

POTENTIAL USE OF ARTIFICIAL INTELLIGENCE
TECHNIQUES IN AIR TRAFFIC CONTROL

Antonio L. Elias and John D. Pararas
Massachusetts Institute of Technology

A Blasted Introduction to Artificial Intelligence

Edward Feigenbaum, in his "Handbook of Artificial Intelligence" defines artificial intelligence (AI) as "the part of computer science concerned with designing intelligent computer systems". This is a very easy definition to make, since it shifts the burden of definition to another one, namely that of "intelligent computer system". Feigenbaum then proceeds to define intelligent computer systems as those which "exhibit the characteristics we associate with intelligence in human behavior". Unfortunately, this one is not very helpful either, since we now must define intelligence itself, a rather formidable task.

But even if we were able to define human intelligence, we would still have a problem, since "intelligent behavior", when applied to computers, cannot be equated with intelligent behavior in general. As little as a hundred years ago, computing the square root of a number was unequivocally a manifestation of human intelligence, since it required a number of decision-making steps depending on the signs of intermediate values, remainders, and so on. Yet today, nobody would call the square root calculating ability of a computer intelligent behavior.

In view of this inability to satisfactorily define intelligent behavior as applied to a computer, some people have slightly altered the classical definition to mean doing with a computer something you normally don't expect a computer to be able to do. This definition appears to be satisfactory, since taking square roots, for example, is something you expect a computer to do, so a system that takes square roots of numbers is definitely not an artificial intelligence system, while a system that composes concert music does appear intelligent, since computers do not usually compose concert music.

The problem with this new definition is that it is self-defeating: the moment one builds a computer system that does something you do not expect a computer to do, it does it, so it ceases to become an artificial intelligence system. As paradoxical as this may seem, it actually makes some sense: today, one can purchase battery-powered toys at a department store that do a better job at synthesizing speech or playing chess than the most sophisticated experimental equipment did just ten years ago.

In view of this difficulty in defining what precisely artificial intelligence is, perhaps we should simply describe a little of its history and some of its typical products. AI research has traditionally had three distinct objectives: first, to understand the high-level workings of the human brain by constructing functional computer models of human activities, such as vision and reasoning; second, to build computers based on the brain model; and third, to build a mechanical man, perhaps by combining the results of the other two efforts.

The motivation of the first line of research is a better understanding of the human brain, and any computer functionality that may result from this research is purely secondary. The motivation of the second, to build computer systems - both hardware and software - patterned after the human

brain, is the goal of building better computer systems, independently of the specific applications.

Now the third goal is the most elusive one; humans had the dream of building mechanical replicas of themselves for at least as long as they dreamed of flying. If we have been able to fulfill the dream of flying, is there any reason we will not be able to build a true robot? Perhaps, but we should be very careful not to identify artificial intelligence with only this goal. If we do, we will miss what perhaps are the most useful benefits from AI research. However, these benefits are not the intelligent machines per se, but rather the computer technology that has been developed as a consequence of the quest for machine intelligence. AI is also not only expert systems. The popularity and press coverage that expert systems have recently received have caused a lot of people to believe that the only useful product of artificial intelligence research consists of expert systems.

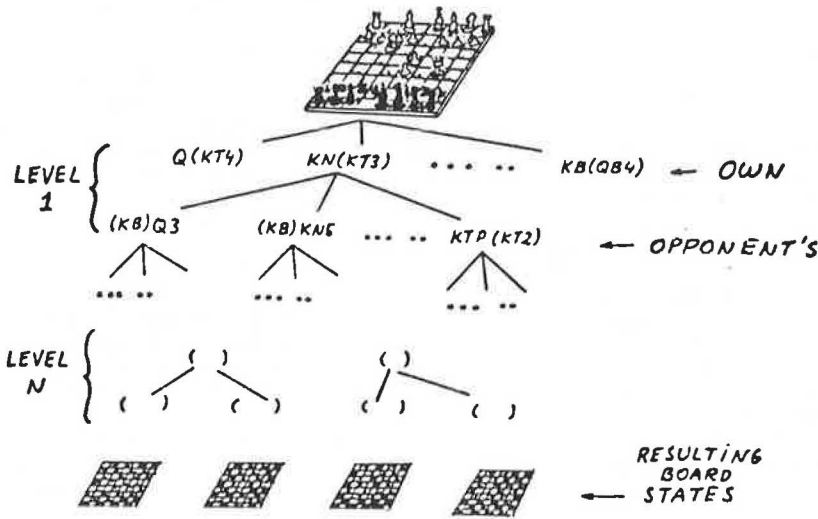
Historically artificial intelligence had its roots in the discipline of mathematical logic, sometimes also called symbolic logic, the study of the processes by means of which we construct the mental models we call mathematics. It was with the discovery, by Turing and others, that these symbols could be manipulated and operated upon mechanically with the same ease as numbers - although with a different set of operations, naturally - that the possibility of a computer performing these intelligent functions was first postulated. Indeed, Lisp, now considered to be the programming *lingua franca* of the AI community, can be considered either a programming language, or a convenient, elegant, and powerful method of expressing mathematical concepts.

The first attempts at using computers to manipulate symbols for a purpose started by defining a simple problem to be solved. The kinds of problems that early AI systems were capable of handling had two common characteristics: the goal, or problem, was very simple to state, but the solution to the problem was complex and non-trivial. The measure of success used in the development of these game-playing systems was this: could the program play a better game than the people that build it?

The common technique used in these systems was the generation of large sets of alternatives, followed by a process of search (for a desired solution), usually coupled with procedures that reduce the number of alternatives to be evaluated to a reasonable subset. In a chess-playing program, the alternatives are the sequence of legal moves and counter-moves that can be made by the program and its opponent from the current state of the board, alternatives that can be structured as a tree; the search consists in the successive evaluation of each branch of the tree to find the most convenient immediate move, evaluation that may include not only the eventual end state of the board at the end of that branch of the tree, but the likelihood of each of the opponent's moves (Figure 1).

Before these solutions can be generated, searched, reduced and evaluated, some symbols and operations must be defined; in other words, a representation of the problem must be designed. For example, a chess-playing system may operate on descriptions of the state of the chess board, that is, the position of each of the pieces; the operations that can be performed on these descriptions would include valid piece movements, or functions that measure the desirability of having a piece in a certain position relative to other pieces. Other symbols that may be involved could include standard

Figure 1. Demonstration of alternative moves on a chess board.



moves, such as the classical chess opening moves, in such a way that the system can easily recognize when the opponent has performed such a move, and know what the consequences of that move are without elaborate analysis.

In spite of the spectacular performances that such systems exhibit - few human chess players can outperform the best chess playing programs today - these efforts were in a way disappointing because of the extremely narrow focus of the results. While some of the searching and problem reduction techniques developed as a consequence of that research are applicable to a large class of problems, the problem representation aspects were extremely case-dependent: the symbols and operations developed to solve chess moves are of little or no value outside that specific problem domain.

This frustration led in the late 1960s and early 1970s to a flurry of efforts to find more universal problem representations; ideas such as problem solving systems and logic reasoning systems seemed attainable at the time. At one time work actually began on a general purpose problem solver system, with no clear limitations on what kind of problems it could solve. When it became apparent that finding truly domain-independent means of representing problems was a little too difficult, researchers then directed their efforts to more restricted, but still relatively generic problems, for example proving mathematical theorems or automatic computer programming.

Also at this time, and perhaps influenced by the success of the early game-playing programs, some individuals began to make exaggerated claims about the practical possibilities of AI systems. Actually, this had already happened before, even before the term artificial intelligence had been coined. Grossly unrealistic estimates of the potential of computers to perform intelligent functions - whatever those may be - were common during the early days of electronic computers, as exemplified in the contemporary label electronic brain.

Figure 2 was produced directly from the terminal screen in the Flight Transportation Laboratory of the Massachusetts Institute of Technology computer complex. Lines with numbers beginning with the letter C are the authors' inputs, while lines identified with the letter D are the outputs from M.I.T.'s MACSYMA program. We begin

this example by typing in an equation in a form which looks very much like FORTRAN. Notice, however, that MACSYMA displays the input back in a form that resembles the way one would write this equation on a blackboard; this is possible because the representation of that equation used in MACSYMA has concepts such as power and denominator.

We then ask MACSYMA to "solve" for the variable Y in that equation; what happens next is that MACSYMA recognizes a binomial equation pattern in that expression, and invokes the rules to solve such equations that we all learned in high school. If, on the other hand, we ask MACSYMA to integrate the right-hand side of that equation, it will recognize a polynomial pattern, and invoke the classical polynomial integration rules. While we could do that ourselves, we would be hard pressed to integrate the expression shown in the next figure, which involves knowing some rather exotic rules of integration. We use MACSYMA frequently in our work, especially to manipulate rotation matrices that transform, for example, position vectors in radar site coordinates to mosaic-relative coordinates. This kind of system which is composed of an internal representation of a domain, a set of rules representing knowledge in that domain, and a set of commands that allow the user to invoke the appropriate rule without knowing the details of that rule is called a knowledge-based system, or more precisely, a stored-knowledge system;

Figure 2. A typical MACSYMA session.

```
(C1) x = 3 * y^2 - 2 * y + 17;
(D1)          2
            x = 3 Y - 2 Y + 17

(C2) solve(d1,y);
(D2)          3          3
            [Y = - -----, Y = -----]
            3          3

(C3) integrate(part(c1,2),y);
(D3)          3  2
            Y - Y + 17 Y
```

Figure 3. A typical MACSYMA session (continued).

```

(C4) sqrt((x+1)/(x-1));
(D4)          X + 1
          Sqrt(-----)
          X - 1

(C5) integrate(d4,x);
          X + 1          X + 1          X + 1
          LOG(Sqrt(-----) + 1) LOG(Sqrt(-----) - 1) Sqrt(-----)
          X - 1          X - 1          X - 1
(D5) - 4 (- ----- + ----- - -----)
          4          4          2 (X + 1)
          X - 1          X - 1          X - 1

(C6) ratsimp(d5);
(D6)          X + 1          X + 1          X + 1
          LOG(Sqrt(-----) + 1) - LOG(Sqrt(-----) - 1) + (X - 1) Sqrt(-----)
          X - 1          X - 1          X - 1

```

the computer program has the knowledge, but does not know when and how to apply it unless specifically instructed by the user (Figure 3).

The indisputable success of these stored-knowledge systems, coupled with the failure of totally generic problem-solving or thinking programs, resulted in the expert system concept. Like the stored knowledge system, an expert system operates with symbols and operations representing knowledge in a particular field, and sets of pre-stored rules which embody knowledge, just like the integration rules of MACSYMA. The expert system, however, has two additional ingredients: first, the capability of chaining the given rules, perhaps with the help of intermediate results, to reach conclusions that are not covered by any single rule; and second, logic to direct both the invocation of the rules and the chaining of simple rules to achieve a specific objective.

Perhaps the best way to understand the notion of expert systems is to observe one in operation. The next figures will show a typical conversation with perhaps the most famous, or successful, of all expert systems: the mycin bacterial infection diagnostic program. In the mycin system a moderately large set of rules (about one hundred) is used to store knowledge about bacterial infections. But whereas in MACSYMA the rules were

scanned to see if one of them satisfied the request typed by the user (e.g., integrate a given expression), mycin rules are automatically activated in a complex way by a logic in the program called the inference engine, whose built-in goal is fixed: to determine the best antibiotic treatment for a patient whose infection is not precisely known.

Figure 5. A typical mycin session (continued).

```

6) Please give the date and approximate time when signs or
symptoms of the primary-bacteremia (infection-1) first appeared
(mo/da/yr)
*** May 5 1980

The most recent positive culture associated with the
primary-bacteremia (infection-1) will be referred to as:

----- CULTURE-1 -----

7) From what site was the specimen for culture-1 taken?
*** Blood

8) Please give the date and time when this blood culture
(culture-1) was obtained. (mo/da/yr)
*** May 9 1980

The first significant organism from this blood culture
(culture-1) will be referred to as:

```

Figure 4. A typical mycin session.

```

----- PATIENT-1 -----

1) Patient's name: (first-last)
*** Fred Smith

2) Sex
*** Male

3) Age:
*** 55

4) Have you been able to obtain positive cultures from a site
at which Fred Smith has an infection?
*** Yes

----- INFECTION-1 -----

5) What is the infection?
** Primary-bacteremia

```

To understand why mycin behaves the way it does, indeed to understand the importance of the development of mycin, we have to review briefly what kind of problem it is solving. There are thousands of bacteria that can cause infections in humans, and hundreds of available antibiotic drugs that act specifically on a bacteria, or groups of bacteria, while having more or less desirable side effects. If the identity of the bacteria causing a patient's infection were to be known, a very precise antibiotic treatment could be prescribed. Unfortunately full identification of a bacteria from a culture requires from a few days to a few weeks, while antibiotic treatment must begin immediately. Thus, treatment is begun with only sketchy data on what kind of bacteria is involved - usually a two to three hour culture yielding only very basic information about the bacteria, not its precise identity.

Figure 6. A typical mycin session (continued).

```

----- ORGANISM-1 -----
9) Enter the identity of organism-1:
*** unknown

10) Is organism-1 a rod or coccus:
*** rod

11) The gram stain of organism-1:
*** gramneg

12) Have there been positive cultures from sites which are NOT
associated with infections about which you are seeking advice?
*** no

13) Did organism-1 grow aerobically?
*** yes (0.8)

14) Did organism-1 grow anaerobically
*** yes
    
```

Figure 7. A typical mycin session (continued).

```

15) What is the suspected portal of entry of organism-1 into
this sterile site?
*** GI (0.6)
    
```

My therapy recommendation will be based on the following probable infection(s) and potential causative organism(s):

```

INFECTION-1 is PRIMARY-BACTEREMIA
<Item 1> The identity of organism-1 may be PSEUDONONAS-AERUGINOSA
<Item 2> The identity of organism-1 may be KLEBSIELLA-PNEUMONIAE
<Item 3> The identity of organism-1 may be E.COLI
<Item 4> The identity of organism-1 may be BACTEROIDES-FRAGILIS
<Item 5> The identity of organism-1 may be ENTEROBACTER
<Item 6> The identity of organism-1 may be PROTEUS-NON-MIRABILIS
    
```

```

46) On a scale of 0 to 4, where higher numbers indicate
increasing severity, how would you rate Fred Smith's degree
of sickness?
*** 3
    
```

Figure 9. A typical mycin rule.

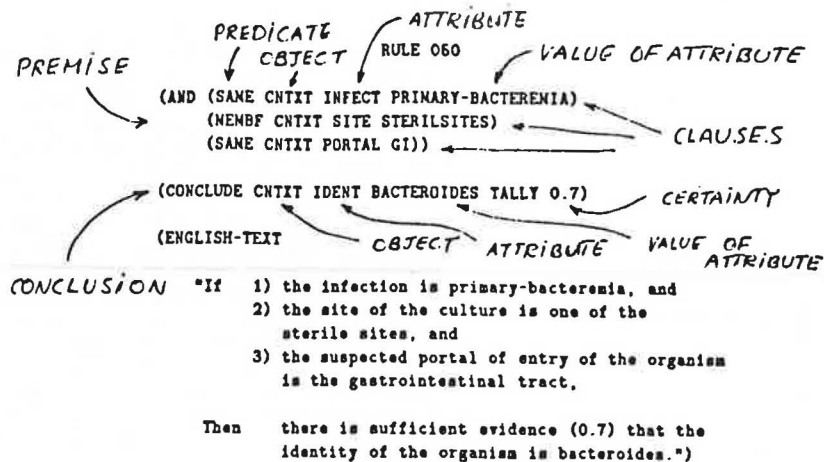


Figure 8. A typical mycin session (continued).

```

47) Does Fred Smith have a clinically significant allergic
reaction to any antimicrobial agent?
*** no

48) Fred Smith's weight in kilograms (or <number> pounds):
*** 70
    
```

[Rec 1] My preferred therapy recommendation is as follows:

In order to cover for Items <1 2 3 5 6>:

```

Give: GENTAMICIN
Dose: 119 mg (1.7 mg/kg) q8h IV [or IM] for 10 days
Comments: Modify dose in renal failure
    
```

In order to cover for Item <4>

```

Give: CLINDAMYCIN
Dose: 595 mg (8.6 mg/kg) q6h IV [or IM] for 14 days
Comments: If diarrhea or other GI symptoms develop,
patient should be evaluated for possible
pseudomembranous colitis.
    
```


The way mycin achieves this objective is by assuming a very large number of possible alternatives and then asking the user to provide information that it can use to eliminate as many alternatives as possible, until all the information is exhausted. This technique is known as backwards-chaining the rules.

The conversation shown in Figures 4 to 9 is a little long, but interesting. Mycin's questions are preceded by a number, while the user's answers are preceded by three asterisks. After the usual basic questions about the patient, mycin checks in question (4) that the basic operating premise, that is, the existence of an infection, is indeed true. If one were to answer no to that question mycin would simply say goodbye.

At the very beginning of the conversation mycin printed the label PATIENT-1; after question (4) it prints the label INFECTION-1; these labels are an indication of the context of the conversation. When humans exchange information verbally we implicitly establish a context in which indefinite articles such as it or the have a unique meaning. Although mycin does not understand English it always has a current context, or implicit object of inquiry which begins with the patient, switches to the first infection (for that patient), and then may change to an organism, to a culture, change back to the patient, and so on.

After establishing that the type of infection is known, so that a series of questions leading to the identification of the type or possible types of infection is not necessary, mycin then proceeds to find out what laboratory information has been obtained on the organism or organisms producing the infection. Answers to a question, including the answer "don't know", dynamically modify the sequence of successive questions. Note also that the user's answers can be followed by a number in parenthesis, such as in question 13; this indicates the degree of confidence that the user has in that piece of information, with 1 indicating absolute certainty, and 0 being equivalent to a don't know answer.

After about forty or so questions mycin is ready to display a conclusion; perhaps it is satisfied that this conclusion has a low enough uncertainty factor, or, more likely, the user has begun to answer "I don't know" to so many questions that mycin decided that to give up asking. In any case mycin displays first, its conclusions regarding the possible identity of the organism causing the infection. As you can see, it is not a single conclusion, but rather six conclusions. Next, after three additional questions mycin proceeds to issue a 'preferred treatment', preferred in that there may be other treatments covering the same set of bacterial infections and which may be preferable to the user for reasons that mycin cannot handle (for example, local availability).

The next figure shows the form of a typical mycin rule. On the top of the figure is the text of the 50th rule, as stored in mycin, while a comment in English at the bottom of the figure explains the meaning of the rule (for the benefit of humans). The rule has two parts: a premise and a conclusion. If the premise is true then the conclusion is true, much like an *if-then* statement in a traditional programming language such as Fortran. A program using this kind of rules is sometimes called a production system.

The premise is in itself composed of the boolean, or logical combination of three clauses; each clause in itself consists of a predicate - a statement that may or may not be true - relating an attribute of an object to a value. For example, in

the second clause of rule 50's premise, MEMBF (meaning "a member of") is the predicate, CNTXT is the object - actually, this stands for "the current context, whatever it may be" - SITE is the attribute, and STERILSITES is the value with respect to which that object's attribute must satisfy the predicate. This clause would be true if the value of the SITE attribute of the current context is a member of STERILSITES (presumably a list of values).

The action part of Rule 50 consists simply of the identifier CONCLUDE followed by a statement of value of an object's attribute, possibly followed by a certainty index: here, the rule affirms that the IDENT attribute of the context is BACTEROIDES with a certainty of 0.7. Note that this fact could have been established by the user if he had answered positively question number 9, which asked "Enter the identity of organism-1". Mycin rules are triggered by values of attributes, and these values can be established either by user's answers or by rules' conclusions. Indeed, mycin's backwards chaining logic determines which questions to ask the user by determining which rules, if triggered, would restrict the potential conclusions the most.

Air Traffic Control Applications of AI Technology

This overview of the world of artificial intelligence has been, by necessity, very brief. It has not covered, for instance, any of the work done in a natural language processing, that is, the analysis of human language - written or oral - to extract specific information. We have not covered speech synthesis and recognition - a different problem than that of understanding natural language. We have not covered robotics, the discipline that deals with mechanical manipulators and touch sensors. Finally, we will only mention vision and image recognition, even though we believe there may be an opportunity for air traffic control (ATC) applications of artificial vision.

It seems that in order to do justice to the title of this presentation we should also briefly mention what we mean by ATC. By air traffic control we do not mean exclusively the activity of the man or woman behind the radar screen issuing vectors and clearances to aircraft and looking out for conflicts; we very specifically include all the activity that, combined, makes for a safe and efficient ATC system, such as planning the command and control structure of the system - that is, determining when and where information is transmitted, and when and where decisions are made - or selecting the set of airways that will constitute the preferential routes from two busy terminal areas in a particular complex weather situation. The possibilities for useful applications of AI technology to the world of ATC go well beyond the radar controller's screen.

Some of the technologies of AI can be of quite immediate applications; others may have to wait five, ten or even twenty years before they can be seriously considered. We will mention both short term and long term applications and will divide these immediate and future applications in a different way, namely two groups which we call visible and invisible.

Invisible applications are those where the AI component is hidden from the final user of the ATC product or system. Perhaps AI technology was used in the design, development or implementations of the system for economic reasons, or perhaps it is the only way in which to mechanize a certain function, but as far as the user is concerned, it is just another computer program.

In a visible application, on the other hand, the particular behavior of an AI product, as typified in the mycin example, is an essential part of the usefulness of the tool, and the user must be prepared and trained to use it in this way. In the invisible category we would like to mention symbolic programming, experimental simulation, radar tracking algorithms, and procedure generation. In the visible category we would like to propose a theoretical flow oriented command and control structure, an expert system to help select runway configurations, two very similar applications of visual scene recognition, and the controller's assistant concept.

"What?" you will say, "they are not going to talk about applications of voice recognition?" About the only application we can foresee for this technology is the simulation of pilots' voices - and ears - in a real time ATC simulation, and we are afraid that the available technology is not capable of doing even this. At the present time voice recognition and synthesis seems to be more of a solution looking for a problem, than a solution to an existing problem.

Artificial Intelligence and the Management of Complexity

The history of aeronautical technology has always been characterized by barriers, or measures of performance that were considered unattainable: transoceanic flight, stratospheric flight, blind flying, the sound barrier, the heat barrier, space. One by one these barriers have been conquered. We believe that the current barrier, the one performance limit we must conquer today, is the complexity barrier. Consider this: Charles Lindberg's aircraft, the Spirit of St. Louis, required 850 man-hours of engineering effort to design; the Lockheed C5A Galaxy transport jet took 49 million man-hours to design. As aircraft become more complex, and as the relationships between aerodynamics, propulsion, avionics, and even radar signature become more and more interrelated in determining the performance of the aircraft, this complexity, and the cost of designing it, will become greater and greater.

Nowhere is this more dramatic than in present and future ATC systems. the United States ATC system has already been dubbed "the most complex man-machine system in the world"; indeed, its complexity has reached a point where nobody quite knows

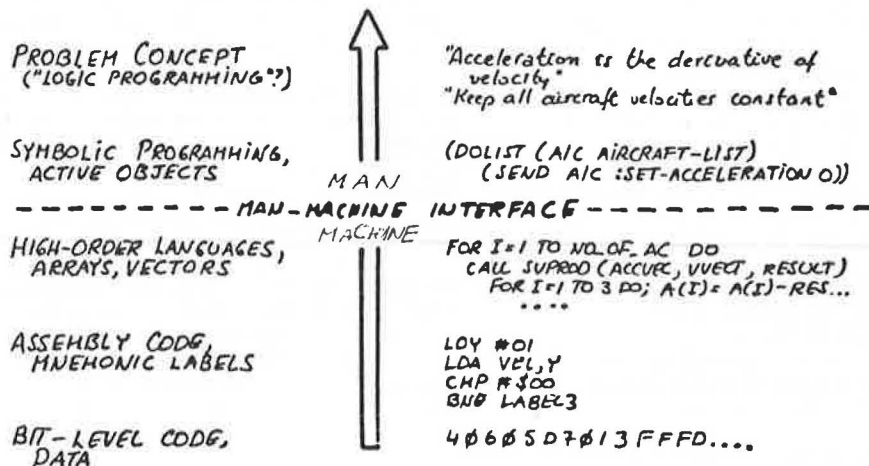
how the entire system operates, and it is becoming more and more difficult to estimate what effect on the entire system the introduction of a new component, such as direct routings, will have.

Another area where the cost of this complexity is quite evident is computer software; it is a well established fact that the cost of developing a software system is not proportional to the size of the system: "two programmers can do in nine months what any of them could do in twelve months" is the popular proverb. A more detailed analysis of the additional costs incurred when a large software project is partitioned in N smaller components is N to the one and one-half power, and this, coupled with the decreasing cost of computer hardware has resulted in a reversal of the relative importance of hardware and software costs. Whereas fifteen years ago hardware costs for a large system were typically ten times larger than software costs, today it is software which is about ten times more expensive than hardware for a typical command and control system.

The differences in programming productivity are tremendous. While the industry standard for fully developed, tested and documented code ranges between 1200 and 2000 lines of code per man-year, project-wide averages of 20000 to 50000 lines are not uncommon in AI projects. In addition to the simple increase in single-programmer productivity, this difference is compounded by the reduction in the number of individual pieces in which a large project must be subdivided in order to meet the required schedule (the N to the one-and-a half power law), with overall differences in software cost of up to 100 to 1, for the same resulting software functionality.

The reason for this difference is actually quite simple. Programming is nothing more than the translation of the original functional specifications of the system to be designed into the simpler elements that can be executed in a computer. In the early days these were individual bits, representing either data or instructions, so that the entire translation process had to be performed by the human programmer. Next came the assembler or machine language which, while operating with the same machine-level elements, at least allowed the programmer to refer to them by names and symbols, rather than by anonymous numbers. The advent of the so-called high-order languages raised the interface to the level of vectors, arrays, strings and passive data structures, and produced what appeared to be a miraculous increase in programming productivity (Figure 10).

Figure 10. Man-machine interface in programming an ATC system.



High-order languages, even in their most complex form such as Ada, are still rooted in the Von Neumann concept of the computer as a sequential executor of instructions. Code and data, for example, are two distinct and unmixable elements, linkable only through the process of compilation. By comparison, symbolic computation removes itself one step further from the details of hardware, and allows truly abstract concepts to be represented and manipulated on a computer. Probably the most spectacular consequence of this increased level of abstraction is that the program itself, or code becomes simply one more abstraction, and thus can be directly manipulated by a program without the compilation or interpretation barrier of high-order languages.

And this is only the beginning. AI research is fast advancing in the direction of declarative programming languages, or rather, programming models, that allow the user to state the functional specifications for a computer system in extremely abstract terms without having to specify, for instance, the sequence in which operations have to be performed to arrive at the desired effect. These languages, while still many years away, may make Lisp look as mechanical and complex as high-order languages look in comparison to Lisp.

It is interesting to observe that while the attempts to build an automatic programming system during the early seventies were dismal failures, the same results are being arrived at by a diametrically opposite route. Instead of a very high level program that transforms any program specification to the detailed instructions that computer hardware requires, we are seeing computer hardware and software that operate at higher and higher levels of abstraction: a bottom-up approach, rather than the top-down approach of the automatic programming concept.

Of course, nothing comes free. This increase in the level of abstraction at which the machine interfaces with the human programmer entails an inevitable increase in the processing power required in hardware. But one should not look at this increase as inefficiency or overhead; in fact, this additional processing is performing an extremely useful function, namely the translation process from abstraction to machine bits and back, of both code and data. Therefore, we will have to learn to accept much higher computer processing requirements as a natural by-product of our increase in complexity. However, the continuing decline in the cost of processing, or, if you wish, the increasing performance of computer hardware will make it more palatable. The important point to consider is that the computer technology, both hardware and software, used today by AI researchers may become the only economical way of implementing very complex software systems in the near future.

Research Simulation Technology

Leaving behind the world of computer software, we find that some of the same problems that plague builders of large software systems also haunt designers of large human systems. Even if the ATC system used no computers at all, the flow of information, and the distribution of decision-making authority makes the system look very much like a gigantic computer, with procedures, rules, regulations, and letters of agreement being its program.

We have long passed the stage where the effects of major changes in procedures or technology can be evaluated effectively by simple analysis: simulation becomes the ultimate evaluation and verification tool. Unfortunately, building and running a

sufficiently good simulation of a very complex system can be extremely costly.

Consider the difference between an aircraft simulator and, for example, the simulation of an advanced ATC controller station of the year 2000. While the basic principles of aerodynamics, structures, propulsion and so on cannot change radically from now to the year 2000, the same cannot be said, at least in principle, of air traffic procedures. There are few physical limitations to what can be displayed on a futuristic controller's screen. So whereas the aircraft simulation can count on a number of essential fixed elements no matter what the configuration of the experiment may be, the same cannot be said of an ATC systems simulation.

The traditional way of designing, implementing, and using large system simulators was this: a detailed specification was drawn of the fixed part of the system, that is, the part that is not expected to change from one experiment to another. Next, the user defined some bounds on the kind of experiments that would be run on the simulator. The simulator designer then would convert the fixed part of the specification to detailed formulations of the core of the simulator, which would include generation of large amounts of data that could be used to feed the expected experiments. Also, the behavior of the core system would be determined, as much as possible, by parameters that could be read from a data file in a simulation initialization time, so that the core could be tailored as much as possible to the particular experiment that was to be run.

The alternative to this traditional approach is to build not a core simulator, and an array of ad hoc extensions for each new experiment to be run, but rather a kit of building blocks with which a customized simulation can be built in a very short period of time. In other words, we not only accept, but actually encourage the notion that a new simulation will have to be built for each new experiment in ATC technology (Figure 11).

The key to this approach is the level of abstraction of these building blocks. Using symbolic programming techniques, it is possible to build blocks such as "VOR", "Aircraft", "Random Aircraft Generation Point", "Airport Runway", "Airway Intersection", "Radar", "Display Screen", and the like. Moreover, there can be many different types of these blocks, not only in terms of their performance parameters - you can do this in Fortran with initialization files - but even in the level of detail being simulated (Figure 12).

For example, the MIT Flight Transportation Laboratory is currently designing a building block kit which will allow the experimenter to intermix three very different levels of simulation at the same time: a Level I, where the smallest geographic unit represented is a control area, say several sectors large, and aircraft dynamics consist only in movements from an area to an adjacent area (Figure 13). At this level of detail, the entire continental United States could be modeled, with some 2000 aircraft, with very little effort required to set up the experiment. A Level II would look into the actual geometry of the airway structure, as well as direct routings, and be able to model individual sectors. At this level of detail, the position of each individual aircraft along an airway or along its direct route would be modeled, but not, for example, the effects of individual radar vectoring. The maximum number of sectors that one would like to model this way is probably ten or fifteen, with a total of one to two hundred aircraft, enough to analyze problems relating to the communications and handoffs between two centers. Finally, Level III of simulation detail would look at individual aircraft

Figure 11. ATC customized simulation building blocks.

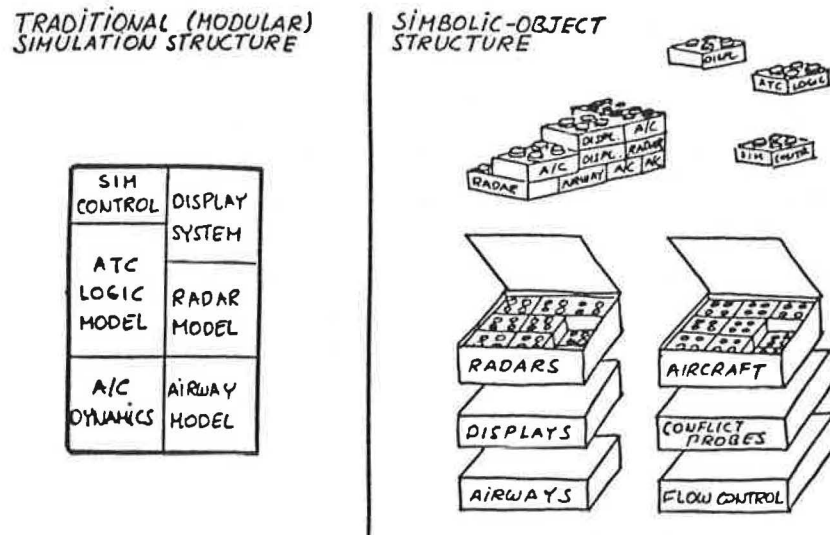


Figure 12. Typical symbolic-object simulation.

OBJECT STRUCTURE

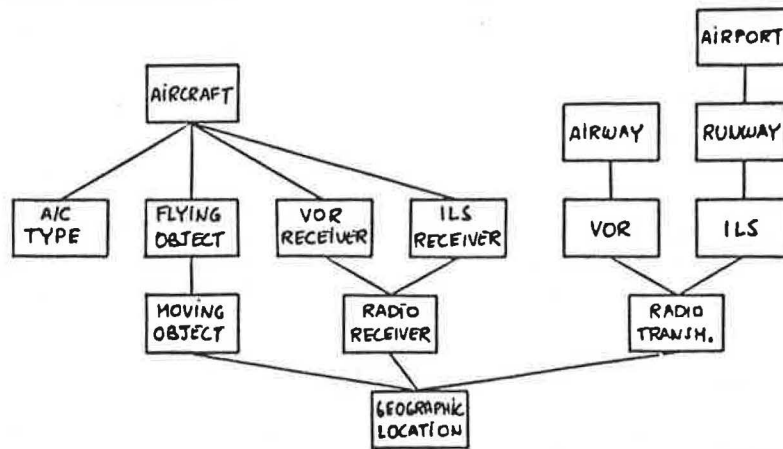
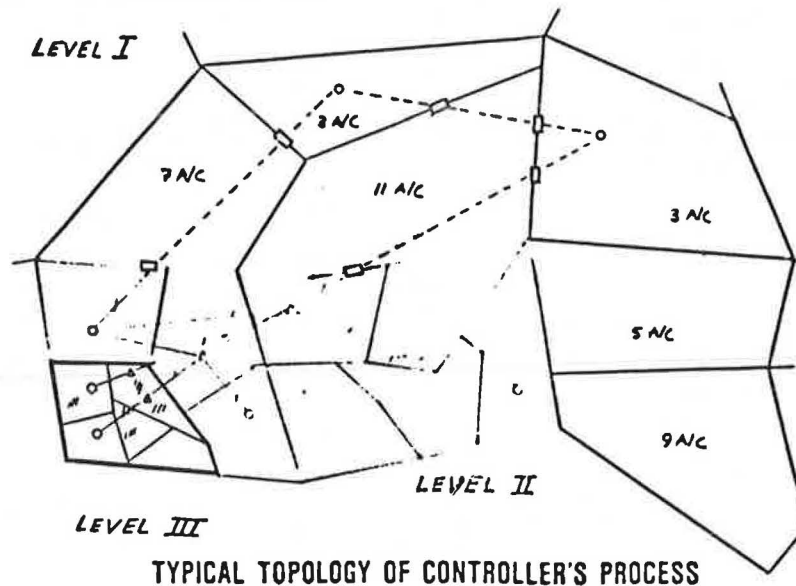


Figure 13. Example of levels of simulation at the same time.



dynamics and the performance of radar sensors, and would be the level of detail at which to look at problems such as simultaneous instrument arrivals to closely-spaced parallel runways, or the sector-to-sector interactions for a maximum of, say, three sectors and thirty or forty aircraft.

This building block kit would then include not only three levels of airspace models and three levels of aircraft models, but also different display format for each level. The important feature of this approach is the possibility, if designed correctly, to run a simulation where the entire country is modeled with Level I elements, except for two centers, which are modeled with Level II elements, and have within these two centers two or three sectors modeled with Level III elements.

Object-oriented and symbolic technology are capable of solving the problem of interfacing these rather dissimilar objects together. Consider a flow control algorithm that wants to know how many aircraft are in a certain area, the smallest Level I unit of airspace. In traditional programming, the programmer would have to know the location of that number in whatever data structure contains that information for a Level I area, but would probably have to write a subroutine to obtain that information from a Level II center, since it would have to add all the aircraft in each of that center's sectors. With object-oriented programming, the burden of providing any information about an object is shifted from the seeker of the information to the supplier of the information.

The technique in question is called message passing; each object in the kit is known to respond to a certain number of requests, or messages. These requests can either ask for information about the object, or ask that the object perform some action that has a side effect, such as displaying a symbol on a screen. All the interactions between objects must be through these publicly advertised messages. Part of the effort required in designing such a simulation is to define what kinds of messages each object should be required to handle.

Once this is decided, though, the task of inter-object communication is enormously reduced; if both Level I areas and Level II centers are required to reply to the message "how many aircraft do you have now", it does not matter to the object requesting the data whether this data is obtained by simply looking it up somewhere, or by laborious computation: it simply is returned in response to the message. If the internal makeup of an object must be modified - say, in response to the requirements of a new experiment - only its way of handling its incoming messages must be modified, whereas in the traditional technology every object that could possibly interact with the modified object would have to be modified as a consequence of this change.

The development of this simulation architecture is the most exciting ATC-related project at the MIT Flight Transportation Laboratory in the last decade. If successful - and there are a number of major technological obstacles still to overcome - it may enable for the first time the testing and evaluation of truly advanced ATC concepts in a sufficiently realistic environment, at reasonable cost.

The concept of building a real-time ATC simulation based on software building blocks as just described has been demonstrated at the Flight Transportation Laboratory, where a full scale Level III simulator using this technique is in daily use. The largest technology risk associated with this simulation is related to its hardware; in addition to the building block software approach described, it is designed around a building block

hardware architecture; the same message-based interaction technique that allows different kinds of objects to interface in a homogeneous manner will also allow these objects, and the functionality they carry, to reside in different processors, with some limitations, so that the exact number of processors available to run the simulation is invisible to the software, although, of course, the resulting performance will be very visible to the user.

This will also allow incremental growth in the capabilities of the simulator, as more processors and display hardware are added without the need for software recoding, but is dependent on very recent, and still untried advances in symbolic computation hardware.

An Expert System for Runway Configuration Management

Curiously, there are fewer opportunities for classical expert systems such as mycin in ATC than one might expect. Indeed, there are few circumstances where accumulated knowledge, as opposed to skill or ability, determines the performance of a control function.

Perhaps one of the most promising short-term applications of classical expert systems may be to the problem of runway configuration management, that is, the selection of what runway configuration to use under changing weather and flow conditions. Complex airports, such as Chicago, or the New York City Metroplex, have hundreds of possible runway and approach configurations. The problem consists in selecting which configuration to use, and, more particularly, selecting when to perform a configuration change. The relative timing of the arrival of a front at the airport terminal area with respect to the peak traffic hour may make a difference as to whether the runway configuration change should be performed in advance, or delayed with respect to the weather-optimum time. Moreover, weather at other airports may affect the normal traffic pattern at an airport so that, for example, a snow storm approaching the Boston area from a westerly direction requires a different runway configuration change strategy than one approaching from the northwest, since the former will hit New York before Boston, therefore causing potential diversion of traffic from the New York City area.

This simultaneous consideration of multiple contradicting factors, some of which may be the result of many years of experience and observation at the station in question, lends itself ideally to mechanization as an expert system. Indeed, the MIT Flight Transportation Laboratory is developing such an expert system, under the code name Tower Chief. This name was selected to bring to mind the notion that the Tower Chief is usually the senior - and therefore the most experienced - controller in that facility, and therefore would be the ideal person to make runway configuration decisions at all times, not just when he is the actual shift supervisor. By capturing his expertise, the expert system would make available to any supervisor having to make such decisions the expertise and accumulated knowledge of the senior person.

Actually, such an expert system would be capable of storing knowledge and associations furnished by a number of individuals, and therefore be of use to the Tower Chief himself, specially in its ability to be comprehensive in examining all the knowledge elements pertinent to the current state of affairs. On the other hand, we dislike the name Tower Chief since, in addition to the concept of

wisdom and experience, it also calls to mind the concept of authority, or responsibility. There is therefore the danger of concluding that such an expert system, by virtue of its superior data base, is able to make superior decisions than a human in this situation. This is clearly not so. In fact, beyond the assurance that the expert system has systematically tested all the knowledge contained in the data base, the greatest benefit that the shift supervisor can derive from the use of Tower Chief is not the final conclusion or recommendation that it may make regarding the runway configuration changes to select, but rather its capability to display the logical process that leads to that conclusion. This display can be used not only to help make a final decision, but also to enrich both the expert system's and the human's knowledge base; therefore, we would have preferred to title this project supervisor's consultant, but it is a little late for this, so we will continue to call it Tower Chief.

Some technical problems must be resolved before rules and knowledge can begin to enter a Tower Chief prototype system. As with all knowledge based systems, expert or not, the work begins with the construction of logic abstractions capable of representing, both to a computer and its user, the elements of knowledge in the particular field. For Tower Chief these may be runway, prevalent winds, primary flow direction, etc. with again, both data and functionality being associated with these abstractions. This is the knowledge engineering phase, and is now under active development for Tower Chief at the Flight Transportation Laboratory.

Simultaneously with the knowledge engineering phase, an expert system systems design must be carried out. This is the design of the process by means of which the abstractions will be entered, searched, activated, processed, and displayed in the operation of the expert system. There are a number of classical methodologies, such as forward chaining, where as many of the rules as may possibly be achieved given the established facts are invoked, until all the rules have been used, and mycin's backwards chaining, where a number of hypotheses are postulated and tested by means of the rules, until as many of them as possible have been weeded out. Other classical techniques address the method of incorporating rules into the knowledge base, requesting specific

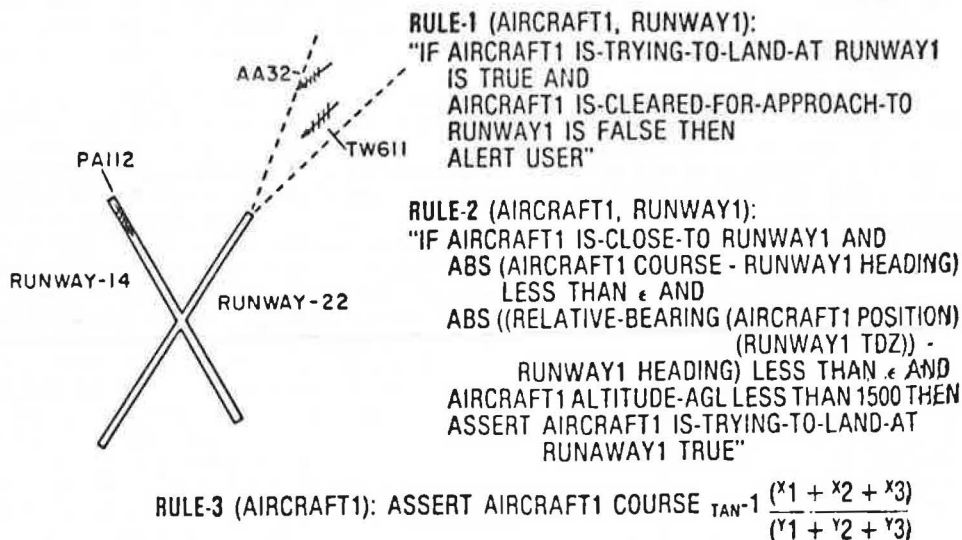
data items as the hypothesis tree is traversed to reduce the number of branches that must be explored. The collection of techniques and the software used to implement them are referred to as expert system cores.

A small but growing industry of pre-fabricated expert system cores offers a large number of more or less off the shelf software systems. These cores consist of a general-purpose structure for representing knowledge, and the inference engine or logic that drives activation of the rules to achieve the final objective. Along with these features, some of these systems also come equipped with fabulous claims about the speed and ease with which useful expert systems can be built around them.

Unfortunately, these claims are usually exaggerated for two reasons: first, because experience has shown that rule-processing procedures are much less universal than previously thought; second, because even if an existing core is adequate to perform the rule processing required in a particular problem, a significant knowledge engineering effort is usually required to case the particular knowledge relevant to the problem in the forms required by the expert system core.

Tower Chief is the second ATC-oriented prototype expert system developed at the Flight Transportation Laboratory. The first, known simply as Rule System One, or RS-1, was only an experimental system in which conventional algorithms could be re-implemented as rules, and was developed to gain familiarity with expert system techniques, and not to demonstrate any useful function (Figure 14). RS-1 showed us, for example, that ATC problems are particularly ill-suited for prefabricated expert system cores. In RS-1, data, or rather assertions about the objects known to the system, arrived in time-sequenced frames, corresponding to entire revolutions of a terminal radar antenna; thus the assertion base, the data base of statements asserted to be true about the objects, was continually evolved. Moreover, rules may refer not only to current assertions, but also to past assertions, or even changes in assertions, as for instance: "If aircraft-1 appears to be on a base leg, and it was previously affirmed to be on final, something is wrong". Among the interesting consequences of the RS-1 work we found that the concept of past, as applied to

Figure 14. ATC experimental expert system RS-1.



computer implementations of knowledge, is more complex than previously thought.

Symbolic computation has taught us that the concept of equality is more complex than the simple equality of numerical values of Fortran. For example, a simple chair and an armchair are clearly not equal, while two identical armchairs are, to a certain degree, equal, although they are two different chairs, two different actions of equality. Similarly, we have two different notions of past. Suppose, for example, a rule which estimates the general direction of an aircraft track; this rule may ask the assertion base for the previous heading of the aircraft in order to compare it with the current heading. But suppose that, during the previous four-second revolution of the antenna, insufficient valid transponder hits were received and a missed reply was declared for that target during that antenna revolution; what should be answered to the question "what is the previous target data?" One possibility is to answer "not known", since there was no reply on that antenna pass. But another is to return the target data for the last antenna pass during which there was valid data. In a way, both are previous data, but the answers may be quite different.

The consequence is, of course, that there are at least two different pasts, one relating to the sequence of known data, independently of the time at which it was asserted, and another relating to a sequence of instants of time. Such a feature was not available in off the shelf cores at the time the RS-1 effort was started.

In addition to this passage-of-time problem, Tower Chief will also be subject to three more time related problems. First, the elements of knowledge that Tower Chief will handle will have themselves a time component, similar, but more complex, than the time related questions asked by mycin.

Second, the goal of this expert system is really a program, or timed sequence of runway configuration changes, so time is one of the components of the answer, as well as of the data used to arrive at the answer; nobody has had any significant experience in designing expert systems that deal with time as one of the parameters of the goal.

Third, and this is a problem faced by all expert systems whose answer is required in real time - the search for answers may be terminated by the time available, rather than by exhaustion of the search, as in mycin, where the time required to arrive at the answer is not really important, as long as it is reasonable. There is little experience about time constrained expert system performance. Indeed, expert systems share with some operation research methods the property that, while monotonic, the rate of improvement of the answer may vary widely with time. In some cases an excellent answer may be arrived at very quickly, with only marginal improvements afterwards. In other cases, the bulk of the solution improvement may only be achieved at the very end of the search, so that an early termination may produce a very unsatisfactory answer. It is not known at this time if the amount of processing required by Tower Chief will be such that time-terminated processing will be required; if it is, its performance may depend on new developments in solution search techniques which guarantee uniform solution improvement with time. As an aside, one of the methods that have been proposed to achieve this uniformity involves the intentional randomization of the search procedure, in a Monte Carlo like process.

Two Simple Applications of Mechanical Vision In ATC

An entire field of research in artificial intelligence is that of visual scene recognition, that is, the processing of raw data from, say, a television camera or other means of converting visual information into bits, with the purpose of identifying objects, positions, three-dimensional shape, and even higher order relationships, such as attachment between objects or their constituent materials.

At first glance there would seem to be no obvious application of this robot vision capability in air traffic control, unless one wished to build a robot tower controller or a robot pilot. Actually there are two very good possibilities, one on the ground, and one in the air.

A useful ground system based on mechanical vision and scene recognition would be a low cost, totally passive LIDAR, or Light-based Radar. Such a system would consist of two, perhaps three television cameras mounted on fast remote-controlled tilting and panning heads, and equipped with fast zoom lenses. Controlled by a computer with visual recognition software, this system could act as a VFR radar in congested small general aviation airports whose traffic density changes from being higher than that of Heathrow during fine VFR conditions, to practically nothing as the weather becomes IFR. Visually scanning for aircraft, this system could present to the local tower controller a plan view display of the aircraft within the airport's traffic area.

In its simplest form this system would periodically scan the horizon surrounding the airport and create a visual map of the fixed features around the cameras: trees, buildings, hills. Some of these features may change periodically, such as the foliage of the trees, but just as in a modern radar's clutter map, they can be immediately recognized by their very, very slow rate of change.

Real scene recognition begins with slow, but really dynamic objects, such as clouds and birds. Clouds have such a characteristic texture, size, and speed that it should be trivial to separate them from aircraft targets. How can this system distinguish a bird at five hundred meters from a light airplane at five kilometers? One possibility is radial velocity: the bird at five hundred meters can move faster across the camera's field of view than a similar-sized aircraft target.

In addition to acquiring all this information the system has some unusual potential for presenting the information to the controller. For instance, instead of the usual bars we are accustomed to in high-intensity radar displays, we could have a small picture of the actual aircraft, in color, obtained by the system's cameras, and processed by the computer so that at any time in that aircraft's flight that picture should look just like what the controller should see with his binoculars were he to look for that aircraft.

Now we have a system that not only is more sensitive than a human controller in detecting and processing visual targets, but may even provide him with additional information about the target that a conventional radar certainly could not. And being only software, it is a cheap system to produce in large numbers, so as to offset its probably large software development cost.

An Abstract Concept of Flow Control

The next concept in air traffic control that we will consider is not a gadget like Tower Chief or the visual radar, but actually a concept. It is related to artificial intelligence because it is the result of building abstract representations of knowledge, capable of being implemented on a computer, but also independent of any computer implementation. Indeed, they could very well be implemented as procedures, with humans performing all the information handling and decision making.

These abstractions are models of how a flow of aircraft could be regulated by control elements that interact only with their neighbors; at what level this flow control would be carried out is immaterial. The test prototype we have implemented in our computer at the MIT Flight Transportation Laboratory operates at the tactical, terminal area level; but the concept could equally well be implemented at the central flow control level. It is far too early to decide whether these abstractions would be of any use in a future ATC environment or not. Our purpose in presenting this work is to show a different kind of product of artificial intelligence thinking in air traffic control research.

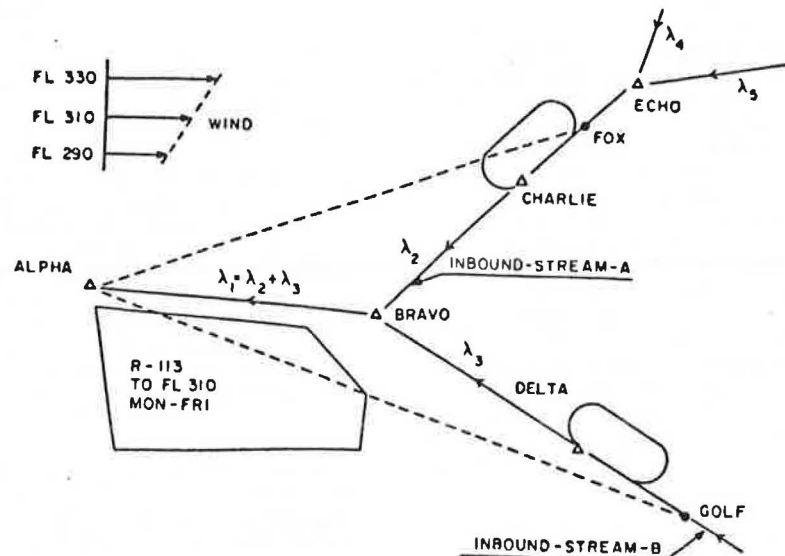
The development of these abstractions began as an attempt to state, in knowledge representation terms, the classical time based metered merge control problem, which can be simply stated as follows: merge two streams of incoming aircraft with random interarrival times to form a single output stream with uniform aircraft separation (Figure 15). This is usually performed by assuming

path arrives at the sink, thus creating a binary converging tree.

Flow control is only one of the tasks to be performed by the ATC system. Indeed, separation assurance is by far more important, in the short term, than orderly flow of traffic. For a number of technical, operational and historical reasons responsibility for separation assurance requires that ATC functions be divided into small sectors under the authority of a single human controller, as opposed to a central control authority. This federated approach, which is optimal for separation assurance and responsibility accounting, conflicts with the centralized approach of traditional flow control algorithms. In a federated approach, each control element, that is, each controller, interacts mainly with his immediate neighbors, rather than with a centralized arbitrator. Handoffs are initiated, accepted, or rejected on a one-to-one basis, and not as a result of the decision-making of a central authority.

For this reason, flow control procedures are difficult to implement and interface with in a federated ATC environment. It would be interesting to develop and test a flow control approach that operated as a number of independent elements which interact only among neighbors, in the same way tactical ATC elements do. This approach, developed at MIT's Flight Transportation Laboratory, is called the Metered Merge Control Element, or MMCE, concept. Again, it is too early to decide if this approach has any merit, and is presented here only to illustrate the kind of product that can be developed using the AI approach to computers.

Figure 15. Stream merge problem of an ATC system.



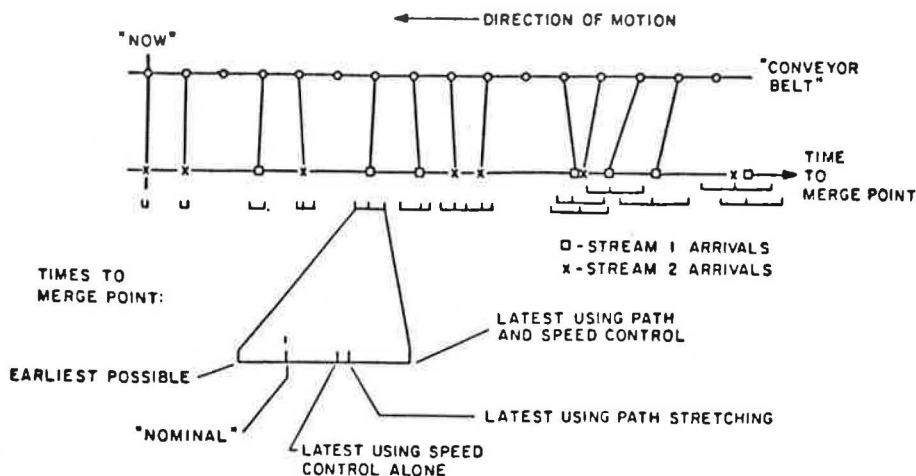
an ideal conveyor belt of time slots, and by assigning aircraft from both incoming streams to a slot in the conveyor belt, and then maneuvering the aircraft - in the time dimension, hence the name time-based merge - to their assigned slot. This maneuvering in time may, of course, require complex maneuvering in two-dimensional space. (Figure 16).

The picture is a little more complicated when not two, but a number of incoming streams must merge into a single one. Each route begins at one of the sources; the routes merge in pairs, until a single

Conceptually the MMCE consists of the following elements: two entry gates, a single exit gate, and two nominal transit times from each of the entry gates to the exit gate. While it is useful to visualize the MMCE as a Y-shaped merging path, the geometry of the MMCE is irrelevant to the concept, except inasmuch as the transit times are related to the size and shape of the paths (Figure 17).

Connected to the exit gate, each MMCE has a downstream correspondent which can be either another MMCE or, in the case of the last MMCE of the tree,

Figure 16. Time-based metered merge-base model of an ATC system.



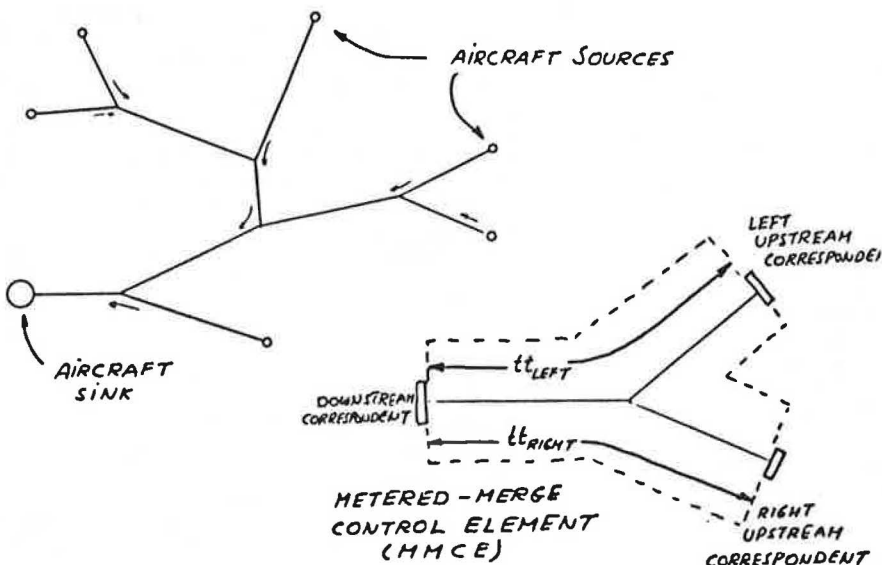
the aircraft sink. Connected to each entry gate is an upstream correspondent, either another MMCE or, in the case of the first MMCE in the tree, the aircraft sources. Sources, MMCEs, and the sink comprise the entire metered merge flow control abstraction. This abstraction is independent of the scale of the problem: it could be the terminal area around an airport, with the sources being the feeder fixes, and the arrival runway; or it could be an enroute problem, with the sources being originating airports and the sink the destination airport's terminal area. In any case the operation of the abstraction is as follows.

When an aircraft appears at a source, its existence is immediately made known to the MMCE immediately downstream of this source. In the absence of any flow control, that aircraft would reach the MMCE's exit gate at a time which is equal to the time at which the aircraft appeared, plus the nominal transit time through the MMCE's right or left branch, as appropriate. Therefore, that aircraft should appear at the entry gate of the

current MMCE's downstream correspondent at that time. This information is passed on by the current MMCE to that downstream correspondent, who then performs the equivalent computation and passing of the information to its downstream correspondent. Finally, the ultimate downstream correspondent, the sink, is told that an aircraft would nominally reach it at a time equal to the current time plus the sum of the nominal times through all the appropriate branches of all the intervening MMCEs.

At this point the sink has to perform its own decision-making, which may include previously received notifications of incoming aircraft. The result of this decision-making is a desired arrival time for that aircraft, which may or may not be the nominal arrival time. This information must then be made known to all the MMCEs that the aircraft must traverse to get there. Since the sink only has communications with the last MMCE, this element receives the desired arrival time at the sink for that aircraft.

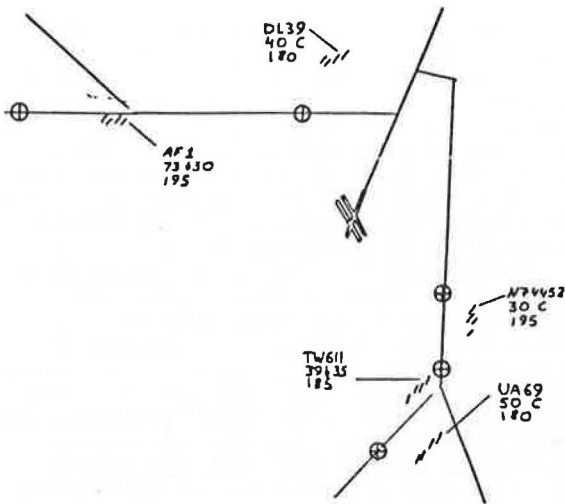
Figure 17. MMCE flow control procedure.



The process used to propagate the nominal arrival time downstream is reversed, in that the MMCE's nominal transit times are subtracted from the desired arrival times before submission to the next upstream correspondent. Finally, the first MMCE (the one currently responsible for that aircraft) receives the time at which the aircraft should leave its exit gate so that, flying at the nominal speed through the remaining MMCEs, it would arrive at the sink at the time that the sink desires it.

Actually, this upstream propagation of information is not as symmetric with the downstream propagation as we described it. Indeed, when propagating the information upstream, each MMCE has to send it to its right or left upstream correspondent, as appropriate, a decision-making not required when propagating the information downstream.

Figure 18. A radar controller's display of the MMCE concept.



In the Flight Transportation Laboratory implementation, the MMCE concept is used to drive a Radar Controller's display. In this display, the MMCEs are made to correspond to actual converging ATC paths. In this way each controller is given an indication as to how early or late the aircraft is with respect to the ultimate sink's wishes (Figure 18). This display concept, or conveyor belt had been proposed before, although it has never been mechanized, even experimentally, beyond the final approach path. It is clear that this kind of display could be constructed without the need for MMCEs, downstream ripples, upstream ripples, and the like.

While the development of this abstraction does not imply its computer mechanization - it could be mechanized, for example, as a series of controller-to-controller interactions - we are able to simulate them, and therefore perform experiments with them, using software objects in Lisp in the MIT Flight Transportation Laboratory's symbolic ATC simulator. A number of instances of sources, sinks, and MMCEs can be created, linked, and positioned interactively. Image objects corresponding to the MMCEs' nominal paths and the previously described slots are created and manipulated as easily as numbers of a calculator or characters on a word processing system.

A Distant Dream: The Controller's Assistant

Finally, and as an example of a truly long-term possible application of AI technology to air traffic control, we would like to propose the idea of a personalized controller's assistant. This device would consist of a knowledge base made up of four parts: a general part reflecting the generic kind of controller know-how that would be reflected, for example, in the Controller's Handbook, or in training material; a second part, at a higher priority level than the first, would include position-dependent knowledge, such as the route and airway structure pertinent to that facility, letters of agreement between facilities, and the like; the third part would include the daily weather, notam and similar information, while the last part would be made up of the individual controller's performances and personal techniques.

Exactly what functions such a system could perform is not very clear at this time; one possibility is to act as a dummy of the controller, that is, display for his benefit what control actions the clone would take. By periodically observing that dummy controller the human controller could detect his own blunders, especially missed control actions, early enough to take effective corrective action.

If such a feature is to be a real help, rather than an additional burden, it is likely that the display of such dummy directives would have to be at a rather high level of abstraction. For example, rather than the clone displaying the command "TW611 turn right heading 220", to which the human controller may think "Why is he doing that?", the display should read something like "I would like to send TW611 west to make him a little late on his turn to final, or else he is going to be too close to that heavy ahead of him".

The key characteristic of such a system would be its personalization capabilities: personalization with respect to the position being assisted, the current weather, navaid and traffic information, and most important, the individual controller. The controller's individual knowledge base could, presumably, be part of his personal equipment for the duration of his career. If we may be allowed to dream for a moment, we can imagine the days when the controller, upon taking over a position from the previous person, would insert his or her magnetically-coded ID card on the console, to indicate to the system that his personal knowledge base is to be used. This knowledge base would replace the previous controller's personal set of rules, and interact with the facility's rule set, as well as the knowledge of the day which was entered by the same shift supervisor that briefed the incoming controller on the day's situation. Thus, there is a one-to-one correspondence between one element of the knowledge base and the controller's basic training, knowledge of the local environment, personal controlling style, and knowledge of the current traffic, weather and facilities situation.

What the form of this knowledge would have will have to wait for the appropriate knowledge engineering to be performed. We can only venture to suggest that it will involve abstract concepts both intuitively obvious to the human and manipulable by the computer, similar to the geographic location and intersection objects of our symbolic ATC simulator. The collection of abstractions, which would include both objects and actions, would in effect create a rich, unambiguous and intuitively attractive language which could be useful not only for humans

and machines to communicate, but even for human-to-human communications, much in the same way that the language Lisp is today used not only to program, but also to describe logical process in scientific publications.

The same uncertainty about how knowledge would be represented in such a system also applies to what kind of inference engine or rule-processing logic it should have. To begin with, several simultaneous goals may be required, and these goals may be more complex than the simple diagnosis-seeking of the mycin or the runway configuration change program of Tower Chief. Certainly today's expert system technology is not sufficient to achieve this functionality.

A Final Caveat

As ambiguous as all these promises are, they appear to hold a lot of promise for performance that we know cannot be achieved by today's computational techniques. It is also fair, however, to point out some potential problems, principally that of software verification and validation. A significant part of the cost of today's software is associated with achieving a satisfactory degree of confidence that the behavior of the software in a system as critical as the air traffic control system will be correct. The cost of this validation increases, of course as the complexity of the desired behavior increases; the problem with the personalized algorithm just described is not only that its behavior is radically more complex than that of any software ever used in ATC automation, but that its behavior cannot, by definition, be completely known and specified *a priori*.

This problem is not unique to the controller clone idea. Indeed, imprecise *a priori* knowledge of the behavior of the system seems to be a fundamental feature of most AI-oriented devices. What is the solution, then? Abandon this class of software as untestable? Abandon the notion that we can validate the software to be used in air traffic control? Both extremes seem unjustified. A new concept of software reliability must be developed, a concept more sophisticated than just the idea that it meets the prescribed specifications. For example, the notion of a software defect could be organized in various categories. Category one would be a software defect that simply and catastrophically causes the entire system to stop functioning. Probably we can devise methods for testing against that type of bug, no matter how complex the software and the expert system rules become.

A second category of bug would involve a less than perfect solution to a problem, such as not finding a solution to a specific problem. In this case it is clear to the user that the system is not functioning properly in that particular instance, but in all likelihood it will function properly on the next problem. This we would categorize more as a performance limitation of the technology than a real bug, and the difficulty here is that we cannot predict, therefore specify, what the performance of an AI-based product will or should be. We will have to learn to live with this type of software deficiencies.

A final and perhaps the most devastating type of bug would be one which involves a definite malfunction whose effects, however, are not immediately apparent to the user. Such a defect, for instance, would involve making decisions about an aircraft on final approach using data pertaining to another aircraft on final approach. Since the aircraft are

in similar situations, the control actions suggested may look reasonable for the aircraft in question, even though they were based on information about the wrong aircraft.

How would one be protected from such defects? Perhaps a way out would be to implement software redundancy in the same way as today we implement hardware redundancy to protect against hardware malfunctions. The notion of redundant software is, however, very different from that of hardware redundancy. While two identical ILS receivers do offer a significant amount of protection against receiver failure, two copies of the same program offer no protection against a programming bug. Indeed, programs, or, in the case of AI products, the rules or other language data that determine the behavior of the program, must be independently developed, implemented and tested, to offer any degree of protection.

We are at the very infancy of software redundancy. With today's programming technology, exhaustive validation and verification are cheaper than redundant software development. With the next generation software technology and systems complexity it is possible that redundant software development may be the cheapest way, or may be the only way, of gaining confidence in critical software.

To summarize, artificial intelligence is a source of extremely powerful tools and ideas, and in particular, it opens up a new viewpoint on the use of computers for any kind of applications. One should not expect miracles from this technology in the near future, except perhaps in the areas of software productivity and simulation technology. We would like to compare the state of AI today with that of the transistor in the late 1950s. At that time there was little a transistor could do that could not be done with vacuum tubes. Admittedly, the transistor was a little smaller and used a little less power than a vacuum tube, but in many respects, such as frequency response, it was in fact inferior. Yet today it would be a little hard for us to walk around with a wrist watch that computes inverse trigonometric functions if it were built with vacuum tubes, even if we had a long enough extension cord. So, sometime between 1960 and 1985 the mere quantitative advantage that the transistor had over the vacuum tube was transformed into an insurmountable qualitative advantage. Perhaps we will wake up some day in the year 2000 and realize that sometime between 1985 and 2000 the mere quantitative differences between artificial intelligence and conventional use of computers was also transformed.

Acknowledgments

Research described in this paper has been sponsored by the Federal Aviation Administration and the Transportation Systems Center of the U.S. Department of Transportation. The authors wish to recognize the important contributions of the following individuals: Professor Robert Simpson, Mr. Lyman Hazelton and Mr. Jim Butler of the MIT Flight Transportation Laboratory; Mr. John Fabry of the FAA Technical Center, Atlantic City; Mr. Richard Wright of the MIT Transportation Systems Center, Cambridge, Massachusetts; and Messrs. Paul Neumann and Steve Alvania of the FAA Headquarters, Washington, D.C.