FLIGHT TRANSPORTATION LABORATORY REPORT R86-9

ATCLAB:

A LABORATORY ENVIRONMENT FOR RESEARCH IN ADVANCED ATC AUTOMATION

CONCEPTUAL DESIGN

Antonio L. Elias
John D. Pararas

Flight Transportation Laboratory
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

June 1986

This work was carried out under DOT Contract DTRS5785C00083
for the Transporation Systems Center, DOT, Cambridge, Ma.

Contents

1 INTRODUCTION

A large number of ideas and schemes have been proposed and are constantly being suggested to enhance the Air Traffic Control system's safety, reliability, and efficiency by means of automation. The capability of the Federal Aviation Administration to properly specify and procure advanced automation systems depends critically on its capability to evaluate these ideas from a number of viewpoints:

1. Functional: i.e., is the proposed idea of any value, assuming it could be implemented?

2. Procedural: i.e., can the proposed idea be implemented in conjunction with existing and/or new ATC procedures?

3. Implementability: i.e., can the proposed functionality and/or procedures be implemented, with sufficient accuracy, reliability, data requirements, etc?

4. Cost/benefit: i.e., are the benefits expected from the proposed functionality sufficient to offset the expected costs and risks?

5. Requirements definition: i.e., is the proposed scheme or system sufficiently well defined to allow the development of meaningful and supportable requirements?

There are two conventional approaches available to answer these questions: analysis and simulation. The effectiveness of analysis to evaluate an automation proposal usually depends on the degree to which the proposed function interacts with other elements of the ATC system. In general, the more isolated and self-contained the function, the more amenable it is to analytical evaluation. Functions that interact with many different elements of the system generally require dynamic simulation for effective evaluation. Traditionally, this has required the development of an ad-hoc simulator to evaluate the proposed automation scheme, or the adaptation of an existing simulation. Both approaches are expensive and risk intensive; attempts at building all-inclusive, general purpose simulations are even more expensive and not entirely risk free.

As an alternative, a mid-ground solution would be the establishment of a flexible computer-based laboratory environment to perform combined analysis and simulation evaluation on an ad-hoc basis in response to the specific automation scheme being evaluated. To be more effective than traditional analysis and simulation techniques alone, this environment must reduce the cost of building prototype code by two orders of magnitude, both in terms of labor and of calendar time, over traditional environments, such as the ones used to develop existing simulators and prototype systems.

Recent developments in computer hardware and software have drastically altered the process of developing software, particularly in the systems simulation area. Symbolic computation and object-oriented languages, along with hardware specialized to execute this type of code, have been shown to produce the two orders of magnitude improvement suggested in the previous paragraph.

This report analyzes the feasibility of establishing such a laboratory environment, including identification of the required technology, a possible architecture that would fulfill these requirements, a tentative implementation plan, and two sample pathfinder projects to show how the proposed environment could be used to evaluate two specific advanced automation proposals. To facilitate references to this environment, it will be referred to in this report as **ATCLAB**; this is *not* an official FAA-approved name.

2 BASIC ASSUMPTIONS

2.1 Building a Simulation Presents Problems

It is impossible to build a single simulator that would satisfactorily fulfill all the expected ATC automation research and evaluation requirements, for the following reasons:

1. It is difficult to predict functional modeling requirements for future, unspecified automation systems. Experience at M.I.T. in testing tactical automation aids has shown that even for a well-specified system, only about 80% of the simulation functional requirements can be predicted; that is, some 20% of the simulation's functions were identified during the progress of the experiments. The fraction of unexpected simulation requirements may be much higher in the case of incompletely specified or developmental systems.

2. Similarly, some 20% of the expected simulation functionality proves to be unnecessary in actual testing, even for well specified systems. The combination of both effects is that the efficiency of an "all bases covered" general-purpose simulation, in terms of needed and actually used functionality cannot be expected to be greater than 60%.

3. There is a large variance in the costs of implementing different functional capabilities in a simulation; building a general-purpose simulation necessarily implies including some expensive functions which may not be cost effective in actual practice, thus increasing the cost/risk product of the project.

Similarly, experience has shown that it is impractical to build a special purpose simulation for every expected ATC research project. Even for major projects, the elapsed

time and labor cost of developing a suitable, custom simulator usually ends up being the major fraction of the project cost and the major contributor to project risk and schedule slippage.

On the other hand, the LISPSIM work at M.I.T.'s Flight Transportation Laboratory has shown that it is possible to develop a set of hardware/software building blocks from which a project-specific simulation can be built in a reasonable time and at a reasonable cost, where "reasonable" means a length of time and a labor cost comparable to that needed to properly specify the simulation requirements. This requires the use of fourth generation software techniques (symbolic manipulation, automatic memory management, and object-oriented programming) and appropriate hardware. These techniques also increase the general level of programming productivity required to support the analysis capability of an ATCLAB.

2.2 The Technical Environment is Changing Radically

The technical environment expected in the next ten years will affect the design of the ATCLAB from three directions: new trends in software/hardware, changes in the hardware/software cost tradeoff, and growth of computational capacity.

2.2.1 New Trends in Software/Hardware

The late 60's and early 70's saw the recognition, on the part of computer scientists, of the importance of *control structures* in the efficient production of reliable code – which gave rise to the "structured programming" approaches and the third generation of programming languages and systems. These developments produced an improvement in the productivity – hence, the cost – of programming estimated between a factor of 2 and a factor of 10. These languages (e.g., PL/1, Pascal, C) were termed "third generation" by comparison with the "first generation" of assembly languages and the "second generation" of the early high-order languages. While the basic architecture of machines until that time adhered faithfully to the Von Neumann model of the mid-50's, the impact of software on the design of processors was beginning to be felt when the "stack architecture" was created in response to the development of the first recursive, "functional" languages such as Algol.

The major development of the late 70's and early 80's was the creation of the "actor" model of computation, where software "objects" with individual identity, state, and behavior, were conceived. The effect of this development was to raise the level of abstraction at which humans interfaced with the programming language one step further than third generation languages, since these software objects could be made to resemble the real-world objects that constitute the human programmer's problem domain. It is interesting to note that simulation of large problems was one of the

original motivations of this development. The military in particular has for some time now been pursuing this approach for their simulation needs (see for example [KLA 82] and [NUG 83]). As a result a number of "object-oriented" simulation languages have been developed ([KLA 82], [BIR 73], [GOL 83]) and established simulation languages like SIMSCRIPT are following the same route ([ELI 84]).

The combination of this "object-oriented" technology with the "functional programming" and the "symbolic manipulation" models of computation (the latter two developed by mathematicians) is loosely referred to as "fourth generation" languages. Most post-1980 developments in programming technology – including ADA[1] – feature some or all of these elements.

The practical realization of these newer models of computation was delayed somewhat by the mismatch between these models and the existing computational hardware architecture[2] – hence the traditional dictum that "Lisp is a very inefficient language" which really should be stated, "Von Neumann machines are very inefficient when executing Lisp." The development of virtual memory systems and microprogrammed architectures made practical the construction of machines capable of executing fourth generation code as efficiently as a Von Neumann machine would execute third generation code.

While the development of this technology was motivated by, and in most cases carried out by, researchers in the field of Artificial Intelligence, it is a mistake to assume that fourth generation software techniques are inseparable from Artificial Intelligence.[3] Other myths, such as the inefficiency of these techniques, the difficulty in learning them, and their inability to be used in real-time work are fast disappearing, while their economic advantage over third generation techniques is becoming more apparent. It is becoming clear that the improvements of this fourth generation over the third generation are much larger than those that the third generation offered over nonstructured high-order languages (second generation), or even what these offered over assembly programming.

---

[1] Some experts may argue that ADA is not a true fourth generation language system in spite of its fourth generation style features. The authors do not wish to express an opinion in this regard.

[2] After all, the Von Neumann model was developed taking into consideration the "limitations" of hardware as they could be foreseen in the mid-50's.

[3] As an example, programming in support of research at M.I.T.'s Flight Transportation Laboratory only four years ago was performed 80% in second generation systems (mostly FORTRAN) and 20% in third generation (mostly PL/1), with no experience in fourth generation systems. Today, the ratio is 80% fourth generation (mostly Lisp) and 20% third generation (mostly C), with only about 20% of the work being related at all to Artificial Intelligence.

2.2.2 Hardware/Software Cost Tradeoffs

Expected continued improvements in hardware price/performance, coupled with the nonlinearity of software cost vs. program size indicate that hardware performance can and should be traded off for increased programming productivity. This will require short-term performance sacrifices and lead to long-term gains as the HW/SW cost ratio continues to decline.

Of particular significance is the so-called Halstead's law, or $\frac{1}{3}$-power hypothesis, which postulates that the cost of an engineering effort (e.g., programming, or systems analysis) increases with the number of production units N devoted to performing this task as $N^{\frac{1}{3}}$.[4] The consequence of this effect in the design of the ATCLAB environment is that it is vital to keep the number of individuals required to program simulations and analysis to a minimum, even at the expense of hardware costs *and traditional, government approved programming techniques.* Of particular concern are government-mandated *approved languages* and government-mandated *mandatory documentation.* Experience has shown that the benefits from the use of a standardized language, i.e., portability and universal recognition, do not overcome the burden of using them for the following reasons:

1. Except for trivial or mathematically oriented programs, large systems development requires access to operating system features or machine characteristics that are both outside the scope of the language standard and different from system to system. Not only does portability disappear in these cases, but attempts at implementing these features with the tools afforded by the standard language are obscure and difficult to understand, even the author.

2. It is an observed fact that most program documentation produced under government standards is far less useful than comparable commercial documentation. This is due to the compliance with a standard template for these documents, a template that assumes the existence of a standard programming problem, standard programming techniques, etc.

In practice, there are significant differences among software engineering problems, and the programming techniques that may be used to solve them. Free-form documentation, where the specifics of *what* to document and *how* to document it are left to the initiative (and, in many cases, the art) of the author, will, therefore, always prove superior to any standardized documentation scheme.

---

[4] Popularly, this phenomenon has been stated as "two programmers can do in nine months what any of them could do in twelve;" the actual ratio, for $N = 2$ is more like *seven* to twelve months.

2.2.3 Computational Requirements

Past attempts at predicting the computational requirements of ATC systems simulations have consistently fallen short of actual requirements. At the same time, rapidly evolving harware technology precludes solving this uncertainty problem by over-procurement of computational power, which will soon become obsolete. This mandates the use of a distributed computational system with incremental growth capability so that short-term computational throughput problems can be met by increasing the number of computational units, while long-term growth is assured by replacement of individual units with higher technology equivalents as they become available.

There are three ways of implementing distributed processing:

1. As an ad-hoc partitioning of a specific problem into two or more parts. This is what usually happens when an existing simulation outgrows its mainframe, and dual mainframes are installed to cope with the problem. Typically in this situation, the simulation is divided into a master/slave configuration, with communications between both halves taken care of by very special means, e.g., shared memory. Characteristic of this approach is the inability to incorporate additional processors in an incremental fashion. This approach usually requires minor operating system support.

2. The development of a program structure that lends itself to partitioning among a wide range of well-defined lines, with established communication and synchronization protocols capable of operating with one, two, or more processors after only minor modifications. This usually requires significant language and operating system support, usually in the form of program-controlled task generation and memory allocation.

3. The use of a language/operating system environment that offers programmer-transparent distributed processing, without specific communication or synchronization actions being required of the programmer.

While there are a significant number of existing simulations that implement the first form of distributed processing, we do not believe that such an ad-hoc approach is the best way to proceed for the ATCLAB simulations. At the same time, even though there are important efforts underway to provide transparent distribution of processing, this capability is not currently available and, in all likelihood, it will be at least five years before such systems are commercially available. Such approach would therefore represent an unacceptable risk for the proposed laboratory. This leaves the second approach – planned partitioning of simulation elements – as the recommended approach for ATCLAB.

Even this level of distribution is not a solved problem; to date, nobody has demonstrated concurrent distributed simulation processing in a fourth generation environment. It is our opinion, however, that the benefits resulting from incremental distributed processing, especially the capability to increase incrementally the computation power, far outweigh the technical risks.

2.3 Labor Considerations

The staffing of a computer-intensive operation has traditionally required three distinct types of labor:

1. Engineer/researchers, whose main task is to define projects, carry out experiments, and write functional specifications for software and tests to validate their operation.

2. Programmer/analysts, whose main task is to convert the specifications into code and carry out the validation tests.

3. Operators/technicians, whose task is to physically operate the machine, carry out repetitive chores such as backups, interface with field maintenance, schedule the machine's usage, etc.

This three-tier staffing is being made obsolete by fourth generation hardware/software for the following reasons:

1. The increase in level of abstraction of fourth generation languages is reducing the burden of programming to the extent that the division of labor between problem analysis/formulation and coding is no longer appropriate, especially in view of the communications overhead between the engineer/researcher and the analyst/programmer; this change is similar to the elimination of keypunch operators, which came about in the late 60's/early 70's with the introduction of interactive terminals.

2. Similarly, the reduction in size, power consumption, and physical complexity of hardware has eliminated the need for dedicated operations people, with networking and file serving reducing the burden of maintenance and backup operations.

Consequently, it is apparent that a facility such as the ATCLAB should have only one kind of employee: a trained engineer/computer sciences specialist with background and knowledge in both ATC and fourth generation computer science. While training in the latter is scarce,[5] it is nevertheless the authors' opinion that the FAA Technical Center

[5] As witness the current high demand for these persons by both industry and the Government.

can indeed provide the necessary personnel with perhaps some retraining required in the area of Advanced Computer Sciences.

It is particularly important to realize that the ATCLAB concept *does not require large numbers of programmers* [6] but rather fewer, properly qualified individuals. As a result, staff planning for it should be carried out accordingly.

3 LABORATORY CONFIGURATION

In view of the premises and environmental factors expounded in sections 1 and 2, we recommend that the ATCLAB be developed and viewed more as a *high-performance computer-based research environment* than as simply just another simulator. This does not contradict the fact that the primary purpose of the facility is to provide fast, low cost simulation capability for the purposes outlined in section 1, but rather implies that this capability be achieved by the creation of a suitable high-power environment – including the kit of simulation building blocks outlined in the next sections.

3.1 Hardware Configuration

The proposed hardware configuration consists of a number of identical high-powered workstation units interconnected by a high-speed Local Area Network (LAN).

3.1.1 Display and Execution Units

The basic hardware unit of the Lab will be a Development and Execution Unit (DEU). The initial DEU is a high-end workstation with computational throughput in excess of 1 MIPS, memory capacity equal to or greater than 4 Megabytes, and a bit-mapped display with resolution equal to or better than 720 by 348 pixels.[7] As its name implies, this unit is both a single-programmer development station and the basic unit of computational resources during simulation execution. There is to be at least one DEU per support person assigned to the Lab, with extra units being highly desirable both for redundancy and for support or outside experimenters and/or visitors. Unless specifically required by an experiment, the DEU's display will also be the principal runtime user interface, requiring a DEU per simulation actor. Additionally, each DEU will be equipped with local I/O capability to support special-purpose devices, displays, or Systems Under Test (SUT's).

The DEU attempts to be the least common denominator of the processing requirements of any system element to be simulated or tested at the ATCLAB, as well as provide the basic horsepower behind the simulation. This approach appears sub-optimal, since

[6] At least compared with a traditional programming effort of similar scope.

[7] With *some* color units being desirable, but not essential.

it is very likely that specialized hardware could be better adapted to each of the tasks: for example, a large mainframe could be used to generate targets and perform data logging during the simulation, with smaller minicomputers used to simulate controller stations, special devices, etc., and special display units used to interface with the simulation actors.

In contrast, the single DEU type has, by necessity, to be an overkill in most situations: the DEU that creates and manages targets will have underutilized display capabilities; the DEU used to simulate an advanced controller display will have excessive computational power, etc.

But this mismatch, and the apparent underutilization of equipment resources is more than amply matched by the economies of using a single language, operating system, display functions, etc. Use of different devices would by necessity require the staffers either to be proficient in a number of different devices simultaneously, or to specialize in one or two devices. In addition, the cost of developing interfaces between identical devices is much lower than the cost of interfacing dissimilar devices. Again, this is a clear case of trading off equipment costs for labor costs and elapsed time.

3.1.2 Interprocessor Communications

DEU's are connected by a high-speed Local Area Network (LAN). During simulation development and post-run data reduction, the LAN operating parameters shall be optimized for maximum throughput in the vicinity of 1 MBaud. During simulation execution, the LAN parameters shall be optimized for a minimum message transmission latency between processes executing in different DEU's of the order of 50 msec or less.

The operating system running in the DEU's shall provide support for three types of network operations:

1. Remote file access and transfer, including backup operations
2. Remote user access of processing nodes ("telnetting" in the network nomenclature)
3. Direct communication between a process running in one DEU and another process running in another DEU (interprocess communications).

This last capability is seen as the natural mode in which the distributed computation capability suggested in section 3.2 should be implemented.

3.1.3 System Support Units

Additionally, System Support Units (SSU's) may be attached to the LAN to provide

file server, back-up storage, remote communications, and printing services. A typical SSU may be a medium-sized minicomputer (e.g., VAX-780) with disks, a tape unit, a laser printer, and serial interface lines. Alternatively, the SSU may be a dedicated and/or specially modified DEU.[8] Special-purpose devices, such as stroke-written display systems, are usually best interfaced from a dedicated processor, instead of the DEU's. Interface with external systems (e.g., the SSF, TATF or even a standalone SUT) should also be performed from an SSU.

3.2 Software Configuration

The Laboratory software can be categorized in two different ways: by source, and by usage. By source, the software will fall into one of three categories: System Tools and Services (STS's), Project-Independent Elements (PIE's), and Project-Dependent Elements (PDE's). By usage, both PIE's and PDE's fall into one of four categories: System Under Test (SUT), Internal Simulation Models (ISIM), External Simulation Models (ESIM) and Simulation Core Elements (SIMCORE).

3.2.1 System Tools and Services

This category includes operating systems, language compilers, editors, and debuggers, communications software, graphic support packages, etc. It is expected that most of this software will be off-the-shelf, with perhaps minor customizations on the operating system and LAN software required to support the ATCLAB. These programs are very hardware dependent, and their availability and performance is expected to impact the selection of the DEU hardware to the extent that the DEU/STS procurement should be considered a single, inseparable one. Mandatory capabilities are: message-passing and functional programming support, source-level debugging and incremental compilation capability, multi-tasking with direct operating system services access from a High Order Language (HOL), and language-sensitive full-screen code editing. A single-language environment (operating system written mostly in the same HOL as applications) is desirable.

3.2.2 Project-Independent Elements

Project-Independent Elements (PIE) are Lab-developed modules from which project-specific simulations can be made. It is expected that the development of PIE's will be evolutionary, with the initial set of PIE's being determined by the earliest projects to be supported by the lab, and continuing development of PIE's during the lifetime of the laboratory. PIE's include both simulation skeleton, or core elements, such as the

[8] This would have the additional advantage of not requiring the staff to master an additional Operating System environment.

master scheduler, data gathering and recording tools, simulation assembly and configuration control elements, etc., and the basic repertoire of models for both internal and external simulation elements (see sections 3.2.5 and 3.2.6 below). The fundamental soundness of the building-block approach to assembling simulations has been demonstrated at M.I.T.'s Flight Transportation Laboratory on Lisp Machines, albeit on a single-processor basis.

3.2.3 Project-Dependent Elements

Project-dependent elements (PDE): these are Lab-developed software elements for execution in the Lab's DEU's for the purpose of running a specific project's simulation. It is expected that program development efforts will be balanced between PIE's and PDE's. As individual projects develop PDE's, they become part of the facility library, thus turning perhaps into PIE's for other projects. For example, a particular project may require a detailed model of the mode S data link; this model may be developed exclusively for that project, but is subsequently made available to other projects which, while not critically dependent on such a model, nevertheless benefit from its availability.

3.2.4 System Under Test

The System Under Test (SUT) includes models of the specific ATC element, component, program, or device which is the primary focus of the experiment, e.g., an Arrival Metering Sequencing function. These software elements are characterized by a high level of functional detail.

3.2.5 Internal Simulation

The Internal Simulation (ISIM) is made up of models of other components of the present or proposed ATC system that, while not the primary focus of the simulation, are necessary to support the simulation of the SUT to the required degree of realism. For example, the Arrival Metering Sequencing Function may require a model of the mode S data link system and ancillary ground processing functions.

3.2.6 External Simulation

The External Simulation (ESIM) comprises models of elements not part of the ATC system assumed in the test, such as aircraft, airborne systems, weather, and wind. This is equivalent to the target generation role of traditional simulations, but will of necessity be much more complex in the ATCLAB to avoid the need for large numbers of actors in simulation runs.

3.2.7 Simulation Core

The Simulation Core elements (SIMCORE) comprises all the software required to prepare a project specific simulation (software cataloguer/simulation compiler), prepare simulation input data (e.g., airways and facilities data base managers, live data), run and control the simulations in real time or in fast time (including Monte-Carlo run managers), and analyze, present, distribute, and archive simulation results.

3.2.8 Intersoftware Communications

The reason for the above categorization is to bound the Intersoftware communication requirements, which affect both the software development cost (due to the programming overhead involved) and hardware performance (both partitioning among DEU's and LAN loading). Typically, the level of detail or aggregation is uniform in each of the three categories, with the SUT models being the most detailed ones, and the ESIM models the least detailed. Most of the intermodel communication will be between elements of each category, with lower bandwidths required between categories. This will tend to favor partitioning of the computational burden along these lines, with one DEU supporting the ESIM, another supporting the ISIM, and one or more supporting the System Under Test.

The boundaries between SUT, ISIM, and ESIM are quite fuzzy: for example, a model that is part of the SUT for one particular run may be considered an ISIM model during another run. Similarly, the SUT may actually be implemented in non-ATCLAB hardware, with a DEU dedicated to the interfacing between the Lab and the SUT hardware rather than to running the SUT software itself.

While the individual software elements may be written in any language (as long as the development environment requirements listed in 3.2.1 are satisfied), interprocess communications between the ISIM and the ESIM, and between processes within these two levels, shall use the *active object* model of computation, also called *message passing*. The objective is to facilitate the partitioning of the ISIM and ESIM among multiple DEU's. It is desirable that the SUT elements also be coded in this style, although alternative communication protocols between the SUT and ISIM/ESIM elements may be required. An example of this situation would be a SUT implemented in non-Laboratory hardware intended to be ported to an actual field location. In this case, the DEU interfacing the simulation to the SUT must mimic the communication protocol of the field system that the SUT will eventually interface with.

3.3 Simulation Configuration

3.3.1 Simulation Timing and Control

The ATCLAB simulations shall be capable of running in one of two modes: real-time, and fast-time. Fast-time simulations are semi-asynchronous simulations where a master processing unit (a DEU in this case) keeps track of a pseudo real-time (presumably faster than real time) which then paces the execution of the rest of the simulation elements in other processing units (the other DEU's). Fast-time pacing represents a compromise between operating system and programming complexity on one hand, and efficient hardware utilization on the other, since the load on the DEU's is likely to be unbalanced, and the effective speed of the simulation is determined by the slowest, or most highly loaded DEU. This compromise is consistent with the hardware/software tradeoff principle stated in 2.2.2.

The DEU responsible for maintaining system time (real or accelerated) is also responsible for allocating and loading the SUT/ISIM/ESIM elements to the appropriate DEU's, and controlling the starting, freezing, terminating, and data collecting of the entire simulation. This Master DEU (MDEU) is also the simulation manager's display and control unit. Under this model, simulation timing shall occur as follows:

1. Each DEU executing ISIM and ESIM models shall have its own executive, with its own scheduler queue. Processes and events executing within a DEU shall be scheduled by this program.

2. Simulation time shall have a finite granularity, determined by the requirements of the experiment being supported.

3. In real-time mode, the MDEU shall issue periodic clock ticks which the executive process in each DEU shall receive and use to advance its own timer queue. If a DEU's schedule falls behind the MDEU clock ticks, a warning message is sent to the MDEU where the simulation manager may elect to take corrective action, e.g., terminate the simulation.

4. In fast-time mode, the MDEU shall propose to all the DEU's the advancement of time to the next time tick; upon receipt of a "ready to advance" message from all the DEU's (indicating that their timer queues have caught up with the proposed new simtime), the MDEU shall broadcast a "simtime is now..." message to all the DEU's, which would then advance their timer queues to the new simtime. All simulator transactions are therefore assumed to occur within that clock tick.

This architecture has the additional advantage that the entire ESIM model group can be exercised (e.g., for development and testing purposes) in a single, separated DEU (or

sets of DEU's, if necessary) by having its scheduler queue advance autonomously (i.e., without receipt of timing messages from the MDEU). It is also conceivable that, given sufficient number of DEU's, and sufficient LAN throughput, more than one simulation can run independently of each other.

3.3.2 Display Requirements

It is expected that the ESIM and ISIM will contain a larger percentage of models of human behavior than existing simulations, thus eliminating the requirement for a large number of actors, or at least reducing the requirements in cases where actors are absolutely necessary, such as when voice communications are involved. As much as possible, ISIM/ESIM models should be manageable from a single display each, thus supporting the partitioning of the ESIM/ISIM computational load to one DEU each. In the case where the ISIM or ESIM load is higher than one DEU's resources, this shall not require an additional operator(s) on the additional DEU(s). Conversely, if the ISIM or ESIM require additional actors (therefore additional displays) but the computational load is still under one DEU, it should be possible to use additional DEU's as simple terminals, with minimal additional software required in them.

The SUT presents a different problem; here, special-purpose displays and display formats may be required. While ESIM/ISIM operator displays may be designed for the convenience of the program developers, the operators, or both, SUT displays must satisfy the requirements of the experiment under way. As far as possible, these requirements shall be met with the DEU displays; when this is not feasible, special project-dependent hardware may be required. Three cases are possible:

1. The SUT may include specialized hardware, including displays. In this case, the ATCLAB simulator's responsibility is simply to communicate with the SUT, and the communication DEU's display can be used, for example, to monitor the SUT.

2. The SUT is modeled by ATCLAB DEU's, but specialized displays or I/O units are required. The special displays may be interfaced with the simulation in one of three ways: directly with the DEU's, using the DEU's local I/O capability; through the SSU's I/O subsystems; or directly to the LAN, if the display device interfaces with the same hardware type and software protocol as the ATCLAB's LAN.

3. An external simulator or system is used; of particular interest is the use of the NAS Simulation Support Facility (NSSF). The NSSF has been examined as a possible source of radar targets for ATCLAB. Its limited capability and expense of operation in the target generation mode indicates such use is not desirable. On the other hand, its 8 ARTS III displays and possible 32 NAS displays could

be very effectively used as SUT displays or, perhaps more interestingly, as ISIM displays (e.g., for the Arrival Metering and Spacing project).

3.4 ESIM Architecture

The main determinant of the computational requirements of a simulation run and the complexity of the ESIM/ISIM software is the level of detail of the simulation, in particular of the data that the ESIM/ISIM has to provide the SUT. A review of the likely SUT's from 44 Research, Engineering, and Development projects seem to indicate that a minimum of three levels of ESIM detail are required. These levels will be identified by the level of detail of the movements of the aircraft modeled in the ESIM: an Area Network level, an Air Route network level, and a Free-Flight level.

3.4.1 Area Network Level

At this level, referred to as the *Area Network Level* or *Level I*, the airspace is modelled as a network of *Areas* which behave as aircraft containers. The size of an area may vary from that of a present sector, to that of several ARTCC's. Each area may contain traffic *sources* which generate aircraft that flow through the system, as well as traffic *sinks* which absorb aircraft that have reached their destination. Even though sources and sinks will typically represent airports, they can also entry/exit points from/to airspace that is of no interest to the current experiment.

Aircraft are generated at traffic sources and are endowed with a *flight plan* consisting of a list of areas that the aircraft has to traverse. The last area in that list is the aircraft destination and must be a sink.[9] Flight plans are indirectly related to actual aircraft flight plans (in that they determine the route of each aircraft at the level of detail allowed by the model). Direct routing can also be modelled by suitably adjusting the nominal travel times within each area in a direct flight plan. Flight plans can be dynamically modified thus allowing rerouting of traffic after the beginning of the flight.

The primary simulation events at this level of detail are area crossings. For each aircraft, there will be information regarding the area in which it currently resides, the time at which the next area crossing is scheduled to occur, etc. Time variation in source and sink rates as well as in area capacities will be modelled, thus allowing simulation of varying weather as well as other operating conditions. The relationship between area loading and area transition times will be modeled. Aircraft movements and other events occurring within an area are not modeled.

This level of detail is expected to support SUT's related to National Flow Control

[9]Sources and sinks are special cases of areas in that they are nodes in the area network with the proviso that aircraft never emerge from a sink and never enter a source.

operations; it should be possible to model several thousand aircraft in $10x$ fast time on a single DEU at a simulation time granularity of 15 seconds.

3.4.2 Air Route Network Level

At this level, referred to as *Level II*, aircraft are assumed to flow through a set of finite (i.e., fixed) airway and direct routes, although the flight plans themselves are dynamically modifiable. The progress of individual aircraft along these fixed routes, using simplified performance models is supported, as are random (asynchronous) queries about the state of the aircraft. The time-varying location and severity of areas of specific weather conditions – including winds – shall be modeled, possibly by statistical means, as well as their effect on aircraft progress.

This level of detail is expected to support simulations involving ACF-level TMU related SUT's; it should be possible to model several hundred aircraft in real time on a single DEU at a simulation time granularity of 4 seconds.

3.4.3 Free Flight Level

At this level, referred to as *Level III*, detailed aircraft dynamics and avionics/flight control responses are modeled; flight plans are modeled at the same level of detail as in a real aircraft flight management computer. High-level models of manual navigation performance are included to simulate non-FMC equipped aircraft, reducing the need for pseudo-pilots to voice response modeling. Provisions will be made to support fully automated aircraft models when the performance of voice recognition/voice synthesis devices allow it. Weather modeling shall be similar to that of Level II.

This level of detail is expected to support tactical and mixed strategic/tactical level SUT's, such as Arrival Metering Sequencing functions; it should be possible to model one hundred aircraft with a single DEU in real time at a simulation time granularity level of 4 seconds.

3.4.4 Model Level Intermixing and Compatibility

The communications between the SUT/ISIM, and the ESIM shall be through clearly established messages using the active actor or message-passing model of computation. Specific sets of messages shall be defined at each level. Aircraft objects in each of the three levels of detail shall be created in one of three ways:

1. Stochastically, by means of aircraft generator objects driven by a set of statistical parameters appropriate to each level.

2. Deterministically, in response to a predefined set of script-like events (e.g., real

traffic data or output from another simulator), with the necessary logic to supply any missing required information (e.g., detailed flight plan data).

3. Deterministically, in response to the termination of an aircraft object at a different level; this allows the simultaneous existence of simulated regions of different levels of detail; when an aircraft object reaches the boundary of its level of detail's region of simulation, it is destroyed, and an aircraft object is created at the other side of the boundary, at a new level of detail. Automatic logic will provide any necessary detailed information required when the new aircraft object is of a higher level of detail than the destroyed object, or will abstract the data in the reverse case.

The coexistence of simulation regions of different levels of detail, which can be thus provided at a modest incremental development cost, will be unique to this facility and can provide the opportunity for research in the vertical organization of the ATC command and control structure. The cost-effectiveness of this feature cannot be ascertained precisely due to lack of previous experience with such hybrid systems. Therefore, while this capability is not seen as an essential component of the laboratory's basic cofiguration and cost considerations may preempt its immediate implementation, we recommend that it be kept open as a possibility in the fundamental architecture of the ATCLAB.

3.5 Interface with Existing Facilities

The effectiveness of the ATCLAB to support evaluation and development of advanced ATC automation concepts can be greatly enhanced by interfacing the Lab with other facilities at the Technical Center. In particular, the NAS System Support Facility (SSF) and the ARTS Terminal Area Test Facility (TATF) can provide significant support of short term projects that involve interfacing with the present NAS or ARTS systems. Indeed, one of the pathfinder projects illustrated in this report assumes the capability of interfacing with these two systems.

Both the SSF and the TAFT are display systems with local intelligence. They communicate with target generator programs through low bandwidth (2400 to 9600 Baud) serial interfaces; close examination of the Common Digitizer interface documentation ([CD 77]) and the GFP to ACP interface documentation ([NSSF 85]) reveals that it would be a *trivial* task to use the ATCLAB facilities to generate pseudo-targets for these two systems; in particular, the required programming would require at most a few labor-weeks. In turn, the SSF/TAFT facilities could provide the test bench for early automation systems, such as the Traffic Management device postulated in section 4.3.

4 FACILITY DEVELOPMENT

4.1 Development Philosophy

As mentioned in subsection 3.2.1, all system tools and service software should be procured in conjunction with, and simultaneously with, the DEU and LAN hardware. All ESIM/ISIM software should be developed at the ATCLAB, as well as applicable SUT software (SUT software and perhaps hardware may be generated outside the ATCLAB). A complete ESIM for each level of detail should be fully developed and tested before the ATCLAB can be considered operational. ISIM elements for the first few projects should be given the next immediate priority, with ATCLAB-developed SUT software close behind. Periods of low facility utilization, if any, should be devoted to completing the ISIM library.

It should be remembered, however, that the "Building Block" concept of the facility implies that the facility should be in a continuous state of growth and evolution; in other words, the facility is *never 100% complete.* This has a number of consequences:

1. The Facility Acceptance Tests must be suitably designed; since the degree of completeness of the facility can only be judged against a particular project, it is imperative that a pathfinder project be established to determine the degree of readiness of the facility.

2. The construction phase and operational use phase of the development of the facility will be ill-defined; as such, great care should be used when applying traditional management measures and techniques that provide, for example, for the segregation and differentiation of construction and operations personnel. Indeed, part of the routine operations of such a facility is recurrent buildup.

3. Traditional measures of progress, such as the PDR and CDR mentioned below, must be applied appropriately, i.e., taking into consideration item (1) above.

4.2 ATCLAB Development Phases

The expected stages of development of the ATCLAB are as follows:

I. Architecture refinement/ESIM specification. In this phase, the overall architecture outlined in this document shall be refined and criticized. The hardware and systems software performance specifications shall be refined to the point where they can support procurement of these elements. ESIM specification shall be to the level of individual interface messages and functional descriptions of the capabilities of each model. These activities must be supported by a first-level design

of ISIM elements, which in turn requires the identification of pathfinder projects and their SUT's. In this phase, selection and initial indoctrination of ATCLAB personnel can begin. It is expected that the personnel running ATCLAB will be inserted in the facility's design process as soon as practicable. A Preliminary Design Review (PDR) of the facility, or its equivalent, should be conducted at the end of this phase.

II. Detailed ESIM design/personnel development. In this phase, contracts for DEU hardware/software have been awarded, and physical facility construction has begun. Since the specific DEU hardware/software environment is known, training of ATCLAB personnel in these areas can begin, so that they will be familiar with the environment when the DEU's are delivered. This training shall emphasize procedures and programming methodologies, as well as the specific mechanics of the environment.

III. Initial ESIM coding/detailed ISIM design. This phase is assumed to begin after acceptance of the procured DEU hardware/system software. In this phase, actual development, coding, and testing of ISIM models shall be performed, as well as detailed design of the ISIM elements required to support the pathfinder projects. Also at this time, initial analysis of pathfinder SUT's is initiated. Establishment of the Facility Acceptance Criteria for the pathfinder projects should be established at this time. The reason these criteria should not be developed earlier is that sufficient information on what to expect from the facility, including the nature of the pathfinder projects and SUT's, to develop realistic acceptance standards may not be available until this moment. A Critical Design Review (CDR) of the facility, or its equivalent, should be conducted at the end of this phase.

IV. Pathfinder project preparation. In this phase, ISIM coding and detailed SUT software design for the pathfinder projects is performed. The ideal number of pathfinder projects is two, one at each of two levels of detail, with the possibility of a single project requiring two levels of ESIM detail being an interesting alternative. Actual production of SUT software is not included in this stage because it is assumed that SUT software production is part of the experiment itself. Standalone testing of complete ESIM sets (for each of the Levels) should be carried out at this time.

V. Pathfinder project execution. In this phase, actual coding of, and experimentation with, the pathfinder SUT's is carried out. At the end of this phase, the results should be matched against the expectations of the Facility Acceptance Criteria established in phase III. Satisfactory achievement of these standards shall

be used to indicate that the facility has reached operational status, even though SUT software coding, ISIM element library development, and ESIM refinement should continue for the lifetime of the facility.

4.3 Pathfinder Project: Traffic Management Unit Support System

The purpose of this hypothetical project is the development of a standalone computer system to aid the Traffic Manager at an Area Control Facility (ACF) approve direct route descents for Flight Management Computer (FMC) equipped aircraft which result both in a smooth metered flow into the terminal area of a high-density airport, and a minimum of predicted clearance conflicts. For purposes of discussion, we shall call this system the TSS (Traffic management unit Support System).

The TSS is to be developed as a standalone unit which, after laboratory tests, could be physically deployed at an ARTCC for operational trials. Experience with the TSS and its algorithms could then be used as a baseline to develop the Advanced Automation Traffic Management function.

4.3.1 Project Description

The TSS is to receive position reports and flight plan data from the NAS system.[10] In addition, the TSS may receive mode S data link messages with proposed descent profiles from the aircraft.

The Traffic Manager at the ACF receives all proposed direct route descents about 10 minutes before top of descent; the TSS must then present to the manager the consequences of that proposal in terms of clearance conflicts and flow into the final controllers. It is unclear whether approval of the proposed path (which then becomes a clearance) is to be made by the Traffic Manager alone, or if any of the affected positions (pre-descent enroute, descent, initial terminal area, final approach) should be consulted.

After approval of the proposed four-dimensional descent profile, appropriate information is presented to the affected positions to allow them to monitor conformance to the clearance. It is unclear whether this information is to be integrated in the NAS/ARTS displays (thus necessitating major modifications to these), or to be displayed on an auxilliary device connected uniquely to the TSS.

The unique development challenges of this project are the short project elapsed time,[11] its interaction with the NAS/ARTS systems, and the lack of a precise initial functional

[10] With the position reports perhaps derived directly fron Common Digitizer messages, thus bypassing the NAS system.

[11] For this project to be useful, it must be carried out in a relatively short time, say 18 months.

description. Thus, the development facility has to be able to cover the entire spectrum of conceptual development and testing, algorithm refinement, NAS/ARTS integration, and predeployment validation.

4.3.2 Initial Development Phase

Rapid conceptual development can be achieved by using two or three ATCLAB DEU's; one DEU will be dedicated to target generation, target control, and controller display. It does not seem necessary to separate the controller and pseudo-pilot functions at this time. The ESIM building blocks required are simple Level III aircraft dynamics. The Flight Management Computer also needs to be modelled. This however can be done at the functional level. Predefined descent profiles will be used and there is no need for accurate aircraft performance (e.g., fuel consumption) models. It is however important to model the aircraft conformance to the descent paths. Finally, modeling the performance of the mode S data link is also not necessary at this stage. ISIM blocks are simply a pseudo NAS or ARTS display screen. A second DEU can be used to provide a second controller screen, if required.

The last DEU is to host the TSS itself; its processor shall model the TSS logic, while its display will be the Traffic Manager's display. The processing power, memory, advanced graphics and nonalphanumeric I/O capability required of the ATCLAB DEU's should be more than sufficient to satisfy even the most demanding requirements at this stage. Assuming that the ATCLAB is properly staffed and equipped at the time the project begins, six months should suffice to develop a demonstrable, functional system and allow the transition to the next phase.

4.3.3 Refinement/Integration Phase

At this stage, the functionality of the TSS would have been sufficiently defined and tested to indicate whether any special display and/or input device is needed, how many control positions are involved, and what the data requirements to/from the NAS/ARTS systems are.

One DEU per involved controller position will be used to simulate the TSS. The use of a full DEU to simulate, for example, a simple D-controller auxiliary display may appear an overkill, but it results in a much shorter development time than the procurement of ad-hoc, simpler units which would require different programming skills.

4.4 Pathfinder Project: Dynamic Special Use Airspace

The purpose of this hypothetical project is to evaluate the feasibility of implementing the concept of Dynamic Special Use Airspace (DSUA) in the context of the Advanced

En-Route Automation System (AERA), in particular, its effects on the Conflict Probe and issuance of conflict-free clearances.

The main problem with such a project is the lack of precise definition both of the AERA functional capabilities *and* of the way these capabilities are to be used (i.e., procedures). Thus, this project is attractive not only because it would support continuation of consideration of the DSUA concept, but also, and perhaps more important, because it would force a refinement in the definition of AERA and its procedural utilization.

4.4.1 Motivation

Currently the use of airspace by civilian and military traffic is static. Controlled aircraft generally follow designated air routes, and military maneuvers use restricted, Special Use Airspace (SUA), which is permanently reserved even when it is not used.

Two developments however will tend to upset the current balance achieved through static allocation of airspace.

1. The introduction of area navigation capability and flight management computers in the cockpit has resulted in an increasing percentage of direct routing requests by the users.

2. The introduction of the new NAS software and AERA is expected to further promote and encourage use of direct routing, and at the same time allow dynamic rerouting of traffic.

Both these factors suggest that static allocation of SUA may result in considerable inefficiencies when AERA is operational and indeed may be totally unacceptable in the dynamic routing environment which is expected to be present in the NAS of the 1990's and beyond. As a result the concept of dynamically allocating SUA to suit short-term needs has been proposed.

In this scenario, the Air Force (or any other user of SUA) would notify the TMU supervisor and specify their requirements for a portion of airspace to be restricted for some future time interval. The TMU supervisor would then review the projected traffic during the interval in question and would grant the request by allocating the airspace which would least disrupt the operation of the ACF. The hope therefore is that, by tailoring the SUA allocation to specific needs, we can better serve those needs while at the same time maximizing the net availability of airspace to civilian traffic.

4.4.2 Initial Development Phase

To support this development effort, two ATCLAB DEU's will be needed.

The first will be used for target generation and control. No controller or pseudo-pilot functions will be required at this stage of development. The simulated area will encompass an entire ACF. Flight Path Network (Level II) components will be used as the ESIM building blocks for this project. A simple traffic display may also be needed for monitoring purposes only. Such a general purpose display would be provided as a basic Level II building block for the ATCLAB.

The second DEU would be used to house the ISIM blocks as well as the SUT. The ISIM for this project is a functional simulation of the AERA system. The System under test can be thought of as an enhancement of AERA and is thus implemented on the same DEU. A special purpose display will be developed to provide an interface between the TMU supervisor and the SUT software.

The bulk of the simulation needs for evaluation of the dynamic SUA concept can be satisfied by the above environment. The major technical task for this project is the development and implementation of the methodology for choosing among portions of airspace that meet the requested specifications. Analytical methods of determining flows and minimum paths in networks can be used in conjunction with a knowledge based system which can identify SUA candidates that meet the specifications and can help make decisions based on nonquantifiable factors such as controller workload as a result of unfamiliar traffic patterns, etc.

At this level of detail we can answer a number of questions:

1. How does a pop-up obstruction affect AERA's routing algorithms and conflict probe?

2. Are those functions capable of dealing with such an event at the local level or do we need to introduce the new SUA in the long-range planning process for AERA?

3. What are the advantages of Dynamic SUA when no request for any restrictions is pending?

4. How do we best decide how to grant the request with minimal disruption of normal traffic?

Given the answers to the above questions we can begin to formulate and test procedures for the requests of SUA's. We can determine what lead times may be required for the requests, and what (if any) bargaining procedures should be available. Finally, we can identify possible trends that are evident in the allocation of airspace (e.g., certain portions of the airspace seem to be vital to the traffic movement and are therefore never allocated) and thus make it possible to simplify the allocation procedures.

We note that the environment described above requires a single operator to run the simulation. In particular, aircraft control is automatic. The only requirement on the aircraft is that they follow specified flight plans which can be dynamically modified. Indeed this is the base capability of Level II aircraft blocks.

4.4.3 System-wide Effects of DSUA

At this point we will have investigated the feasibility and effectiveness of the DSUA in relative isolation from the rest of the ATC system. Indeed it is conceivable that very little coordination with other ACF's and Central Flow Control is required to implement DSUA.

There are several additional questions however which may need further investigation once the basic concept is well defined and understood. For example:

1. Is the TMU the appropriate level in the command chain to make such decisions?
2. Are there any consequences of SUA allocation in the global traffic flow?
3. If a request for a SUA severely limits the capacity of a ACF to handle traffic, what are the consequences in the global flow patterns?
4. If the initial phase indicates that it is possible to implement DSUA in the 30 minute to one hour time horizon, how is the tactical control of traffic affected? How is the information disseminated to the sectors that are affected by the SUA allocation?
5. How does the new restricted airspace affect the sector boundaries? Static SUA is well suited for static sector units since they can both be planned in advance for maximum operational compatibility. Does dynamic SUA also require dynamic sectorization of the ACF?
6. How does a change in the traffic flow within an ACF affect the tactical control at the sector level? For example, what happens in sectors adjacent to the newly allocated SUA? Does the new traffic significantly change the control problem the sector controller is faced with?

This list is by no means exhaustive. One however can see that there is a wide range of questions which require simulation of almost the entire ATC command structure. This seems to be a very common phenomenon among all interesting problems in the operation of command and control systems and emphasizes the primary reason for proposing that the ATCLAB be capable of simulating the ATC system in three distinct levels of detail.

If any of these questions is still open at this point a second phase of simulation will be required. We will need to use a third DEU in order to simulate the central flow control environment. For this we will use building blocks from the Level I simulation. In addition, one or two sectors may need to be simulated in using a fourth and fifth DEU. As in the case of the TSS project, Level III blocks will be used for this purpose.

4.4.4 ISIM Development

Unlike the TSS project, much of the ISIM blocks required for the DSUA concept simulation will clearly not be available in the standard ATCLAB kit.

In order to realistically simulate the dynamic allocation of airspace one has to first implement the AERA conflict probe, the AERA planning process, as well as other AERA support functions, at least at the functional level. In other words, in order to implement DSUA one must first have a functional simulation of the entire AERA capability. Using our nomenclature, AERA functions will become the ISIM to support the development of this SUT.

This realization points out two facts that have been already emphasized. First, the evolutionary development of the ATCLAB is a central prerequisite for its success. Even though AERA I and II are at this point in too much of an advanced stage of development to be SUT's for the ATCLAB, it is a good example of a system which after testing becomes a part of the ATCLAB internal simulation capability and can then be used by (indeed be a crucial part of) future SUT's. Second, in order to simulate a new concept, the existence of a functional specification is absolutely necessary. If such functional specification is not well defined, the implementors are forced to clarify the fuzzy elements by making ad-hoc decisions. In other words a complete functional specification of a concept will emerge out of the implementation even though this was not originally intended.

In the case of AERA such a specification does not exist due partly to the fact that a functional simulation of the AERA concept was never implemented.

APPENDIX A

REVIEW OF OBJECT ORIENTED PROGRAMMING

Object Oriented Programming is fast emerging as one of the most important developments in computer sciences, far more consequential than the "structured programming" and "top down programming" concepts of the late 60's and early 70's. It is an approach particularly suited to programs that model the physical world. This appendix presents a brief introduction to the concept of Object Oriented Programming and explores its applicability to ATC simulations.

Object Oriented Programming is a paradigm for computational processes, i.e., a model of how computation is performed by a machine as seen by the programmer. There are a number of such paradigms, each leading to a different view of what a computer is and how it behaves. Some of these are:

1. *register machine model,* typified by a classical assembly language program
2. *functional programming,* which is the essence of, and the original motivation for Lisp, and
3. *logic programming,* commonly associated with the language Prolog, but implicit in most Database query applications, such as Honeywell's MRDS (Multics Relational Data Store) systems.

There are two uses for these paradigms: first, they can serve as the basis for the construction and use of computing machines, either at the hardware or, more commonly, at the software level in the form of computer languages; second, they can be used as inspiration for programming styles in any language or machine. Thus, we find both object-oriented languages, such as Smalltalk-80, and applications written in the object-oriented style in other languages, such as M.I.T.'s LISPSIM Air Traffic Control simulator.

Reference [ABE 85] is a basic textbook on the structure and interpretation of computer programs, and includes a comparative study of various paradigms within the framework of a single pedagogic language. Reference [ELI 84] describes the authors' suggestions for development of an object oriented version of SIMSCRIPT, a widely used simulation language based in the FORTRAN programming language. Finally, [STA 84] includes a very good discussion of Object Oriented Programming and documents the features of the flavor system, a Lisp-based object oriented package.

A.1 Objects are Individual Artifacts with Local State and Functionality

We view the world as a set of individual and interacting objects, each of which has a state which may be changing over time. Each object can itself be an aggregate, composed of various objects. Thus depending on our objective, we may view objects at different levels of aggregation. An aircraft for example, can be seen as a single entity capable of flying. We can then talk about its mass, its center of gravity, its lift-to-drag ratio at specific flight configurations, etc. At the same time it can be seen as made up of components (wings, engines, control surfaces, instruments, etc.), which are objects themselves. Finally, to a NAS computer an aircraft may be nothing more than a call number and a flight plan.

Object Oriented Programming views programs as being built around conceptual entities that can be likened to real-world things. Each of these entities, called *objects*, can be characterized by:

1. *internal state*: The current instance variables of the object summarizing its history,
2. *methods*: A set of operations that can be performed on the object, and
3. *individuality*: The property of being distinguishable from other objects of the same type.

In an Air Traffic Control (ATC) simulation, the objects may include aircraft, – which carry altimeters, airspeed indicators, heading indicators and other instruments – radars, communication links, displays and display images, etc.

Immediately one can see the suitability of the Object Oriented Programming approach in modeling physical systems. Using a conventional programming approach, the programmer has to constantly maintain two mental models: the model of the physical world and the computer represenatation of that model. The mapping between the two models is the key to understanding the correspondence. At worst this key is hidden in the programmer's mind. At best it is hidden between the lines of page after page of program documentation. In both cases, any large program is very likely to become unintelligible before the initial coding is complete.

Using the Object Oriented Programming approach, on the other hand, does not require a key since the mapping is trivial. The model of the physical world and the computer model are in one-to-one correspondence. Each object abstraction has an equivalent in the computer model. Understanding the program logic therefore is independent of any mapping and requires only understanding of the simplifications and assumptions

inherent in the modeling of the physical system. In other words we now only need to document the engineering assumptions rather than the coding conventions.

Like their real-world counterparts, objects can be grouped into *classes* or *types*[12] so that each member of a class exhibits similar behavior. Indeed the aircraft, altimeters, etc. do not describe specific objects but classes of objects. An object-oriented program, therefore, defines a number of object types, a set of operations allowable for each object type, and can create a number of *instances* of each type. For example, an object class `AIRCRAFT` may be defined, and then three instances (three actual objects) of type `AIRCRAFT` may be created and manipulated by the program.

In order to distinguish two instances of the same object type, each object must maintain its own internal state information. A number of terms are used to describe an object's internal state; state variables, attributes, slots, instance variables, are but a few. We will use the term *instance variables.* An object's instance variables can be examined and altered using the operations that are defined for this object class.

We will consider an aircraft as an example of an object class. The class `AIRCRAFT` may have instance variables which include its current position (`LATITUDE`, `LONGITUDE`, `ALTITUDE`), its current speed vector (`NORTH-SPEED`, `EAST-SPEED`, `VERTICAL-SPEED`), all the onboard instruments, etc. Some of the operations that can be performed on aircraft might involve simply accessing the appropriate instance variables, such as `GET-LATITUDE`, `GET-LONGITUDE`, `GET-ALTITUDE`, etc. In addition, we can define operations like `SET-LATITUDE`, to alter those instance variables. Finally, we can define operations such as `GET-SPEED` and `GET-DIRECTION` which, rather than simply returning the value of an attribute, perform the calculations required to compute and return the polar coordinates of the aircraft's velocity.

Finally, objects have individuality which is distinct from their state description. Consider two instances of `AIRCRAFT`: `AC1` and `AC2`. Neglecting for a moment the physical impossibility of having two things in the same place at the same time, these two aircraft may have exactly the same state. It can be said therefore that they are equal. However, they are not the same aircraft. The only way to find out if `AC1` is really the same as `AC2` is to change the value of one of the instance variables of `AC1` and subsequently compare the states of the two aircraft again. If the states remain the same independent of what attribute we modify then the two objects are really the same object, and we state that `AC1` is *equal* to `AC2`. If `AC1` and `AC2` do not pass this test (that is, they are not equal), but the values of all their instance variables are identical, we state that `AC1` is *equivalent* to `AC2`.[13]

---

[12]The two terms will be used synonymously.

[13]Note that in conventional programming, the identity question does not arise, so that there is no dichotomy between the terms *equal* (i.e., being the same object) and *equivalent* (i.e., having the same

Even in this very basic form, Object Oriented Programming style helps and encourages the design of simple, modular programs. Since the state of any object can only be manipulated directly by a well defined set of operations, these become the natural interface of the object with the rest of the world. The object becomes a black box which behaves in a well defined way, while the remainder of the program is not required to have any knowledge of the internal logic and structure of the object. Again looking at the aircraft example, we could have chosen to implement speed and direction as the aircraft instance variables and define operations for the aircraft's north and east speeds. To the world outside the aircraft object, however, our choice between rectangular and polar coordinates would be transparent.

A.2 Generic Operations on Objects

Let us consider the ATC simulation example. As aircraft are moving in the simulated airspace, other objects will need to find out their current location. To accomplish this they will need to access the aircraft's latitude, longitude, and altitude. Those instance variables, as we know, can be accessed through the operations `GET-LATITUDE`, `GET-LONGITUDE`, and `GET-ALTITUDE`. Looking a little ahead we can see that this presents a serious problem. Other objects in the ATC environment are characterized by (i.e., need to have as instance variables) latitude, longitude and altitude. However `GET-LONGITUDE` as defined previously will work only for aircraft and cannot be used for any other object type.

The traditional solution to this problem is to use the name `GET-AIRCRAFT-LONGITUDE` for the function that is specific to aircraft, so that we can then use `GET-RADAR-LONGITUDE` for radar objects, `GET-VOR-LONGITUDE` for VOR's etc. This approach solves the immediate problem, but creates an even more formidable one. In a typical ATC environment we would often need to find distances between objects. Aircraft require to find their distance from the next waypoint, conflict alert systems need to find the distance between two aircraft, etc. Can we create a general purpose function to find the distance between two objects? Obviously not. In fact if there are n objects defined in our program, we would need n^2 such functions.[14] The problem spreads very quickly since every function that requires latitude and longitude could be specific to a particular object type. In practice, the Object Oriented Programming structure of our program would have to be abandoned. We can witness this type of breakdown in conventional computer languages which implement structures (e.g., C, PASCAL, PL/1, and lately FORTRAN) but do not truly support Object Oriented Programming.

To overcome this problem we introduce the concept of *generic operations*. We can

---

value).

[14] or equivalently a single function with n^2 distinct branches.

think of GET-LATITUDE not as an operation on a particular object type but rather as the name of generic operation applicable to all objects which have LATITUDE as an instance variable. *Sending a message* is a commonly-used term, inherited from Smalltalk, denoting a request for the performance of a generic operation on an object. To perform this operation we still need to know the object's type and the name of the operation to be performed. In this case however the system keeps associations between the name of the operation and the actual function to be invoked for each object type. The function invoked for an object type in response to a message is called the object type's *method* for the generic operation. Thus all aircraft objects share a GET-LATITUDE method, a GET-LONGITUDE method, etc, and these are distinct from the GET-LATITUDE and GET-LONGITUDE methods for objects of type RADAR, or VOR.

In summary, we have the following universal protocol for performing a generic operation on some object: we send the object a *message* consisting of the name of an operation (e.g., GET-LATITUDE) and possibly some arguments. The actual method, when executed, may return a value, or may perform a side-effecting operation, but in any case the effects of the message can depend on the type of object which receives it.

Prior to introducing generic operations, we mentioned operations being performed *on* rather than *by* objects. This implied that objects were passive elements of the program since the caller determined the function which was to be invoked. With generic operations and message passing, the object itself plays an important role in determining which function is invoked and therefore the exact effects of the message. From the user's standpoint, the object is active: it receives messages and it responds by returning a value, or by some side effect, or both.

The concept of generic operations is not new. The need for generic arithmetic functions was identified and implemented since the early days of FORTRAN. What is new is the mechanism by which the user defines and uses generic operations. This simple mechanism has become one of the most powerful concepts of Object Oriented Programming.

A.3 Inheritance of Instance Variables and Behavior

Often we find we can abstract common behavior from objects of different classes. In the examples above, we found that many objects have common instance variables. In addition we found that distance computation is a common feature for all objects that have latitude, longitude and altitude as their instance variables.

In fact what is evident is that aircraft, radars, VOR's etc, all belong to a more general class of objects: namely those that have geographic location.[15] Object Oriented Pro-

[15]It is true that this can be said of all physical objects. We however are interested in objects whose

gramming provides a mechanism which allows us to define GEOGRAPHIC-LOCATION as an abstract object type with instance variables latitude, longitude and altitude, and then define other object types like aircraft, radar, VOR etc., which inherit both the instance variables *and all the methods* of GEOGRAPHIC-LOCATION.

Inheritance opens a horizon of new capabilities which are absent from any other programming methodology. In addition to breaking the programming task along the well understood interactions of distinct objects, we can look at a complex object as an amalgam of simpler, more manageable behaviors. Thus we can look at an aircraft as a mixture of the following simpler abstractions:

1. *Geographic Location*: maintains information on the aircraft's location and provides methods for calulating distances from other objects of the same type.
2. *Moving Object*: maintains speeds and accelerations and provides methods for integrating the aircraft's state.
3. *Flying Object*: maintains bank angle and relates longitudinal and lateral acceleration to the accelerations provided by *moving object.*
4. *Aircraft Control System*: maintains current commanded state and implements all control loops for the aircraft's autopilot.
5. *VOR Receiver*: maintains VOR receiver status and implements lateral deviation indicators for use by the aircraft control loops.
6. *Altimeter*: provides current altitude indications for altitude control loops.
7. *Flight Plan Mixin*: maintains flight plan clearances and provides appropriate editing and display capabilities.

New capabilities can be added to aircraft by defining new abstract object types and including them in the mixture. Run time mixing of capabilities is also possible. For example it is possible to define 4D-RNAV capability and only include it in a portion of the aircraft that are generated for a specific run.

A.4 Conclusion

There are three basic reasons Object Oriented Programming should be considered the most appropriate approach for implementing system simulations:

---

geographic location is a significant attribute in their behavior within the context of our model.

1. The abstraction barriers provided by Object Oriented Programming enhance the desirable modularity in programming that makes large systems manageable.

2. The software architecture that results from programming in object-oriented style often match the experiential perception of the system being simulated; this simplifies the mapping between elements of the real system being simulated and the corresponding software elements simulating them, and between the flow of causality in the simulated system and the flow of control in the simulation.

3. The Active-Object/Message-Passing paradigm of computation seems to lead to a practical and effective way of implementing concurrent, synchronized multiprocessing, and indeed has been proposed as an approach to concurrent simulations (reference [JEF 85]).

On the other hand, it must be made very clear that there is nothing in an object-oriented program that could not be coded in a conventional way, much in the same way that there is no Lisp program that could not be implemented in FORTRAN, BASIC, or, for that matter, machine code. It is a matter of convenience. And time.

In reading this appendix, one is tempted to translate the Object Oriented Programming concepts and examples to traditional function calls – after all, that is what the compiler would be doing; however, when one's mind is in simulation rather than in language technology, objects and message-passing are much closer to the real world one is trying to emulate than function calls.

Developers and promoters of Object Oriented Programming styles face the following two problems: first, the advantages of these styles – as well as their drawbacks – can only be perceived by practical use; second, since the fundamental advantage of these methods lies in the management of complexity, it is hard to convey with the simple programming examples that one is likely to encounter in a textbook or a report. The authors predict that user experience will be the only real promoter of these styles.

APPENDIX B

SAMPLE OBJECT DEFINITIONS IN LISP

The following code is extracted from LISPSIM, the lisp-based ATC simulator developed at the Flight Transportation Laboratory at MIT. Note how the object type TRANSMITTER is defined first, and then both VOR and ILS are built on this type. At the same time TRANSMITTER is itself built on another object type called GEOGRAPHIC-LOCATION.

```
;;; -*- mode:lisp; readtable:cl; package:atc -*-
;;;
;;;  ATC Ground instruments.
;;;
;;;  Defined instruments are:
;;;  VOR, ILS, and soon to come RADAR
;;;
;;;  Transmitter is a base flavor for ILS and VOR.
;;;
;;;  Transmitter mixin
;;;

(defobject transmitter
  ((frequency 000.0)) ; instance variables
  ()
  (:included-flavors geographic-location)
  (:init-keywords :frequency)
  (:documentation "Base flavor for all transmiting objects.")
  (:equality-criteria :frequency)
  :gettable-instance-variables
  :flavor-not-instantiable)

(defmethod (transmitter :before :init)(init-plist)
  (let* ((freq (get init-plist :frequency)))
    (or (null freq)(send self :set-frequency freq))
    self))

(defmethod (transmitter :print-self)
    (&optional (stream *standard-output*) printlevel slashify-p)
```

```
    (oustr (format nil "#<~a ~a ~a>" si:name (typep self) frequency)))

(defmethod (transmitter :set-frequency)(new-frequency)
  (cond ((numberp new-frequency)(setq frequency new-frequency))
((stringp new-frequency)(char-to-number frequency))
(t (send self :set-frequency
 (cerror t () :wrong-argument-type
 "~s is not a number nor a string" new-frequency)))))

;;;
;;;  VOR transmitter
;;;

(defobject vor
  ((type 'h)) ; instance variables
  (transmitter) ; included flavors
  (:init-keywords :magvar :type)
  :gettable-instance-variables
  :inherited-equality-criteria)

(defmethod (vor :before :init)(init-plist)
  (let ((vor-type (get init-plist ':type)))
    (or (null vor-type)(setq type vor-type))
    self))

(defun make-vor(&rest keyword-arguments-to-make-instance)
  (apply #'make-instance 'vor keyword-arguments-to-make-instance))

;;;
;;;  ILS transmitter
;;;

(defobject ils
  ((glide-slope-tangent 0.0524077793) ; instance variables
   (true-course 0.0)
   (om-distance 9260.0))
  (transmitter) ; included flavors
  (:init-keywords :true-course :glide-slope :om-distance)
  :gettable-instance-variables
  :inherited-equality-criteria)
```

```
(defmethod (ils :before :init)(init-plist)
  (let* ((ils-course (get init-plist ':true-course))
 (glide-slope (get init-plist ':glide-slope))
 (om-dist (get init-plist ':om-distance)))
    (or (null ils-course)(setq true-course
       (* *degrees-to-radians* ils-course)))
    (or (null glide-slope)(setq glide-slope-tangent
(tan (* *degrees-to-radians* glide-slope))))
    (or (null om-dist)(setq om-distance (* *nm-to-meters* om-dist)))
    self))

(defun make-ils (&rest keyword-arguments-to-make-instance)
  (apply #'make-instance 'ils keyword-arguments-to-make-instance))
```

GLOSSARY AND ABBREVIATIONS

ACF	Area Control Facility.
ADA	New Department of Defense standard language.
AERA	Automated EnRoute Air traffic control.
ARTCC	Air Route Traffic Control Center.
ARTS	Automated Radar Terminal System.
ATC	Air Traffic Control.
ATCLAB	Authors' acronym for the proposed ATC automation development and testing facility. Not an official FAA name.
C	A third generation programming language.
CD	Common Digitizer.
CDR	Critical Design Review.
DEU	Display and Execution Unit.
DSUA	Dynamic Special Use Airspace.
ESIM	External Simulation Model.
FMC	Flight Management Computer.
FTL	Flight Transportation Laboratory, M.I.T.
HOL	High Order Language.
HW	Hardware.
I/O	Input/Output.
ISIM	Internal Simulation Model.
LAN	Local Area Network.
NAS	National Airspace System.
NSSF	National Airspace System Simulation Support Facility.

MB	Megabytes.
MDEU	Master Display and Execution Unit.
MIPS	Million of Instructions Per Second.
MIT	Massachusetts Institute of Technology.
PDE	Project Dependent (software) Element.
PDR	Preliminary Design Review.
PIE	Project Independent (software) Element.
PL/1	A third generation programming language.
SCE	Simulation Core Element.
SSU	System Support Unit.
SSF	System Support Facility (of NAS).
SUA	Special Use Airspace.
SUT	System Under Test.
SW	Software.
TATF	Terminal Area Test Facility (of ARTS).
TMU	Traffic Management Unit.
TSS	Traffic management unit Support System.

Bibliography

[ABE 85] Abelson, H., and Sussman, G. J., *Structure and Interpretation of Computer Programs*, M.I.T. Press, Cambridge, MA, 1985.

[BIR 73] Birtwistle, G. M., Dahl, O-J., Myhrhaug, B. and Nygaard, K., *SIMULA BEGIN*, Auerbach Publishers Inc., Philadelphia, PA, 1973.

[GOL 83] Goldberg, A. and Robson, D., *SMALLTALK-80: The Language and its Implementation*, Addison-Wesley, Reading, MA, 1983.

[JEF 85] Jefferson, D., and Sowizral, H., "Fast Concurrent Simulation Using the Time Warp Mechanism." *Proceedings of the Conference of Distributed Simulation 1985*, Society for Computer Simulation, January 1985, pp. 63-69.

[KLA 82] Klahr, P. and McArthur, D., "The ROSS Language Manual." Rand Note N-1854AF, The Rand Corporation, Santa Monica, CA, September 1982.

[NUG 83] Nuggent, R. O., "A Preliminary Evaluation of Object-Oriented Programming for Ground Combat Modeling." Working Paper WP-83W00407, The MITRE Corporation, Mc Lean, VA, 1983.

[STA 84] Stallman, R., Weinreb, D., and Moon, D., *Lisp Machine Manual*, Sixth Edition (system Version 99), Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA, June 1984.

[BAS 79] Basak S., "ATCSF Data Link Communication Interface Analysis." Report No ATCSF-97-022, Prepared for the FAA Technical Center by Computer Sciences Corporation, July 1979.

[NSSF 85] "National Airspace System Simulation Support Facility Baseline Simulation Software: Ground Facility Program / ARTS III Controller Display Program Software Interface Control Document." Technical Report No NSSF-85-005-P00, Prepared for FAA Technical Center by Sperry Corporation, FAA Technical Center, Atlantic City, NJ, April 1985.

[CD 77] "Common Digitizer Record Program: Functional Specification." Technical Report FAA-4106F-3, Airway Facilities Service, FAA Technical Center, Atlantic City, NJ, September 1977.

[ELI 84] Elias, A. L., and Pararas J. D., "Object-Oriented SIMSCRIPT." Unpublished Memo prepared for CACI Federal, La Jolla, CA, October 1984.