessors are also available `latitude()`, `longitude()`, and `altitude()`. The methods `xCoordinate()`, `yCoordinate()`, and `zCoordinate()` return Euclidean data in units of `nmi`, `nmi`, and `ft`. Note that short names return internal units (SI) and long names return ATM customary units.

```
    Vect3 v = new Vect3(1,2,3);
    Velocity vel1 = Velocity.make(v);
    Velocity vel2 = Velocity make(Vect2 v);
    Velocity vel3 = Velocity mkVxyz(vx, vy, vz);
    Velocity vel4 = Velocity makeVxyz(vx, vy, vz);
    Velocity vel5 = Velocity mkTrkGsVs(trk, gs, vs);
    Velocity vel6 = Velocity makeTrkGsVs(trk,gs, vs);
```

Figure A3: Velocity constructor and factory methods

The Velocity class extends a `Vect3` class with additional machinery. Figure A3 illustrates how a `Velocity` object is created The following methods are useful:

`track()`	returns track angle in radians (i.e. $-\pi$ to π).
`compassAngle()`	returns the compass angle (i.e. 0 to 2π).
`groundSpeed()`	returns the ground speed
`verticalSpeed()`	returns the vertical speed
`Hat()`	returns a unit velocity vector in same direction

Appendix B

Debugging Tools

Several methods have been included to aid the user in debugging programs. These methods are found in the `PlanUtil` class. There are also several print methods available in the `Stratway` class including: `dumpParameters()`, `currentPlansToOutput()`, `containmentPolysToOutPut()`, `localDataToOutput()`, and `toString()`.

`void savePlan(Plan plan, String fileName)` Write the plan to the output file `fileName`.

`void savePlans(List<Plan> plans, String fileName)` Write the set of plans to the output file `fileName`.

`boolean isConsistent(Plan p, boolean silent)` Check the plan `p` to make sure the plan is well-formed and that the beginning and ending TCP pairs have a correct mathematical relationship. Problems are written to standard output. If `silent` is set to true, then the output messages are surpressed.

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704–0188

1. REPORT DATE *(DD-MM-YYYY)* 01-05-2016	**2. REPORT TYPE** Technical Memorandum	**3. DATES COVERED** *(From - To)*
4. TITLE AND SUBTITLE The Stratway Program for Strategic Conflict Resolution: User's Guide		**5a. CONTRACT NUMBER** **5b. GRANT NUMBER** **5c. PROGRAM ELEMENT NUMBER**
6. AUTHOR(S) Hagen, George E.; Butler, Ricky W.; Maddalon, Jeffrey M.		**5d. PROJECT NUMBER** **5e. TASK NUMBER** **5f. WORK UNIT NUMBER** 154692.02.40.07.01
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) NASA Langley Research Center Hampton, Virginia 23681-2199		**8. PERFORMING ORGANIZATION REPORT NUMBER** L–20698
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) National Aeronautics and Space Administration Washington, DC 20546-0001		**10. SPONSOR/MONITOR'S ACRONYM(S)** NASA **11. SPONSOR/MONITOR'S REPORT NUMBER(S)** NASA/TM–2016–219196

12. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified-Unlimited
Subject Category 03
Availability: NASA CASI (443) 757-5802

13. SUPPLEMENTARY NOTES
An electronic version can be found at http://ntrs.nasa.gov.

14. ABSTRACT
Stratway is a strategic conflict detection and resolution program. It provides both intent-based conflict detection and conflict resolution for a single ownship in the presence of multiple traffic aircraft and weather cells defined by moving polygons. It relies on a set of heuristic search strategies to solve conflicts. These strategies are user configurable through multiple parameters. The program can be called from other programs through an application program interface (API) and can also be executed from a command line.

15. SUBJECT TERMS
air traffic, conflict, detection, resolution, avoidance, secondary conflict, weather polygons, application programming interface, verification

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			STI Help Desk (email: `help@sti.nasa.gov`)
U	U	U	UU	57	**19b. TELEPHONE NUMBER** ***(Include area code)*** (443) 757-5802