Autonomous Agents for Air-Traffic Deconfliction

Michal Pěchouček, David Šišlák, Dušan Pavlíček and Miroslav Uller

Gerstner Laboratory – Agent Technology Group
Department of Cybernetics, Czech Technical University
Technická 2, Prague, 166 27, Czech Republic

pechouc@labe.felk.cvut.cz

ABSTRACT

This contribution presents a deployment exercise of multi-agent technology in the domain of deconflicted air-traffic control among several autonomous aerial vehicles (manned as well as unmanned). Negotiation based deconfliction algorithm have been developed and integrated in the agent-based model of the individual flight. Operation of the underlying multi-agent system has been integrated with freely available, geographical and tactical data sources in order to demonstrate openness of the technology. An additional, web client visualization and access component has been developed in order to facilitate a multi-user, platform independent use of the system. The features and application design is illustrated in the demonstration video clip[1].

1. INTRODUCTION

In the future warfare and humanitarian relief operations (especially in the surveillance and monitoring domains) there will be a strong potential for integration of the technologies and mechanisms supporting coordinated flight among the collective of autonomous manned and unmanned aerial vehicles. In this paper we argue that agent technology, relying on collective decision making among several autonomous intelligent agents, can provide solution for such a challenge. Even though this is not a fundamental research paper, this contribution provides good evidence of practical applicability of agent technology by reporting on a technology deployment exercise in this specific domain.

According to [11] the current air traffic control methods based on rigidly structured airspace have shown to be inefficient even for the future coordination of the piloted aircraft. This is true mainly due to: (i) inefficiency of airspace utilisation that is based on fixed predefined flight corridors, (ii) increased air traffic workload given by ever increasing air traffic density and (iii) use of obsolete technologies, that are in many cases 30 years old.

[1] `http://agents.felk.cvut.cz/atg-videos/atc_video_divx_titles.avi`

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
AAMAS'06 May 8–12 2006, Hakodate, Hokkaido, Japan.
Copyright 2006 ACM 1-59593-303-4/06/0005 ...$5.00.

In the future these will be important for both UAVs and manned aircraft:

- requirements to increase flexibility of the vehicles operations,
- requirements to fly higher number of vehicles in substantially smaller airspace and
- requirements for higher precision of the flight plan and its accomplishment.

Due to the first two reasons of inefficiency listed above, the classical air traffic control methods are not very suitable for coordination of higher numbers of dynamically tasked aerial vehicles. In our work, we have built on top of the concept of *free flight* [9], [4] – an approach to autonomous routing of the aircraft and continuous self-optimization of the flight trajectory complemented by the peer-to-peer deconfliction mechanism.

The presented work resulted in a demonstration prototype (ATC) of a multi-agent system modeling deconflicted free-flight of collective of fully autonomous aircraft.

ATC has not been primarily designed as flight planner producing a set of deconflicted route plans for a number of aerial vehicles. There is a central planning component that provides (during the *planning phase*) the flight plans without consideration of possible operation of other aircraft. This planning process can be easily distributed among the individual aircraft and thus no central planner would be necessary. The plans are likely to contain possible collisions. Working with no-collisions makes planning considerably faster and much more flexible. This also results in substantial scalability improvement affording to operate higher number of aerial vehicles in condensed airspace (can be used e.g. in situation where a higher number of unmanned aerial vehicles carries out rapid surveillance tasks).

Following the planning phase, ATC performs the *simulation phase*, where the aircraft implement the provided flight plans in a free-flight fashion. In a danger of possible collisions, the embedded deconfliction mechanism makes sure that aircraft negotiate deconfliction maneuvers, while still optimizing their operation.

2. DECONFLICTION BY NEGOTIATION

ATC is designed to operate in distributed manner, which is why the deconfliction technology (while developed within the multi-agent ATC model) is ready for deployment on autonomous vehicles without any central point of control.

The aircraft are modeled by agent containers hosting several agents. In this contribution we will be referring to

agents representing an aircraft auto-pilot. This agent is a self-interested entity that is in charge of (i) preparing a detailed flight plan for the airplane respecting time-specific waypoints for the airplane's mission and (ii) executing the flight plan by performing the flight.

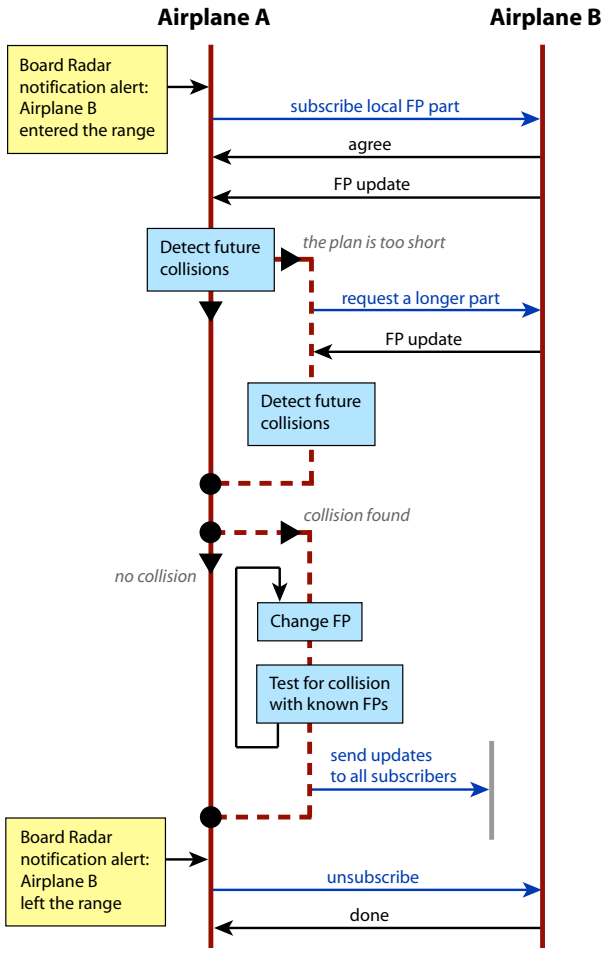

Figure 1: Negotiation protocol

Each simulated aircraft is surrounded by a number of concentric spherical zones: **Communication**, **Alert**, **Safety** and **Collision** zones. The **communication zone** is the outermost one. It represents the communication range of the data transmitter onboard the aircraft. Using this data, the aircraft can send data packets to other aircraft that are positioned within the specified spherical zone defined by its radius. The **alert zone** defines the operation range of the radar onboard the aircraft. If another aircraft is located within the alert zone, the aircraft are periodically notified about its relative position and its flight code. The **safety zone** encapsulates the area around an aircraft that other aircraft are not allowed to enter in order to minimize the mutual influence of the aircraft movements, e.g. airspace turbulence. If two aircraft do enter each other's safety range, they can still continue flying but their flight path may be influenced by e.g. turbulence. This is not the case when two or more airplanes fly together in a close formation. The **collision zone** is the innermost zone. It defines the critical contact area. When the mutual distance between two aircraft is smaller than the sum of their collision zone radiuses, the physical collision happens.

Cooperative deconfliction. In the simulation phase the ATC system solves collisions *cooperatively* by negotiation between the aircraft, see Figure 1. Let us suppose an aircraft `A` to fly along its planned optimal flight path through its mission waypoints. An airplane `B` enters the *alert zone* of the airplane `A`. The pilot agent of the aircraft `A` is notified about its position and flight code by the on board radar system. The pilot agent of the aircraft `A` tries to establish negotiation connection with the pilot agent of `B`. In the case when the connection cannot be established or the communication is not trusted, the pilot agents should use *non-cooperative approach*, described later in this section. If the connection was established successfully, the pilot agent `A` subscribes for a local area flight plan of the aircraft `B` (representation of the flight plan is described in Section 5.1). The pilot agent of aircraft `B` sends an update to the subscriber every time it changes its own flight plan. The update contains the part of the flight plan of the aircraft for the specified amount of time depending on the flight speed. The update is also sent when the time span of the previous update was not long enough and it needs to be updated again. When the pilot agent `A` receives an update from the pilot agent `B`, it executes the **collision detection process** on its own flight plan and the received one.

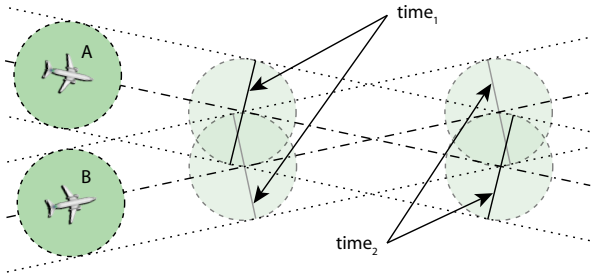

Figure 2: Flight plan collision interval

The collision detection process is a linear algorithm that analyses two flight paths and tests the condition of safe flights. If there is a specific point in time detected when the distance between the positions of the aircraft is smaller than the maximum of the safety ranges of the aircraft (see Figure 2), the detection process returns $time_1$ and $time_2$ which represent the first and the last collision point between the two flight plans. This information is needed by the pilot agent for better handling of the situation.

If the detection test is negative, both flight paths are safe for flying. If a collision is found, the airplanes `A` and `B` must modify their flight paths. The first prototype of the designed system uses a rule-based approach for modifying flight plans described in Section 2.1.

Non-cooperative deconfliction: The distributed deconfliction approach is open to be extended to *non-cooperative deconfliction*. The non-cooperative deconfliction is useful in situations when an airplane has a malfunctioning transmitter/receiver on its board or in the situation when there is an intruder/enemy airplane with adversarial behavior [7]

which intentionally sends incorrect future flight path parts to the others. The most suitable approach to the *non-cooperative deconfliction* is the game theory. In this case the pilot agent tries to change its own flight plan in a way that would guarantee a minimal collision risk for any future position of the other airplane. To determine all possible future positions of the other plane, information about its current position, direction and information about its type can be used. The monitored object's flight path is always continuous but there are also certain restrictions that depend on the airplane type – e.g. minimal/maximal flight speed, minimal radius of turning, etc. When the pilot agent wants to identify whether or not it should use the non-cooperative deconfliction for a particular airplane, it can integrate a special *detection module*. The detection module compares the notification received from the board radar with the known flight plan part of the aircraft in the radar range.

2.1 Collision Avoidance Mechanism

The specific collision type is determined based on the angle between the direction vectors of the concerned planes at $time_1$ projected to the ground plane (defined by X and Y axes), see Figure 3.

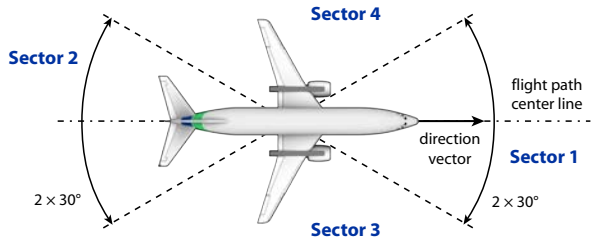

Figure 3: Identification of the collision type

Depending on the computed angle, the airplane `B` falls into one of four sectors surrounding the airplane `A`. Depending on that sector, one of the following rules is applied on the flight plan of the airplane `A` to avoid the collision:

- **Sector 1** – head-on collision, in this case the airplanes avoid each other by both of them turning to the right. The flight plan is changed as shown in Figure 4. The pilot agent shifts the plan points at $time_1$ and $time_2$ to the right, perpendicularly to the old direction vector. The length of the shift is equal to a minimum of safety ranges of both airplanes. Beyond $time_2$, the flight plan follows the shortest way to the next mission waypoint.
- **Sector 2** – rear collision, there are two subcases: i) the front aircraft is faster – airplanes do not change their current flight plans; ii) the rear airplane is faster – it has to change its flight plan so that it turns to the right and passes the front airplane without endangering it. The flight plan is similar to that in Figure 4. To achieve this, the rear airplane shifts its flight plan points at $time_1$ and $time_2$ to the right, perpendicularly to the old direction vector. The length of the shift is at least 1.1 times of the safety range.
- **Sector 3** – side collision, the airplane `B` has higher traffic priority. The aircraft `A` needs to slow down its speed so that it reaches the collision point at $time_1$ later than the airplane `B`. If this is not possible due to the minimal flight speed defined for each airplane type, the airplane `A` slows down as much as possible and shifts its flight plan point at $time_1$ to the right so that there is no collision between the two flight plans.
- **Sector 4** – side collision, the airplane `B` has lower traffic priority. The aircraft `A` changes its flight plan by increasing its flight speed so that it passes the collision point before the airplane `B`. The airplane `A` only accelerates as much as needed.

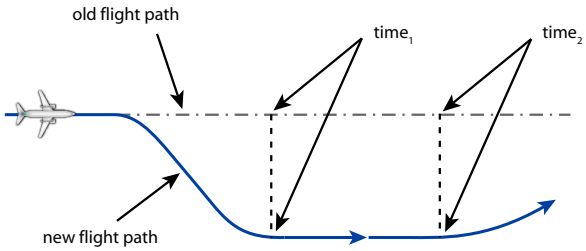

Figure 4: Change of the flight plan.

The above rule-based changes to the flight plan are carried out by both planes independently because each aircraft detects the possible collision with the other airplane from its own point of view. After applying the changes to the flight plan, the airplane sends an updated local flight plan part to all airplanes that subscribed for it. The change is also verified against all other known flight plans of all aircraft monitored by the board radar system. If another collision is detected, new changes are applied.

The pilot agent internally uses the *flight plan wrapper* interface for manipulation with its flight plan. The requests for each plan modification are handled as a special set of solver time-constrained waypoints. A special handling algorithm takes care of the execution of each modification that overrides the previous one.

3. A-GLOBE: MULTI-AGENT PLATFORM

The agent-based part of the designed prototype of the Air Traffic Control (ATC) system runs on the **$\mathcal{A}$-globe** Java multi-agent platform [10, 1]. **$\mathcal{A}$-globe** is a fast and lightweight platform with agent ***mobility*** and ***inaccessibility*** support. Besides the functions common to most of agent platforms it provides ***Geographical Information System***-like service to the user. Therefore, the platform is ideally suited for testing experimental scenarios featuring agents' position, position dependent environment simulation and communication inaccessibility. The platform provides support for permanent and mobile agents, such as communication infrastructure, storage, directory services, agent migration (including library migration with version handling), service deployment, etc.

$\mathcal{A}$-globe is optimized to consume just a limited amount of resources. **$\mathcal{A}$-globe** platform is not fully compliant with the FIPA [3] specifications, still it implements most protocols and respects the spirit of the specification. It does not support communication between different agent platforms (e.g. with JADE, JACK, etc.). For large scale scenarios, the problems with system performance that interoperability

brings (memory requirements, communication speed) outweigh any advantages, as heterogenous environment is of limited interest for simulations.

The $\mathcal{A}$-**globe** operation is based on several core components: The **Agent Platform** provides the basic components necessary to run one or more *agent containers*, the *container manager* and the *library manager*. The **Agent Container** is a skeleton entity that provides basic functions such as communication, storage and management for agents. The **services** provide common functions for all agents in a container. The agents have two means of communication with a service – either via standard messages or by using a service shell – a special proxy object that interfaces service function to a client so that they appear to be synchronous function calls, but the calls are actually handled in the service thread thereby preventing deadlocks and synchronization overhead.

The $\mathcal{A}$-**globe** platform is primarily aimed at large scale, real world simulations with fully fledged agents. To support this goal, it includes a special infrastructure for environmental simulation. *Actor agents* play roles in the simulated world, while *Environment Simulation (ES) agents* implement the simulated world itself. ES agents only rarely use messages to communicate with actor agents. Instead, they communicate via *topic messaging*.

Topic messaging implements container to container messaging reserved for easy environmental simulation. Topic messaging is built on top of standard messages and is managed by the Geographic Information System (GIS) Services – server and clients. GIS services provide distribution and subscription mechanism for the agents. ES agents can be responsible for nearly any simulation layer, depending on the wishes of the developers. Accessibility agent, which controls the availability of communication links between containers holding the actor agents, is one of the most important of ES agents. $\mathcal{A}$-**globe** messaging layers use the information provided by the accessibility agent to prevent sending messages between inaccessible nodes. Accessibility simulated by the system can depend on many factors, typically including the distance and simulated link reliability.

4. PROTOTYPE STRUCTURE OVERVIEW

Air Traffic Control (ATC) system is mainly written in Java except for the real-time visualization component which is written in C++. Agent-based part of the system runs on the $\mathcal{A}$-**globe** Java multi-agent platform. The system consists of several components, see Figure 5:

– **ATC core** is a mandatory component of the system responsible for aircraft simulation and airways planning. The component also provides interfaces for connecting a number of real-time visualization components and remote web clients. See Section 4 for details.

– **Real-time 2D/3D visualizer** is an optional component that provides the user with a real-time overview of the simulation state with all important information in a 3D/2D environment. See Section 7 for details.

– **Remote web client** is an optional component allowing a remote user to connect and interact with the ATC core system via a client application executed from an internet browser. If necessary, all communication data can be secured using asymmetric cryptography but it comes with higher processor load requirements. See Section 7 for details.

Agent-based core system encapsulates one *server component* (described in Section 4.1) and one or more *platform components*, Figure 5. The *platform component* is used as a registration unit for starting the simulated *aircraft container* (described in Section 4.2) inside Java Virtual Machine (JVM). When ATC system is used for planing/simulation of a huge number of aircraft, it is highly recommended to use several host computers with their own JVMs and *platform components*. The number of running aircraft is proportionally split between the registered platform components. This enables the ATC system to balance the overall load between all registered computers.

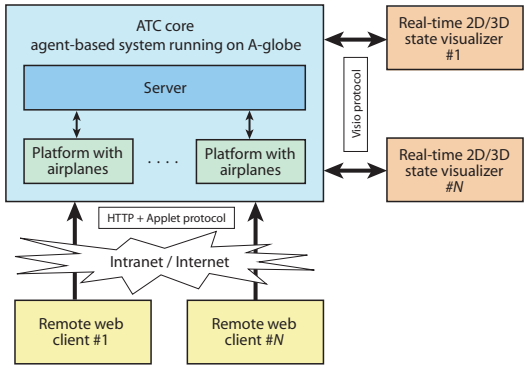

Figure 5: ATC System Structure Overview

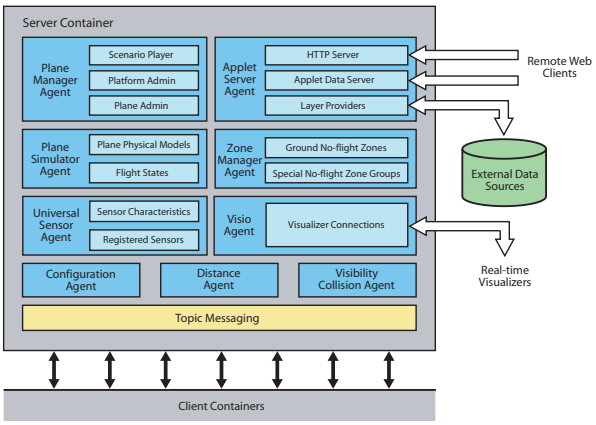

Figure 6: Server component of the ATC Core system

4.1 Server Component

The *server component* of the ATC core system is a sole central element of the system. It simulates positions of aircraft and other objects in the simulated world, aircraft hardware, weather conditions, communication ranges given by the ranges of board data transmitters, etc. It is also responsible for acquiring information about all airplanes and provides them to both types of visualizers. If the proposed distributed agent system for flying on deconflicted airways was used to control real aircraft, this *server component* could be removed from the system.

The *server component*, Figure 6, consists of several agents. **Configurator Agent** loads initial configurations from the specified configuration files and distributes them to other agents. **Plane Manager Agent** administrates connected

JVM platforms and running aircraft containers, it spawns new airplanes and removes the existing ones, and assigns initial flight missions to the airplanes. New airplanes can be started automatically as specified in the scenario script. The server also acts as a load balancer between connected platform components. **Plane Simulator Agent** computes the current position of the aircraft in the simulated world. It contains all physical models for all plane types and keeps all current flight plans and states of the running aircraft. When a plane *pilot agent* changes some part of the flight plan, the change is propagated via the *plane agent* to the *plane simulator agent* in the form of a difference flight plan update. The agent can be asked for the current airplane position by the *pilot agent*. **Distance Agent** calculates Euclidian distances between each pair of existing aircraft using their current positions. **Visibility Collision Agent** prepares **$\mathcal{A}$-globe** visibility updates [10] for controlling communication restrictions between airplanes. It also detects whether there has been a physical collision between flying aircraft. The airplanes that have collided with any other object are uncontrollable and they fall down to the ground. Falling aircraft can endanger any airplane that flies under it. **Universal Sensor Agent** represents all radar sensors on aircraft boards. **Zone Manager Agent** takes care of no-flight zones. It transforms any defined no-flight zones to a compressed octant tree. The ground surface is also represented as a special no-flight zone. This allows to use the planning mechanism even for generating flight plans that do not collide with the ground surface. No-flight zones can be dynamically changed during the planning/simulation. **Visio Agent** is an interface between the Core agent system and the C++ real-time visualizers. To ensure fast communication between them, a special binary communication protocol defined is used. **Applet Server Agent** runs the HTTP server, Applet Data server and all external data providers. It provides a communication interface between the agent system and the *remote web client* (described in Section 7).

All server agents communicate together using **$\mathcal{A}$-globe *topic messaging*** described in [10].

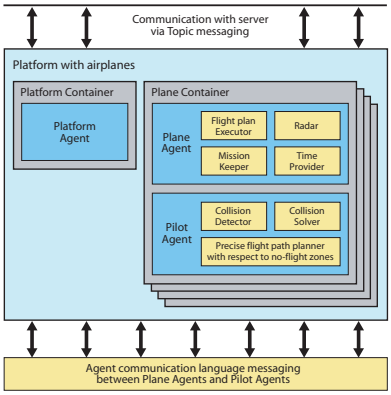

Figure 7: Platform Architecture

4.2 Platform Component

The *platform container* is used as a registration unit for starting containers with agents simulating aircraft behavior. Design of the platform component is shown in the Figure 7. The only agent running in the *platform container* is the *platform agent*. The agent acts as a control bridge between the *plane manager agent* and the local JVM on which it is running. Through this agent the *plane container* can be started or removed. Many *plane containers* can run in each *platform component* (inside one JVM) or there can be no *plane containers* at all.

Each running *plane container* hosts two agents. **Plane Agent** provides high-level airplane functions, such as flight plan execution in cooperation with the *plane simulator agent*, radar and detector readings, airplane configuration and time synchronization ticks. **Pilot Agent** is the main control unit of the simulated aircraft handling negotiation based deconfliction as described in Section 2. It processes notifications about every new visible objects on the radar and tries to communicate with the respective *pilot agent* if there is any. It generates a detailed flight plan based on the initial mission specification given by the *plane manager agent* with respect to the no-flight zones (Section 5), described in Section 5.2.

The airplane container agents communicate with the server container via **$\mathcal{A}$-globe *topic messaging***, while together they communicate using standard agent communication language (ACL) messages.

5. FLIGHT MODELING

The modeling of flights in our system proceeds in two main phases: the *planning* of the flight plans for each airplane and the *simulation* of these flight plans. In this section, we give a brief description of the planner component and other key concepts related to the flight modeling. We use a very simple physical model of the airplanes: they are modeled as mass points moving along a previously planned trajectories.

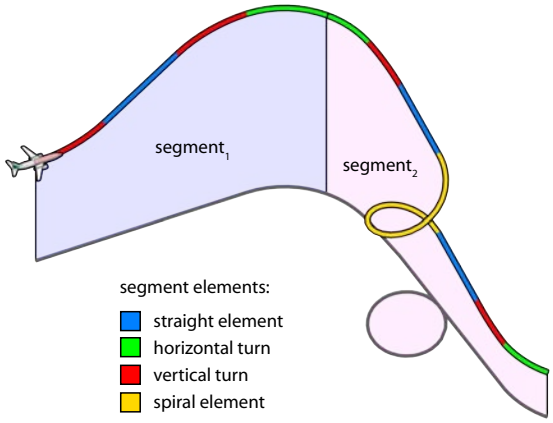

Figure 8: Flight plan, segments, elements

5.1 The Structure of Flight Plan

The flight plan describes a trajectory which the plane follows during its flight. Its principal building blocks are *waypoints*, *segments* and *elements*. A waypoint is an important navigational point used for rough definition of the flight route; it represents a certain location and also specifies the time interval when the airplane should fly through it. The waypoints serve as an input to the planner, which generates the precise flight path consisting of the segments and elements. A segment is a portion of a planned flight path

between two adjacent waypoints. The segments are composed of elements, which constitute the most basic parts of a flight plan with a simple geometry.

Our planner is able to generate flight paths from four types of elements: straight elements, turns (horizontal and vertical), spirals and a warp element. The warp element is a special kind of element allowing for an abrupt change of certain parameters of the flight route (e.g. position, direction or velocity). Without warp elements, the flight trajectory is always continuous and smooth and each element is connected smoothly to the adjacent ones. This is desirable in most cases; exceptions to this rule will be mentioned later. An example of a flight plan is shown in Figure 8.

5.2 Flight Path Planning

By the flight path planning we mean creating a smooth and continuous path passing through all input waypoints, which will also satisfy the time constraints associated to the waypoints. The planning proceeds in three phases: the computation of the actual path without regard to the time constraints, the adjustment of the flight plan to satisfy the time criteria and the replanning of the plan in the parts where it collides with no-flight zones (if defined).

During the **Path Planning**, the planner generates a detailed flight path which passes through all the waypoints. For each couple of successive waypoints, a segment is created, initially without any elements. A segment represents the smallest part of a flight plan which can be planned independently on the other parts of the plan. By planning, we mean filling of a segment with elements; the planned segment is typically composed of 2 to 8 elements.

After that, the **Time Planning** follows. For each segment, the actual flight plan is adjusted so that the time of flight through the segment matches the time intervals defined in the start and end waypoints belonging to the segment. The adjustment is done in two ways: through stretching the segment by altering some of its parameters (so that it becomes longer and thus the flight takes more time) and through setting the velocity of the plane at the beginning of the segment. For this reason, we place a warp element at the start of each segment, which allows for a step change of velocity at the segment boundaries. By using this simplification, our planning algorithm is very fast.

The flight path for an airplane can be planned with respect to predefined no-flight zones (such as areas around power plants, military facilities or natural terrain obstacles). **Avoiding the no-flight zones** is done as follows: the segments of the flight plan generated in previous phases are tested for collisions with the no-flight zones. If any segment intersects a no-flight zone, it is replaced with a flight path which bypasses the respective no-flight zone(s). This is accomplished through finding suitable "bypassing" waypoints, which roughly define the collision-free flight route, and through planning a flight path through them. The situation is shown in Figure 9.

The problem of finding the bypassing path is thus equivalent to finding suitable waypoints that define it. These waypoints have no time constraints. Our approach to finding these waypoints is based on a spatial subdivision of the area of interest by an octal tree to a set of cubic cells. The cells intersected by a no-flight zone are marked as full, while the rest of them is marked as empty. By using the well-known A^* algorithm we find a sequence of empty cells, connecting

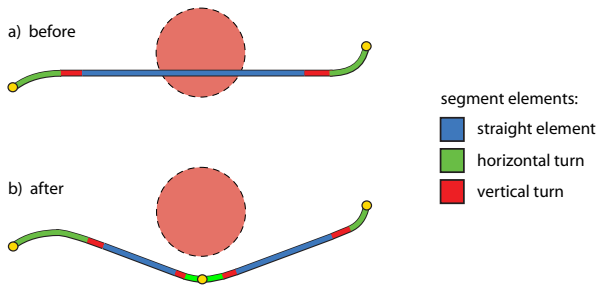

Figure 9: Avoiding the no-flight zone.

the cells containing the start and end waypoints of the replanned segment; this sequence forms a continuous "tunnel" which bypasses the no-flight zones. The waypoints are then placed in the "tunnel". . This approach is similar to that described in [6].

6. REAL DATA INTEGRATION

The designed ATC system integrates four data sources:

- **National Aeronautics and Space Administration (NASA)** – a mosaic of Landsat7 images[2] at the maximum resolution of approximately 50 meters per pixel was used as an underlying ortophoto map of the United States
- **U.S. Geological Survey (USGS)**[3] – detailed vector shapes of 50 U.S. state boundaries; a set of more than 650 U.S. airports, including their names, GPS coordinates with the corresponding average numbers of enplanements per year; a set of more than 26 thousand major U.S. highway segments
- **Geographic Names Information System (GNIS) database**[4] – a set of more than 24 thousand U.S. populated places, including their names, GPS coordinates with the corresponding population; a set of more than 80 U.S. powerplants, including their names with GPS coordinates to act as the no-flight zones
- **ATC flight data** – airplanes including position and direction; rough flight plans in the form of series of points in space through which the airplane is supposed to fly on its way; detailed flight plans

7. FLIGHT VISUALIZATION

The **real-time visio** allows the user to overview the entire simulation in a 2D/3D environment, to efficiently navigate through it, but also to provide the user with all the important data and information at the same time.

An open-source 3D game engine CrystalSpace [2] was used as a platform on top of which the visualization component is built. For efficiency and performance reasons, the entire visualizer is written in C++ and it internally utilizes the OpenGL graphics engine. A binary-coded network communication protocol has been defined and implemented to allow fast data exchange between the core system and the visualizer component.

[2] `http://onearth.jpl.nasa.gov`
[3] `http://seamless.usgs.gov`
[4] `http://geonames.usgs.gov`

The communication is duplex: by sending messages to the visualizer, the core system notifies the visio that an airplane has been spawned, destroyed, it has changed its position etc. On the other hand, the visualizer sends messages to the core system e.g. when it requests specific information about a certain airplane upon user's demand, such as its flight plans. In response to this request, the simulation system selectively sends the required data. This behavior was introduced to reduce the amount of data that need to be sent over the network – instead of sending all the data continuously even if they are not needed.

The visualizer provides two main display modes: a two-dimensional (2D) view (Figure 10) and a three-dimensional (3D) view (Figure 11). They both work with the same underlying data, only the way of depicting them differs.

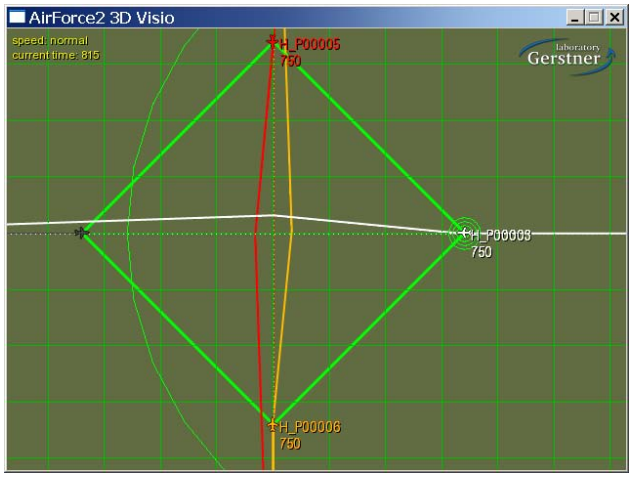

Figure 10: Real-time visualizer: 2D view

The **2D mode** provides a radar-like top view of the scene. Airplanes present in the system are displayed as 2D icons that visualize both current position and direction of each airplane. Once the airplane is selected, the camera starts following it so that it stays in focus. A set of zones is displayed around the selected airplane as a group of concentric circles. Most importantly, these refer to the collision range (innermost) and the visibility range (outermost) of each airplane. The current flight plan can be displayed for the selected airplane and/or for all airplanes that are visible to it. Flight plans in 2D view are represented by solid polylines that interpolate through the intended route.

The **3D mode** (see Figure 11) is useful for observing the entire simulation in a natural 3D environment that allows the user to get a better insight of the spatial relations between airplanes, the actual 3D shapes and possible collisions of flight plans, series of waypoints etc. In 3D, flight plans are displayed as three-dimensional semitransparent corridors of a rectangular profile. As the visio is required to cope with as many as hundreds of airplanes, massive optimizations of visualization of 3D space have been implemented. One of the key features in this respect is the incorporation of multiple levels of detail (LOD) for all 3D models.

Another important feature of real-time visualizer is the ability to interactively control the speed of the simulation. The visualizer displays the current speed of the simulation in the top left-hand corner of the screen, together with the current simulation time.

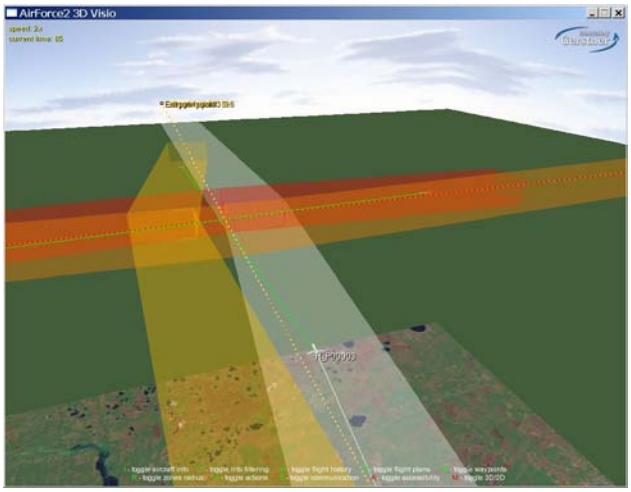

Figure 11: Real-time visualizer: 3D view

The simulation can also be accessed using a **remote web client**, a Java application (see Figures 12 and 13) that connects via network to the simulation system which acts as a server and provides all its clients with regular data updates. This way a number of users can concurrently observe and interact with the same simulation. Accessing the simulation system via network is as simple and straightforward as opening a web browser and entering the IP address of the computer on which the *core system* is currently running.

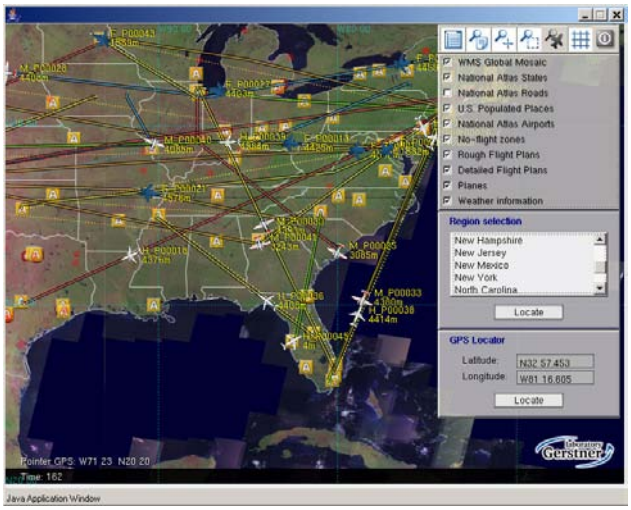

Figure 12: Remote client GUI overview

Before the user can access the simulation, he needs to log in the system by entering a valid combination of username and password. Remote client authentication procedure is secured using one-time hashes for password validation. To minimize network traffic between the remote client and the ATC core system, a combination of HTTP and a special binary protocol is used.

The ATC core system uses a Java Web Start application for loading and starting the client. The client is built on top of JOGL libraries (Java binding for OpenGL [5]) which are

used for accessing graphics 3D acceleration.

The icon palette located in the top right hand corner of the client window gives the user access to application controls, e.g. data layer menu, zooming to a named region, zooming to a selection, zooming to a particular airplane, GPS locator, coordinate grid control and the logout button.

As the user changes the view, the appropriate data segment of the virtual map is requested and downloaded from the server, and therefore at any moment the client needs to keep only a minimal amount of data, which results in its fast and efficient performance.

Figure 13: Flight plans avoiding no-flight zones

8. CONCLUSION

In the paper we present how agents can be used for the negotiation-based deconfliction in the Air-Traffic Control. We describe the prototype of the ATC system where we integrate our agent-based solution with several external data sources. The external online data sources provide necessary information about ground surface, airport locations, weather conditions, no-flight zones and area boundaries for the ATC system. We introduce two types of system visualization: real-time and remote using an internet browser.

The ATC prototype has only simple deconfliction rule-based mechanism for the flight plan changes which assumes that all airplanes use the same deconfliction rules. The system will be extended so that airplanes that detected future collisions would iterate through *monotonic concession protocol* (MCP) to find new flight plans that are collision free. The monotonic concession protocol is a simple protocol developed by Zlotkin and Resenschein for automated agent to agent negotiations [12, 8]. Both airplanes prepare a set of possible flight plan changes scored by the utility function. The utility function includes pilot's own intentions including flight priority, fuel restrictions, time restrictions, etc. From all collision free combinations of flight plan pairs, the possible solution set is created. The iteration protocol results in a commonly accepted solution of the collision. Then each airplane applies the respective flight plan changes. However, the iteration solution can lead to a situation when the solution is not found fast enough. The process has to be extended with an emergency solution that is used when the iteration process does not lead to any fast solution. As the emergency solution, the game theory approach can be used.

Acknowledgement

Effort sponsored by the Air Force Office of Scientific Research, Air Force Material Command, USAF, under grant number FA8655-04-1-3044-P00001 extension of the FA8655-04-1-3044. The U.S. Government is authorized to reproduce and distribute reprints for Government purpose notwithstanding any copyright notation thereon[5].

9. REFERENCES

[1] A-globe. A-globe Agent Platform. `http://agents.felk.cvut.cz/aglobe`, 2005.

[2] CrystalSpace. Crystal Space 3D – opensource game engine. `http://www.crystalspace3d.org`, 2004.

[3] FIPA. Foundation for intelligent physical agents. `http://www.fipa.org`, 2004.

[4] J. C. Hill, F. R. Johnson, J. K. Archibald, R. L. Frost, and W. C. Stirling. A cooperative multi-agent approach to free flight. In *AAMAS*, pages 1083–1090, 2005.

[5] JOGL. Java Bindings for OpenGL. `http://jogl.dev.java.net`, 2005.

[6] S. Kambhampati and L. Davis. Multiresolution Path Planning for Mobile Robots. *IEEE Journal of Robotics and Automation*, RA-2(3):135–145, 1986.

[7] M. Rehák, M. Pěchouček, and J. Tožička. Adversarial behavior in multi-agent systems. In M. Pechoucek, P. Petta, and L. Z. Varga, editors, *Multi-Agent Systems and Applications IV: 4th International Central and Eastern European Conference on Multi-Agent Systems, CEEMAS 2005*, number 3690 in LNCS, LNAI, 2005.

[8] J. S. Rosenschein and G. Zlotkin. *Rules of Encounter*. The MIT Press, Cambridge, Massachusetts, 1994.

[9] R. Schulz, D. Shaner, and Y. Zhao. Free-flight concept. In *Proceedings of the AiAA Guidance, Navigation and Control Conference*, pages 999–903, New Orelans, LA, 1997.

[10] D. Šišlák, M. Rehák, M. Pěchouček, M. Rollo, and D. Pavlíček. **$\mathcal{A}$-globe**: Agent development platform with inaccessibility and mobility support. In R. Unland, M. Klusch, and M. Calisti, editors, *Software Agent-Based Applications, Platforms and Development Kits*, pages 21–46, Berlin, 2005. Birkhauser Verlag.

[11] C. Tomlin, G. Pappas, and S. Sastry. Conflict resolution for air traffic management: A case study in multi-agent hybrid systems. *IEEE Transactions on Automatic Control*, August 1997.

[12] G. Zlotkin and J. S. Rosenschein. Negotiation and task sharing among autonomous agents in cooperative domains. In N. S. Sridharan, editor, *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 912–917, San Mateo, CA, 1989. Morgan Kaufmann.

[5]The views and conclusions contained herein are those of the author and should not be interpreted as representing the official policies or endorsements, either expressed or implied, of the Air Force Office of Scientific Research or the U.S. Government.