7th AIAA Aviation Technology, Integration and Operations Conference (ATIO)
18 - 20 September 2007, Belfast, Northern Ireland

AIAA 2007-7803

Design of a Research Platform for En Route Conflict Detection and Resolution

James R. Murphy* and John E. Robinson†
NASA Ames Research Center, Moffett Field, CA 94035-1000

This paper describes the process of designing a software tool for testing en route air traffic management decision support tool functionalities in a research environment. This effort focused on the development of a scalable, low maintenance software application to replace a legacy en route conflict prediction tool. Previous development efforts provided the lessons learned that guided the development of the new en route conflict prediction tool. This new tool met near term functional requirements while allowing flexibility to support future research objectives both inside and outside of the en route domain. The software process encouraged incremental prototyping and short development intervals. The development team used best practices to increase the reliability and maintainability of the software. The tool was also ported to multiple hardware and operating system platforms to further increase its scalability. When compared to the legacy en route conflict prediction tool, the new application had seven times fewer highly complex functions, while maintaining data integrity to ensure consistent predictions.

I. Introduction

THE National Aeronautics and Space Administration (NASA) is exploring automated concepts for detecting and resolving airborne traffic conflicts. At the core of these concepts is the Center TRACON Automation System (CTAS), which is comprised of legacy software that has had many successes.[1,2,3,4] Unfortunately, this legacy software has become increasingly difficult to maintain and extend. For example, in order to rapidly prototype the initial en route conflict detection functionality, developers reused runway sequencing software written for the terminal area.[4] At that time, software efficiency was a high priority due to limitations in computer hardware performance, while data integrity and software maintenance issues had less importance.[5] As the next generation air traffic management (ATM) research requirements are being defined however, it is clear that the CTAS en route conflict detection and resolution software must handle increased algorithmic complexity without sacrificing software reliability if it is to form the basis for future technologies.[6,7,8]

Fortunately, both hardware performance and software development techniques have improved greatly during the lifecycle of the legacy CTAS en route conflict detection software. In addition, recent focus of en route ATM research has migrated from testing an operational prototype to developing a research platform. These conditions provided an opportunity to redesign the CTAS en route software infrastructure with an increased priority placed on data integrity and software maintenance. This new research platform provides a basis for future ATM software development by applying the lessons learned from the original CTAS en route conflict detection functionality to the revamped ATM project guidelines.

This paper describes the development process used during the redesign of the CTAS en route conflict software. Specific software development practices, the rationale for their use, and ways they can be applied to other ATM software projects are discussed. Analyses of the quality and efficiency of the redesigned software, as compared to the original system, and performance improvements at key design decision points in the software development cycle are presented.

* Software Engineer, Aircraft Operations Software Development Branch, MS 210-8, AIAA Member
† Software Manager, Aircraft Operations Software Development Branch, MS 210-8, AIAA Member

II. Background CTAS Design

Before discussing the redesign of the CTAS en route software, brief descriptions of CTAS and the development of the original conflict prediction module are presented. For a more in-depth discussion of CTAS processes, refer to Ref. 2.

CTAS is a collection of software processes that generate predictions of the future location of aircraft. These predictions are used to produce a wide variety of data used by ATM and ATC personnel including: estimated and scheduled times at the meter fix and runway, vector, speed and altitude advisories to meet those times, and predictions of potential losses of aircraft separation. CTAS runs as a distributed system of processes each designed to perform a specific task. The core components of the CTAS system include the Communications Manager, Route Analyzer and Trajectory Synthesizer. Figure 1 depicts data flow through these core CTAS processes. Aircraft flight plan and state information enter the system via the Communications Manager, which acts as a central data distribution hub. The Communications Manager sends the aircraft information to one or more Route Analyzers, which derive the horizontal and vertical profiles based on these data and request future position estimates from the Trajectory Synthesizers.[5]

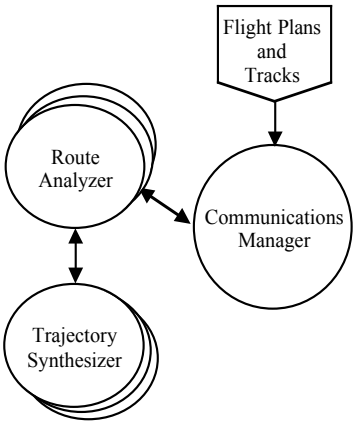

Figure 1. CTAS Core Components

From this core functionality, the CTAS suite of tools has been developed. One of these tools, the Traffic Management Advisor (TMA), uses an enhanced version of the Route Analyzer to produce a range of likely trajectories for aircraft arriving to an airport. These predictions represent the fastest and slowest times an aircraft is expected to reach the boundary between the en route and terminal airspace (typically known as the Meter Fix). These time ranges are sent to a planner, which produces an arrival schedule that helps improve the overall throughput of the terminal airspace while maintaining a safe flow of aircraft.[9] Another CTAS tool, the Final Approach Spacing Tool (FAST), extends the TMA algorithms into the terminal airspace by providing controllers with estimated and scheduled times of aircraft at the runway, as well as advisories that help the controllers meet those scheduled times.[4] FAST's built-in infrastructure, known as "profile selection", provided the ability generate multiple trajectories for aircraft based on many different criteria and formed the basis for the CTAS en route conflict detection baseline.[5]

It is important to note that the CTAS tools share much of the same basic software infrastructure. Figure 2 shows how processes are connected to the core CTAS system to provide additional functionality. Notice, the Dynamic Planner (DP) can be added to generate time-based schedules for TMA, while the Profile Selector (PFS) generates runway assignments and landing sequences for FAST. The advantage of this separation of processes is two-fold:

1) Development of new algorithms and tools will have less impact on existing algorithms and tools.
2) New processes can be run on separate machines making the overall system more scalable.

The En Route profile selector was developed in much the same way. A new process, the Center Profile Selector (PFS-C), was created. As the name implies, it borrowed the basic multi-trajectory generation capability from the PFS, and expanded it into the en route airspace. The software reused as much of the existing baseline as possible to minimize development time for the prototype, which was a necessity at the time.[10] However even as this en route

focused development effort began, design flaws such as the use of global data structures in the baseline software became evident.

The redesign of the terminal profile selector software to allow for controller vector and speed advisories followed a few years later and provided developers an opportunity to take a fresh look at the software infrastructure. The Active Profile Selector (PFS-A) was designed with many of the same requirements as the original PFS, but introduced concepts like data management and integrity to the software design by taking a snapshot of the data for use throughout an advisory determination run.[11]

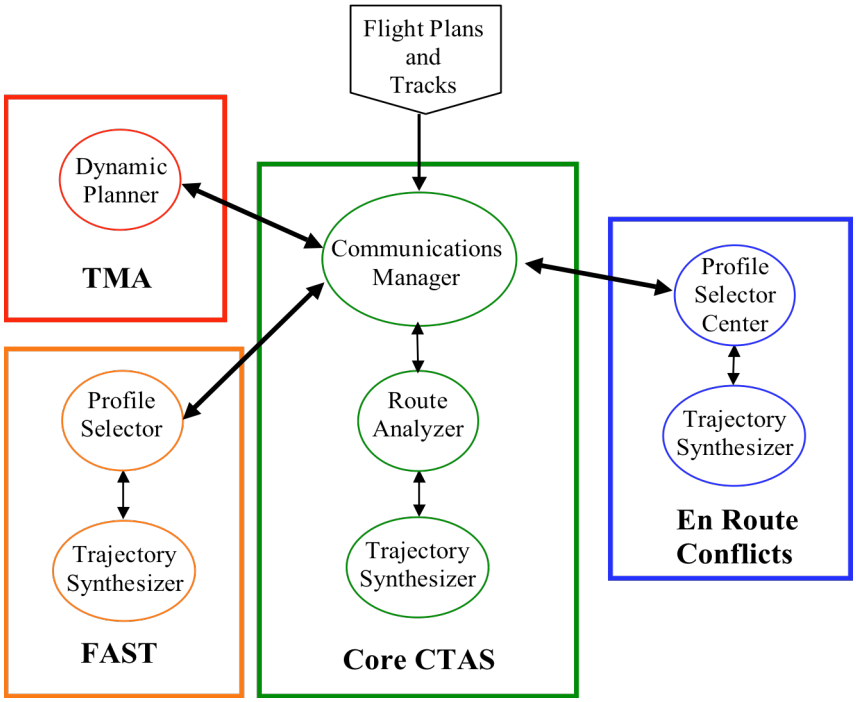

Figure 2. CTAS Functional Components

As the complexity of the en route conflict detection algorithms increased and features such as efficient routing and conflict resolution were added, the PFS-C software became increasingly difficult to maintain. The En Route Profile Selector (PFS-E) was thus developed to ensure a software baseline that could be more easily used to prototype new en route capabilities, as well as to provide a scalable architecture for future research efforts. The development of the PFS-E will be discussed for the remainder of the paper.

III. Profile Selector – En Route

The primary goal of the PFS-E development was to implement the existing and potential en route scheduling, conflict prediction and conflict resolution algorithms on a reliable, maintainable and scalable software foundation. The PFS-E development process used several practices to achieve more results in less time and leveraged the lessons learned regarding design and implementation best practices from earlier PFS, PFS-A and PFS-C development.

A. Basic Functional Requirements

The PFS-E functional requirements include the major capabilities implemented by the PFS-C. These basic capabilities are:

i. *Basic Conflict Detection*: The PFS-E must build "flight plan" trajectories using the aircraft's latest state and intent information. It must also build "dead-reckon" trajectories using only the aircraft's latest state information. These trajectories must be searched for aircraft conflicts (trajectory-to-trajectory) and airspace incursions (trajectory-to-airspace) that may occur within the next 20 minutes.

ii. *Trial Planning Aides*: The PFS-E must build "trial plan" trajectories using potential aircraft intent information changes needed to resolve a projected conflict or simplify the aircraft's routing. These

trajectories must be searched for aircraft conflicts (trajectory-to-trajectory) and airspace incursions (trajectory-to-airspace) that may occur within the next 20 minutes.

iii. *Air Traffic Flow and Efficiency*: The PFS-E must build "meet-time" trajectories using a schedule that alleviates a particular airspace constraint such as the terminal acceptance rate. It must also build "direct-to" trajectories that reduce aircraft flying times without causing additional conflicts downstream.

iv. *Conformance Monitoring*: The PFS-E must compare each aircraft's latest state and intent information to identify conditions such as flight plan deviations, safety critical maneuvers and holding patterns.

The PFS-E functional requirements also include expanded capabilities, which are:

v. *Automatic Conflict Resolution*: The PFS-E must determine the aircraft intent changes needed to resolve a projected conflict without causing additional near-term conflicts.

vi. *Severe Weather Avoidance*: The PFS-E must search aircraft "flight plan" trajectories for intersections with severe weather that may occur within the next 20 minutes. It must also determine the aircraft intent changes needed to safely and efficiently avoid regions of severe weather.

Together, the basic and expanded capabilities represent the software functionality necessary for the research and development of NASA's en route ATM decision support and automation tools.

B. Software Development Process

The PFS-E development began with the aggressive goal to have a limited capability prototype within its first year. These limited capabilities included all basic conflict prediction and some air traffic flow and efficiency capabilities, like direct-to planning. The intention was to formulate the PFS-E design with these well-defined basic capabilities while, at the same time, augmenting the design to support the expanded capabilities needed for the future. It was also desired to have progress measured and course-corrected by allowing immediate testing of the PFS-E. As a result, an Agile approach to software development was used to address these needs. The four statements of the Agile Manifesto[12] summarize this approach:

i. We value individuals and interaction over processes and tools
ii. We value working software over comprehensive documentation
iii. We value responding to change over following a plan
iv. We value customer collaboration over contract negotiation

These four simple statements describe the key aspects of the PFS-E software development process. First, software development experience with PFS, PFS-A and PFS-C showed that exhaustive development procedures do not compensate for less-skilled individuals participating in the software development effort. Thus, the development process must increase the domain knowledge and improve the software development skills of all team members through constant interaction. Traditional methods often treat software development as a mere result of requirements definition without considering the need for design iterations. Meanwhile, Agile methods believe that domain-knowledgeable developers are indispensable, because they must communicate the inevitable trade-offs to the customer. Therefore, frequent formal and informal technical meetings of the researcher and developer staffs were held to discuss new requirements, raise issues and demonstrate the latest PFS-E capabilities.

Second, working software was the principal measure of progress. A common attribute of all Agile methods of software development is a reliance on working code as the low-level documentation that details the specific functionality of the software.[13] External documents should only provide high-level descriptions of the software design and system architecture, since these elements are relatively stable. Therefore, several coding standards that improve the readability of the PFS-E code as low-level documentation have been adopted.

Third, it is more important to rapidly and effectively respond to change, rather than to follow a comprehensive plan (e.g., plan weeks rather than months in advance). Plans need to have regular feedback and adaptation mechanisms, so they can be used as guides rather than control mechanisms. While this approach is critical when the software product is a research platform that is continually evolving, it is not only limited to research platforms. A Harvard Business School case study of the development of the customizable personal webpage My Yahoo! found that half of the code changed in the last four weeks of the project, culminating in almost three entire rebuilds of the product over the development cycle.[14] Traditional methods of software development often attempt to develop a start-to-finish plan for the product from initial design to final release. Meanwhile, Agile methods of software development attempt to minimize this risk by developing software in short release intervals typically lasting two to

four weeks.[15] CTAS experience showed that the longer the planned development period, the greater the risk that the end product would not meet the needs of the customer as initially defined. Therefore, the PFS-E development focused on the implementation of new capabilities and features in manageable two-week intervals.

Lastly, continuous customer interaction with the software is critical to the overall success of the project. Traditional methods of software development often start with a lengthy period of design followed by initial implementation that does not easily allow for early customer interaction with the software. This creates considerable risk, as the customer is not able to refine requirements or evaluate early progress. Conversely, the short release interval of Agile methods of software development is well suited for early customer interaction with the software. Initially, the PFS-E had a small fraction of the PFS-C functionality. However, after an early release, customers used the PFS-E with its limited capabilities and were able to provide constructive feedback about its problems and limitations. The short release intervals of the PFS-E allowed the results of this early collaboration to immediately guide subsequent development.

C. Software architecture

The PFS-E design has been evolutionary and based upon lessons learned from the development efforts of earlier CTAS profile selectors. Figure 3 shows these evolutionary pathways. The PFS-E leveraged its architecture from the PFS-A and its functional requirements from the PFS-C. In order to increase the initial speed of development, a large amount of the PFS-A software was reused, largely without change. Some examples of this low-level software included the socket connection to the Communications Manager and the shared memory connection to the Trajectory Synthesizer. The reuse of this existing infrastructure allowed the PFS-E to go from research concept to generating flight plan trajectories in just four months. However, the reuse also imposed a few constraints on the design. First, the PFS-A was written in C, so the PFS-E would necessarily need to have a C foundation in order to leverage this software. Second, the PFS-A still shared many of its libraries with the PFS and PFS-C. These libraries needed to be refactored[‡] and even redesigned to attain the levels of multi-thread safety and data integrity required by the PFS-E.

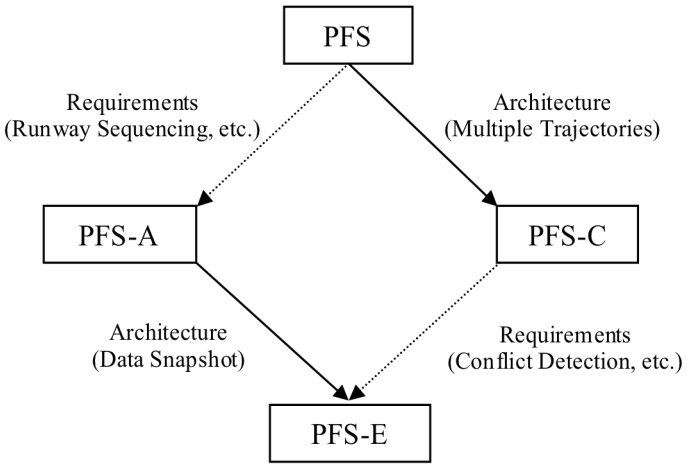

Figure 3. Evolution of Profile Selector Design

A core functional requirement of the PFS-E was the need to generate many different types of trajectories for each aircraft in order to predict conflicts, determine preferred routing, etc. Considering the lessons learned from the previous profile selector designs, this *functional* requirement imposed several *architectural* requirements on the PFS-E. It needed to be data-centric, it needed to ensure data integrity, and it needed to support scalable execution. In this section, these elements of the PFS-E design are discussed in detail.

Data-Centric Architecture

A significant motivation for the development of the PFS-E was the ever-increasing cost of adding new capabilities to the existing profile selector. To improve its maintainability, the PFS-E uses a software design that is

[‡] The term refactor refers to making changes to the software that improve its readability, change its internal structure and design, and remove dead code, to make it more maintainable.

similar to the data-centric model described by Ref. 16. Figure 4 shows the notional organization of the PFS-E, which is comprised of two main components: the persistent data storage (Manager) and the transitory algorithms (Workers). The manager maintains the persistent data, including airspace data, conflict prediction parameters and aircraft intent and state information. The workers contain the algorithms that operate on the persistent data, including conflict detection, conflict resolution and trajectory generation. The separation of the data storage from the algorithms and the allocation of different algorithms to separate workers allow division of the otherwise complex software into manageable elements.

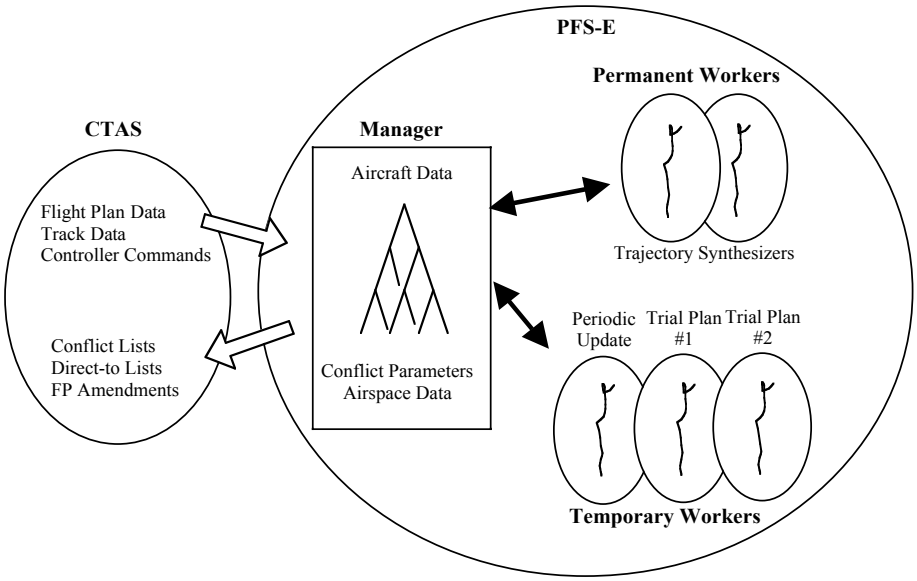

Figure 4. Schematic of PFS-E Data Flow

Additionally, each worker operates on a well-defined set of inputs to provide an equally well-defined set of outputs, such as an aircraft trajectory or a list of aircraft conflicts. There are two types of workers: those that persist indefinitely, such as trajectory synthesizers, and those that may be temporary, such as a trial plan generator or conflict predictor. Regardless of type, each worker communicates only with the manager using a two-way message queue. This design ensures that each worker has a single input and output access point of communication.

Data Integrity

The lack of data integrity was another significant motivation for the development of the PFS-E. Software development experience with the PFS and PFS-C showed that data integrity is difficult to maintain when asynchronous updates to persistent data and inline exception handling are allowed. The PFS-E leverages its separation of the data storage and the algorithms and borrows the concept of a data snapshot from the PFS-A to avoid these problems.

Each worker uses a complete and independent snapshot of the persistent data for the duration of its execution. The manager is permitted to update the persistent data arbitrarily at any time. The worker does not need to implement exception handling to account for these changes, because it has a private copy from an earlier instance in time. At the conclusion of a worker's execution, the results of the execution are communicated back to the manager, and the manager is responsible for maintaining data integrity by rejecting any result that may be irrelevant, incorrect or undesired.

Multi-Threaded Execution

The desire for improved scalability was another motivation for the development of the PFS-E. Therefore, a fundamental requirement of the PFS-E design was the use of separate threads for the manager and each worker. A thread is a part of a program that can execute independently of its other parts. The combined increase of algorithmic complexity and air traffic levels rendered a single-threaded profile selector design obsolete.

There are two main advantages of a multi-threaded PFS-E design. First, it provides a software architecture to rapidly implement new algorithms without the overhead associated with creating a new CTAS client. Furthermore, it

simplifies the code needed to mix algorithms with different update intervals and execution times. Some of the algorithms that have currently been implemented as individual workers include:

- Aircraft-to-aircraft conflict prediction
- Severe weather incursion prediction
- Special use airspace incursion prediction
- Manual trial planning
- Aircraft-to-aircraft conflict resolution
- Arrival metering trajectory selection

Second, the scalability of the PFS-E improves, because it can take advantage of hardware with multiple processors and multiple cores per processor. A multi-threaded PFS-E allows multiple instances of certain resource-constrained elements to be executed simultaneously. For example, multiple trajectory synthesizer threads reduce the service times to the algorithms using trajectories. Similarly, multiple trial planning and conflict resolution threads reduce the response times to the customer using their results.

D. Software Implementation Practices

Several implementation practices were applied to improve the flexibility, maintainability, and reliability of the PFS-E software. These practices were intended to ensure that the PFS-E is a scalable ATM research computing platform capable of supporting both existing functional requirements on today's hardware platforms and future functional requirements on different hardware platforms. These implementation decisions are discussed below.

Platform Independence

An important aspect of the PFS-E implementation is reliable cross-platform support. Initially, NASA's ATM research software was developed on Sun Microsystems hardware. The principal motives for this decision were the amount of computing power needed to run a CTAS system and the reliability needed for safe and effective operational field-testing. In the early 1990's, when this work was initiated, the only hardware capable of this level of performance was a high-end server-class product, such as from Sun Microsystems. However, the hardware options have now changed. In recent years, lower cost PC-class products that meet or exceed the necessary computing power have become available. Also, work focus has switched from operational field tests to the development and demonstration of long-term research concepts in the laboratory.

To eliminate platform dependencies, the PFS-E has been made fully POSIX compliant.[17] In addition, the PFS-E is being enhanced to support heterogeneous networks of different hardware architectures. This capability will allow the PFS-E to run on most Unix-based platforms, including little-endian and big-endian byte orders, and 32-bit and 64-bit architectures. Table 1 shows a list of the hardware platforms and operating systems, which have been tested during PFS-E development.

Table 1. Supported Platform Configurations

Vendor	Hardware	Operating System(s)
Apple	IBM PowerPC Intel Xeon Intel Core and Core2	MacOS X 10.2, 10.3, 10.4
ASL	AMD Opteron	SUSE 10.1
Sun	Sun UltraSPARC AMD Opteron	Solaris 8, 9, 10 RedHat 4.4

There are several important benefits of cross-platform support even if the final deployed system will only use one particular platform. First, the set of possible hardware solutions available to achieve the necessary computing power is maximized. Second, some software defects that are hard to find on one platform may be easily found on other platforms. For example, we have seen that the Solaris operating system is more tolerant of memory allocation bugs than the MacOS X and Linux operating systems. Third, each platform has its own unique set of software development tools, each with their own strengths. For example, MacOS X has an extensive set of CPU and memory

profiling tools built into the operating system, Solaris has a large selection of third-party commercial software, and Linux has an unrivaled amount of open-source free software.

Language Flexibility

Initially, the PFS-E borrowed heavily from the PFS-A software, and, as a result, a large amount of the PFS-E basic infrastructure is written in C. It was recognized from the outset that restricting the software to only C would significantly limit its ability to rapidly implement future requirements. As a result, the development of the PFS-E has since implemented portions of its algorithms in C++, Java and Pascal.

The primary motivation for language flexibility was to increase code reuse, which translated into faster development and less maintenance. The PFS-E components associated with worker thread management, worker algorithm execution and data archiving are written in C++. These components benefit from the Object Oriented features of C++ like encapsulation, polymorphism and inheritance. For example, the features shared by all workers are implemented using inheritance to maximize code reuse across the workers. Similarly, the PFS-E components associated with trajectory plotting are written in Java. These components customize Ptolemy Plot,[18] a free Java plotting package, and benefit from the superiority of Java's Swing GUI toolkit. Finally, the PFS-E components associated with conflict resolution[19] are written in Java, and those components associated with traffic collision avoidance[20] are written in Pascal. These components are provided by external sources and have well-defined interfaces. As a result, these components can be treated as black boxes by the PFS-E, and their code is best used in their original native implementations.

Limited Optimization

Another fundamental principle of the development of the PFS-E has been that simplicity is more important than elegance, especially in the early stages of development. Code optimization, when done early and inappropriately, makes the code harder to understand and thus more difficult to maintain. Code that is hard to understand hinders all aspects of the software development process, making it more difficult for developers to add new features, for testers to find bugs in existing features, and for team leads to review code modifications. Consequently, the PFS-E development attempted to make the initial implementation of every new feature as simple and flexible as possible, instead of making it as fast as possible. The value of specific code optimizations can only be understood after a particular feature has been sufficiently implemented for use by the customer.

One example of this approach was the development of the airspace incursion algorithm, which determines incursions into Special Use Airspace. When first developed, the airspace incursion algorithm reused significant portions of the existing conflict prediction algorithm. This decision allowed the airspace incursion algorithm to be implemented quickly. However, this decision also meant that the airspace incursion algorithm calculated its own trajectories, which duplicated the existing conflict prediction trajectories. Later in the development of the PFS-E, it was decided that the airspace incursion algorithm, as well as several others, would be significantly more responsive if they retrieved flight plan trajectories stored by the conflict prediction algorithm rather than recalculating their own copies of the flight plan trajectories. However, maintaining the integrity of stored trajectories is more complex than recalculating new trajectories, so this code optimization was only done when the PFS-E responsiveness was not acceptable to the customer.

Self-Documenting Code

Another practice intended to improve the maintainability of the PFS-E software was to follow the concept of self-documenting code.[21] Instead of relying upon internal comments or external documents to describe the software, the code itself is the low-level documentation. Often, this concept can degrade into no low-level documentation at all. However, in the case of the PFS-E, coding standards ensure methods and variables names are clear, concise and representative of their purpose. This process reduces code maintenance, since the method and variable names (and hence the low-level documentation) are automatically updated when the algorithm is changed.

For example, CTAS defines many different units of time in its algorithms, and each unit can have a different numeric precision–integer or floating point. Some examples include milliseconds, seconds since midnight, seconds since the last Unix epoch (January 1, 1970) and minutes before the next predicted conflict. Since variable names may not clearly define the associated unit of time, custom variable types were introduced to unambiguously describe some common units of time. For seconds since the last Unix epoch, we use custom types, such as "CTAS time type" for integer values and "CTAS precision time type" for floating-point values. Thus, this simple practice completely eliminates the need to explicitly document the units of time.

Integrated Testing

Another practice intended to improve the long-term reliability of the PFS-E software was the use of fully integrated verification tests, also known as sanity checks. The use of integrated testing combines elements of the Agile practice of automated testing with elements of Test Drive Development (TDD).[22] TDD is a software development technique that mandates test cases be written before implementing the code. The PFS-E incorporates a series of verification tests that it labels as "Unexpected Conditions." Instead of relying upon separate functional and unit tests to find bugs, the code is instrumented to automatically perform its own low-level tests. These checks are intended to verify the software is doing exactly what it is expected to do – nothing more and nothing less.

The PFS-E systematically checks for two types of problematic conditions: assumption violations and incomplete logic. An assumption violation will occur when a method's required preconditions are not met (i.e., it's doing something more than expected). An incomplete logic violation will occur when a method's required post-conditions are not met (i.e., it's doing something less than expected).

Traditionally, it was common practice to handle most of these situations by simply returning from the method with, at most, a warning to a log file that was frequently ignored. To prevent the serious warnings from being lost, the PFS-E is configured to immediately halt execution and provide diagnostic information about the unexpected condition. This implementation strategy only succeeds when the developers are able to rapidly provide fixes or temporary solutions for unexpected conditions as they occur. This practice allows developers to concentrate on basic algorithm and feature development without having to test for every possible combination of inputs. Also, it allows developers to quickly recognize when a new feature is violating an explicit, yet undocumented, assumption of another component of the PFS-E software.

IV. Benchmark Analyses

As an integral part of the PFS-E development cycle, performance checks and regression tests are used to ensure that the software remains both scalable and maintainable. In particular, the software is analyzed to determine if recent functional changes have reduced its maintainability or performance. The software is tested on existing and emerging hardware and operating system configurations to provide continued platform independence. And, the developers and customers continually evaluate the qualitative performance of the system. The results of these tests are used to justify improvements to the software. Examples of the PFS-E software tests and their results are provided below.

A. PFS-E Static Software Metrics

During the development of the PFS-E, the software was evaluated using standard software metrics to help determine if its design and implementation are sound. Source lines of code (SLOC) is a static software metric that measures the size of a program. For our analyses, we used a physical SLOC (pSLOC) that measured the number of lines with anything other than comments and white spaces (tabs and spaces).[23] Studies have shown a positive correlation between a module's size and its defect rate.[24] Complexity is another widely used static software metric that may be considered a broad measure of the soundness and confidence of a software module (or method). For our analysis, we used the cyclomatic complexity that measures the number of independent paths through the module.[25] This measure provides a single ordinal number that can be compared across modules and averaged across entire applications. Studies too have shown a positive correlation between a module's cyclomatic complexity and its defect rate.[24] Furthermore, cyclomatic complexity is also a strong indicator of a module's testability. A low cyclomatic complexity improves a module's readability and indicates that it can be modified with less risk of introducing a new defect than a more complex function. The typical use of cyclomatic complexity is to compare it against a set of threshold values. Table 2 shows a commonly accepted set of these threshold values and their meaning.[25]

Table 2. Cyclomatic Complexity Thresholds

Complexity	Type of Module	Risk Level
1-4	Simple	Low
5-10	Marginal	Low
11-20	Complex	Moderate
21-50	Complex	High
>50	Untestable	Very High

Historically, each profile selector design has implemented a set of specialized methods in conjunction with a set of common library methods. The cyclomatic complexity of each design's specialized methods was used as an indication of the design's soundness. Table 3 provides summary statistics about the specialized methods of each profile selector design. Figure 5 shows a histogram of the cyclomatic complexity divided into bins of 5 for each profile selector design. Figure 6 shows a comparison of each profile selector design in terms of the cyclomatic complexity risk levels.

Table 3. Summary of Specialized Methods

Module	Methods	pSLOC	SLOC per method
PFS	586	25,680	44
PFS-C	1138	50,492	44
PFS-A	810	27,547	34
PFS-E	2060	52,176	25

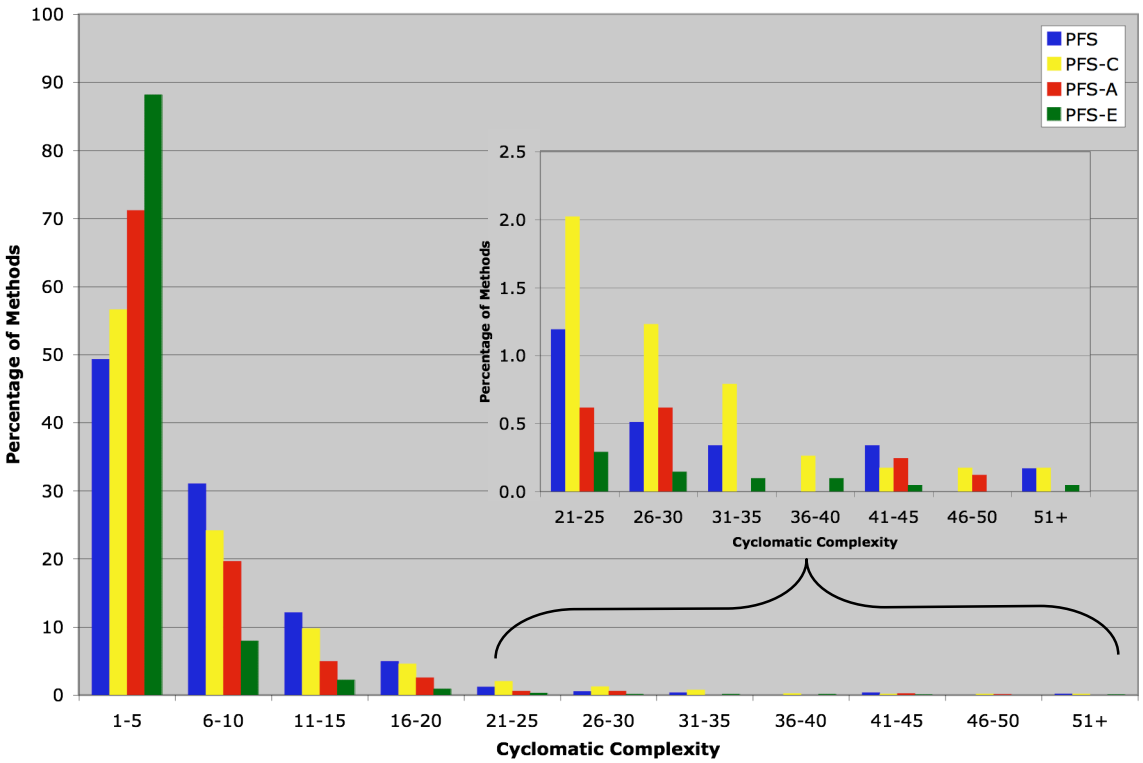

Figure 5. Cyclomatic Complexity of Profile Selector Designs

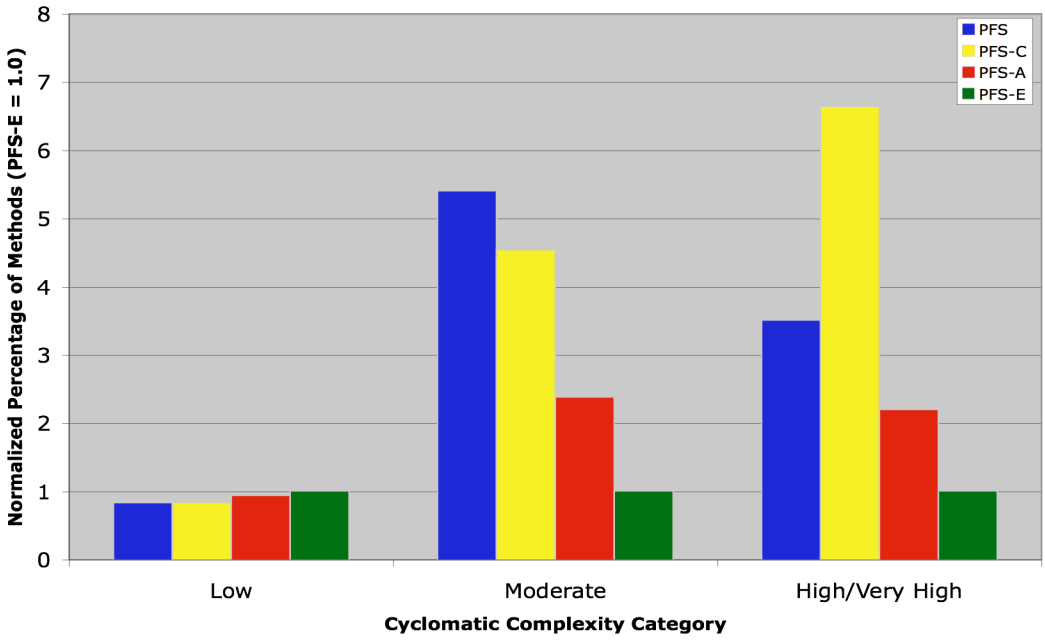

Figure 6. Aggregate Cyclomatic Complexity of Profile Selector Designs

An important caveat is that some types of methods, while simple to understand and maintain, will yield a high cyclomatic complexity. For example, a simple message parsing method employing a switch block and case statements will have a cyclomatic complexity that scales with the number of message types. These inaccurate complexity measurements are results of the McCabe algorithm used to calculate cyclomatic complexity. Therefore, it is important to compare the designs on the basis of the aggregate results and not on the basis of individual complexity values.

The results show two promising attributes of the PFS-E design. First, the PFS-E has significantly shorter methods than the other profile selector designs. The average PFS-E method is two-thirds the size of the average PFS-A method and about one-half the size of the average PFS and PFS-C methods. Second, the PFS-E has significantly fewer moderate and complex methods. The PFS-E has 2.4 times fewer moderate methods than the PFS-A, and 4.6 times fewer moderate methods than the PFS-C. And, it has 2.2 times fewer complex methods than PFS-A and 6.6 times fewer complex methods than PFS-C. In fact, the PFS-E size and complexity metrics are reduced further if its integrated verification tests were removed. The PFS-E by far has the most sophisticated built-in debugging and testing support of any of the designs, and these features only act to inflate the metrics. When these optional features of the code are removed from the PFS-E, its marginal and complex method counts are reduced by an additional 6% and 15%, respectively.

Overall, these results are not surprising. The PFS-A was designed based upon lessons learned from the PFS design, and the PFS-E was designed based upon lessons learned from the PFS-A design. As a result, there is a steady migration to shorter and less complex methods. Furthermore, these objective results are consistent with the general consensus that the PFS-C was hardest to maintain, modify, and test.

B. PFS-E Dynamic Hardware Profiles

During the development of the PFS-E, the software was repeatedly profiled using a nominal traffic sample to help determine if its performance remained acceptable as features were added. Figure 7 shows the current relative performance of some platforms used during PFS-E development. Each bar in the chart corresponds to a specific type of machine, and each color represents the time it takes to execute a particular stage of the PFS-E basic conflict prediction algorithm. First, system and algorithm initialization identify the relevant aircraft and build their flight plan trajectories, respectively. Then, conflict prediction performs the conflict search, and advisory generation broadcasts the list of predicted conflicts. Finally, algorithm and system cleansing move the results to persistent storage. The approximate production date for each platform is shown at the top of the bar. Note that stages denoted with asterisks in Figure 7 can take an insignificant amount of time for some platforms and may not be shown.

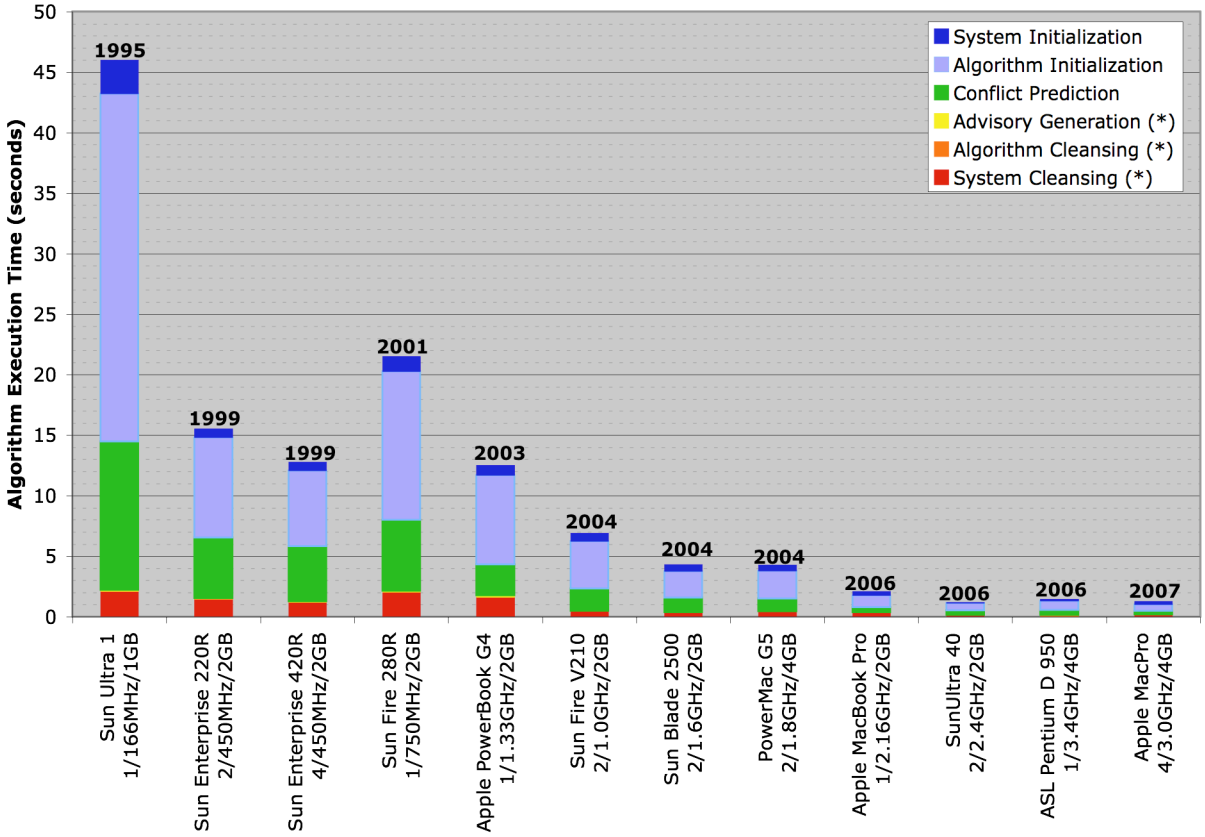

Figure 7. Comparison of PFS-E Performance on Various Platforms

Several key findings can be taken from these results. First, the empirical observation is that PFS-E performance has been doubling every two years for the past eight years. Not surprisingly, this is similar to Moore's Law.[26] Second, platforms based upon the Intel Xeon (e.g., Apple Mac Pro) and AMD Opteron (e.g., Sun Ultra 40) chipsets are now outperforming those platforms based upon the UltraSPARC and PowerPC G4/G5 chipsets. Thus, the PFS-E, and more broadly CTAS, would have been at a considerable performance disadvantage if it did not achieve platform independence. Furthermore, laptop computers, like the Apple PowerBook and MacBook Pro achieve performance high enough to be used as development and demonstration platforms.

In order to justify the "simple first, optimal later" design approach, it is important to compare the PFS-E performance improvements due to hardware advancements to the PFS-E performance improvements due to code optimizations. Figure 8 shows the relative performance increase associated with various code optimizations for the PFS-E basic conflict prediction algorithm and the airspace intrusion algorithm that determines the intersections of trajectories and Special Use Airspace. The code optimizations ranged from compilation changes to algorithm changes to runtime changes. They included using compiler-based optimization, streamlining use of low-level memory management, storing less analysis data, using analytical equations for weather instead of measurements, and using multiple trajectory synthesizer processes. The PFS-E basic conflict prediction and airspace intrusion algorithm are analyzed separately since the effectiveness of certain optimizations will depend on the type of algorithm.

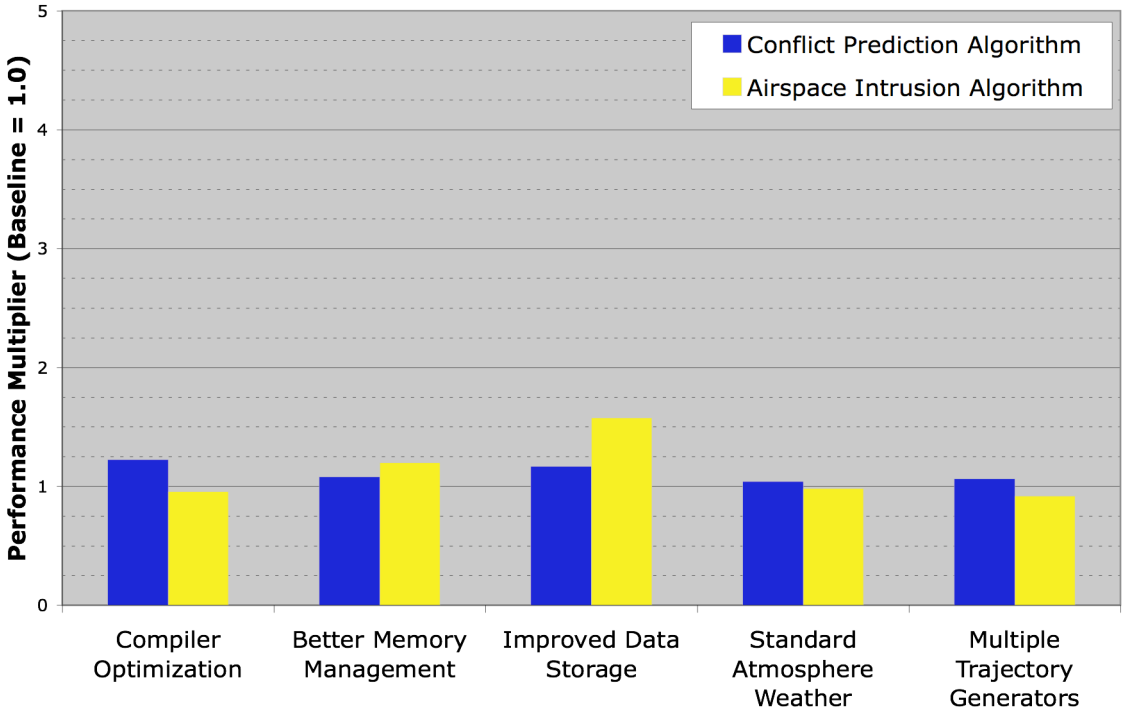

Figure 8. Effects of Code Optimizations

With the exception of storing less analysis data, none of the code optimizations achieved more than a 20% increase in PFS-E performance. Small apparent performance decreases are actually the result of measurement variability. These measurement errors (5%-10%) can be seen most easily with the addition of multiple trajectory generators. For the PFS-E's basic conflict prediction algorithm, compiler optimization achieved a 20% decrease in execution time due to the algorithm's numerical intensiveness. For the PFS-E trial-planning algorithm, storage of less analysis data achieved a 60% decrease in execution time due to the algorithm's use of stored data. These performance improvements are minimal when compared to the performance improvements due to hardware advancements, and they underscore the "simple first, optimal later" approach. Only in cases where a performance improvement is needed in a timely manner (and waiting for new hardware is not an option) are software changes justified.

C. Implementation Benchmarks

Although not the primary concern for the PFS-E design, the computational efficiency of the system was routinely measured. Qualitative measurements were provided by the customers via informal feedback on the speed of conflict resolution and the responsiveness of the trial planner. The quantitative performance of the software was analyzed with simple regression tests and by measuring the execution times for particular tasks. These regression tests were executed periodically during the development cycle. Occasionally, the results of these tests motivated changes on the grounds of computational efficiency when the expected benefit was large.

Figure 9 shows bar graphs depicting the relative speed of the PFS-E airspace intrusion algorithm. The first bar represents the original nominal execution time of the algorithm. In this case, the intrusion algorithm calculated all of the necessary trajectories itself. The execution time decreased by a factor of approximately 2.3 when the algorithm was redesigned to retrieve the raw trajectories from storage (second bar). Additional changes to allow the algorithm to retrieve the processed trajectory products and then to reuse the processed trajectory products (without retrieval) decreased the execution time by factors of approximately 2.9 and 4.3, respectively.

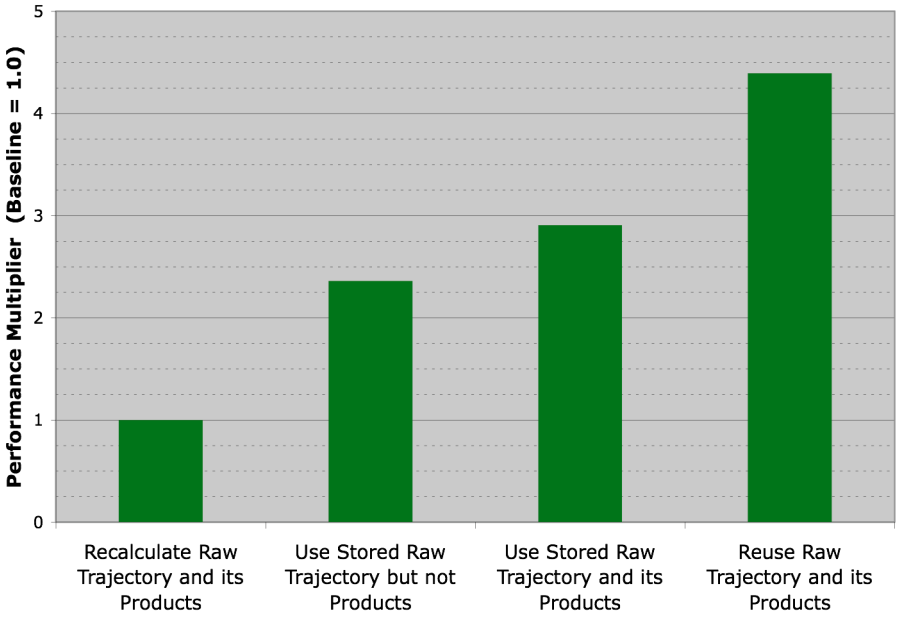

Figure 9. Effects of Design Improvements

The previous example illustrates the importance of design improvements, and it further reinforces the "simple first, optimal later" approach, because it shows that the performance benefits of code optimization (Figure 8) simply do not compare to the performance benefits of a better design (Figure 9). While the redesign of the trajectory storage added some complexity to the software, all algorithms that analyze flight plan trajectories reap the benefits of that increased complexity. These changes are now part of the PFS-E infrastructure.

V. Conclusions

ATM research requirements made it necessary to redesign the legacy CTAS en route profile selection software. The primary goal of the PFS-E development was to implement the existing and potential en route scheduling, conflict prediction and conflict resolution algorithms on a reliable, maintainable and scalable software foundation. The functional requirements of the PFS-E included conflict prediction, trial planning, traffic flow and efficiency analysis and conformance monitoring, as well as, conflict resolution and severe weather avoidance.

An Agile approach to software development was followed to achieve more results in less time. The PFS-E software development process stressed constant team interaction, short two-to-four week development intervals, incremental demonstration of working software and early customer involvement to define requirements. The PFS-E design and its associated implementation were evolutionary and based upon lessons learned from the development efforts of earlier CTAS profile selection software. The key elements of the PFS-E design included a data-centric architecture to divide the complex software into manageable elements, an explicit consideration of data integrity by using complete and independent snapshots of persistent data and multi-threaded execution to provide scalability for future capabilities. Several implementation best practices were followed to improve the flexibility, maintainability and reliability of the software. These principles included platform independence, limited optimization, self-documenting code and integrated testing.

The PFS-E was found to have fewer marginal and high-risk methods than the PFS, PFS-A and PFS-C. Also, the PFS-E was found to have significantly smaller methods than those other profile selectors. Studies show positive correlations between cyclomatic complexity and defect rate and between method size and defect rate. Therefore, it can be expected that the PFS-E will have fewer defects and be more maintainable.

The PFS-E performance was analyzed on a wide range of hardware platforms, ranging from a 10-year-old Sun Ultra 1 to a brand new Apple Mac Pro. The results showed that PFS-E performance has been nearly doubling every two years, and platform independence has allowed the PFS-E to take advantage of today's fastest hardware. Furthermore, the PFS-E "simple first, optimal later" design approach was justified, because the performance improvements due to code optimizations were shown to be much smaller than the performance improvements due to hardware advancements. However, the airspace intrusion algorithm also illustrated the benefits of limited design

changes. In particular, the responsiveness of the airspace intrusion algorithm was greatly improved by allowing the algorithm to reuse stored trajectories instead of recalculating duplicate trajectories.

Acknowledgments

The authors would like to thank Dave McNally and Chester Gong for providing the customer feedback that is so critical to the Agile approach. We would also like to thank Greg Wong, Liang Chen, Joe Walton, Don Shawver and the rest of the PFS-E development team. Lastly, but most importantly, we would like to thank Jinn-Hwei Cheng and Thien Vu for their tireless pursuit of bugs.

References

[1] Erzberger, H., and Nedell, W., "Design of Automated System for Management of Arrival Traffic," NASA TM-102201, Ames Research Center Moffett Field, CA, Jun. 1989.

[2] Erzberger, H., Davis, T. J., and Green, S., "Design of Center-TRACON Automation System," *AGARD Meeting on Machine Intelligence in Air Traffic Management*, Berlin, Germany, 11-14 May 1993.

[3] Green, S. M., and Vivona, R. A., "En Route Descent Advisor Concept for Arrival Metering," *AIAA Guidance, Navigation, and Control*, Montreal, Canada, Aug. 2001, AIAA 2001-4114.

[4] Davis, T.J., Krzeczowski, K. J., and Bergh, C., "The Final Approach Spacing Tool," *IFAC Thirteenth Symposium on Automatic Control in Aerospace*, Palo Alto, CA, Sep. 1994.

[5] Erzberger, H., Paielli, R. A., Isaacson, D. R., and Eshow, M. M., "Conflict Detection and Resolution In the Presence of Prediction Error," *1st USA/Europe Air Traffic Management R&D Seminar*, Paris, France, Jun. 1997.

[6] Erzberger, H., "Transforming the NAS: The Next Generation Air Traffic Control System," *24th International Congress of the Aeronautical Sciences*, Yokohama, Japan, 29 Aug. - 3 Sep. 2004.

[7] "Next Generation Air Transportation System: Integrated Plan," Department of Transportation, Joint Planning and Development Office, Dec. 2004.

[8] "Next Generation Air Transportation System (NGATS) Air Traffic Management (ATM)-Airspace Project: Reference Material," National Aeronautics and Space Administration Transportation, Jun. 2006.

[9] Swenson, H. N., Hoang, T., Engelland, S., Vincent, D., Sanders, T., Sanford, B., and Heere, K., "Design and Operational Evaluation of the Traffic Management Advisor at the Fort Worth Air Route Traffic Control Center," *1st USA/Europe Air Traffic Management R&D Seminar*, Saclay, France, Jun. 1997.

[10] Slattery, R. M., and Green, S. M., "Conflict-Free Trajectory Planning fir Air Traffic Control Automation," NASA TM-108790, Ames Research Center, Moffett Field, CA, Jan. 1994.

[11] Robinson III, J. E., and Isaacson, D. R., "A Concurrent Sequencing, and Deconfliction Algorithm for Terminal Area Air Traffic Control," *AIAA Guidance, Navigation, and Control Conference*, Denver, CO, Aug. 2000.

[12] Highsmith, J. (ed.), Agile Software Development Ecosystems, The Agile Software Development Series, Addison-Wesley, Boston, 2002, p. xvii. (URL: http://agilemanifesto.org [cited 29 Jan. 2007])

[13] Abrahamsson, P., Warsta, J., Siponen, M.T., and Ronkainen, J., "New Directions on Agile Methods: A Comparative Analysis," *25th International Conference on Software Engineering*, Portland, OR, May 2003.

[14] Iansiti, M., *Technology Integration: Making Critical Choices in a Dynamic World*, Harvard Business Press, Boston, 1998.

[15] Schwaber, K., "Scrum Development Process," *10th Annual Conference on Object-Oriented Programming System, Languages, and Applications*, Austin, TX, 15-19 Oct. 1995.

[16] Murphy, J. R., Reisman, R., and Savoye, R., "A Data-Centric Air Traffic Management Decision Support Tool Model," *6th AIAA Aviation Technology, Integration and Operations Conference*, Wichita, KS, 25-27 Sep. 2006.

[17] The Open Group, "IEEE Std 1003.1, 2004 Edition," The Open Group Technical Standard Base Specifications, Issue 6.

[18] Ptolemy Plotter, Software Package, Version 6.0.2, University of California Berkley, Berkley, CA, 2007.

[19] Meyn, L., Windhorst, R., Roth, K., Van Drei, D., Kubat, G., Manikonda, V., Roney, S., Hunter, G., Huang, A., and Couluris, G., "Build 4 of the Airspace Concept Evaluation System," *AIAA Modeling and Simlation Technologies Conference and Exhibit*, Keystone, CO, Aug. 2006, AIAA 2006-6110

[20] Fast Time Encounter Generator (FTEG), Software Package, Version 7.1, Federal Aviation Administration, Atlantic City, NJ, 2007.

[21] McConnell, S., *Code Complete*, 2nd ed., Microsoft Press, Washington, 2004.

[22] Beck, K., *Test Driven Development: By Example*, Addison-Wesley, New York, 2002.

[23] SLOCCount, Software Package, Version 2.26, David A. Wheeler, URL: http://dwheeler.com/sloccount. [cited 25 Jul. 2007]

[24] Khoshgoftaar, T.M., and Munson, J.C., "Predicting Software Development Errors Using Software Complexity Metrics," *IEEE Journal on Selected Areas in Communications*, Vol. 8, Issue 2, Feb. 1990 pp. 253-261.

[25] McCabe, T. J., and Watson, A. H. "Software Complexity," *Crosstalk: The Journal of Defense Software Engineering,* Vol. 7, No. 12, Dec. 1994, pp. 5-9.

[26] Moore, G. E., "Cramming more components onto integrated circuits," *Electronics*, Vol. 38, No. 8, 19 Apr. 1965.