RL-TR-91-275
In-House Report
December 1991

A VISUAL PROGRAMMING METHODOLOGY FOR TACTICAL AIRCREW SCHEDULING AND OTHER APPLICATIONS

Douglas E. Dyer, Capt, USAF

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

Rome Laboratory
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RL-TR-91-275 has been reviewed and is approved for publication.

APPROVED: Samuel A. DiNitto, Jr.

SAMUEL A. DINITTO, JR., Chief
C3C Software Technology Division

FOR THE COMMANDER: Raymond P. Urtz Jr.

RAYMOND P. URTZ, JR.
Technical Director
Command, Control & Communications Directorate

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify RL (C3CA), Griffiss AFB NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1991	In-House Sep 87 - Apr 90

4. TITLE AND SUBTITLE
A VISUAL PROGRAMMING METHODOLOGY FOR TACTICAL AIRCREW SCHEDULING AND OTHER APPLICATIONS

5. FUNDING NUMBERS
PE - 62702F
PR - 5581
TA - 27
WU - 40

6. AUTHOR(S)
Douglas E. Dyer, Capt, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Rome Laboratory (C3CA)
Griffiss AFB NY 13441-5700

8. PERFORMING ORGANIZATION REPORT NUMBER
RL-TR-91-275

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Rome Laboratory (C3CA)
Griffiss AFB NY 13441-5700

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
Rome Laboratory Project Engineer: Douglas E. Dyer, Capt, USAF/C3CA/(315) 330-3528

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This thesis describes the aircrew scheduling problem faced by US Air Force units flying A-10 aircraft and a visual programming methodology used to automate A-10 aircrew scheduling. Database, scheduling, and programming technologies are discussed in the context of automated aircrew scheduling. The visual programming methodology developed is based on Microsoft Excel, a commercial spreadsheet with database functionality, and is unique because it focuses on the use of Excel as a general-purpose programming language. Using Excel, an A-10 aircrew scheduler was constructed with greedy heuristics which schedule based on priority, event requirements, and currencies subject to pilot and resource availability. Three other applications were developed using the methodology described, and, from the programming experience to date, recommendations for improvements are made.

14. SUBJECT TERMS
Planning, Resource Scheduling, Visual Programming

15. NUMBER OF PAGES
162

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	U/L

ABSTRACT

This thesis describes the aircrew scheduling problem faced by U.S. Air Force units flying A-10 aircraft and a visual programming methodology used to automate A-10 aircrew scheduling. Database, scheduling, and programming technologies are discussed in the context of automated aircrew scheduling. The visual programming methodology developed is based on Microsoft Excel, a commercial spreadsheet with database functionality, and is unique because it focuses on the use of Excel as a general-purpose programming language. Using Excel, an A-10 aircrew scheduler was constructed with greedy heuristics which schedule based on priority, event requirements, and currencies subject to pilot and resource availability. Three other applications were developed using the methodology described, and, from the programming experience to date, recommendations for improvements are made.

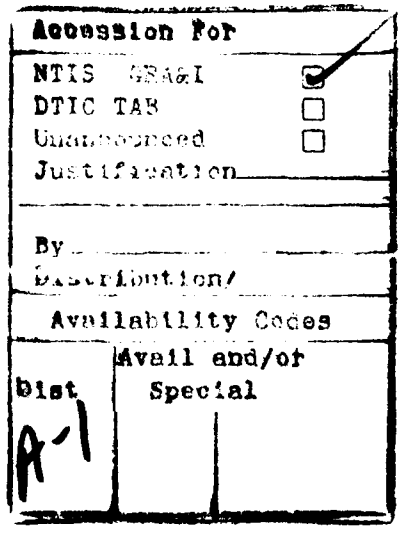

TABLE OF CONTENTS

Page

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

The Rome Air Development Center (RADC) is a large Air Force laboratory that does exploratory development of Air Force command, control, communications, and intelligence (C3I) systems. In 1987, RADC developed a knowledge-based aircrew scheduler to meet the needs of single-seat aircraft unit continuation training. (See APPENDIX A for a system description.) Unfortunately, the 1987 prototype was developed in a LISP environment under a commercial expert system shell called Knowledge Engineering Environment (KEE). The cost of the required LISP machine and KEE prohibited direct installation of the prototype in operational units. Furthermore, the complexity of the code comprising the 1987 prototype made modification an unpleasant prospect. Because each operational unit schedules a little differently, the ability to modify the scheduling system is an important consideration. In addition, the Air Force faces many different types of scheduling problems other than single-seat aircrew scheduling. Some examples of these other problems are scheduling of operational missions, multi-person aircrews, air refueling, and aircraft maintenance. It would be nice if there were some simple programming methodology for developing software solutions for these problems as well. During 1989 and 1990, the 1987 KEE aircrew scheduler was ported to the Apple Macintosh running in Microsoft Excel. (See CHAPTERs 4 and 5 and the APPENDIX). In the process, a new methodology for programming generic scheduling systems was developed. This thesis describes the new aircrew scheduling prototype and the methodology used in the context of other relevant research.

The following chapter describes the specific scheduling problem which must be solved by Tactical Air Command (TAC) units flying single-seat aircraft to complete required

continuation training. Requirements and constraints for A-10 continuation training are described. Flying units are described organizationally because responsibilities and information pathways are important to the problem solution. The manual process for scheduling is described along with the current database support tool. The goals of scheduling are identified and an automated information system for scheduling is proposed.

The third chapter discusses current technology applicable to scheduling and programming in general. A scheduling system typically involves a moderate amount of data, so database technology is addressed. Specifically covered are shortcomings of traditional database models and efforts to bridge them. Scheduling algorithms from operations research and artificial intelligence are discussed. Different programming languages, environments, and methodologies are described for implementing an aircrew scheduler. Visual programming, a new methodology based on the computers increasing ability to handle graphics, is presented as a new way to reduce the complexity of programs and programming.

In CHAPTER 4, a visual programming methodology based on Microsoft Excel is introduced. Using visually explicit data structures and animated programs, the representational complexity of programs is reduced, making development and debugging simpler. Software engineering concerns which arise from representational complexity such as modifiability and verification and validation are treated. A visual programming methodology also helps make the code easier to change and believe in. The graphical programming possible in Excel is described. The use of other attributes inherent in Excel and the Macintosh are discussed. The high-level database functionality of Excel is credited with simplifying scheduler applications. Windowing, pull down menus, and mouse input are also given their due.

CHAPTER 5 is a system description of the new Excel aircrew scheduler prototype. Data structures and data representation are described in detail. The algorithms and their operation are discussed. User functionality is described. Shortcomings of the current prototype are identified.

In CHAPTER 6, the use of Excel for other applications is discussed. There are a number of difficulties associated with the use of Excel which could be reduced if suggested improvements were implemented. The way data is referenced in Excel should be modified to reduce inconsistencies. Excel's macro programming was designed to provide analysis functionality, not as a complete programming language. The macro language needs to be formalized or a complete language like LISP or Pascal should replace it. Finally, Excel would serve better with additional means for data abstraction, including the ability to support the object-oriented paradigm of programming.

CHAPTER 7 summarizes the thesis and indicates the direction of on-going and planned research.

CHAPTER 2
PROBLEM DESCRIPTION

To fulfill its mission, the Air Force requires highly trained pilots. After undergraduate pilot training in a particular aircraft, pilots move to their assigned squadron and enter into continuation training which continues throughout their flying career. Continuation training is designed to make pilots combat capable and to keep them that way by frequently retesting critical skills. In addition, continuation training is used to upgrade pilot qualification, a certification of enhanced ability and greater experience. Flying skills are supplemented by a variety of ground training classes. Ground training includes simulator time, bail-out practice, and survival training.

Pilots have additional duties (duties not including flying; DNIF) which impact their availability for training. Staff jobs, such as Squadron Scheduling Officer or Training Officer, can detract significantly from training opportunities. Sickness, medical and dental appointments, and immunizations often require pilots to be suspended from flying for physiological reasons. Daily tasking assignments like Squadron Safety Officer also preclude flying. Currency in all ground training events is a prerequisite for flying and can limit the pool of pilots available to fly. For safety, pilots are not allowed to fly longer than 12 hours (length of flying day concept). Furthermore, pilots must be given 12 hours of crew rest after flying before they fly again.

The rules governing pertinent aspects of flying and flight training are spelled out in detail in specific Air Force regulations. A-10 flight training requirements are listed in Tactical Air Command Regulation TACR 51-50 Volume II. The hard constraints imposed by TACR 51-50 make aircrew scheduling for A-10 continuation training a very difficult

process. For example, there are 17 different training events required and each of them must be completed a certain number of times by each pilot during the six-month training term. Each event has an associated currency period which reflects the frequency of training. For example, for the landing event, the currency is thirty days. If a pilot fails to land for thirty days, that pilot is out of currency for landing and must fly with an instructor pilot to regain landing currency. "Instructor pilot" is a qualification rating which may be achieved through the upgrade process. A-10 pilots have one of five different qualifications: mission qualification training (an initial qualification given after completion of undergraduate pilot training), mission qualified, two-ship flight leader, four-ship flight leader, and instructor pilot. To upgrade, a pilot must complete a certain number of different training events satisfactorily and must normally fly with an instructor pilot for evaluation purposes. In continuation training, "flying with an instructor pilot" implies that an instructor pilot flies in a separate aircraft but observes performance and issues instructions and corrections orders as needed.

In addition to the hard constraints listed above, there are a number of soft constraints (preferences) which vary with the situation and can make scheduling objectives nebulous. Pilots have preferences about when they take leave, who they fly with, and when they perform training and other duties. Pilots often negotiate with the scheduling officer to attain these goals. Squadron supervisors also have special desires, such as the desire to improve a particular pilot's qualification. However, supervisors usually dictate their needs to the scheduling officer, rather than negotiate with them. The primary goal of the scheduling officer is to attain resources necessary to give all pilots ample training opportunities, build partial schedules using these resources, and fill all partial schedules with the most appropriate pilots.[1] Because of the differing goals of different elements in

[1]Scheduling tasks are done in the order implied by the last sentence; relative difficulty of tasks increases also in the order implied.

the organization, it is very difficult to arrive at a metric of schedule goodness that is acceptable to all.

Operational Air Force units take flight training very seriously because of its impact on combat capability and aircrew safety. It is a primary measure of unit mission effectiveness and reflects directly on the leadership of the commander. Therefore, commanders normally structure their organization to achieve required training and improve the overall ability of their pilots. Operational A-10 units (wings) are often subdivided into squadrons. Training officers report training progress to supervisors at each level and use scheduling officers on staff to ensure that all pilots receive sufficient sorties to complete training. Training officers, scheduling officers, and others are normally flying pilots who must also complete continuation training. That is, their staff functions are additional duties assigned which generally do not preempt the primary job of flying. Because pilots can ill-afford additional work, automation of the scheduling process is highly desirable.

The problem of scheduling aircrews to single seat aircraft continuation training is similar to the classical operations research problem of choosing the best way to allocate resources to competing activities amidst constraints. Using this view, each pilot represents an activity, or job, which requires resources to complete. Resources include aircraft, munitions, gunnery ranges, and instructor pilots. (Note: instructor pilots are both activities and resources).

In current practice, the resources (except instructor pilots) are typically allocated based on availability and compiled in the form of a partial schedule based on the training needs of the squadron. (See Figure 1.) After a partial schedule has been developed, with aircraft, configurations, and ranges filled in, it is completed by assigning pilots to each sortie. This two-step process is used to reduce scheduling complexity and allow the

214 356TFS MON

LINE	TOT	LAND	MSN	PILOT	CONFIG	RANGE	REMARKS
601	0800	1000	WD		B6I	P	
602	0800	1000	WD		B6I	P	
603	0830	1015	WD		B6IMP	P	
604	0830	1015	WD		B6IMP	P	
605	0830	1015	WD		B6IMP	P	
606	0830	1015	WD		B6IMP	P	
607	1200	1330	ACBT		J	A	
608	1200	1330	ACBT		J	A	

Figure 1. A Partial Daily Schedule for A-10 Continuation Training

scheduling officer to use knowledge available at the time. For example, resources must typically be reserved before pilots have to be placed on the schedule. Squadrons which

must negotiate with each other and support organizations for aircraft, munitions and other external stores, and weapons ranges about a month before they will be used. The scheduling officer normally has only general knowledge about training needs that far in advance. On the other hand, pilots must be scheduled to fly only one week in advance. The scheduling officer therefore can make a partial schedule consisting of sorties first, then complete the schedule at a later time by assigning pilots who need the scheduled training missions the most . Events often preclude a pilot from flying (i.e., assignment to an exercise) or increase a pilots requirements for a mission (i.e., failure to accomplish a previous training event satisfactorily), so it is effective to schedule in this manner.

Scheduling is a dynamic process. Once a schedule has been completed, it must be changed if any assumption used to make it is invalidated. Aircraft may fail or weather may preempt a planned mission set. Pilots become ill, require emergency leave, or fail to complete a previous event requirement. The squadron supervisor may re-order preferences in light or in lieu of changes. These types of changes force the scheduling officer to modify the schedule from four to eight times before flying it.[2] However, schedule consistency is also desirable because pilots must have time to plan for missions. Therefore, the scheduling officer often posts a suboptimal schedule with a few changes, rather than a completely reordered, optimal schedule.

To add to the complexity, day-to-day scheduling is only the default case. Operational units frequently deploy to joint or service exercises far from their assigned base. Also, special missions are occasionally required to demonstrate unit capability in non-standard tasks, aid in recruiting, or perform a ceremonial or public relations flight. The scheduling officer must arrange for pilots and flight resources for these events as well

---

[2]Data from one squadron.

as for normal (local) training. These additional activities often require coordination between units for needed support such as airfield support, air refueling, etc. The primary communication medium between units is the telephone, implying that requests may be missed and support may fail to materialize when needed.[3]

Pertinent pilot data are currently tracked in a database (AFORMS) managed by the centralized computing facility on each Air Force base. The AFORMS database tracks pilot event completions, currencies, and qualifications and generates reports of the data which are distributed weekly. Squadrons update the database using optically-scanned forms or an interactive input routine through a terminal.

From a squadron scheduler's viewpoint, the AFORMS database is not adequate because it does not store all the data, outputs the data infrequently and inflexibly, and is difficult to get the data into. AFORMS does not track all required data, such as availability, staff assignments, and event completion in preparation for qualification or rating change. Therefore, the scheduling officer must remember or record this additional data using pencil and paper or grease board. The AFORMS data is not readily available in real-time. Therefore, schedule inputs and, ultimately, schedules are based on old data. In addition, when the AFORMS data is delivered, it is in the form of a "core dump" resulting in a 20-50 page printout. Some reports in the printout are useful for determining priority pilots for different events, but user requested reports are not possible. Nonstandard reports would probably be of questionable value anyway, given the time frame of data delivery, but real-time manipulation of the data would be valuable in determining who should be scheduled or which pilots can feasibly be flown in a particular slot. Scheduling officers usually present data to squadron or unit supervisors (e.g., the Wing Director of Operations (DO) at standup

[3]Strategic Air Command now uses a system called MASMS to automate reservations for low-level tracks for B-52 training.

briefings. Slides used in these briefings contain data from AFORMS and could conceivably be generated automatically, but the AFORMS is too inflexible to support this function currently. AFORMS cannot present data graphically, a function which would aid analysis of training status and point out the need for resources of different types. In addition, if automated scheduling were integrated with the database, schedules could be generated and printed automatically, rather than being generated by hand from hard copy data and then typed by a stenographer. Finally, AFORMS has no intelligence or friendliness with regard to terminal data entry. The data input routine remembers nothing, cannot suggest inputs, and lacks even menus or mouse support. Instead, each piece of data must be painstakingly typed in by hand.

Clearly, the current level of automation (AFORMS database) does not meet the scheduling needs of operational flying units. Nearly all scheduling officers use a grease board or pencil and paper to record additional data and current scheduling status, despite the availability of automation equipment. The reasons for depending on the grease board include a distrust of automation, the lack of powerful automated tools for data handling and analysis, and significantly, the need for data visibility between pilots, scheduling officers, and supervisors.

Scheduling officers have different methods of generating schedules using the data available to them. Although some scheduling officers have formal training in operations research, nearly all use heuristics and "common sense" to develop satisficing[4] schedules. Human schedulers are able to determine relatively good heuristics and apply them, so long as they have the data and the patience to manipulate it to fit the preconditions on the heuristics. Unfortunately, human memory is volatile and human patience has its limits. In

[4] near optimal; satisfying

practice, human schedulers using heuristics constantly find themselves asking for data manipulations to determine an appropriate pilot, for example, *"Who is at least a two-ship flight lead, current in night air-to-air refueling, has not flown since 3 A.M. on Tuesday and needs a night landing the most?"* In addition, its easy for human schedulers to forget one or more hard or soft constraints during the scheduling process, resulting in an inappropriate or disgruntled pilot. There are few scheduling officers who have not generated a very bad schedule, despite knowing reasonable scheduling heuristics.

As mentioned above, schedule optimality is desired, but "optimal" means different things to different people. Fortunately, ample training resources generally exist for completing minimum acceptable training requirements. Ordinarily, in day-to-day operations, it is more important to arrive at a schedule which satisfies requirements, rather than one that is optimal (in whatever sense). In context of a longer time frame, unit commanders strive to maximize training, but also emphasize a host of other people- and mission-oriented performance metrics. Training can be measured in terms of the percentage of combat capable pilots, number of upgrades, and currency delinquency rates. Commanders protect their people and therefore give attention to things like safety and workload distribution. Commanders are tasked with enhancing unit performance which is measured in terms of rapid generation capability, weapons delivery and simulated air combat scores, and other competence and effectiveness metrics. In short, schedule optimization is the global goal, but the objective function for aircrew scheduling is a very complicated equation.

Given the nature of the aircrew scheduling task and the level of automation available, the scheduling officer does an admirable job of producing aircrew schedules. However, the current process used is time-consuming, tedious, and error-prone, despite the professional dedication of the scheduling officer. Current technology exists which

could improve the scheduling process dramatically. It consists of an integrated information system based on a set of automated tools which flexibly transfers and presents data in real-time and assists in the schedule generation. Automatic scheduling is included in the tool set by employing heuristics (or another type of solution method) to generate "good" schedules which do not violate constraints. The tool set adapts to changes inherent in the domain and still generates robust, rather constant schedules. The tool set supports resource and training analysis for additional decision making. The information system is designed to allow for rapid, lossless data flow between unit supervisors, local and distant support elements, and the scheduling officer, if not the pilots as well. That the information system described is possible with current technology is proved by the existence of one example: the system described in CHAPTER 5.

CHAPTER 3
COMPUTER TOOLS AND METHODOLOGIES FOR AUTOMATIC AIRCREW SCHEDULING

Simply put, creating an effective schedule depends on having some generic method of scheduling and applying the method to some specific data set which adequately describes the world.[5] Because data is the more fundamental problem for current tactical aircrew scheduling, database technology will be discussed first and scheduling procedures will follow. Then, because a program must be written to automate the scheduling process, programming methodologies will be considered.

Database Technology

Most problems which are interesting or useful to solve using a computer have large data sets associated with them. Database management systems (DBMS) are used to store and manipulate data efficiently, but the different views of the data required by different users has made it desirable to improve upon current database models. The AFORMS database management system is representative of early military database technology. AFORMS and the centralized data processing model on which it is implemented unnecessarily restrict the flow of information required to schedule pilots effectively. Current DBMS products are friendlier than AFORMS, but many still lack needed functions. DBMS shortcomings are being addressed by research, primarily research aimed at folding in lessons learned from other fields to integrate richer representational schemes

[5]In fact, problem solving in general depends on having some solution method and using it on current world data. Typically, military planning suffers not from lack of a solution method, but from a lack of data or an easy means to elicit data from clogged, unfriendly command and control systems. This is especially true when a crisis arises.

and add general programming features.[39] Several attempts have been made along these lines, but data modeling is still a research issue. Finally, database systems based on a distributed processing model and implemented across a wide-bandwidth network facilitates data dissemination better than the centralized data processing does.

Classical (machine-oriented) Data Models. There are a number of different data models used by current database management systems. Most DBMS products are based on one of the classical models: hierarchical, network, or relational. Data is organized as a forest of trees in the hierarchical model and as a directed graph in the network model. Both the hierarchical and network schemes have difficulty representing many-to-many relationships and tend to require procedurally-oriented operations based on a knowledge of the data structures involved. In contrast, the relational model is mathematically simple. Data elements are stored as tuples in tables. Queries may be expressed in a declarative fashion, freeing the user from having to think about the underlying procedures which actually deliver the answer. As a result, users prefer relational DBMS over systems based on one of the other two traditional models, and the commercial software industry has rapidly responded with an increasing number of relational DBMS products. [3]

DBMS Queries. The relational model supports declarative queries to some extent, making possible a simple query language based on a description of the desired data set, rather than a procedure for getting it. However, the user must still know which data is in which table and specify that information in the query. In addition, although the user may get at any data view using the three available operations (join, select, project) on the global database, there are two problems which can arise from this flexibility. First, query results can be incorrect unless the schema are carefully designed. Second, it would be nice to be able to update the data base when viewing it from any angle. However, it is not always

easy to input information into the database from a given virtual database relation because ambiguity may result.[3]

Do any of the classical DBMS models satisfy current user's requirements? For many applications (including a portion of the aircrew scheduler prototype), the relational model is sufficient. However, classical models are based on the need for efficient machine-implementations of databases, not on user's needs. Most DBMS models only have two levels: the schema and the data. Inheritance of information inherent in a taxonomy of objects is impossible to capture using two-level models. Also, classical models often blur the distinction between a set of data and a type of data. Different perspectives are possible, but difficult to separate. The problem of updating virtual relations cannot be addressed using a purely relational approach. Finally, no classical model supports temporal modeling, the changes in data over time.

Other Models. Semantic data models have been developed to address the shortcomings of classical approaches. Semantic data models like the entity-relationship model (Chen) focus on capturing the meaning of data, rather than on being machine-implementable. Other examples of semantic models are the Relational Model/Tasmania (Codd), Semantic Data Model (McLeod and Hammer), and the Event Data Model (King and McLeod). All semantic data models improve on the richness of data representation over classical models. They differ primarily in the types of relationship between data which may be expressed.[42]

Object-oriented Databases. Recently, some researchers have investigated the use of an object-oriented approach to database modeling. Object-oriented programming is a methodology which focuses on active data elements rather than procedures and passive data. The data elements (objects) are encapsulated and modular because they communicate

with one another only by message-passing. If the state of an object must be changed, typically another object passes it a message requesting the change and the procedures bound to the requested object make the necessary change. Therefore, each object knows how to alter its own state and react to incoming messages.[3, 4, 42]

The object-oriented data model also allows the programmer to take advantage of a class hierarchy and use the inheritance property to reduce storage requirements. A class is defined with known attributes, default values, and encapsulated procedures. An object which is a particular member of a class becomes only an instantiation of the class. If subclasses exist, they may inherit data and procedures from their superclass. For example, a programmer might wish to represent a class *transport-asset* with subclasses *truck*, *cargo-plane*, and *ship*. Attributes of *transport-asset* might be range, speed, location, and capacity. *Transport-asset* would also have an associated set of procedures, for example, a procedure for changing location. *Truck* and *ship* would, as subclasses, inherit all three of those attributes, but *trucks* might also have a clearance attribute and ships might have a loading-resource attribute. *Ship* might have additional subclasses like *tanker*, *frigate*, and *fast-cargo*. An instance of the class *fast-cargo* might be the *USS Atlas* with loading-resource for the instance having a value *self-loading-crane*. By inheritance, the data object *USS Atlas* would also have data slots for range, speed, location, and capacity, as well as all default values and procedures known to all superclasses.[33, 40]

Using the object-oriented paradigm, database researchers can easily model generalization and aggregation relationships (is-a and a-part-of relationships). Class information is easily captured and manipulated using the inheritance property. In addition, objects can be easier to manipulate when a collection is presented in a non-standard view. Several object-oriented DBMSs have been implemented; there is increasing support for graphical manipulation of data from an analytic view. Furthermore, complicated real-world

objects are more easily modeled and data modularity is enhanced. Therefore, the object-oriented data model addresses some of the difficulties associated with classical models. The primary disadvantage of the object-oriented model is that the mathematical simplicity and declarative beauty of the relational model are forsaken for a highly procedural, somewhat constricted message passing scheme.[6] [3, 4, 42]

Lessons from Artificial Intelligence Research. Knowledge representation has long been a research topic in the field of artificial intelligence (AI). Object-oriented programming (frames/slots) is one representation method used widely for AI applications. Other representations include predicate calculus, production rules, semantic networks, and expectation schema. Data manipulated by AI systems are less structured and less certain than data associated with most conventional databases. Therefore, knowledge and data representation schemes in AI tend to be more flexible and less pleasing in a mathematical sense. Moreover, there are a variety of ways of expressing uncertainty in databases due in large part to AI research. Temporal aspects of data have been treated by a number of AI systems. The biggest drawbacks from the rich expressive power of databases based on AI knowledge representation are that they are generally complex and large databases are difficult to manage and maintain.[3]

Database Interface. From a user's standpoint, a DBMS can be painful to use. Oddly, the typical DBMS restricts input and output. It requires data in a certain way and is not extroverted about "showing what it knows." Normally, a user must learn a command language to do sorts, attain virtual views, edit data, or even to get raw data regurgitated from an existing database table. Generating a non-standard report is beyond the capability of most casual users. The man-machine interface becomes crucial to the utility of the

---

[6]For more information regarding object-oriented programming, see APPENDIX C.

DBMS, and most commercial DBMS products are incorporating features such as menus, windows, and mouse input to make the user's tasks easier.

Database Access. Data must be accessible to those who need it. The current AFORMS database is a good example of how a constricted data flow can hinder decision making. The centralized data processing approach that AFORMS epitomizes separates the data from the user both in terms of the flexibility of data manipulation possible and response time of data access and update. Rather than giving the user terminal access to an inflexible DBMS, it is wiser to distribute the data and the ability to manipulate it with powerful tools in a cooperating information system. A distributed system comprised of personal computers, high-power workstations, and minicomputers all networked over a high-speed channel is becoming standard in many Air Force units.[3]

Scheduling Methods

There are a number of variations on scheduling one of multiple agents to one of several tasks (i.e., scheduling activities). Scheduling algorithms discussed in the literature generally have their basis in one or more of operations research, traditional AI search, or knowledge-based greedy heuristic search or some combination.[7] The systems with the most impressive performance result from a combination of scheduling methods. Constraints, both soft and hard, are pervasive and have great impact on scheduling algorithms, especially in the area of bottleneck resources.[5, 10] Optimization is an implicit goal, although optimization normally means different things to different evaluators in the

[7]Human schedulers tend to use the heuristics iteratively, with iterations occurring only after a sufficiently hard constraint is violated and discovered. It is a mistake to believe that human planners consider multiple alternatives, especially in a time-constrained environment.[21]

real world. Schedule flexibility, constancy, and explainability are all desirable for real world systems.

Algorithms Based on Operations Research. Generally, linear programming can be applied to scheduling problems only as an approximate method because the divisibility assumption does not hold. That is, some or all decision variables must be integer-valued or binary. In addition, some scheduling models are based on nonlinear equations which require nonlinear programming techniques to solve. Nonlinear programming can become computationally intensive; integer programming is not too bad, but soft constraints are difficult to model and solutions tend to be sensitive (lack consistency) when new constraints are added. Operations research methods attempt to provide a global maximum over time, but in many domains, the world situation changes for reasons outside of any plan (e.g. machines break, orders are re-prioritized). In practice, schedulers based on operations research algorithms often cannot keep up with unplanned events in complex systems, and a human supervisor must intervene to repair the plan. Operations research algorithms are not easily explained automatically.[5, 8, 9, 10, 12, 14, 18, 26, 38, 41]

Algorithms Based on AI Search. Search may be applied to find a `satisfying (or` optimal), robust solution based on some objective function. Search techniques include forgetful backtracking or memory-intensive depth- or breadth-first forward search. The best first search (A*) may be applied if an estimation heuristic can be defined. The objective function is typically knowledge-based as well. These search techniques are commonly used in classical AI planning systems (generative planners) to generate a solution path, and they work the same way for scheduling purposes. The A* algorithm was used in the LISP machine/KEE version of the RADC Aircrew Scheduler.[11, 28, 32]

Algorithms Based on Heuristics. If enough knowledge can be derived from the particular domain, a knowledge-based scheduling algorithm may be developed using heuristics. Greedy heuristics are those that improve the solution the most at the point they are invoked. An example of a greedy heuristic is "schedule a pilot who has less than five days of currency remaining without taking into account future placement possibilities." Like search-based methods, heuristic methods deliver robust, flexible solutions which can be explained by a little extra code. These AI systems differ from traditional search-oriented models by the degree of backtracking or search-memory required. Backtracking is normally required only when constraints are changed or may not be implemented at all. Some rule-based production systems are good examples of this type of system.[5, 7, 32]

Programming Environments and Methodologies for the Aircrew Scheduler.

Design requirements on the final aircrew scheduling product indirectly constrained the choice of programming environment (because prototype programming environments often become delivery environments). As implied by earlier discussion, data storage and database functionality were required to manipulate the half megabyte or so of pilot and schedule data. Several different types of data representation were required. Implementing the scheduling algorithm made the flexibility of a general purpose programming language desirable. A simple, aesthetically pleasing interface was considered mandatory to gain acceptance by the user. Specifically, multiple windows were thought to be important for intuitively generating different data views; mouse input was considered very desirable (pilots prefer pointing devices to keyboards); and direct editing of displayed data was also needed. The entire interface had to be simple and natural, with menus and context-sensitive help available. The use of multiple fonts, color, and sound were considered non-essential bonuses but would add appeal. The entire scheduling system had to run

acceptably on an IBM personal computer; portability was considered an unnecessary bonus.

The choice of programming environment and methodology were also directly affected by the complexity of the programming project and the relatively short time allocated to completing it. An object-oriented methodology or functional programming approach would have addressed the complexity issue; the available development time pointed toward the use of a software tool for building scheduling systems. A tool is a software environment consisting of a higher ordered language together with powerful functions useful for rapid development of a specific type of application. Tools cut development time by raising the level of programming; they are essentially higher- higher-ordered languages. For example, a tool for building expert systems might be Lisp-based but have functions for specifying a windowing interface and an object-oriented knowledge representation scheme lacking in Lisp. There was historical impetus for using a software tool -- the original LISP machine version of the aircrew scheduler was built on KEE which provides graphics, object-generation, and procedural functionality over and above the environment of a LISP machine. Unfortunately, generic tools often restrict flexibility; another way of saying this is that they lack needed functions and procedures. Tools are useful when the application to be built fits entirely within the scope of the high-level capability they provide. Furthermore, the tools available at the time[8] were not appropriate because high licensing fees would preclude distribution. Therefore tools were not considered further.

The choice of programming methodology depends to a large degree on the amount of program modification that will be required. Software has a life cycle that differs for

[8]Intellicorp's KEE and Gold Hill Computer's Gold Works.

different applications, but the stability of the code generally increases throughout its life cycle. Different types of software have different life cycles and different levels of stability. For an application with a large user base like a commercial word processor, there is economic justification for releasing a very complete product that will require few modifications. A word processor may be prototyped using one methodology and when complete, the prototype is ported to a hardware-oriented language for efficiency. Unlike word processors, a very specific application like a single-seat aircrew scheduler for A-10s has a very small user base; there may not be funds for release of a complete product, but more importantly, specific applications require user input not available to the software developer before release. Therefore, a specific application like an aircrew scheduler is generally released sooner in the life cycle, and will normally require modification. A specific application may never be hard coded into a faster language because of lack of funding, the need for continuous modification, or suitability of current execution. Artificial intelligence applications are typically very user-specific because current expert systems have narrow domains. Very few expert systems ever get out of the development stage for the reasons stated above.

If the programming methodology and environment of a user-specific application remains with and is delivered as part of the application, then there is another factor to consider: understandability. User acceptance depends on user understanding of the software. This is especially true of artificial intelligence applications. Therefore, the more the user can understand how the program arrives at an answer, the better the chosen environment. If users can understand the program and the programming methodology is simple enough, they may even be able to modify the code themselves.

Higher Ordered Languages (HOLs). Standard languages like Pascal or C offer the flexibility, execution speed, power, and portability required for the end product scheduling

system. The primary disadvantage of using a higher ordered language is that many needed functions and routines must be programmed; an alternative tool might provide these functions and routines and thus speed development. The tradeoff between speed of development and flexibility allowed by the programming environment is a prime design consideration. Initially, we attempted to port the LISP Machine/KEE version of the RADC Aircrew Scheduler to KEE running on a COMPAQ 386 personal computer running UNIX. That effort ground to a halt because of differences in KEE versions and inflexibility of the COMPAQ KEE environment, among other reasons. Our second porting attempt used the popular and powerful Turbo C compiler. We made progress using C, but development time was too slow to meet our milestones. Using our eventual programming environment, Microsoft Excel, we were able to achieve in 3 days the same functionality that had taken us 20 days worth of C programming earlier, given our meager programming experience. (See APPENDIX B).

Languages like Pascal and, more particularly, C are machine-oriented, procedural languages. They guard computer memory resources diligently and ensure fast execution speeds using efficient primitives and library routines. However, they require the user to think procedurally, and some programmers find that thinking procedurally stifles creativity. Other languages have been developed as alternatives to procedural languages. Declarative languages like Prolog focus on a specification of a data set (like relational databases do) and use a built-in backtracking procedure to generate query responses. In fact, it is easy to implement a relational database in Prolog[9]; Prolog was considered as a possible aircrew scheduler development environment because of its ability to answer questions which arise when using a heuristic scheduling methodology.[23] Functional languages like LISP and its derivatives (for example, Scheme) are similar to Pascal and C in some respects. LISP,

[9]The efficiency of a relational database implemented in Prolog is limited by the linear search mechanism used by its interpreter.

unlike C or Pascal, almost demands the use of recursion and automates memory allocation and reclamation. LISP was developed for artificial intelligence programming and allows the programmer to use any data without requiring type declarations. LISP has a simple syntax composed of seven primitives, the ones of primary importance to programmers being those that construct and select (dissect) LISP data objects (lists).[1,17]

Object-Oriented Programming. The object-oriented programming paradigm can be used in almost any language, but languages like Smalltalk-80 enforce it. The object-oriented model is data-centered, promotes data abstraction and modularity, uses procedures bound to data to make active data objects, and uses messages between data objects to execute a program.[10][33]

Experienced programmers tend to build up software modules that are later reused. Commercial C libraries are now available; LISP programmers tend to develop whole "worlds" of procedures useful in many contexts. A primary goal of object-oriented programming is software module development and reuse. Software reuse requires much documentation (writing and reading), but saves programming time, avoids undetected errors, and standardizes higher-level functions. The distinction between a "tool" and a programming language supported by an extensive library is getting blurry, but "tool" still implies a neater, less flexible package.[15, 33, 40]

Visual Programming. Using a LISP machine or a Smalltalk development environment clearly demonstrates that the total development environment impacts program development at least as much as the particular language chosen. Specialized programming environments such as these are typically single-user workstations having large displays with

---

[10]See APPENDIX C.

windowing/menu interfaces, mouse input, and seamless integration of all programming tools: editor, debugger, context-sensitive help, and compiler. All of these elements of the environment aid in software development to some extent, although it is very difficult to quantify the effect of environment attributes on software development. Rather, attribute "goodness" must be described qualitatively and subjectively. Personal preferences and histories have an impact. For example, many programmers (and users) would agree that a mouse coupled with a point-and-click interface is preferable to typing a response, but some never use a mouse because they can type faster or prefer not to switch from keyboard to mouse and back again.[2, 34]

One attribute of the environment, the display, is of particular importance and is now recognized as an important area of research commonly referred to as visual programming:

> With the availability of graphic workstations has come the increasing influence of visual technology on language environments. In this article we trace an evolution that began with the relatively straightforward translation of textual techniques into corresponding visual techniques and has progressed to uses of visual techniques that have no natural parallel using purely textual techniques. In short, the availability of visual technology is leading to the development of new approaches that are inherently visual.[2]

Visual programming focuses on making computer systems easier for people, rather than enhancing hardware performance. Part of the impetus for paying attention to visual interfaces comes from the widespread use of personal computers by nonprogrammers. Artificial intelligence research and expert systems have also helped make interfaces important. Display technology has advanced to the point where high-resolution bit-mapped graphics are available to almost anyone. Using high resolution graphics, a more visual mode of programming is possible and attractive. Interactive graphics has the potential for making input and output not only meaningful, but fast, interesting, and flexible as well.[34]

Why are vision and graphics important to computer input and output? Humans deal naturally and quickly with visual input and not so easily with serialized, one-dimensional text or speech. One reason that a picture conveys so much more information than a stream of words is that the "language" of pictures is a much richer, truer representation of objects in the real world. Things like shape, relationship to other objects, color, and texture are instantly recognized in a scene that would take hours to fully describe verbally. Another advantage that visual images have over text strings is that humans can focus on information they find interesting. The use of multiple fonts and columns (as in a newspaper, for example) allows a reader to focus on information of interest. In contrast, a listener must access information sequentially (as in a radio news broadcast), waiting for the desired information. Both of these reasons result in a higher information transfer rate for visual images than for one-dimensional text strings. The human visual-processing bandwidth is much wider than the audio-processing one. Animated pictures are an even better representation of dynamic real world objects. Animation further increases the potential transfer rate of information to humans.[34]

Currently, programmers use a variety of non-automated visual techniques to support programming. Among these are control-flow representations, like flow charts, Nassi-Shneiderman diagrams, state diagrams, and Petri nets. Data flow diagrams, which focus on data, rather than algorithms, are also used. In addition, more informal drawings are used to help visualize the state of the system. For example, student programmers often sketch out data structures like linked lists and trees to learn how they must be manipulated. Even when the concept of a particular data structure is known, drawings are often used to analyze programs and fix bugs. Finally, overall program structure is often conveyed as a topological arrangement of code modules (boxes) as are used for top level diagrams.[16, 34]

Program Animation. Some programs, particularly simulations, provide an animated representation of the world being modeled. Programs can also be written to simulate the internal state of the computer through animation. Animated programs display pertinent variables, program instructions, and their interactions. As the program executes, each line of code is shown along with the variables it accesses and the changes it causes. Animated programs help programmers check program correctness, analyze execution speed in different parts of the program, and determine which sections of code are inherently parallel by explicitly showing what the program is doing.[2, 24, 29, 34]

Graphical Input to Programs. Languages are being developed for integrating graphical images along with text as input symbols or output results in programming languages. Some of these developments take the form of syntax-directed editors which provide a template which allows for "programming by example" and syntactic error checking.[2,34] Others support a graphical view of programming by allowing the programmer to see a visual representation of data structures or code in execution.[6, 24, 27, 34] Windows may be used to support different views. Icons are used in some tools to assign a visual abstraction to code or data. Still other tools are useful for designing and documenting software, or generating it from a visual specification.[34] Newer research has focused on the use of graphical symbols as inputs to the programming language (either along with or in lieu of text).[6, 37] The use of symbols to index data is also a research topic.

Principally from artificial intelligence research and man-machine interface design, the impact of vision on computing environments is now known to be great. Seeing programs and data reduces the complexity of both by providing a way to move from the abstract to the concrete very easily. The field of visual programming languages has arisen as a result. It turns out that the best display is the largest display, as humans already have

the ability to focus attention on important parts of images. Debuggers and spreadsheets[35] are examples of this "more is better" rule.[2, 34]

Visual programming makes it easier to write programs. It enhances the programmer's ability to debug programs while running various data sets. Thus, validation and verification are easier in a visual environment. Modifying programs requires finding a portion of code causing some behavior and changing it to alter the behavior. Animation helps the programmer find the code causing a behavior, and a visual environment simplifies writing and testing new code. Therefore, visual programming also makes it easier to modify programs. [13, 25, 36].

CHAPTER 4
A VISUAL PROGRAMMING METHODOLOGY BASED ON MICROSOFT EXCEL ON THE MACINTOSH

The new aircrew scheduling prototype is based on Microsoft Excel, a spreadsheet/database/language with a spreadsheet interface. By using Excel, the new scheduler is portable to IBM/MS-DOS personal computers or Apple Macintosh computers. The principle development was done on a Macintosh IIcx. The methodology used to construct the Excel prototype is described in detail below.

A visual programming methodology based on the environment provided by Microsoft Excel has two cornerstones. One is the continuously updated display of a two-dimensional data array, the most apparent feature of any spreadsheet. The second cornerstone is program animation resulting from using a particular style of programming. Both features make program development and debugging easier by making data structures and program execution explicit in a visual sense.

Excel is an integrated tool having spreadsheet utility, but also featuring an interpreted macro programming language, graphics routines, and a powerful library of user functions.[11] The database functions are the primary ones useful for programming scheduling systems in Excel. Using the database functions and the macro programming language, a heuristic scheduling system with a sophisticated visual interface may be written with only a few hundred lines of code.

---

[11]Excel is representative of several spreadsheets having similar functionality. Many comments made about Excel are also true of other spreadsheet products.

Excel also supports graphical programming methods for generating dialog boxes. These interactive, graphical, user input windows further refine the overall interface and make the program even easier to use.

The visual programming methodology described is based on wide-bandwidth output and animated execution, but takes full advantage of all attributes of Excel, including the windowing operating system it resides on and other attributes of the Macintosh: graphics, mouse, and built-in networking.

A Visual Programming Methodology Using Excel. The general methodology used is outlined below and discussed in the following sections.

1. Without regard to procedures required, generate the applicable data structures in portions of the Excel spreadsheet array. Use concrete representations which are as similar to the modeled data structures as possible Group related data structures in appropriate locations in the spreadsheet array. Fill the data structures with real or example data.

2. Make full use of functional programming to display abstract data.

3. Using the continuous display, mouse, and available commands, manually manipulate the data to learn how to cause the desired program behavior as needed.

4. Using available functions, write small program modules to incrementally automate data manipulations. Group related program modules in appropriate locations in the two-dimensional program area. Modules should be called as subroutines. Access data only by visiting its location to provide animation. For example, move data using *'select-copy-select-paste.'* Continue until data manipulation is fully automated, testing modules using animation. Rearrange data structures and program modules as appropriate.

5. Generate the user interface by graphical dialog box programming and menu bar commands.

Excel's Interface. Like other spreadsheets, Microsoft Excel displays an array of data cells which are continuously updated, should they contain a formula. Higher-ordered languages (HOLs) like C, Pascal, and Ada require extra programming to get output, but a spreadsheet interface is always the same (with minor variations): maximum output. The programmer must only determine how to partition the data array into a useful display.

Data structures in Excel are visually explicit, in contrast to the hidden ones in C or Pascal. Novice students learning about computers are often told, "Computer memory is like a row of mailboxes at post office. Each mailbox has a physical address which the computer knows about. Each mailbox has space to store data in. You name the mailboxes so that you can access the data in them." Thus, the abstract notion of assignment becomes concrete and understandable. In a Pascal program, the assignment might be "A = B + 1;" and suddenly, the notion is abstract again. What is value of B? What does A look like? (Perhaps A is an array!) Arrays, linked lists, and trees all had to be drawn and visualized by every programmer who now understands them, yet in the language, these structures are invisible. No wonder it is so difficult to program using them! A mental image of what is going on is constantly required. What happens when the number of variables becomes large? Questions like "What is the value of B?" begin to slow progress. Those questions are not so difficult when their answers are continuously displayed.

Part of the representational complexity of programs comes from data structures (the remainder comes from algorithms). The more concrete a data structure is, the easier it is to understand. For many data structures and some real-world things (e.g., schedules), a visual, two-dimensional array is a more concrete, truer representation than the invisible data structures available in Pascal or C. In Excel, an array looks like the array we visualize in Pascal (at least, to two dimensions). A linked list in Excel looks natural also, and it is a lot

tougher to get lost in an explicit one. A real-world schedule is often represented as a table; an Excel schedule can look identical.

Because of the two-dimensional array displayed, Excel lends itself to two-dimensional data structures. Two-dimensional data structures include single variables (1 x 1 arrays), one-dimensional arrays (n x 1 arrays), two dimensional arrays, and tables, relational database tables included. With a little imagination (using relative pointers), linked lists, stacks and queues (as n x 1 arrays) can be implemented. Higher dimensional arrays and trees are difficult to implement as visually explicit structures. For writing "real world" applications, lack of support for higher dimensional data structures can be unimportant; many "real world" data structures, such as invoices, bank statements, time-tables, etc, are inherently two-dimensional because they are expressed on paper or some other two-dimensional surface. For example,the data structures most used for scheduling are a schedule (a table) and relational database schema (other tables).

In Pascal or C, variables are typed and allocated by name. After that they may be used in the program. Strong typing allows for some error checking and allows efficient use of memory. However, in the iterative style of most programmers, there is a necessary cycle between the discovery of the need for a variable and its required typing and allocation. Languages such as LISP require no typing and allocate using definitions at run time. In Excel, typing is discovered by the system from syntax (as in BASIC), and allocation is made incrementally as data is entered into each cell. Furthermore, each location where data may be stored has a default name.[12] Thus, the programmer never needs to scroll back to the top of a file to allocate additional variables, as in C or Pascal. In

[12]A12 or B3, for example. User specified names are also supported.

practice, it is more natural to put data and data structures onto the spreadsheet, and write programs to manipulate them afterwards.

In Excel, simple variables, higher data structures, and algorithms are all physically located someplace on the two-dimensional spreadsheet. For example, one subroutine may be east, west, north, or south of another. This is fundamentally different from a language like C or Pascal. In C or Pascal, data structures are invisible, existing somewhere in the ether of the computer's memory; algorithms are a little better off, existing in a linear (one-dimensional) text file. By inserting white space, Pascal and C algorithms can be written with added dimensionality. In other words, properly indenting code can give additional meaning; thus, much of the power of structured programming arises because of appearance, a visual extension of the one-dimensional alternative. However, indented code is not really two-dimensional in the same sense as Excel programs can be. In Pascal, one code module must come before or after another. The added dimensionality in Excel provides additional impetus to write modular code for cognitive reasons (discussed below).

From a cognitive standpoint, Excel's visual, two-dimensional spreadsheet interface is far superior to the programming environment of typical higher-ordered languages. The primary advantage of a spreadsheet interface is that additional associations are possible that relate a data structure or piece of code to familiar objects. There are three practical benefits which result. First, it is easier to find a data structure or a piece of code because the data or code always has some relation to what is known (visible). In a traditional HOL, "finding a data structure" has no meaning, but finding a piece of code equates to scrolling a particular distance from the current cursor position. The difference between finding something in Excel versus Pascal or C is somewhat analogous to navigating by map as opposed to getting directions. Directions tell you how far to go on a one dimensional route before taking some action (e.g., turning), while using a map allows the use of external points of

reference. A second benefit of two-dimensional associations is added flexibility in structuring programs. Different arrangements are possible. A top-level design diagram may be implemented in similarly arranged code; alternatively, code modules may be arranged hierarchically. The methodology used to create the Excel scheduling prototype only requires related code modules to be placed close together in some natural order. A third benefit from extra-dimensional associations arises from the way human memory functions. Remembering a piece of information is related to the number and strength of the associations attached to it which relates it to something else already in memory. For example, mentally picturing a new acquaintance standing with old friends with the same name is a common procedure for learning names at a cocktail party. Current HOLs attach a name to a data structure or procedure, although a procedure has a second memory "handle," its location is in a linear text file. In Excel, both data structure have at least three handles: absolute cell address, user-defined name (if one is assigned), and a relative position from some other cell. Note that the last handle is really many handles, e.g., a variable called *'current pilot'* in cell F19 is 3 cells west of *'current comment'*, 12 cells northeast of *'priority pilot'*, etc. ad infinitum. The utility of the extra memory handles is entirely semantic but is closely tied to locating code and structuring it. Six months after writing a procedure, not only can it be found and known to be related to an adjacent procedure, but its meaning and function may also be recalled (especially when the meaning of adjacent procedures is known). Because of this, writing code in a modular fashion does not add to the complexity of the program as much. Modularity combats complexity, but the overhead of remembering what all the modules do can, at some point, begin to add complexity of its own. Because modularity is well supported by the Excel environment, it is also a goal of the visual programming methodology described.

Control Structures. The Excel aircrew scheduling prototype makes extensive use of IF-THEN-ELSE, subroutines, and GOTO control structures. Excel has a WHILE

primitive, but most modules are so small that the GOTOs used do not cause too many problems because they are restricted to the module. Going outside the module should be done using a subroutine call, under the methodology used. Using small modules keeps branching limited. Although this is the most natural type of structured programming supported by Excel (perhaps with the GOTOs replaced by WHILE), it occasionally made modules slightly longer than necessary. Program branching can make code smaller, but it can also add complexity.

Data Structure Design. When programming a solution to an unfamiliar, complex problem, its not clear, from the start, what data is important, what data structures are suitable, or what the relationships between data structures are. Using a visually explicit language like Excel allows one to begin programming before having a complete understanding of the problem and aids understanding along the way. This approach was used during development of the Excel aircrew scheduler: First, the data thought to be important was arranged in tables or simple variables in a way thought to be correct. Then, simple programs were written to manipulate the data and discover what manipulations were possible and natural. This process uncovered several important relationships between data structures and showed what additional data and data structures would be useful. The two-step process was iterated a number of times to arrive at a reasonable product.

Program Animation. In Excel, there is always an active cell and a selected array of cells, just as there is always some location stored in the program counter of a microprocessor. (The selected array may be just the active cell, a 1 x 1 array). Macro programs which operate on Excel data use the active cell and selected array to access data in a fashion very similar to the way a microprocessor gets data from memory (one byte or word at a time). While this arrangement appears rigid (and can be overcome), it forces explicit manipulation of displayed data. By differentiating the active cell and selected array

from other cells, animated programs are possible. In Excel, the active cell is outlined in color and the selected array (except for the active cell) is filled in with color. By programming Excel to visit each data cell to access it, the active cell will, during program execution, move around on the spreadsheet data array, thereby showing the programmer exactly what it is doing. For example, a typical operation is to move data from one cell to another (for example, the criteria array). To do this in Excel, a four line program is needed to select the first cell, copy its contents, select the second cell, and paste the contents of the first cell. The *'select-copy-select-paste'* program appears as a three step visual program wherein the active cell visits the cell to be copied, visits the cell to be pasted, then pastes the value of the first cell. Using this methodology, every time a program runs, the programmer sees which data cells are visited and what changes occur to cells. Furthermore, Excel has a single stepping feature which allows the programmer to slow execution to see just what effect each line of code has.

The program animation possibilities using Excel makes it very easy to debug a program. The programming process is also much improved because it is simple to see inefficient pathways to the same end.

Rearrangements of Data and Programs. Excel uses relative addressing as a default means. Relative addressing is more natural than absolute addressing if data is going to be moved around. Excel expects rearrangements of data and provides for it in sophisticated ways. For example, data arrays may be cut and pasted elsewhere on the spreadsheet. When data is cut and pasted, all references to it in any Excel file are transferred to the new, correct address. Functions on the spreadsheet which return data values also refer to new addresses, making it possible to cut a function from one cell and paste it into a large array. The formulas pasted into the array all return different values because they use relative, rather than absolute references. Easy rearrangement promotes modular programming,

supports experimentation with different data structures, and supports experimentation with the physical layout of code and data structures.[30, 31]

Verification and Validation. The combination of visually explicit data structures and animated program execution are powerful tools for verification and validation (V&V). There are a number of static and dynamic methods for ensuring V&V for traditional and artificial intelligence software. Static methods include anomaly detection, structured walk-throughs, and mathematical proofs of program specification and correctness. Dynamic methods include random, regression and thorough testing. Static analysis of data structure and algorithms is directly related to their visibility, a metric of relative obscurity. Visual programming methods allow the easiest static analysis of data structures due to their explicit representation. Proofs of program correctness are generally too cumbersome for complex software. Routine testing has been found to be one of the most effective V&V methods, especially for artificial intelligence applications. All testing methods are simple when the program displays itself executing as Excel programs do.[13,25,36]

Graphical Programming. Excel supports graphical programming for generating data for interface dialog boxes. The Excel dialog editor displays an empty dialog box and the programmer may add elements such as list boxes, option boxes, cancel and accept buttons, and text from a menu. The dialog box and its elements may be resized and moved using the mouse. When an acceptable interface is designed, the data which creates it can be cut and pasted into an Excel spreadsheet. Then a one line program will generate the interface and user inputs are stored as data next to the interface data. Additional lines of code are required to access stored user inputs.

Use of Database Functions. Although not part of the visual programming methodology, use of Excel's fourth-generation functions make many applications simpler

to program. In the case of scheduling, the database functions are most important. To heuristically select tuples from a database based on information in a schedule, items from the schedule (e.g., qualification) may be pasted into the criteria array. This is one method of applying a constraint to push solutions into a feasible region. Database functions such as selection, the number of satisfying tuples, the maximum or minimum of an attribute column, and extraction (projection) all use the current criteria array to select tuples and make programming easier by raising the level at which it is done. For example, a very simple aircrew scheduler might be based solely on training need. To implement the scheduler, a pilot database, schedule, and algorithm are needed. One algorithm which satisfies the requirement would copy the mission type from the schedule, go to that mission type in the criteria array, and paste in the maximum value of the corresponding mission column in the database. That action would constrain a database selection to the pilot with the largest training requirement for the mission under consideration. The algorithm would simply select the pilot name, copy it, and paste it in the schedule.[30,31]

Windowing, Menus, and Mouse. The Macintosh environment is a visual operating system. The use of windows, a standardized pull-down menu interface across all applications, and mouse input all contribute to the ease of using any application. The integration between applications epitomized by the ability to cut and paste information between applications is very helpful during application development and use. Graphical icons are more meaningful than text identifiers when it comes to file manipulation. The mouse simplifies the user interface for most applications and allows file icons to be manipulated in a natural way. In Excel, using the mouse to select and move data is much simpler and faster than using the cursor keys.

CHAPTER 5
THE EXCEL AIRCREW SCHEDULING PROTOTYPE

Using the visual programming methodology based on Microsoft Excel described in the last chapter, the functionality of the LISP Machine/KEE version of the RADC Aircrew Scheduler was ported to run in Excel on the Apple Macintosh IIcx. (Excel also runs under Microsoft Windows on IBM personal computers and under Sun-OS (a Unix derivative) on a Sun workstation).

Design Considerations. To gain user acceptance, the Excel aircrew scheduler prototype was designed to do aircrew scheduling in the same way as the current manual method. The focus was not on improving the current scheduling algorithm, but on improving the communication and presentation of data, providing analysis capability, and automating the current algorithm. By automating the heuristic scheduling algorithm used, the computer can aid the scheduling officer by finding appropriate pilots and never forgetting constraints. Using this approach, the computer is allowed to do what it does well (i.e., store and manipulate data) and the scheduling officer is left to do what he does well (handling anomalies and determining smarter ways of doing things). Heavy emphasis was placed on allowing the user to have control of the system. For each action implemented, there is an analogous procedure for retracting it. An uninhibited display attitude was a prime design requirement from the start, and Excel's constant output of all data and functional results supported that requirement well. There was some concern that the user would be flooded with data, but humans are well tuned for focusing on what they consider important, and scheduling officers have not complained thus far. Using the standard window interface, the user can access more data by simply scrolling the window. Editing is important for changing schedule data, and the direct editing interface supports the

user's need for natural interaction. Also, the same animation that helped the program developer build the prototype will help the scheduling officer understand exactly what the prototype is doing and thus build confidence in its scheduling choices. Because Excel exists in a windowing environment, multiple user views are possible, including one which minimizes the animation window, should it become boring. The prototype has been built making maximum use of graphical dialog boxes, resulting in interactive routines that are simple point-and-click operations.

System Description of the Excel Aircrew Scheduler Prototype:

Data Structures. Consolidated database. The current Excel prototype has a database of pilots, their qualifications, event completions for the current training term, and currency days remaining for each event (basic AFORMS data). (See Figure 2.) In addition, the consolidated data base includes a row for preferences and the availability status of each pilot based on the mission under consideration, pilot availability data, and timing constraint data. (See Figure 3.) Timing constraint data are those data items which arise from the length of flying day and crew rest constraints. This data is added onto (i.e., linked with) tuples in the consolidated data base. Pilot availability data is stored in a linked list which is also attached to tuples in the database. Pilot availability status appearing in the consolidated database is actually the result of a function operating on availability and timing constraint data, using current mission start and stop times. Thus the abstract notion of availability that the user has in mind is the actual data presented, and the details of how availability is calculated are buried.

Associated with the consolidated database is a criteria array used to specify criteria for selecting data. (See Figure 2.) It is composed of attributes from the consolidated database and is displayed in tabular form. Because there is a one-to-one correspondence

PILOT	Qual	Availability	WD	WD Cur Days	ACBT	ACBT Cur Da	DACBT	DACBT Cur D	Preference
Able, Adam	5	Unavailable	7	10	10	12	9	8	
Baker, Barry	5	Unavailable	3	9	14	22	8	9	
Charlie, Chuck	5	Free	10	18	9	20	8	9	
Dingo, Dave	5	Free	23	29	9	27	3	28	
Edwards, Eric	4	Free	14	1	3	11	9	13	
Frank, Fred	4	Free	1	16	18	18	4	21	
Gonzo, Greg	4	Free	6	-6	11	-24	6	-21	
Harris, Harry	2	Free	17	4	4	25	20	2	
Iggy, Ian	2	Free	9	7	19	16	1	11	
James, Jim	2	Unavailable	11	18	20	8	15	16	
Kee, Ken	4	Unavailable	9	11	7	0	20	8	
Lint, Larry	1	Unavailable	22	-34	0	-2	13	-30	
Mason, Mike	1	Unavailable	10	22	40	9	9	11	
PILOT	Qual	Availability	WD	WD Cur Days	ACBT	ACBT Cur Da	DACBT	DACBT Cur D	Preference
13	Pilots fit criteria		23	-34	40	-24	20	-30	
Below is space allocated to alternate criteria rows:									
	5	Free			14	>=0			
	5	Free							

Figure 2. The Consolidated Database and Criteria Array

PILOT	Prev	Today last Land	Match	Availability list - each pair is a start and stop of available time								
Able, Adam		1015	1	0	830	1015	2400					
Baker, Barry		1000	2	0	800	1000	2400					
Charlie, Chuck			1	0	2400							
Dingo, Dave		1330	1	0	1200	1330	2400					
Edwards, Eric			1	0	2400							
Frank, Fred			1	0	2400							
Gonzo, Greg			1	0	2400							
Harris, Harry			1	0	2400							
Iggy, Ian			1	0	2400							
James, Jim		1000	2	0	800	1000	2400					
Kee, Ken		1000	2	0	800	1000	2400					
Lint, Larry		1330	1	0	830	1015	1200	1330	2400			
Mason, Mike		1000	2	0	800	1000	2400					

Figure 3. Availability Data Linked to Pilot Tuples

between column headings (attributes) in the criteria array and the consolidated database, the criteria array is physically located directly below the consolidated database for aesthetics. There are two areas below the criteria array used as a swap space for the current criteria. In

addition, there is an array below event and currency attributes in the criteria which hold maximum and minimum values of corresponding database attribute tuples. Again, "maximum" and "minimum" are abstractions, the result of functions.

The partial schedule to be filled is presented as a table containing schedule line number (a key), take off time, landing time, mission type, pilot qualification requirement, aircraft configuration and range data for each sortie. (See Figure 4.) There are slots left open in the schedule for pilot and scheduler comments. The schedule flight date and schedule generation date are attached. Today's date, the result of a function using the computer system clock, is displayed and may be copied into the schedule generation date.

The priority list for the schedule appears as a table but is dynamically converted to a database during program execution. (See Figure 4.) It contains relative priority number, pilot, mission, and a requirement comment, if desired, from the supervisor who generated it (typically the training officer or DO). There is an attribute heading for a comment by the scheduler because, when a supervisor establishes a priority, the scheduling officer needs to communicate how the priority was treated. As with the consolidated database, there is a criteria array associated with the priority list (priority list criteria) containing the same attribute headings and located directly below the priority list. In addition, there is an area below the priority list criteria used to extract (project) the priority number and pilot name from the priority list based on the selection criterion.

Flight data are displayed in another table. (See Figure 5.) This data specifies which sorties are parts of which flights. Two- and four-ship flights are common, although

Schedule for			Date Gen	Today's Date					
1-May-90			2-Apr-90	2-Apr-90					
Line Number	Take off time	Landing	Mission	Pilot Requ	Pilot	Confiuration	Range	Comments	
100	800	1000	ACBT	>=4		J	A		
101	800	1000	ACBT			J	A		
102	800	1000	ACBT	>=3		J	A		
103	800	1000	ACBT			J	A		
104	830	1015	DACBT	>=3		B61MP	A		
105	830	1015	DACBT			B61MP	A		
106	1200	1330	WD	>=3		B61	P		
107	1200	1330	WD			B61	P		
Priority List	1-May-90								
Number	Pilot	Mission	Requirem	Comment					
1	James, Jim	ACBT							
2	Lint, Larry	WD							
3	Lint, Larry	DACBT							
4	Gonzo, Greg	DACBT							
End									
Number	Pilot	Mission	Requirem	Comment					
				=					
Number	Pilot								

Figure 4. Partial Schedule and Priority List

Flight Data					
1-May-90					
Line Number	Pilot R	No. of aircraft	Other aircraft		
100	>=4	4	101	102	103
101		4	100	102	103
102	>=3	4	100	101	103
103		4	100	101	102
104	>=3	2	105		
105		2	104		
106	>=3	2	107		
107		2	106		

Figure 5. Additional Flight Data

other configurations are possible. The configuration of the flight dictates what qualifications are required by the flight leaders. For example, a two-ship flight would require a two-ship flight leader, but a four ship flight would require a four-ship flight leader as well as a two ship flight leader because tactical aircraft normally fly in pairs (leader and wingman). Currently the flight data are used only to reset pilot qualification when pilots are removed from the schedule.

There are four tables which help the system provide different data views to the user. Three of them result from projecting data from the consolidated data base. Pilot qualification, event requirements, and event currencies may be viewed as tables or graphs using these projections. The fourth table is a manipulation of the pilot availability data used to create a time line chart of pilot availability. (See Figures 6, 7, and 8.)

There are other abstractions displayed and used by the program. Among these are size data for the consolidated database and the number of pilots from the consolidated database who meet the current criteria. There is a vector of match values for the availability data. The match values are used to update data and as an intermediate result for availability calculation. Other data that are displayed and used are the number of days remaining in currency before the scheduler becomes concerned and the last DNIF times.

Most arrays used are not the typical Pascal-like array but rather dynamic data structures which may expand in length or width. The consolidated database lengthens by the addition of pilots (tuples) and widens by the addition of training events. The associated criteria also expands in width when training events are added. The availability data stored in linked lists attached to each pilot (tuple) are able to expand to the limits of memory; however, current algorithms used do not take into account more than five blocks of free time or occupied time for a day. This limit is considered adequate for tracking

Pilot Qualifications			Event Requirements				
PILOT	Qualification		PILOT	WD	ACBT	DACBT	
Able, Adam	5		Able, Adam	7	10	9	
Baker, Barry	5		Baker, Barry	3	14	8	
Charlie, Chuck	5		Charlie, Chuck	10	9	8	
Dingo, Dave	5		Dingo, Dave	23	9	3	
Edwards, Eric	4		Edwards, Eric	14	3	9	
Frank, Fred	4		Frank, Fred	1	18	4	
Gonzo, Greg	4		Gonzo, Greg	6	11	6	
Harris, Harry	2		Harris, Harry	17	4	20	
Iggy, Ian	2		Iggy, Ian	9	19	1	
James, Jim	2		James, Jim	11	20	15	
Kee, Ken	4		Kee, Ken	9	7	20	
Lint, Larry	1		Lint, Larry	22	0	13	
Mason, Mike	1		Mason, Mike	10	40	9	
			Event Currencies - Days Remaining				
			PILOT	WD Cur Days	ACBT Cur Days	DACBT Cur Days	
			Able, Adam	10	12	8	
			Baker, Barry	9	22	9	
			Charlie, Chuck	18	20	9	
			Dingo, Dave	29	27	28	
			Edwards, Eric	1	11	13	
			Frank, Fred	16	18	21	
			Gonzo, Greg	-6	-24	-21	
			Harris, Harry	4	25	2	
			Iggy, Ian	7	16	11	
			James, Jim	18	8	16	
			Kee, Ken	11	0	8	
			Lint, Larry	-34	-2	-30	
			Mason, Mike	22	9	11	

Figure 6. Data Projected from the Consolidated Database

availability and was chosen to constrain file sizes, but it may be easily changed. The schedule will obviously be longer or shorter depending on the number of scheduled sorties. The length of the priority list is also variable. The four tables derived from other data are also dynamic. All program modules have been written to take into account the dynamic nature of the data structures used.

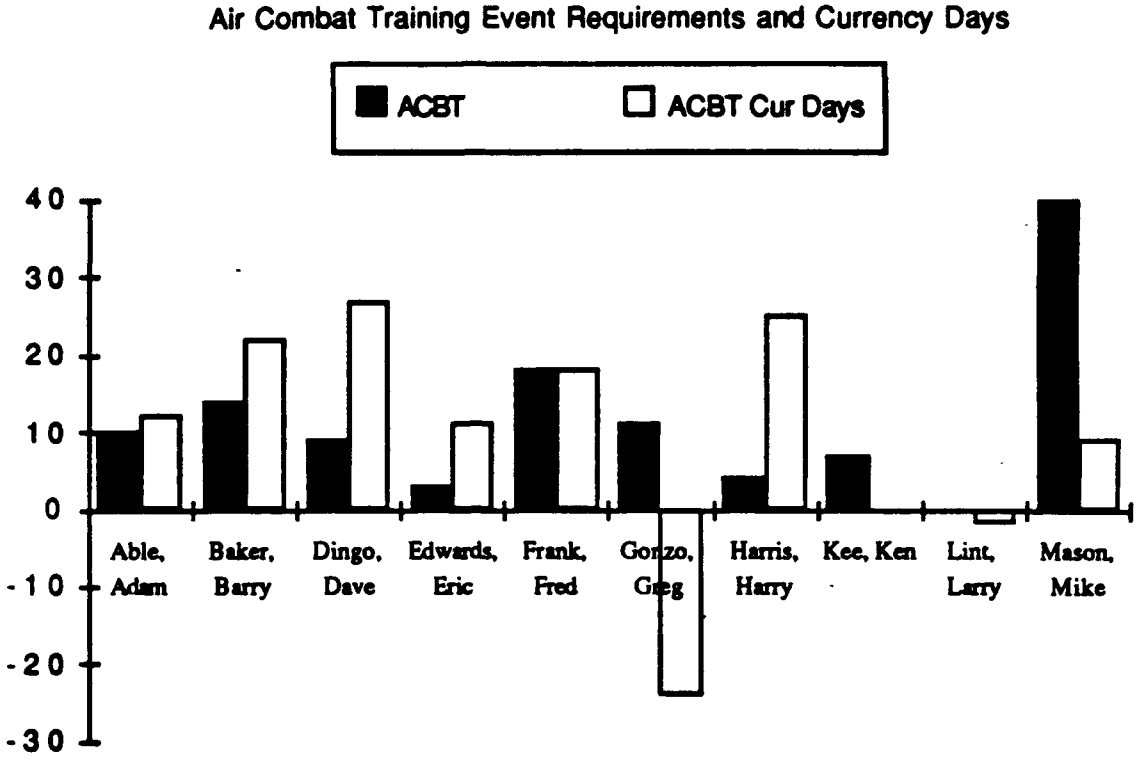

Figure 7. Example Analysis Graph

Data Representations. Data appearing on the interface window are either of simple type, such as numbers, dates, or character strings, or an abstraction, the result of a function applied to other data. Simple data may be edited directly without possibility of error. Abstractions may also be edited, but the details of the function are presented in the editing window. To edit screen objects, the user mouses on it (points to it and clicks) to select it. When it appears in the editing window, normal editing commands are available and a carriage return completes editing.

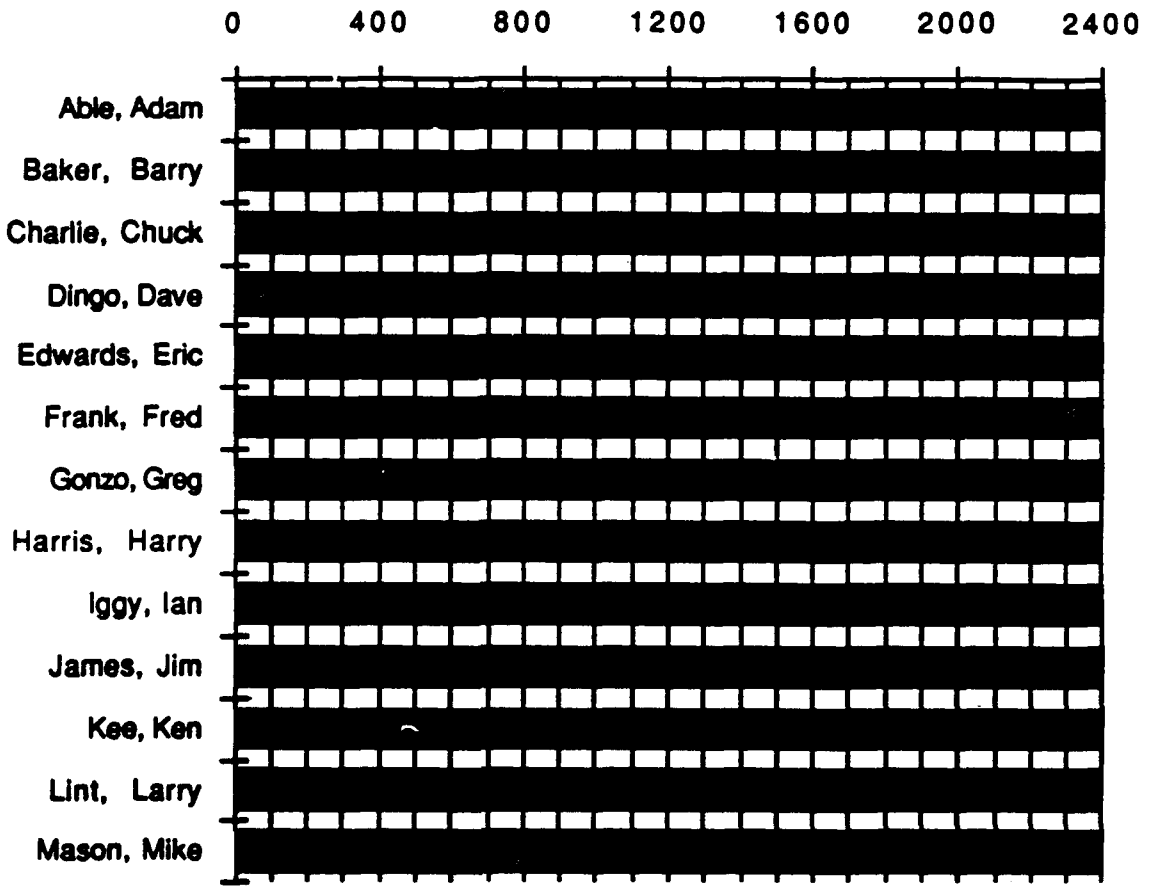

Figure 8. Pilot Availability Timeline

Pilots are represented by a string consisting of their names in the format shown. The user must only type the name in once; thereafter, the name may be more easily copied and pasted. Pasting names is useful when the user wants to manually insert a pilot into the schedule. Pilot qualification levels are mapped to numbers in the following way: those in mission qualification training are level 1, those who are mission ready are level 2, two-ship flight leaders are level 3, four-ship flight leaders are level 4, and instructor pilots are level 5. This mapping is useful for selecting pilots with at least *some* level of qualification and is simple for humans to assimilate. Pilot availability is displayed as either "Free" or

Schedule for			Date Gen	Today's Date						
1-May-90			2-Apr-90	2-Apr-90						
Line Number	Take off	Landing	Mission	Pilot Req	Pilot	Confiur	Range	Comments		
100	800	1000	ACBT	5	Baker, Barry	J	A	14	Events Remaining	
101	800	1000	ACBT		Mason, Mike	J	A	40	Events Remaining	
102	800	1000	ACBT	>=3	Kee, Ken	J	A	0	Days of Currency	
103	800	1000	ACBT		James, Jim	J	A	1	on Priority List	
104	830	1015	ACBT	5	Able, Adam	B61MP	A	9	Events Remaining	
105	830	1015	ACBT		Lint, Larry	B61MP	A	3	on Priority List	
106	1200	1330	WD	5	Dingo, Dave	B61	P	23	Events Remaining	
107	1200	1330	WD		Lint, Larry	B61	P	2	on Priority List	

Figure 9. Automatically generated Daily Schedule

"Unavailable" in context of the beginning and ending time of the activity under consideration at the moment. Event requirements are numbers indicating the number of events yet to be completed before the end of the six month training term to maintain combat ready status. Event currencies are also numbers which are the days remaining before going out of currency in a particular event. Operational units think of currency as a date, but numbers were more easily manipulated, and again, humans can quickly adapt to the altered representation. Preferences are normally user-specified character strings; the system currently uses the preference attribute to temporarily preclude the scheduling of unsupportable pilots. Times are represented in military format by numbers from 0 to 2400, although there is no type checking to flag meaningless values such as 1062 or 4000. The availability data are pairs of times indicating the start and stop time of a free block of unallocated time (or, shifting one data point over, the start and stop of a slice of allocated time). A manipulation of this representation is a list of durations of free time and allocated time slices arranged in an alternating fashion. The duration list is used to create a time line chart. This representation is acceptable for graphing purposes, but it is not aesthetic as a primary representation because of the error induced by using normal numbers as time (there are only 60 minutes in an hour). Missions, configurations, and ranges all are represented

by strings of their commonly used abbreviations. Explanations may be user-specified, but automated ones are formed by appending a number to a string. (See Figure 9.)

There are a number of ways to access or reference data, as described earlier. At all times, there is an active cell in a selected array which is similar to the address pointer in assembly language programming. As discussed in the last chapter, data may be referenced by its relative position to the active cell. This method was used commonly because the visual nature of the display made relative addressing natural and understandable. Of course, the spreadsheet interface names each data cell in an absolute sense as well. A third alternative, often used to abstract data or procedures, is the assignment of a user defined name to data.

Procedures. The primary focus of the aircrew scheduling prototype was, from the beginning, data centered. The Excel-based visual programming methodology supported a data orientation very well. However, Excel's high level functions and relatively clean separation between data and algorithms allow the procedures which manipulate the data to appear very powerful. Three hundred lines of code implemented a sophisticated heuristic scheduler which took over 2000 lines of LISP in the previous prototype. In reality, the macro procedures have very little to do, and the power comes from the high level functions and continuous display and update inherent in Excel's spreadsheet interface.

Some algorithms used were automatic functions whose results appear as data. Availability and attribute maximums are examples. The availability calculation results in "Free" or "Unavailable," in part depending on whether or not the activity under consideration fits entirely within a given pilot's free time block. (See Figures 3 and 8.) The block of time required by an activity (e.g., flying a mission) is defined by its start and stop times. Pilot availability data are stored as blocks of free time (start and stop times) in

the linked lists attached to each pilot tuple in the database. Therefore, its simple to calculate availability by making sure the activity occurs completely within unscheduled time, i.e., does not spill into a previously allocated time block. The other constraints placed on availability arise from Air Force regulations limiting the length of the flying day and providing crew rest. The availability calculation uses last landing times for the current and previous day to enforce these constraints.

Most algorithms are not the result of functions, but programs known in Excel lingo as command macros. These procedures are distinct from the data; they're stored in a separate file and are manipulated via another window. Modularity was used extensively. The top level design is shown in Figure 10.

The scheduling algorithm to find an appropriate pilot uses three approaches to heuristically select a pilot if one is needed. If a pilot has been suggested by the user, the algorithm will check the pilot to make sure no constraints are violated by the user choice. If constraints are violated, the system defaults to automatically finding an alternative, or the user may direct the algorithm to halt. If no pilot is suggested, the routine *'find pilot'* first checks the priority list for applicable pilots and attempts to schedule them in order of their assigned priority. If no priority pilot can be scheduled, the algorithm tries to find a pilot who has only a few days of currency remaining. Why is currency important? Non-current pilots require instructor pilots to regain currency, and instructor pilots are a resource for achieving training goals. Therefore, it makes sense in most cases to assign greater priority to flying pilots who will soon go out of currency. "Low" currency is a visible, user-specified parameter; currently it is set at 7 days. If there are no pilots who may be scheduled because of low currency, the algorithm looks for pilots with the largest number of training events remaining. The primary task of the scheduler is to provide opportunities

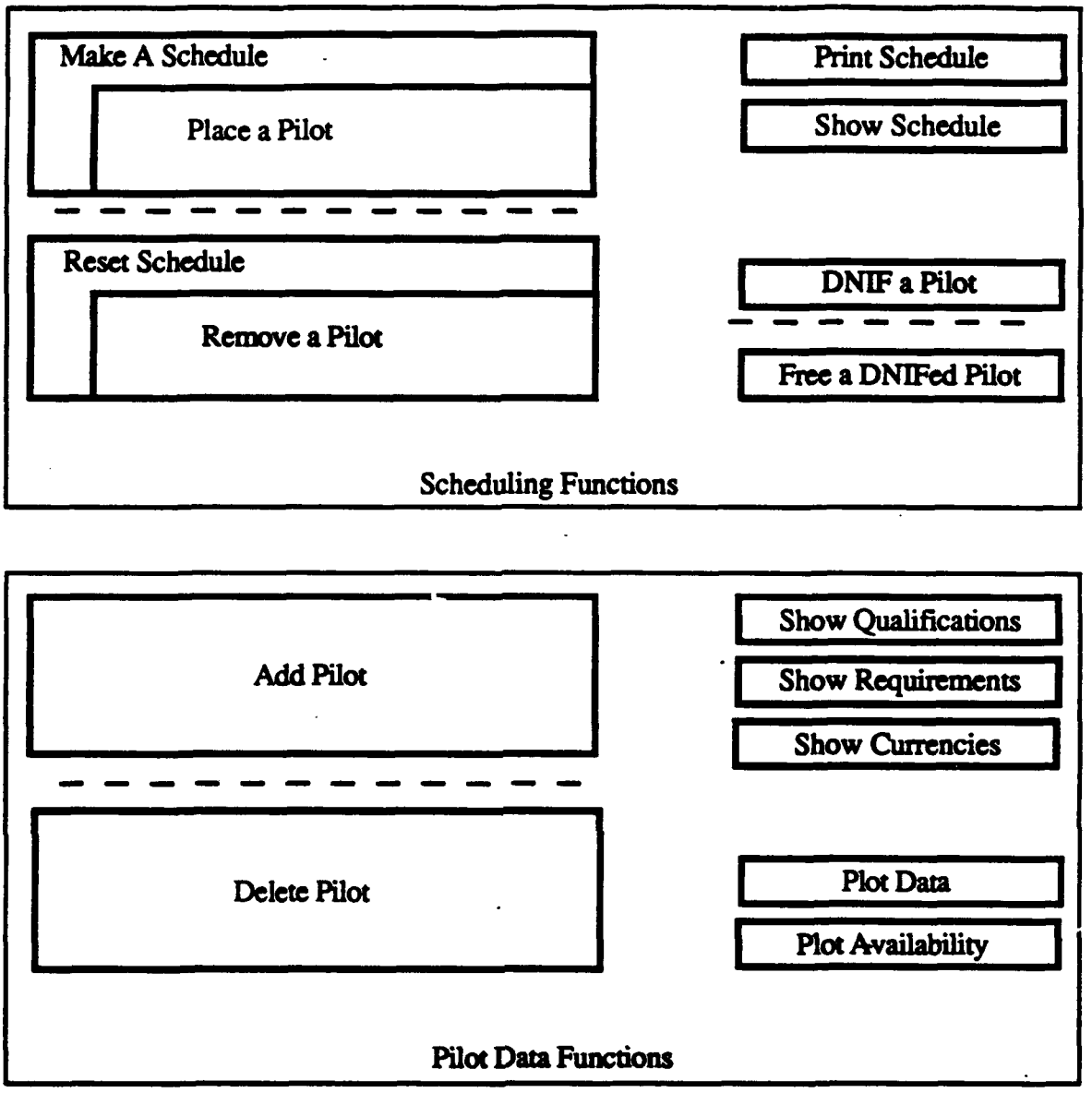

Figure 10. Excel Aircrew Scheduler Top-level Design

for required training, regardless of currency status, qualification, or other factors. If no pilot can be found[13] using these three approaches, the scheduler reports **NO PILOT**. In later versions, the algorithm will backtrack and rearrange the schedule to fill all slots.

[13]This may occur in resource rich schedules.

Regardless of how a pilot is picked by the system, no pilot who violates constraints may be scheduled. A pilot must be qualified, current, and available as specified by the schedule. A pilot who is not current or who is in mission qualification training status must fly with an instructor pilot. Qualification, currency, and availability are pasted into the criteria array as needed by the algorithm; the criteria inputs serve to limit the selection of pilots. When a pilot is picked, the need for an instructor pilot is calculated. There are three possible results: no instructor pilot is required, an instructor pilot is required and there is at least one who is available and current to fly, or an instructor is required, but none are acceptable because of availability or currency. The first and third results are easily handled by scheduling or disallowing scheduling. The second case, required instructors exist, must be handled more carefully because the algorithm does not know (remember) the status of scheduling done so far and must check. If the schedule slot where the instructor pilot is needed has not yet been filled, the algorithm can upgrade the required qualification for that slot and fill the slot below with the non-current or underqualified pilot currently being considered. However, if the associated instructor pilot slot has been filled, the algorithm must check to see if the pilot filling it is an instructor. If so, the algorithm will place the non-current or underqualified pilot. Otherwise, it discards the non-current or underqualified pilot, just as if no acceptable instructor pilot could be found, and selects another pilot for consideration. The dual interpretation of instructor pilots as pilots (jobs) and resources adds to the complexity of the problem and makes handling it in this procedure the preferred way of scheduling.

If an available pilot qualifies for a mission and is supported by an instructor pilot (if required), the algorithm updates the pilot's availability and places him on the schedule with a comment explaining the choice. If the pilot was scheduled based on a specified priority, the priority list is commented to communicate success and so that the pilot will not be selected from the list again.

There is an analogous routine for removing a pilot from the schedule. Operations performed include clearing the pilot slot and comment, resetting any upgraded qualification from the schedule, freeing the allocated time from the pilot availability data, and clearing any associated comment from the priority list.

Routines for completing or resetting an entire schedule reuse the smaller routines. For example, to automatically make a schedule, the *'make schedule'* routine finds the last slot on the schedule and, proceeding upward, fills all unfilled slots with the most appropriate pilot using *'find pilot.'*

There is a routine to update the availability of pilots who are busy with duties not including flying (DNIF, e.g., medical, ground training, a staff job, etc.). A corresponding routine for freeing DNIFed pilots is also implemented.

Several routines have been written to display different views of the data. One shows the schedule, another prints the schedule, and three others display the three projections from the consolidated database: pilot qualification, events remaining, and currencies. A flexible graphics routine allows the user to specify up to four attributes from the consolidated database to produce graphs in a number of standard formats. (See Figure 8.) Another graphics routine produces a time line or pilot availability. These graphs open as windows and may be left open, although their continuous recalculation slows execution.

Pilots may be added or deleted without distorting the data structures or affecting the algorithms in any way. Similar routines could be written to add or delete training events, but at this time, that process is manual.

Data updates are currently done using serial debriefing and data propagation routines. There are important issues yet to be addressed concerning the uncertainty in the database, i.e., expected versus real data. These issues are discussed below.

Additionally, there are no routines or data structures to support check rides or variable currency values (30 days assumed for all events) in this demonstration prototype.

The Interface. On the Macintosh, Excel's interface is the standard pull down menus. The aircrew scheduler prototype retains these menus and adds two additional menus, Schedule and Pilot_data. All user functions described are available from mousing on the different menu choices. (See Figure 10 for functions available on menus.)

Using the Excel Scheduler. As stated, the Excel scheduler has been developed to provide the scheduling officer with real-time data, analysis, and flexible scheduling support. In practice, the scheduling function and data updates will be used most because automated scheduling reduces the need for specific data and analysis is needed infrequently. Data updates include initial development of the schedule, insertion of a priority list, printing the final schedule, replying to the priority list sender, and posting updates to the database resulting from flying the schedule. For the purposes of discussion, it is assumed that the Excel prototype and supporting software exists on a local area network (LAN) of Macintosh IIcx hosts (a LAN of different hosts is possible).

The scheduling officer typically schedules two weeks in advance. For each day, the scheduling officer must develop a partial schedule manually (as is done currently) and insert it in the appropriate format in the schedule data file. Schedule development on Excel's tables is a natural analog to filling out the current schedule template form. The level of automation support provided for this operation is similar to using a word processor,

instead of paper and pencil. The training officer or other supervisor will have free access to pilot data and is expected to develop a priority list in an Excel data file to be sent electronically, over the network, to the scheduling officer's computer. The scheduling officer pastes the priority list into the appropriate day's schedule data file which is now ready for scheduling.

Ordinarily, the scheduling officer might choose to let the system generate a straw man schedule by itself. (See Figure 9.) The straw man schedule may be perfectly acceptable, or the scheduling officer may choose to rearrange it or try unassigned pilots in place of the prototype's picks. The prototype's *'remove pilots'* and *'place pilot'* routines support the scheduling officer by updating data automatically and constraint checking changes. The scheduling officer may also use the consolidated database to find pilots meeting certain criteria. Why might the scheduling officer want to schedule manually? The scheduling algorithm used by the prototype is a kind of default which ordinarily works well. However, the scheduling officer may have additional information which impacts the schedule or simply know a better way to schedule. The Excel prototype, through simulation, clearly shows the scheduling office what it is doing and explains its choices. This visual feedback makes the program more understandable; it may help the scheduling officer discover cases which are not supported and suggest alternative algorithms. Because the algorithm requires less than 300 lines of code, the scheduling officer may even choose to implement changes himself.

Once the schedule is completed, it may be printed out directly and posted. The priority list, having been commented as to who is scheduled, may be sent back to the training officer or other supervisor. Using the appropriate commercial network software, sending files across the network equates to dragging an icon into a folder.

When pilots return from flying the schedule, it is their responsibility to provide data regarding which training events they completed. Normally, completed training events differ slightly from scheduled ones. The current prototype does not support anything but sequential updates of the data using the same scheduling data file as the scheduling algorithm uses. A software upgrade is planned to improve the debriefing process. The scheduling officer is given the opportunity to review a pilot's data before inserting it into the database (this is done currently in practice). An improvement on this manual check would be an automatic constraint check using knowledge from the applicable regulations, but the current prototype does not support such a check.

When analysis is required, the scheduling prototype allows the user to put multiple graphs on the screen and can calculate average events remaining by a single standard function. These analysis tools are helpful in determining the type of missions which are needed in the future and what resources are required to fly those missions. Pilots may be analyzed in relation to their peers as to their training progress; deviations of data may be quickly highlighted. Graphs may be printed out or used to generate slides for briefings.

Constraints on Operations. The interface allows direct editing of displayed data, so any errors may be corrected easily. However, this wide-open, unprotected interface can lead to data inconsistencies and wild algorithms. Excel supports dynamic protection of data, but the current prototype uses standard programming techniques, rather than cell protection, to put constraints on operations. For example, the *'place pilot'* routine will not execute unless it is over a slot on the schedule. As another example, unDNIFing a pilot reports an error if the requested time block was not previously allocated.

Uncertainty in Schedule Development. A current development task is the handling of uncertainty in pilot data during different times of the schedule life cycle. At some point,

generally yesterday, the pilot data should be complete, up-to-date, and true. (This is assuming all pilots debriefed on time.) However, schedules are made for two weeks in advance. It makes more sense to schedule based on an expectation of what the data will be for the day the schedule is to be flown. Therefore, future scheduling data files are created with the database updated for expected events. When actual data are obtained, data inconsistencies must be resolved. Resolving inconsistencies is not difficult, but has not yet been implemented. Currently, expected data and its repair is a manual process.

Scalability. The current prototype database has 3 events and 14 pilots. A typical A-10 squadron may have 30 pilots and must track at least 17 different events. The prototype generates an eight-slot schedule in just over three minutes using the current database. With a full complement of pilots and events, the same Excel prototype may require six minutes to schedule eight pilots. This execution speed is considered acceptable for A-10 scheduling, but may be too slow for B-52 aircrew scheduling. (B-52 pilots alone must track over 100 events.) The Excel prototype's speed is limited by its interpreted macro language and display updates. Continuous display is very helpful for determining data structures, relationships between them, and algorithms which operate on them to produce the desired effects. Therefore, an Excel-based visual methodology may be useful for prototyping a scheduling system even if the system must ultimately be ported and compiled using a non-visual language to enhance execution speed.

Expanding the Network. Using a tabular data format is an effective way of communicating a moderate amount of critical information. A computer network with user friendly file transfer allows rapid transmission of tabular data and opens up the possibility of automating data handling and analysis. For example, a unit-wide LAN might include maintenance as well as aircrew scheduling and provide for communication between the two types of schedulers as well. A wider area network would allow for better communications

between squadrons and distant support (airfields, air refueling, etc.) Already, some of these communication links are in place, but automated data analysis has been slow in coming.

CHAPTER 6
OTHER APPLICATIONS AND IMPROVEMENTS TO EXCEL

A visual programming methodology based on Excel's spreadsheet display and program animation is useful, not only for scheduling applications, but as a general programming methodology. Of course, there are a number of improvements which, if integrated, would benefit the methodology: formalizing references, adding programming functionality, and increasing support for abstraction in a number of ways.

Other Applications. Using Excel and the visual programing methodology described in CHAPTER 3, several other applications were developed and appear in the APPENDIX. They range from a simple form generator based on functional programming to a long distance telephone data recorder which is used to record and deposit telephone data over a network to an analysis center.

A solution to the eight puzzle described by Nilsson was programmed to demonstrate the utility of visually explicit data structures. The eight puzzle is a matrix with nine positions, eight of which are occupied by a moveable tile with a unique number. Tiles are numbered from 1 to 8 and are arranged in what appears to be random order in the initial state. The objective is to arrange tiles in increasing order around the periphery of the matrix. Nilsson's first solution uses hill climbing. The hill climbing algorithm has been implemented in Excel as a short program supported by several abstract procedures. The resulting visual program displays the matrix of the eight puzzle in a concrete representation. The eight puzzle solution appears as an animated program; the algorithm used is quite easy to see.

Difficulties Associated with Programming in Excel.

Spreadsheets evolved into programming languages and databases because of user demand. The primary users of Excel and other spreadsheets are business analysts, not programmers. Therefore, the Excel programming language was developed for ease of use and power. It is not formally complete in the sense that LISP or Pascal are complete. Yet its data presentation and representation capabilities are superior. There are two alternatives to improving Excel programming capability. The first is to patch up the Excel macro programming language. The second is to replace it with a standard language such as Pascal.

Data References. In documentation and in operation, Excel often confuses the location of data with its value. This is because Excel converts references to values whenever it deems appropriate. Excel also converts types to other types when it needs to, rather than giving any error indication. This amiable behavior is nearly always appropriate. However, Excel can be inconsistent with both reference and type conversion.[14] For example, cells may be referenced by absolute address or user-specified name. However, the standard command for visiting a cell, SELECT, does not recognize user-specified names. If the macro programming language is retained, it can be improved by allowing any naming convention to be used for all instructions.

Programming Flexibility. Instead of enhancing Excel's macro language, it would be nice to be able to use a standard programming language with Excel's spreadsheet display. An already complete language like Pascal or Lisp extended to allow access to Excel's

---

[14]The Excel aircrew scheduler prototype was developed using Microsoft Excel version 1.5 rather than the new version 2.2 currently available. Version 2.2 may be more consistent in treating reference and type conversion.

spreadsheet cells would blend the best of both tools. Program animation could be retained by having the Excel portion highlight cells as they are accessed or changed, still updating changes continuously. A less ambitious environment would simply allow Excel code to call a Pascal routine which accesses spreadsheet data and transfers control back to Excel after execution. This would allow more sophisticated search algorithms to run on visual data. Excel has the capability to exchange data with an external file; if it is difficult to call an external routine from Excel directly, one could be still be run from the operating system. A third alternative is implementation of a spreadsheet interface to Pascal (not a trivial undertaking).

A spreadsheet naturally supports data structures such as tables and two dimensional arrays. Queues and stacks are projections of two-dimensional structures and are, therefore, supported as well. Excel fails to support other data structures quite so well. It takes a little creativity to implement a linked list or a tree in Excel. A LISP list would be very difficult to represent.[15] However, Excel's spreadsheet interface strikes a good balance between the appropriateness of a representation and the ease with which it can be implemented in a program. It would be very difficult to program a better way of displaying a tree visually. Until the job of programming a better visual representation becomes trivial, it is simpler to use the standard interface provided by a spreadsheet.

Abstraction. When programming, there is a need to travel in both directions along the spectrum of abstraction. When the program is not operating correctly, it is necessary to look closely at the details. Excel supports this view very well. However, to address large programming problems, one has to create abstraction barriers to avoid becoming

---

[15]It is interesting to note that visual improvements could be made to LISP as well. Consider how much easier lists would be to read if different font sizes were used for parentheses and elements based on their relative depth in the list!

overwhelmed by the details. Excel, like other programming languages, provides naming as a procedural and data abstraction tool. However, in Excel, data abstraction is not supported to the extent needed. Using LISP or even Pascal, it is possible to create very complex data structures useful for data abstraction. In particular, objects in the object-oriented paradigm are very useful for modeling complex, real world objects. Object-oriented programming is not supported well in Excel currently, although some abstraction is possible. For example, a functional result like availability is quite a cognitive leap, considering the calculation going on in the background. However, it would be nice to model a pilot as an object which encapsulates everything in a pilot tuple, as well as procedures for updating data and responding to requests for information. That is not possible using Excel because there is nothing to prevent access to cells. However, it is easy to imagine a spreadsheet which can support the encapsulation required by the object-oriented paradigm. The requirements are that cells must be able to be bound together and accessible only through a specified interface. Cells may be bound by naming them as an array. By arranging object-arrays physically like nodes off a bus, a program can be written to ensure that only a single interface to each object is possible. To do so, the program must continually check to make sure the active cell is either on the bus or in an object.

Inheritance and dynamic binding are also elements commonly associated with object-oriented programming. Inheritance is not supported by Excel and would require a significant effort to implement by programming. Excel currently translates types so well that dynamic binding is not needed for primitive types. User-specified types (objects) would require additional programming.

Originally, Excel version 1.5 was used to develop visual programs. Version 1.5 failed to provide enough abstraction support: visual abstraction. Every detail of the program was animated. It is desirable to hide details, again suppressing them when

abstraction is desired, and looking at them when details become important. There are three approaches which come to mind. First, a small amount of hidden memory which act as registers in a microprocessor might be included. These registers would take the form of additional buffers for copy and paste operations. A second alternative is to use higher - level procedures which combine visit-copy and visit-paste (get and put, or perhaps, get&put). The third possibility is to allow the programmer to specify which modules are animated. The newest Excel (version 2.2) has the capability to turn off screen updates during macro execution. With this functionality, the programmer now has control over program animation.

CHAPTER 7
SUMMARY

From experience in developing the Excel aircrew scheduler prototype, a number of conclusions are obvious. Further work is required to complete the prototype, but user input is required to attain full functionality.

Conclusions

Visual environments are very important to program efficiency and program development. Being able to see data and data structures makes them explicit and concrete. Coupled with program animation, a dynamic display of changing data, visual data structures improve the debugging process and support verification and validation of the program. Because Excel supports rearrangements of data and code very well, modifications in the visual environment are relatively painless.

A spreadsheet is a natural interface for tabular data. The two-dimensional array of the spreadsheet makes maximum use of the surface area of the display and serves as a good generic interface for any data structure. The array default should be abandoned only when it becomes trivial to program a better interface for a specific data structure.

A programming language with a spreadsheet interface is very useful for heuristic scheduling systems but works well for other applications as well. Because spreadsheet data structures are concrete, a visual programming language based on one is very useful for prototyping and for teaching people about data structures. For example, an assembly

language programming class could benefit from using a spreadsheet to model a microprocessor in operation.

When programming, there is a need to work near both ends of the abstraction spectrum, either focusing on concrete representations or viewing large parts of the program abstractly. The visual programming language suggested in CHAPTER 4 supports examining details, but does not support multi-level flexibility when it comes to program animation. Improvements made to Excel version 2.2 allow the programmer to switch animation on and off, improving the visual programming process considerably through suppression of unwanted detail. Object-oriented programming might be useful in a visual programming environment, but is probably too difficult to implement using Excel.

Future Work

The current aircrew scheduling algorithm is relatively complete and flexible. However, it fails in the cases which the scheduling officer has the most difficulty with: resource-rich schedules. When there are too few pilots, the current algorithm schedules **NO PILOT** instead of checking the schedule to see if its greedy choices might be rearranged to complete the schedule. Work is continuing to fix this problem.

Another on-going effort addresses the generation of expected future data and resolution of expected future data with actual data. No research is required to address this problem; it is an implementation issue only. However, research is required to develop an expert system for constraint checking pilot debriefing inputs.

The aircrew scheduling prototype is an example of very specific application software designed to solve a very specific problem. Unfortunately, specific applications

require knowledge and expertise which is available only from the user. Successful development of a specific application requires involving the user up front. Therefore, the next step in improving the current aircrew scheduling prototype is to ask for user support in testing and evaluation of the software. Without user feedback, it is easy to design software which does not quite solve the user's problem.

BIBLIOGRAPHY

[1] Abelson, Harold, and Gerald J. Sussman. Structure and Interpretation of Computer Programs. The MIT Press, Cambridge MA, c. 1985.

[2] Ambler, Allen L., and Margaret M Burnett. "Influence of Visual Technology on the Evolution of Language Environments," Computer, October 1989, pp. 9-22.

[3] Bic, Lubomir, and Jonathan P. Gilbert. "Learning From AI: New Trends in Database Technology," Computer, March 1986, pp. 44-54.

[4] Blaha, Michael R., William J. Premerlani, and James E. Rumbaugh. "Relational Database Design Using an Object-Oriented Methodology," Communications of the ACM, April 1988, pp. 414-427.

[5] Bourne, David A., and Mark S. Fox. "Autonomous Manufacturing: Automating the Job-Shop," Computer, September 1984, pp. 76-86.

[6] Brown, Gretchen P., Richard T. Carling, Christopher F. Herot, David A. Kramlich, and Paul Souza. "Program Visualization: Graphical Support for Software Development," Computer, August 1985, pp. 27-35.

[7] Bruno, Giorgio, Antonio Elia, and Pietro Laface. "A Rule-Based System to Schedule Production," Computer, July 1986, pp. 32-39.

[8] Chow, We-Min, Edward A. MacNair, and Charles H. Sauer. "Analysis of manufacturing systems by the Research Queueing Package," IBM Journal of Research and Development, July 1985, pp. 330-342.

[9] Engelke, H., J. Grotrian, C. Scheuing, A. Schmackpfeffer, W. Schwarz, B. Solf, and J. Tomann. "Integrated Manufacturing Modeling System," IBM Journal of Research and Development, July 1985, pp. 343-355.

[10] Fox, Mark S., and Bernard Nadel. Tutorial notes entitled "Constraint Directed Reasoning," from the Eleventh International Joint Conference on Artificial Intelligence, given Monday, August 21, 1989.

[11] Georgeff, Michael P. "Planning," Annual Review of Computer Science, 1987, pp. 359-400.

[12] Gershwin, Stanley B., Ranakrishna Akella, and Yong F. Choong. "Short-term production scheduling of an automated manufacturing facility," IBM Journal of Research and Development, July 1985, pp. 392-400.

13 Gries, David. The Science of Progamming. Springer-Verlag, Inc., New York, c. 1981.

14 Haines, C. L. "An algorithm for carrier routing in a flexible material-handling system," IBM Journal of Research and Development, July 1985, pp. 356-362.

15 Halbert, Daniel C., and Patrick D. O'Brien. "Using Types and Inheritance in Object-Oriented Programming," IEEE Software, September 1987, pp. 71-79.

16 Harel, David. "On Visual Formalisms," Communications of the ACM, May 1988, pp. 514-530.

17 Helman, Paul, and Robert Veroff. Intermediate Problem Solving and Data Structures: Walls and Mirrors. The Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA, c.1986.

18 Hillier, Frederick S., and Gerald J. Lieberman. Introduction to Operations Research, Fourth Edition. Holden-Day, Inc., Oakland CA, c. 1986.

19 Jacky, Jonathan P., and Ira J. Kalet. "An Object-Oriented Programming Discipline for Standard Pascal," Communications of the ACM, September 1987, pp.722-76.

20 Jacob, Robert J. K. "A State Transition Diagram Language for Visual Programming," Computer, August 1985, pp. 51-59.

21 Klein, Gary A. "Recognitional Decision Making in C2 Organizations," a paper presented at the 1989 Symposium on Command and Control Research sponsored by the Basic Research Group, Joint Directors of Laboratories, and National Defense University, June 27-29, 1989, in Washington D.C.

22 Lakin, Fred. "Visual Grammars for Visual Languages," Robotics, 1987, pp. 683-688.

23 Lassez, Catherine. "Constraint Logic Programming," Byte, August 1987, pp. 171-176.

24 Levien, Ralph. "Visual Programming," Byte, February 1986, pp 135-144.

25 Linden, Theodore A., and Same Owre. Verification and Validation of AI Software. Technical Report prepared under US Air Force Contract F30602-88-C-0087 by Advanced Decisions Systems, available as TR-3209-02.

26 Luh, Peter B., Debra J Hoitomt, Eric Max, and Krishna R. Pattipati. "Schedule Generation and Reconfiguration for Parallel Machines," 1989 IEEE International Conference on Robotics and Automation, May 1989, Scottsdale, AZ, pp. 528-533.

27 Madhavji, Nazim H. "Visibility Aspects of Programmed Dynamic Data Structures," Communications of the ACM, August 1984, pp. 764-776.

[28] McDermott, Drew, and Ernest Davis. "Planning Routes through Uncertain Territory," Artificial Intelligence, Volume 2, 1984, pp. 107-156.

[29] Melamed, B., and R. J. T. Morris. "Visual Simulation: The Performance Analysis Workstation," Computer, August 1985, pp. 87-94.

[30] Microsoft Excel Arrays, Funtions, and Macros (for the Apple Macintosh). Microsoft Corporation, c. 1987.

[31] Microsoft Excel Users Guide (for the Apple Macintosh). Microsoft Corporation, c. 1986.

[32] Nilsson, Nils J. Principles of Arificial Intelligence. Morgan Kaufmann Publishers, Inc., c. 1980

[33] Pascoe, Geoffrey A. "Elements of Object-Oriented Programming," Byte, August 1986, pp. 139-44.

[34] Raeder, Georg. "A Survey of Current Graphical Programming Techniques," Computer, August 1985, pp. 11-25.

[35] Ronen, Boaz, Michael A. Palley, and Henry C. Lucas, Jr. "Spreadsheet Analysis and Design," Communications of the ACM, January 1989, pp. 84-93.

[36] Rushby, John. Quality Measures and Assurance for AI Software. Technical Report prepared under NASA Contract NAS1-17067 by SRI International.

[37] Shu, Nan C. "FORMAL: A Forms-Oriented, Visual-Directed Application Development System," Computer, August 1985, pp. 38-49.

[38] Stark, Walter A., Jr., and Richard A. Reid. "An Operations Research Scheduling Program," BYTE, September 1983, pp. 549-579.

[39] Stonebraker, Michael, Jeff Anton, and Eric Hanson. "Extending a Database System with Procedures," ACM Transactions on Database Systems, September 1987, pp. 350-376.

[40] Wilson, Ron. "Object-oriented languages reorient programming techniques," Computer Design, Vol. 47, November 1 1987, pp. 52-62.

[41] Wittrock, Robert J. "Scheduling algorithms for flexible flow lines," IBM Journal of Research and Development, July 1985, pp. 401-412.

[42] Zhao, Liping, and S. A. Roberts. "An Object-Oriented Data Model for Database Modelling, Implementation, and Access," The Computer Journal, February 1988, pp.116-124.

APPENDIX A

The following paper provides a system description of the 1987 LISP machine/KEE version of RADC's aircrew scheduler. It is useful as background, but doesn't indicate the complexity of the KEE software.

Aircrew Scheduling: An Application of Expert System Technology

Capt Doug Dyer and Ms. Sharon Walter

Rome Air Development Center
Griffiss Air Force Base, New York

ABSTRACT

This paper describes an expert system developed by the RADC for scheduling pilots of single seat aircraft in training sorties. The aircrew scheduling problem and technologies for solving it are discussed.
A full system description of the RADC aircrew scheduler is presented, along with the algorithms that it uses. Present and planned developments are listed.

Introduction

The Rome Air Development Center (RADC) is a large Air Force laboratory responsible for research and development of command, control, communications and intelligence (C3I) systems. Artificial intelligence research is pervasive across the Center because of its importance to command and control (C2) systems. In 1986, RADC initiated the Air Force Innovative Applications program, an in-house effort designed to capture the expertise of Air Force officers in deliverable, "bite-sized" expert systems. Maj Don Henager had served as a squadron aircrew scheduler and, under the program, developed software to automate aircrew scheduling for A-10 aircraft. RADC's current aircrew scheduler is an expert system designed to assist the squadron scheduling officer of a single seat fighter squadron in his daily task of assigning pilots to a limited number of sorties in order to meet semi-annual training requirements. The purpose of this paper is to describe the aircrew scheduling problem and RADC's knowledge-based software for solving it.

The A-10 Squadron Aircrew Scheduling Problem

Globally, flying of the A-10 and other Air Force aircraft is determined by the resources available and the training requirements of the pilots. Higher headquarters dictates the total number of flying hours for a given year, based on funding allocations. Each unit, or wing, tries to maximize training within the flying hours constraint. Aircraft maintenance limits the number of aircraft available and specifies turn-around times. Munition ranges and runway times constrain sortie profiles.

A wing scheduling officer negotiates with and resolves conflicts between squadron scheduling officers and is responsible for setting the type and mix of sorties for each day. As an example, for a given day a squadron may be allotted 30 total sorties consisting of 18 weapons delivery, 4 air-to-air, and 8 instrument sorties. Each sortie type can only fill certain training requirements.

The squadron scheduling officer is responsible for assigning pilots to complete a daily schedule, like the one shown in Figure 1. The scheduler may not place pilots in sorties arbitrarily; pilot qualification, currencies, flight training requirements, ground training requirements, and availability constrain aircrew scheduling. Qualification is based on training and reflects overall experience. Qualification levels include mission qualification training, mission ready, two-ship flight leader, four-ship flight leader, and instructor pilot. Currency is the last date of completion of a particular training event and reflects frequency of training. For example, a pilot who hasn't landed in 30 days is not current for landing and must

land with an instructor pilot. Flight training requirements are specified by regulation and include many separate events designed to train pilots in all aspects of flying the A-10. Ground training events must be accomplished before flight training. Pilot availability is subject to duties not including flying (DNIF) and leave. Pilots may be placed on DNIF status for many reasons including crew rest, illness or medical appointment, TDY, staff responsibilities, or ground training. Under the crew rest concept,

214 356TFS MON

LINE	TOT	LAND	MSN	PILOT	CONFIG
601	0800	1000	WD		B6I
602	0800	1000	WD		B6I
603	0830	1015	WD		B6IMP
604	0830	1015	WD		B6IMP
605	0830	1015	WD		B6IMP
606	0830	1015	WD		B6IMP
607	1200	1330	ACBT		J
608	1200	1330	ACBT		J

Figure 1. A Daily Schedule

pilots cannot fly for longer than 12 hours and must have 12 hours of crew rest after flying. Training often requires more qualified pilots to fly with less qualified ones. Two- and four-ship flight leaders are always required for A-10s, depending on the sortie. Instructor pilots are required for qualification upgrade and to bring pilots back into currency. Therefore, scheduling a less-qualified pilot to fly generally implies a constraint that a more-qualified pilot must also be available to fly.

The current manual method of daily aircrew scheduling is tedious, time consuming and error-prone. The squadron scheduling officer typically builds daily schedules two weeks into the future and revises them as needed when the scheduling situation changes. The scheduler often receives a priority list from the a squadron supervisor (the supervisor may be privy to special information, such as leave or TDY plans, for example). If possible, the scheduler will place pilots from the priority list on the schedule or may negotiate with the supervisor to remove pilots from the priority list. The scheduler then completes the schedule by essentially adding pilots to the priority list and placing them on the schedule. The scheduler makes every attempt to maintain currencies and provide opportunities for pilots to achieve training events. Currency and training event data are provided to the scheduler in the form of hard copy reports from a centralized database called AFORMS. Using the reports, the scheduler selects pilots who will go out of currency soon or pilots who are getting behind in training. The scheduler also tries to fly those pilots who are attempting to upgrade their qualification level. An upgrade to instructor pilot is particularly appealing as instructor pilots are a valuable resource to the scheduler.

Most operations done manually by the squadron scheduling officer are analogous to the relational database operations of join, selection, and projection. Selection of pilots based on currency data, training event data and appearance on a priority list could be done automatically by a relational database management system. However, additional inference is required to take into account the constraint information contained in applicable regulations and knowledge of pilot availability.

The training and currency information from the AFORMS database is neither updated nor available in real-time. Therefore, the scheduler must remember recent information to work effectively. AFORMS stores only training progress and currency data. The scheduler must keep track of all other constraints which affect the scheduling process. In addition, for pilots attempting to upgrade or return to flying status from a staff position, AFORMS fails to record training events accomplished prior to attaining upgrade status.

The AFORMS data are used by the Training Officer, Squadron Commanders, and others in addition to the scheduler. To insure training opportunities, these supervisory officers order corrective actions based on the AFORMS data which can be incomplete and slow in coming. AFORMS is managed by a centralized base data processing center and is not amenable to local manipulation. Typically, the AFORMS data consists of less than one megabyte of information; neither storage nor processing requirements lie outside the capability of a standard Air Force microcomputer. Updates to the database are accomplished using hard copy optical scanner sheets filled out by pilots on sortie completion. Pilot claims are checked for feasibility and consistency by a review board prior to updating the AFORMS database.

Although the scheduler and others in the flying unit benefit from the data provided by the AFORMS database, there is a need for additional data tracking and faster response. In addition, data reports must be generated in a more flexible manner, allowing different users to obtain specialized reports in a timely fashion. There is no reason, for example, that briefing charts cannot be generated automatically from existing data. Finally, using database operations and knowledge-based heuristics, the aircrew scheduling process may be completely automated, thus relieving the squadron scheduling officer from about 80 percent of the scheduling effort.

Scheduling is frequently assigned as an additional duty in conjunction with flying and requires 50-60 hours per week, typically. Squadron aircrew schedulers, after weeks of training, serve only 12 to 18 months in the job before "burnout" occurs. The rapid turnover of pilots in a fighter squadron, the task complexity, and the frequent turnover in squadron scheduling officers in the Air Force makes aircrew scheduling an excellent domain in which to provide computer assistance.

Scheduling Technology

Operations research techniques have been developed to solve optimization problems. Particularly, linear programming algorithms such as the simplex method result in an optimal value of some objective function on solution. A scheduling problem may be cast as an optimization problem; the simplex method has been successfully applied to scheduling problems. Although aircrew scheduling constraints are not all linear, the problem may be modelled as a linear combination and linear programming may be applied to solve it. However, there are at least three reasons why a linear programming approach is not the best one to take for aircrew scheduling.

In practice, aircrew schedules change many times between the initial draft and flight date. Changes are required because pilots become ill, fail to maintain currency, go on TDY, or are required for a staff duty. Often the scheduler arrives at a draft using incomplete data and must revise the schedule once the actual situation becomes known. When a scheduled pilot becomes unavailable to fly, another pilot must fill the void in the schedule. The substitute pilot usually must be as qualified as the original one. Occasionally, large portions of the schedule must be rearranged to fill the void; however, it's best to keep changes to a minimum, as the pilots must be aware of and plan for the mission they are flying. The scheduling changes that become necessary in the aircrew scheduling domain are the primary reason that linear programming techniques are not suitable. Because a linear programming solution is always optimal, a void in the aircrew schedule can cause the entire solution to change. In contrast, when done manually, the schedule can stay relatively stable.

Computational efficiency is a secondary reason that operations research methods are less useful for aircrew scheduling. Casting the aircrew scheduling problem as a linear problem results in combinatorial explosion. For example, given 30 pilots to fill 8 slots on a schedule, there are nearly 236 billion

different solutions, neglecting any constraints. Linear programming techniques must be tempered with heuristics to limit the search space before being applied.

Finally, a squadron scheduling officer is generally an operational pilot with little training in computer science. The officer does not appreciate software which returns recommendations without explanation, as linear programming algorithms do. Instead, the software should be able to describe why a particular pilot was chosen or why some other pilot wasn't. The scheduling officer wants to retain control and understanding of the scheduling process.

Aircrew scheduling does not require an optimal solution, only a good one. The algorithm used for manual scheduling is adequate to provide training opportunities for all pilots. These two facts suggest that a knowledge-based approach is more suitable for aircrew scheduling than linear programming. A knowledge-based approach attempts to model the algorithm and other knowledge used by the human scheduler and to build an scheduling expert system from that model. The resulting expert system should solve problems in the same way as the human expert does.

Development of a successful expert system frequently depends on the availability of a human expert and iterative knowledge engineering to mine and refine the human's expertise. In the tactical aircrew scheduling domain, the constraints governing the scheduling process are well-known and published in regulations. Therefore, although RADC's aircrew scheduling software was developed by a human domain expert, it could have been developed using documentation alone.

Unlike linear programming, a knowledge-based method generates a robust solution, just as a human scheduler does. For example, small changes in pilot availability will normally result in small changes to the schedule. In addition, the expert system developed from the manual scheduling model avoids searching the large solution space. Instead, a few heuristics guide the search to a nearly optimal solution in just a few seconds. Another advantage of a knowledge-based approach is in the presentation and explanation of information. Expert systems are a spin off from artificial intelligence research; they are typically rich in friendly man-machine interface features including windows, mouse input, and inference chain dumps. With a little extra programming, the inference chain may be used as the basis of a good explanation capability, as it has been in the RADC aircrew scheduling software. The level of human expertise involved, the size of the knowledge base, the need for an intelligent search mechanism, and the large number of disjoint and interrelated constraints posed by the domain makes it appropriate for expert system application.

Object-oriented programming is a programming methodology for modelling real world objects. The objects are represented as active data modules which communicate by message passing. Classes of objects may be defined and objects may be defined to be instances of a class; instances of a class take on all procedures and default data values of the class. Hierarchially organized classes may inherit both data and procedures from classes above. RADC used object-oriented programming in the development of it's aircrew scheduling prototype because the methodology allows abstraction of data, rather than procedures, strictly enforces modularity, and saves coding through inheritance.

A Description of the RADC Aircrew Scheduler

The aircrew scheduling expert system was developed in-house at RADC by Major Don Henager. The software was written in KEE (Knowledge Engineering Environment) on a Symbolics 3670 Lisp Machine. The KEE development environment is an expert system shell which supports object-oriented programming and a transparent interface to the Lisp environment of the Symbolics. The combination of KEE and the Symbolics Lisp Machine provide a powerful environment for the development of artificial intelligence applications. For portability

reasons, Common Lisp and Common Windows were used; extensions to Common Lisp (other than Common Windows) and Zetalisp were avoided.

The interface to the aircrew scheduling software is through a series of windows. Pop-up windows are used to display menus of available operations. Items on the menus and on other selected objects have been programmed as mouse "hot-spots" and cause the software to react appropriately on mouse input. A mouse documentation line indicates what operations will occur when one of the three mouse buttons are pressed when the mouse cursor is resting on a hot-spot. These features make the interface very much a "point-and-click" affair. Keyboard input is required only for inputting items such as a pilot's name or a new date. Mouse input is even used to edit some items, as in the case of pilot availability.

There are three classes of scheduling operations: system administration, database, and scheduling. System administration operations allow the human scheduler to alter characteristics of the system display and to make changes to some of the scheduling criteria without modifying the software code. Those operations will not be described in this paper.

Database operations are used to store, display, and edit all information pertinent to the scheduling process. The scheduling algorithm has access to all data and propagates changes to the database as needed. Like the AFORMS database, the aircrew scheduling software stores an identifying key (a name, in this case), qualification level, and flying hours for each pilot, as well as events remaining and currency for each training event applicable to the pilot. In addition, the system stores pilot availability, flight information, the current daily schedule (minus the pilots), and the priority list for scheduling pilots. These four data items are required inputs to the scheduling system; their determination lies outside the realm of aircrew scheduling. At this time, they must be manually entered by the squadron scheduling officer, but the system has been built for easy interfacing, should the data become available on-line. As the current aircrew scheduling software is written in Lisp, the data structures used to store the data are lists. Most data is stored in a single large database and is associated with the appropriate pilot. This does not present a problem because of the relatively small amount of information required.

Pilot qualification level, events remaining, and currencies can be displayed in a tabular format, just as they are in the current AFORMS reports. However, because currencies often drive the scheduling process, currency information can also be displayed graphically to allow the scheduling officer to easily see who should be flown rather immediately. The graphical display is currently discreet, rather than analog; only two types of pilots are displayed: those who will go out of currency within a week and those who have already gone out of currency. Pilot availability is also displayed graphically and can be edited graphically using the mouse. Flight information, the priority list, and the daily schedule are all displayed in a tabular form to retain consistency with the hard copy analogs currently being used by operational squadrons.

The procedures used to update the database are tailored for the particular user (most often the squadron scheduling officer). On sortie completion, pilots can update event and currency data using a debriefing routine. The debriefing routine is displayed as a window similar to the optical scanning sheets used currently and can be filled out using the mouse. There are also routines built for the training officer to edit training or currency data. For the scheduler, ordinary editing of pilot data may be accomplished on the display screens as all data items comprise mouse hot-spots. Clicking on an item allows it to be edited. The flight data, priority list, and schedule may also be edited in this fashion. In addition, there are a several routines for handling abnormal conditions. For example, there are routines for altering the data items tracked; training regulations governing flight training change relatively

frequently. When pilots enter or leave the flying squadron, there are special routines to allocate database records for them, enter them into training, and prorate their training requirements. Prorating a pilot reduces the number of training events required to match the time remaining in the training term. Routines exist for beginning a new training term (six months in duration) or zeroing out requirements for a particular pilot. Zeroing a pilot out is necessary for pilots who leave flying status but remain in the squadron.

Some updating procedures are used to automatically propagate changes caused by the scheduling procedure. For example, once a pilot has been scheduled, the availability display shows the pilot in "unavailable" status.

The third type of operation available on the system is associated with the scheduling process itself, rather than data. Automatic scheduling consists of matching different types of sorties with pilots of varying qualifications and training needs. There are two different algorithms used based on the nature of the training cycle. During the first three months of the training term, the training requirements of the pilots have no noticeable trouble areas so the scheduling algorithm can be simpler. The simple algorithm ranks sorties in increasing order of slots available and fills them with the pilots who most need the training, taking into account any requirements for instructor pilots or flight leaders. For example, given 8 weapons delivery, 2 air-to-air, and 4 instrument sorties, the air-to-air sorties would be filled first, followed by the instrument sorties, and, finally, pilots would be assigned to weapons delivery sorties. The idea behind this simple algorithm is that air-to-air sorties are most precious, as there are least of them. This is very simplistic reasoning, as the next day may include 12 air-to-air sorties. However, during the first three months, the algorithm works well.

During the second three months of the training term, the scheduling algorithm evaluates the current status of pilots and of the squadron as a whole to determine scheduling priorities. If any individual pilot or the squadron average falls behind in an event, that event becomes a high priority for the pilot or the squadron, respectively. For example, if a pilot has 50 percent of his weapons delivery events remaining and only 16 percent of the training time remains, weapons delivery becomes a high priority event for that pilot. Alternatively, if the squadron has 20 percent of its weapons delivery requirements remaining with 16 percent of the training time remaining, weapons delivery events would become a priority for all pilots. The more complex scheduling algorithm ranks sorties according to their relative ability to fill the priority training requirements, rather than on the relative number of slots available. Those sorites that can fill the highest number of priority requirements will be filled first. Using this ranking, slots are filled as with the simpler scheduling algorithm, using pilots who require the training the most.

If the distribution of future sorties were known, the scheduling process would be much simpler. However, the distribution of future sorties is not known. Indeed, if the squadron requires additional sorties of one type, the distribution may shift to accommodate the need. However, the distribution is also influenced and constrained by other factors; therefore, the flexible algorithms above are needed. The algorithms described are based on the experience and knowledge of an expert human aircrew scheduler and have proven to give robust, acceptable solutions in near real time.

After ranking the sorties, the aircrew scheduling system assigns pilots to slots. If possible, all pilots on the priority list are scheduled. The rationale for doing this is that the squadron supervisor should have the same level of control over the software scheduling system as he does over the human scheduler. Currencies are the next criteria, and training needs are considered last. The criteria order may be altered without programming to match the scheduling philosophy of the particular squadron.

The actual scheduling procedure is accomplished using rules as a means of disqualifying pilots from sorties. The sample rule shown below would disqualify a pilot from a sortie for which he is not available:

```
(IF
  (AND
    (THE TOT OF ?SORTIE IS ?START)
    (THE LT OF ?SORTIE IS ?END)
    (NOT (AVAILABLE-PILOT ?
            PILOT ?START ?END)
       (INVALID-SOLUTION ?PILOT
?SORTIE)
```

A pilot must pass through the entire gauntlet of disqualification rules before being assigned to the sortie. As the system progresses through the daily schedule, it builds a list of unused partial solutions. In the course of assigning pilots to sorties, the system may arrive at a point where the remaining sorties cannot be assigned. Instead of backtracking to the starting point, the algorithm will look for alternative solutions on the unused partial solution list.

Once an aircrew schedule has been generated, the scheduling officer can click on any pilot and receive an explanation as to why the pilot was selected (pilot names are also mouse hot-spots). This feature is extremely important because scheduling officer often need to be able to explain their choices to scheduled pilots and supervisors. Moreover, schedule lines may be added or deleted on the display screen to reflect, for example, when an scheduled aircraft must undergo maintenance instead of flying.

The squadron scheduling officer may use the aircrew scheduling system in a semi-automated mode as well. For example, the officer may change a particular pilot on a generated schedule. The system will automatically check the candidate pilot and the entire schedule for constraint violation. If the new pilot violates constraints, the system will report the violations, but it will not rudely remove the pilot. The scheduling officer can use the system in this way to manually generate a schedule, if desired. If two pilots appear on the schedule, the scheduling officer can cause them to swap positions. In addition, there are special database operations available on the scheduling window. With these operations, the system can find all pilots who satisfy a given criterion. For example, all two-ship flight leaders who are current in weapons delivery may be displayed. Also, the priority list may be displayed and edited.

Future Development

The RADC aircrew scheduler prototype has been demonstrated to many high-level DOD members and has been enthusiastically reviewed. However, its current hardware and software requirements are too great for an operational unit to afford. Parallel efforts are ongoing at RADC to transport the functionality of the system to an Intel 80386-based personal computer and an Air Force-standard Zenith Z-248. Those developments are expected to be complete by 1990. The software will be modified to include reasons for non-availability, such as leave and TDY.

The artificial intelligence technology required for well-defined scheduling is generic enough to be applied to many other types of scheduling problems that exits in the Air Force and other DOD services. For example, aircraft maintenance and air refueling schedules can be automated using knowledge-based techniques. These types of scheduling problems are also being considered by the Air Force Innovative Applications program. It is hoped that the expertise gained from treating these problems may be formalized into a generic "language" for scheduling.

The data which serve as inputs to the aircrew scheduling problem seem to be supplied and used in a distributed fashion. Neither processing nor storage of the data requires anything more powerful than an 80286-based personal computer. Local storage of data would enable less constricted data flow and greater flexibility in the use of the data. Moreover, different automated tools for scheduling and database operations are anticipated. For example,

pilot claims could be screened automatically for feasibility and consistency before being reviewed manually and inserted into the database. Therefore, the ideal architecture for scheduling in an operational squadron is probably a local-area network (LAN) of personal computers. RADC is currently developing a LAN test bed for different schedulers to operate in a cooperating fashion.

References

Henager, Donald E. Unpublished notes. 1986-88.

Hillier, Frederick, and Gerald Lieberman. Introductions to Operations Research, Holden-Day, Inc., Oakland California, 4th ed., c. 1986.

APPENDIX B

The following paper discusses initial ideas related to a visual programming methodology using Microsoft Excel and describes a currency-driven aircrew scheduler developed in the summer of 1989 at RADC. The current Excel prototype uses two additional scheduling drivers to arrive at better schedules that expand solutions to more of the feasible space, in addition to providing many more user functions. However, the currency-driven scheduler demonstrates how quickly a simple scheduler may be built using the visual programming methodology based on Microsoft Excel. The currency-driven scheduler required only 120 lines of code

AIRCREW TRAINING SCHEDULER: AN EXPERT SYSTEM APPLICATION USING VISUAL PROGRAMMING LANGUAGE

Capt Doug Dyer
Lt Jennifer Skidmore
Stephen Platis

Rome Air Development Center
Griffiss AFB, New York

Introduction

The difficulty with most expert system programming problems is that not only is building an application complicated in itself, but also the complexity of the computer system discourages the operational user, who is usually a novice programmer. What is needed is a simple programming system so the domain expert doesn't have to be a computer expert. Although high-performance architectures allow for greater flexibility and speed, their complexity adds undesirable overhead to the development effort.

Some of the recent attempts to simplify software development have been expert system shells, object-oriented programming, graphical interfaces, and fourth-generation programming languages. Simple expert system shells do little more than formalize rule representation and restrict ordinary programming languages to sequential if-then-else structures. More powerful shells provide additional knowledge representation and inference alternatives at the expense of increased complexity. The object-oriented programming approach promotes the useful attributes of abstraction on data objects and modularity between objects, but this paradigm is accompanied by new languages (Smalltalk) or language extensions (C++) that are non-trivial and somewhat counter-intuitive to programmers traditionally trained on procedural languages.

Graphical interfaces are more successful at conveying information than text, and they treat the computer's output limitation very well from a user's standpoint. Unfortunately, getting graphical ouptut requires complex programming. Fourth-generation programming languages treat powerful procedures on complicated data structures as language primitives, reducing complexity by abstraction. A fourth-generation language endowed with graphics capability allows the programmer to utilize the tools provided by the higher-level language to get output in the desired form, without the need for explicit programming. One example of this kind of tool is a debugger, which displays variable values while the programmer executes a program. This type of continuous visual display has a logical limit that has been known to business users for years as a spreadsheet. A spreadsheet consists of a large, two-dimensional array of cells which may contain constants or variables (i.e., the results of a formula). These cells are continuously calculated and displayed, an effective way of treating the computer's shyness about output and avoiding complex graphics programming. Although spreadsheets began life as "what-if" tools for business users, they quickly expanded into database management systems and added the flexibility of programming through macros. Current spreadsheets, like Microsoft Excel, feature powerful functions and utilities, placing them in the category of fourth-generation languages.

Although spreadsheets are not often thought of as programming languages, spreadsheets are appropriate for solving some of the problems of interest to the Rome Air Development Center (RADC). In particular, well-defined, constraint-based scheduling problems (among others) are easy to solve using a spreadsheet. As part of an aircrew scheduler development, RADC has developed an innovative programming methodology based on the

use of a spreadsheet as a visual programming language. Not all of the desirable properties of the visual programming language we envision are embodied in current spreadsheets. However, for many problems, the advantages of using a visual programming language far outweigh the shortcomings of current spreadsheets. We consider visual programming methodologies to be important in reducing programming complexity, especially for novice programmers.

We devote the first portion of this paper to the advantages of visual programming languages over conventional higher-ordered languages such as Pascal, C, or Ada, and then admit some disadvantages. The second portion of the paper is a system description of the particular application of interest, a daily aircrew scheduler which assigns pilots to training missions. The scheduler is a prototype expert system which was ported from a LISP machine in the KEE 3.1 environment. The new prototype currently does not support the entire functionality of the KEE version, but has been implemented by novice programmers in only three weeks and a few hundred lines of code. This represents a code reduction of at least one order of magnitude for a complete port.

Advantages of a Visual Programming Language Over Conventional Languages

We have found this portion difficult to write, as many of the advantages are subtle and they range from matters of convenience to cognition. In the discussion that follows, we do not differentiate much between what we call a visual programming language and our current spreadsheet, Microsoft Excel, except for Excel attributes which are clearly not associated with the visual nature of the tool.

When developing a program using Pascal or C, the programmer must iteratively determine the need for a variable in the program and allocate it in a type declaration. This is because these languages efficiently guard memory resources. Each variable type is allocated the least amount of memory that it needs, so the particular type is important and must be declared by the programmer. Swapping from the program to the declarations block is distracting to programmers, but useful for runtime efficiency. Languages like LISP relax the need for strong typing and dynamically allocate memory. Spreadsheets go even further, by pre-allocating memory into a two-dimensional array (most convenient for visual display on a two-dimensional screen). Visual programming languages also allow any data type or even a formula (functional procedure) to occupy a data cell. Furthermore, cells already have their own names, i.e., A1, J45, etc. Moreover, assigning values to data locations is at least as simple as for conventional languages.

Also when using Pascal or C, if output is desired, output must be programmed. A program that provides no output has no value, yet the programmer must go to special lengths to obtain output. Visual programming languages supply continuous output in a two-dimensional window stream without programming, which is clearly a more sensible approach for data-centered problems. When executing a conventional program, special display code (or a debugger) is required to discover values of pertinent variables for program verification or debugging. The continuous calculation and display of a visual programming language makes this extra code unnecessary.

From a cognitive standpoint, visual programming languages are superior. The continuous display of data relieves programmers from having to remember variable names and meanings and also provides additional (human) memory association opportunities. By default, pre-defined names for data cells are displayed and can be mentally derived by projection (e.g., A1 is the cell in column A and row 1). If the meaning of the data cell needs commenting, the comment may occupy an adjacent cell. Furthermore, the cell can be referred to with a user-defined name, just as variables in a conventional programming

language are. User-defined names act as aliases for absolute addresses (e.g., A1). The two-dimensional display also has a more subtle advantage: data exists in planar space. That is, each cell is in a definite location relative to other cells. A programmer knows a cell both from its name and from its location on the display. Finally, data structures in conventional programming languages are invisible, abstract, and seemingly unrelated to one another in space; programmers often draw individual data structures to conceptualize them (for example, linked lists). Both of these issues are addressed by visual programming languages. Although limited by current spreadsheets to rectangular arrays, the data structures of a visual programming language are displayed by the language. They are quite concrete. Furthermore, because of the two-dimensional space, data structures are physically related as well. Conceptually related data may be placed physically close together, if desired. The physical display of data also works for the display of the program source code. Typical conventional languages are edited essentially as linear strings. Despite structured programming, one procedure follows another. This is not at all true for the two-dimensional display area of a visual programming language. Although each procedure (macro) occupies a linear column of cells, a second procedure can be placed north, east, south, or west of the first. By using a two-dimensional programming area, a higher degree of structure is added, and the advantages of structured programming are amplified as a result.

Visual programming languages support data abstraction. A cell may be data or a functional procedure, in which case the result is continuously calculated and displayed as data. Each cell is similar to an object in the object-oriented paradigm, although the interface is rather open and only one "method" may be stored inside. This is a limitation of the currently available spreadsheets; new "three-dimensional" spreadsheets might be better as visual implementations of object-oriented languages.

As a general rule, languages should be extensible. Current spreadsheets support extensibility of their fourth-generation capabilities by supporting user-defined functions. In Excel, these are called "functional macros" and differ conceptually from the "command macros" which are executable programs. Functional macros are applicative, rather than procedural.

In terms of artificial intelligence programming (and many others domains as well), it is often difficult for the knowledge engineer to know what data is relevant and what relationships exist between different data sets. Using a visual programming language encourages the programmer to place data in displayed data structures before writing procedures. The visual display of data structures seems to help in defining what procedures are possible and determining the relationships between data. The idea of throwing lots of data onto the computer screen without much regard for its relevancy is similar to the approach chosen by many neural net programmers. It is not costly to either type of programmer to use this approach, and the data that turns out to be unimportant can be thrown away later.

Because data structures in spreadsheets are two dimensional arrays, relative addressing is used much more frequently to access data than for other languages. For example, it's common to go "one column over and two rows down." This characteristic is more useful than absolute "by name" addressing when the data structure must be modified, because not every cell has a user-defined alias. All current spreadsheets recognize the need for data array changes; relative addressing is the default operating mode. Data structures may be cut and pasted, and the system updates references to them automatically.

Command macros, the programs of spreadsheets, typically access data on a spreadsheet by "visiting" the cell where the data is located. The macro language can use the data to make

calculations and copy the results into another visited cell. This is exactly what happens in any programming language, but with a spreadsheet, the process is displayed visually, rather than being kept invisible. For example, Excel's active cell is displayed as a colored outline. As a macro executes, the active cell indicator moves around on the screen. Debugging is much simpler when every step of a program's process is displayed. With a visual programming language, there is never any need to write test procedures. In the particular case of Excel, macros may be single-stepped which is a good debugging feature.

Command macros can be executed by other macros. This feature encourages procedural abstraction and code modularity. Although programs like C and Pascal also have this feature, the two-dimensional placement opportunities and loose data typing of a spreadsheet encourage programmers to break up the program properly.

Spreadsheets have many primitive functions which are really sophisticated procedures, qualifying them as fourth-generation tools. Among these primitives are mathematical, statistical, database and date functions. In addition, advanced graphing utilities are included. Excel features custom menus and dialog boxes, making user-interfaces easy to construct. These features are not unique to visual programming languages, but they are a useful aspect of all spreadsheet systems.

A Few Disadvantages of Visual Programming Languages

Visual programming languages have an inherent processing overhead associated with display and continuous calculation of formulas in cells. However, it is reasonable to trade runtime execution speed for programming benefits in an era of faster hardware. Also, in our experience, a visual programming language is useful as a prototyping aid, even if the eventual product will be coded in an efficient language such as C. We initially tried to port the KEE aircrew scheduler directly into C code, but development was slow because relationships between data were unclear. Now that a spreadsheet prototype exists, we feel confident that a C version of the prototype could be quickly coded.

For many problems, a two-dimensional array is not the best data structure to use. The idea of a visual programming language doesn't preclude other data structures, but current spreadsheets tend to discourage them. For example, current spreadsheets treat single cells (variables) or arrays as default data structures. A linked list is not terribly difficult to implement, but a tree structure might be. New three-dimensional spreadsheets certainly offer other possibilities.

Designed as business tools, current spreadsheets are not as rigorous as a visual programming languages should be. For example, there should be a clear distinction between data meaning, value, and address. Our current Excel version frequently uses "reference" to mean either address or value. In addition, control and branching constructs require more attention. Spreadsheet developers seem to be cleaning up thier products; newer versions are reported to formalize macro languages and fix irregularities.

Finally, the database utility of spreadsheets could be enhanced by adding a join operation. Selection and projection operations are currently supported. The addition of join would make spreadsheets adequate as relational database management systems.

A Description of the Daily Aircrew Scheduling Problem and the Scheduler Prototype

Single-seat aircraft aircrew scheduling is typical of well-defined, constraint-based scheduling problems. Briefly, an aircrew scheduling officer in an operational squadron must complete a schedule like the one shown in Figure 1 by filling in appropriate pilots. Constraints include pilot qualification, pilot availability, training event requirements, and event currencies. Pilot qualification ranges from Mission Qualification Training through Mission Ready, 2-Ship Flight Leader, 4-Ship Flight Leader, and finally, Instructor Pilot. These values are mapped into the numbers 1-5 in the database for convenience in manipulation (See Figure 2). Pilot availability is subject to having been previously scheduled to fly or a number of Duties Not Including Flying (DNIF). Typical DNIF are things like medical reasons, leave, TDY, ground training, or staff duties. Each training term, pilots must complete a certain number of training requirements for each mission or event type (See Figure 3). Weapons delivery (WD), air combat training (ACBT), and other types of training sorties give pilots training opportunities for different events. For most events, pilots are required to maintain a currency; for example, a weapons delivery mission must be flown every 30 days to maintain currency (See Figure 4). Pilots who go out of currency in an event must have an instructor pilot fly the same mission along with them the next time they fly that event. Instructor pilots are required for any pilot in Mission Qualification Training and for any qualification upgrade, as well.

Training event requirements and currencies act as drivers in the scheduling process, while currencies, qualifications, and availability constrain scheduling. Currencies are particularly important because instructor pilots are a valuable resource to the scheduler. Therefore, it is important to schedule pilots who will soon go out of currency before an instructor pilot will have to be scheduled to fly with them. It is not immediately necessary to schedule non-current pilots since they will have to fly with an instructor pilot anyway, but they have to be scheduled sometime or they will be unable to complete training requirements. All other things being equal, pilots requiring the most training events should be scheduled first.

Before scheduling pilots, it is necessary to decide which the mission type should be scheduled first, as filling these slots will impact pilot availability. Our current prototype doesn't consider future availability of training missions. Instead, it schedules the scarcest mission type first. Algorithmically, it fills the schedule from the bottom up, the idea being that the scheduling officer should place the missions on the schedule in order of increasing scarcity. Although this method is simplistic, it is used by many squadrons, particularly in the early months of the training term. Future sortie types are not completely constrained and are often not known. However, more rigorous algorithms are possible. For example, our KEE scheduler uses scarcity initially, but then switches to an algorithm based on assigning individual and unit mission priorities which are based on training events and time remaining.

While the above description of the scheduling process is admittedly simplistic, it is sufficient for describing the scheduling process and the visual programming methodology used to construct the aircrew scheduling prototype. Additional elements of the aircrew scheduling problem are contained in [1].

The data used by the prototype consists of the contents of Figures 2-4 and is consolidated in the prototype into one database (See Figure 5). Below the consolidated database, there is a row containing the same attribute headings as the database. This row and the row below it make up a criteria array which essentially is the query specification for selection in the database. By editing the criteria, different rows from the database will be returned when a selection is requested. For example, the Availability criteria is "= [nothing]," meaning that a selection will return any row with a blank value for Availability. Just under

1-Aug-89						
Line Number	Take off time	Landing Time	Mission	Pilot Required	Pilot	Confiuration
100	800	1000	ACBT	>=4		J
101	800	1000	ACBT			J
102	800	1000	ACBT	>=3		J
103	800	1000	ACBT			J
104	830	1015	DACBT	>=3		B61MP
105	830	1015	DACBT			B61MP
106	1200	1330	WD	>=3		B61
107	1200	1330	WD			B61

Figure 1. Daily Schedule

PILOT	Qualification	Availability
Able, Adam	5	
Baker, Barr	5	
Charlie, Chu	5	
Dingo, Dave	5	
Edwards, Er	4	
Frank, Fred	4	
Gonzo, Greg	4	
Harris, Har	2	
Iggy, Ian	2	
James, Jim	2	
Kee, Ken	4	
Lint, Larry	1	
Mason, Mike	1	

Figure 2. Pilot Qualification and Availability

Event Requirements			
PILOT	WD	ACBT	DACBT
Able, Adam	7	10	9
Baker, Barry	3	14	8
Charlie, Chuck	10	9	8
Dingo, Dave	23	9	3
Edwards, Eric	14	3	9
Frank, Fred	1	18	4
Gonzo, Greg	6	11	6
Harris, Harry	17	4	20
Iggy, Ian	9	19	1
James, Jim	11	20	15
Kee, Ken	9	7	20
Lint, Larry	22	0	13
Mason, Mike	10	13	9

Figure 3. Training Events Required

Event Currencies - Days Remaining			
PILOT	WD Cur Days	ACBT Cur Days	DACBT Cur Days
Able, Adam	10	12	8
Baker, Barry	9	22	9
Charlie, Chuc	18	20	9
Dingo, Dave	29	27	28
Edwards, Eric	1	11	13
Frank, Fred	16	18	21
Gonzo, Greg	-6	-24	-21
Harris, Harry	4	25	2
Iggy, Ian	7	16	11
James, Jim	18	4	16
Kee, Ken	11	0	8
Lint, Larry	-34	-2	-30
Mason, Mike	22	9	11

Figure 4. Remaining Currency Days

PILOT	Qualification	Availability	WD	WD min days	WD max days	ACBT	ACBT min days	ACBT max day
Able, Adam	5		7	10	10	10	12	12
Baker, Barry	5		3	9	9	14	22	22
Charlie, Chuck	5		10	18	18	9	20	20
Dingo, Dave	5		23	29	29	9	27	27
Edwards, Eric	4		14	1	1	3	11	11
Frank, Fred	4		1	16	16	18	18	18
Gonzo, Greg	4		6	-6	-6	11	-24	-24
Harris, Harry	2		17	4	4	4	25	25
Iggy, Ian	2		9	7	7	19	16	16
James, Jim	2		11	18	18	20	4	4
Kee, Ken	4		9	11	11	7	0	0
Lint, Larry	1		22	-34	-34	0	-2	-2
Mason, Mike	1		10	22	22	13	9	9
PILOT	Qualification	Availability	WD	WD min days	WD max days	ACBT	ACBT min days	ACBT max day
		=						
			23	>=0	<7	20	>=0	<7
				<-10	<-10		<-10	<-10
				>=0	>=0		>=0	>=0
1-Aug-89								
Line Number	Take off time	Landing Time	Mission	Pilot Required	Pilot	Confiuration	Range	Comments
100	800	1000	ACBT	>=4		J	A	
101	800	1000	ACBT			J	A	
102	800	1000	ACBT	>=3		J	A	
103	800	1000	ACBT			J	A	
104	830	1015	DACBT	>=3		B61MP	A	
105	830	1015	DACBT			B61MP	A	
106	1200	1330	WD	>=3		B61	P	
107	1200	1330	WD			B61	P	

Figure 5. Consolidated Database, Criteria, and Daily Schedule

the criteria array is the daily schedule being filled in as the algorithm executes; its rows contain data or data abstractions to be pasted into the criteria section each time a new selection needs to be made.

During execution, the scheduler prototype selects the last unfilled mission, pastes the pilot qualification requirement in the criteria, and pastes a currency range in the criteria for the mission being filled. At that point formulas on the spreadsheet which calculate greatest training requirements are recalculated. If these values are zero (blank), that indicates that no pilot meets the current criteria, and the program relaxes the currency constraint by pasting in an alternate range. Alternatively, the program pastes the result of the greatest training requirement into the criteria, and selects the pilot specified. Next the program tests to see if an instructor pilot is required and, if so, searches for one who is available and current. If an instructor pilot is needed and none can fly, the program will not fly the unqualified or non-current pilot. Otherwise, the program marks the pilot's Availability attribute "flying" and places the pilot on the schedule. If an instructor pilot is required, the program updates the qualification requirement for the next pilot to be scheduled. The algorithm continues until all sorties are scheduled or until pilot resources are exhausted.

The algorithm is not rigorous and doesn't optimize on things like instructor pilot utilization. In addition, it currently doesn't do any backtracking to remove previously scheduled pilots to fill other slots where they might be needed more. Any manual entries made to the schedule should be constraint-checked, a feature not currently implemented. However, the algorithm is less than 120 lines of code; these additional enhancements are planned and could be implemented by almost anyone.

Additional Features of the Scheduler Prototype

Our KEE-based aircrew scheduler has a number of utility features which are desirable. Some of these features have been implemented in the new prototype, while others are yet to be implemented. The prototype allows the addition and deletion of pilots, color graphs of pilot data, pilot debriefing, automatic propagation of pertinent data throughout the database, and display of database projections. The prototype runs under Microsoft Excel, version 1.5 on an Apple Macintosh or on any IBM-compatible personal computer. The Macintosh version includes pull-down menus and plays a portion of Beethoven's Fifth Symphony on completing the schedule.

Our current scheduler lacks the ability to incorporate priorities from squadron commanders or other supervisors. Events must be added, changed, and deleted manually. Certain data structures on the prototype are sensitive to changes and should be protected by locking the spreadsheet cells. Once the scheduler design is relatively stable, a user's manual is required. All of these enhancements are planned, as is a more rigorous scheduling algorithm.

Concluding Comments

The use of spreadsheets as a visual programming language is particularly useful for a number of different problems which have some or all of the following attributes: (1) data-intensive, (2) hard to define, (3) require fourth-generation functions, or (4) benefit from data abstraction. In addition, spreadsheets are friendly for novice programmers and helpful for rapid prototyping as well.

References:

Dyer, Doug and Walter, Sharon; "Aircrew Scheduling: An Application of Expert System Technology," presented at the 1989 Command and Control Research Symposium sponsored by the Joint Directors of Laboratories and the IEEE Control Systems Society, 24-26 Jun 89 at National Defense University.

Microsoft Excel User's Guide.

Microsoft Excel Arrays, Functions, and Macros.

APPENDIX C

The following paper describes the object-oriented paradigm. It resulted from graduate seminar research into methodologies for neural network simulation, but many comments made apply to object-oriented programming in general.

On Object-Oriented Programming and Simulation

Douglas E. Dyer

Introduction

It is well known that small computer programs are easier to develop, debug, and modify than large programs are. More accurately, it is the complexity of the program that dictates how well humans can work with a it. Some very large programs are quite well understood because they are relatively simple. For example, government accounting and payroll applications are thousands of lines long, but are made up of many similar modules; accounting principles are well understood. In contrast, a much smaller expert system can elude understanding for years because the process it models is so difficult to characterize.

The complexity of programs arises from two basic reasons: the real-world system to be modeled and its coded representation. If the system is not well understood, as in the study of cognition, then a computer program which attempts to model it cannot be successful. Successful computer programs are those which exploit the power of the computer to solve problems which (1) people find difficult to solve and (2) can be programmed. The computer is powerful because it has non-volatile, expandable memory and can process information quickly and painlessly, once programmed. Unfortunately, computers lack common sense and most do not tolerate noisy input. The above discussion is not meant to imply that artificial intelligence research is a waste of time. Programming can be used as a tool for thought, for example, an aid in cognitive model development and testing.

If the system to be modeled is not well understood, the main problem is elucidating the system, and that problem lies somewhat outside of the realm of computer science.

However, complexity can also arise from the program and coding process. That is, given a well understood system, a program which models the system may still be unmanageably difficult to develop, debug, and maintain. A real-world system does not have to be very large before the complexity of its coded representation becomes overwhelming. Computer scientists have been struggling with this problem for over thirty years. Many solutions and methodologies have been proposed and accepted. The formal discipline of software engineering is dedicated to improving the process of building large programs. In my opinion, representational complexity is the fundamental problem of computer science.

Some of the developed tools and methods for dealing with complexity in computer programs have been very good. Operating systems relieved the programmer of much work by abstracting house-keeping details. Higher-ordered languages reduced the amount of code required to do a task and also work by abstracting lower level operations. Structured programming encompasses modularity and structure; modularity allows us to mentally break programs into separate pieces and structure allows us to mentally stack the pieces in a recognizable way.

Other techniques for improving programs and the programming process include proofs, compiler syntax checking, smart editors, using different programming languages, and using a single, standard programming language. (Some languages are better at certain problems than others are. Another school of thought attempts to use only one programming language to reduce fluency requirements and give

programmers a common language. Features of programming languages are now studied extensively to find strengths and weaknesses.) However, none of these tools or approaches have been shown to impact complexity like abstraction, modularity, and structure, which are clearly more fundamental ideas.

Object-oriented programming, a relatively new programming methodology, is based on a data-centered viewpoint. Object-oriented programming grew from fundamental roots: abstraction, modularity, and structure. Its four identifying elements are object encapsulation, message passing, dynamic binding, and inheritance. Those four elements can be thought of as unique extensions of abstraction, modularity, and structure.

In object-oriented programming, objects in the code represent elements (nouns) in a real-world system. This approach differs from most algorithmic languages like Pascal or C, which tend to focus on procedures. For this reason, object-oriented programming is a good methodology for handling representational complexity of data-centered models. Simulation is a one example of a problem in which real-world events are often data-centered, rather than algorithmically-centered.

This paper discusses the three traditional ways of handling representational complexity, abstraction, modularity, and structure, as well as their extensions as elements of object-oriented programming. Three of the elements of object-oriented programming, object encapsulation, message passing, and dynamic binding, will be further described in a discussion of a Scheme digital circuit simulator. Additional examples of the use of object-oriented programming in simulation will be briefly described.

Abstraction, Modularity, and Structure.

Abstraction. Because humans have volatile memory, the number of items we can mentally manipulate is limited. Only through abstraction are we able to cogitate on any but the simplest concepts. It has been theorized that only seven mental objects can be stored and recalled; the abstraction process to remember more than seven objects is called "chunking" to indicate the need to aggregate several objects as one. The idea of chunking was fundamental in breaking up identification numbers such as social security numbers and telephone numbers, but it also give insight into the need for abstraction in the programming process.

By definition, abstraction is associating a group of objects with a single "group-of-objects" idea. Often we use a label to represent the concept. As an example, the process of addition is signified by "+" when it actually is a procedure for mapping two numbers into a third.

The power that abstraction offers is that it relieves us of having to think about details which we don't need to think about and allows us to focus on the real problem. Imagine how difficult it would be to calculate the sum of two numbers if not for abstraction! If you have a calculator and want to calculate "2 + 3" you have to know how to push the buttons but not how the calculator works. You don't need to know that the calculator has to have registers, an arithmetic/logic unit, control software, data paths, and an input/output interface. At lower level, you don't need to know how to implement a gate or flip-flop; at a still lower level, you don't need to know the solid-state physics behind semiconductor behavior. Clearly, you don't care how the calculator works; that's not important. You view it as an abstract

black-box that does the addition that you need.

Using a label as a mnemonic representation of an abstract idea seems to be important for memory maintenance and mental manipulation. Once you have a name for something, you can toss it around in you head, add or delete facets from it, and relate it to other mental objects without fear of losing or misaligning it. It's yours to keep. We use labelling or naming all the time, but it is especially apparent in technical fields in the form of jargon. For example, names like "method," "message," and "package" have special, complicated meanings to those who study object-oriented programming.

Abstraction can be used to develop a higher-ordered language. A set of related abstract objects can be thought of as a language which can be used to solve problems. As an example, if you consider the set (+ - * / = {real numbers}), you get the language of elementary arithmetic. In computer science, all higher-ordered languages like Pascal are based on a set of abstracted machine language operations and routines which have been named things like "read," "write," "if," and "while-do" as well as "+," "-," etc. Object-oriented programming, as a methodology, helps generate abstract languages. The language of digital circuit simulation is one example.

Whenever the level of abstraction is not great enough to facilitate problem solving, further abstraction may be used to make a still higher level language. For example, it may not help to think in terms of the functions available in Pascal, like "read" and "write". Instead, you may prefer "input-procedure" and "output-procedure." Abstracting clusters of code in a given computer language can yield a new, higher-level language of software modules. The language of software modules is currently a goal of software engineering and is supported by object-oriented programming. (Cox calls these modules "software integrated circuits.")

There are drawbacks to using an abstracted language such as the language of computer modules. Naturally, you have to learn the language; that can involve reading a lot of documentation. Sometimes the language is not appropriate to your problem. Sometimes the language was not well thought out and, although intended to solve your problem, it doesn't do a very nice job. Sometimes the language is reasonably good, but the documentation is lacking. If the programmer doesn't trust the language, much time may be wasted by wading down into the depths of lower abstraction layers to make sure they work. Although all these problems are real, they don't detract from the power of abstraction; rather, they are implementation issues.

Gerald Sussman has said that the best solution to complex problems often looks more like layers of languages rather than pieces of code which solves pieces of the problem. In his Computer Exercises work book, there are many more examples of languages built from abstracted objects, including languages for, drawing Escher diagrams, drawing squares and triangles, and simulating space mission operations. These examples clearly illustrate that abstraction is a powerful method for controlling complexity in computer programming.

Modularity. Modularity is the practice of building walls around pieces of code and forcing different pieces to communicate only by well-defined communication channels. The idea of "wall" and "contract" in Helman and Veroff is the essence of modularity.

Modularity goes hand in hand with the notion of abstraction. Modularity is the package which binds

an abstraction and allows you to manipulate it as a whole entity. Modularity also allows different pieces of code to be structured -- you must be able to clearly define a piece before you can arrange a group of pieces.

Code is modular when the package binding it keeps it from affecting anything outside of it except through well-defined interfaces. Theoretically, modular code may be slapped in or out of a software environment by altering only the interface between modules. Code behavior can be isolated to each module; bugs are easily contained, found, and corrected.

In practice, modularity does aid in debugging and makes a program more tractable. However, the interface on each package must be explicitly specified, constructed, and documented. In addition, a good implementation requires that the code inside each package be "bullet-proof" so that it's never necessary to give up abstraction to fix an implementation detail.

Structure. The advent of structured programming in the early 1970s gave programmers a new, powerful tool for dealing with representational complexity in programs. Structured programming made "goto" a four-lettered word for programmers; it showed that the dangerous, undisciplined "goto" code could be replaced in all cases by sequential coding.

The major contribution of structured programming is that modularity is enforced. However, an important secondary contribution is that the modular pieces are placed with some relation to one another, and hence, have a structure. The structure of the modules can be rearranged to change the program -- in effect, these rearrangements and the changes which result make up the language of structure in a program. More importantly, the structure of the program is important in helping the programmer to mentally manipulate the program. This is true, or programmer's would not care whether a program were properly indented or merely a long string of statements with no white space at all.

Structure also encompasses the decision about how large to make the program modules. In practice, a certain balance between components at different levels of abstraction is important. It doesn't help to have ten high- level components and just 1 low-level one (or vice versa). Instead, it often makes more sense to use two or three higher-level components for every five to seven lower-level ones. This heuristic extends for multiple levels.

Abstraction, modularity, and structure are all somewhat related. In addition, they all serve to help the programmer combat complexity when trying to represent a model as a computer program. Modularity turns a program into individual code pieces. Abstraction gives each piece a meaning and a name. Structure shows how the pieces are related to one another. If all three are done correctly, the program can fit into the brain and make sense.

Elements of Object-oriented Programming: Object Encapsulation, Message Passing, Dynamic Binding, and Inheritance.

Object-oriented programming takes advantage of abstraction, modularity, and structure, but extends them in unique ways. In object-oriented programming, the focus is on the "object." What is an object? In most programming languages, programs consist of procedures to and from which data are passed. The procedures "do" something and are like verbs in the spoken language. In

object-oriented programming, programs consist mainly of "objects" which consist of "non-passive" data. That is, if procedures are the verbs of spoken language, objects are really more like the nouns.

What does it mean to be "non-passive?" Conventional programming languages draw a clear distinction between procedures and data. Languages like Scheme question that philosophy and treat both as first-class objects. In many Scheme programs, it's very difficult to say that a particular entity is one or the other. Even in speech, nouns rest on a scale between extremes of passiveness and action. Most humans would interpret the sequence of nouns "rock," "hammer," "recipe," and "programmer" to have an increasing implication of action.

Object Encapsulation and Message Passing. Objects are composed of data and the procedures which operate on it, as well as an interface to other objects. Objects maintain an internal definition of state. In this way, an object can be said to "know" something about itself -- it knows its own state and the operations which can change its state.

Conventional programming languages often abstract procedures primarily and data structures secondarily. That is, the abstract data type has relevance only in context of the procedures which operate on it. In object-oriented programming, this viewpoint is reversed. It is data which is abstracted in a primary sense. Procedures which operate on an object are only a part of the object itself. This type of abstraction is unique to object-oriented programming.

The object data, procedures, and interface are all enclosed in a hard shell of encapsulation. By design, it is only through the interface that the object can communicate with other objects. In this manner, strict modularity is enforced. Instead of using a direct procedure call with data passing, the object-oriented paradigm uses message passing between objects to execute a program. A message is a request from one object to another to cause some action to take place. Control is retained by the object -- a request may be denied, and it often is if the object is not working properly. The "procedure- with-arguments" call in a conventional programming methodology seems to be a weaker kind of modularity. Message passing is a unique way implementing modularity and is an extension on "procedure call" modularity. In addition, by using messages, it is easier to create a standard interface for objects.

Dynamic Binding. Binding refers to putting data together with the procedure which affects it. Conventional languages like C and Pascal require the programmer to manage binding up front. It's up to the programmer to determine the correct data type and pass that type to a type-dependent procedure. Some languages (Ada, for example) delay binding until compilation. This approach allows the programmer to use just one procedure name for different types of data. The compiler determines the data type and selects the correct procedure to apply. Only low-level procedures are type-dependent. The advantage of delayed binding is that the programmer is allowed to change data types without changing high-level procedures. Other languages (Smalltalk) delay binding still further until run time. This adds still greater flexibility as data types may be changed on the fly. The program itself may even change the type and still operate properly. Again, low-level procedures will be type-dependent.

In terms of the object-oriented design, late (dynamic) binding offers some real advantages. With dynamic binding, all but the lowest class of object is type-independent. This is another rather unique implementation of

modularity. When objects are type-independent, they are more easily manipulated and reused. Dynamic binding does incur some overhead in terms of space and execution time; it is responsible for the object-oriented paradigm's reputation for sluggish performance. However, the added flexibility derived from delayed binding is very useful during prototyping.

Inheritance. Inheritance refers to a certain way of structuring information. It is recognized as a primary means of classifying information in humans. Inheritance is information stored in a hierarchial structure, as a tree with parents and children, and the children gain information (inherit) from their parent nodes. The "ISA" and "A-KIND-OF" hierarchies found in artificial intelligence literature are examples of inheritance. Object-oriented programming is relatively unique in using inheritance as a way of structuring a program.

Using inheritance, an object has access to data and procedures of all of its superclasses, as well as its own. For example, an object "SHERMAN TANK" might contain information about the the thickness of its armor, its maximum speed, and the size of its gun. However, "SHERMAN TANK" would also have access to general information from its immediate superclass "TANKS" which might include situations in which tanks are effective and weapons which are effective against tanks. Progressing further up the hierarchy, "SHERMAN TANK" would also have access to information from higher superclasses. If "GROUND VEHICLES" were a superclass, "SHERMAN TANK" might gain useful information from it, for example, ground vehicles can't operate in deep water.

Inheritance makes it possible for object-oriented designs to avoid storing a lot of repetitive information. This makes the code more compact and simpler. It is an effective structuring mechanism for reducing program complexity. In addition, inheritance adds greatly to the modularity and modifiability of the code. Information that is stored in only one place is easier to modify than when it is scattered throughout the program. Furthermore, inheritance is powerful; a change to information in that one place can affect many child objects at once.

The elements of object-oriented programming build upon and extend traditional methods of dealing with representational complexity. Object encapsulation treats data as the primary object to be abstracted and made modular. Message passing is made necessary by object encapsulation. Messages are abstract, weak procedures that act as requests for action by objects. Messages promote modularity because they are independent of the procedures they request. Dynamic binding enhances modularity by allowing higher-level objects to remain type independent. Inheritance increases modularity dramatically and is a structural arrangement relatively unique among programming methodologies.

Object-Oriented Programming in Simulation.

In object-oriented programming, there is often a one-to-one correspondence between real-world objects and objects in a programmed model. In a war simulation, for example, "TANKS" would represent real tanks and would be expected to model real tanks appropriately. Therefore, the object-oriented programming methodology would seem to be well suited for simulation.

The Scheme programming language is a dialect of LISP. The Texas Instruments implementation of Scheme includes SCOOPS, an object-oriented programming environment. Scheme is a small language developed

by MIT and other universities to show that a single language could treat a wide variety of different problems, including traditional and symbolic ones.

Although there are some very important differences between Scheme and Common LISP, they both dynamically check data types. However, Scheme treats both procedures and data as first-class objects. That is, procedures can be passed as arguments to other procedures and can be returned by still other procedures. This treatment necessarily blurs the distinction between data and procedure.

Abelson and Sussman present a Scheme digital circuit simulator which can be used to make elements of object-oriented programming more concrete. The SCOOPS environment was not used to develop the code, however, so the inheritance aspects are implied, rather than explicit. Refer to the Appendix for a modified version of the simulator code.

In Abelson and Sussman's circuit simulator, wires are the primary object. Other objects such as inverters, and- and or-gates and probes send messages to wires requesting various actions. Higher-level objects like half- and full- adders are made from lower level ones. An agenda and the various data structures needed for it are also implemented in the code.

Consider the wire object identified in the Scheme code as the MAKE-WIRE definition (under ;; Objects ==========). In the code, a wire knows its voltage value initially and at any time during the simulation as SIGNAL-VALUE. The data type of SIGNAL-VALUE is numeric, but that doesn't matter until runtime, because of Scheme's dynamic binding. If we choose, we can alter SIGNAL-VALUE's type without changing MAKE-WIRE. The wire object also knows the procedures that can affect its data, namely SET-MY-SIGNAL and ACCEPT-ACTION-PROCEDURE. The latter procedure propagates signal value changes to other wires to carry out the simulation. Notice that only the wire object can run either procedure. A wire must receive an external message before dispatching on it and executing whatever procedure it chooses as appropriate. Thus, the wire object is firmly encapsulated in a hard shell. The interface between the object and external objects is identified as the message handler DISPATCH. Messages received are those to get or set a wire's signal (GET-SIGNAL, SET-SIGNAL!) or add a propagation action (ADD-ACTION!). Any other message received is erroneous. Because messages only request action (as opposed to procedures which, when followed, cause it to happen), the program can be quite modular. Inheritance is not demonstrated by this code, as it was not implemented using SCOOPS; the simulator hasn't taken full advantage of the object-oriented paradigm.

The digital circuit simulator serves as a new language. Instead of using Scheme, it enables us to think in terms of wires, probes, and- and or-gates, and inverters. With this new language, we can build even higher-level abstractions like half and full adders. Although the code is really quit simple, its object-oriented design makes it very powerful. The simulator shown in the appendix served as the core of a very complex computer simulation.

Those not familiar with Scheme or other LISPs may not see the simplicity and power of the digital circuit simulator. However, object-oriented methodology has been developed for other, more conventional languages as well. As an example, Jacky and Kalet successfully developed a large Pascal program using object-oriented concepts.
There are other examples in the literature of the use of object-oriented

programming in simulation. For example, Eilbert and Salter developed a neural network simulator in Scheme. Their Scheme simulator was shown to be more effective at modifying network structure and the node updating process than simulators based on standard numerical languages. In addition, the Scheme simulator was more successful at modeling networks which are hierarchially organized. There are three major Scheme functions in the simulator: "(1) a network generator which specifies the structure and single node response of the model; (2) the network evolver, which controls activity initialization and the updating process for the network; and (3) the Monte Carlo simulator, which finds the stable states of the network and records them." Nodes in the system are the primary objects. Computations on nodes are distributed to the nodes themselves. This is fundamentally different from a FORTRAN simulator that the authors studied; the FORTRAN representation used a matrix of connection weights between nodes to calculate convergence. The authors point out that the FORTRAN program is intrinsically faster on von Neumann machines, but is also more difficult to transfer to a parallel architecture. As a neural net is a distributed processing architecture, it seems more natural to model it using an object- oriented design. The flexibility of using the more accurate Scheme representation is an added bonus. To generate a representation for a particular model, the object-oriented Scheme design only has to call the network generator which makes new instances of nodes as appropriate. Also, the updating procedures of a node and its response curve are attributes contained within the object. Local or global changes can be made very quickly without affecting the rest of the program, because of modularity. Eilbert and Salter were very attracted to an object-oriented simulator because they wanted to model neural nets that are hierarchially structured. The message-passing style of Scheme allowed them to model interactions between hierarchial levels quickly and accurately.

Larkin, Carruthers and Soper implemented a simulator of a ship's navigation system in Flavors, an object-oriented programming language often found on LISP machines. They found that object-oriented programming was ideally suited to their simulation for three reasons. First, the one-to-one correspondence between code objects and real-world objects helped make development clearer. Second, the advanced form of modularity inherent in object-oriented programming helped reduce the complexity and ease maintenance of the simulation tools developed. Finally, the structure and modularity of the resulting code made it very extensible.

Stairmand and Kreutzer built a process-oriented simulation environment (POSE) in a locally developed object-oriented programming environment (flavors) running under Scheme. The hierarchial structure afforded by inheritance is key to reducing complexity of the POSE representation of process models.

Conclusions

Object-oriented programming is an effective methodology for reducing the complexity of programmed representation of models. The focus on objects, rather than procedures, is fundamentally different from programming in a standard way using conventional languages. The four elements of object- oriented programming, object encapsulation, message passing, dynamic binding, and inheritance all extend traditional ways of dealing with program complexity: abstraction, modularity, and structure. Object-oriented programming is well suited to simulation because of these unique attributes.

Bibliography

Abelson, Harold, and Gerald J. Sussman. Structure and Interpretation of Computer Programs. The MIT Press, Cambridge MA, c. 1985.

Abelson, Harold, and Gerald J. Sussman. Structure and Interpretation of Computer Programs. (Computer Exercises) The Massachusetts Institute of Technology, Cambridge MA, c. 1986.

Cox, Brad J. Object-Oriented Programming: An Evolutionary Approach. Addison-Wesley Publishing Company, Reading MA, c. 1986.

Eilbert, James L., and Richard M. Salter. "Modeling neural networks in Scheme," Simulation, Vol. 46, May 1986, pp. 193-99.

Helman, Paul and Robert Veroff. Intermediate Problem Solving and Data Structures. The Benjamin/Cummings Publishing Company, Menlo Park CA, c. 1986.

Jacky, Jonathan P., and Ira J. Kalet. "An Object-Oriented Programming Discipline for Standard Pascal," Communications of the ACM, Vol. 30:1, September 1987, pp.722-76.

Larkin, Timothy S., Raymond I. Carruthers, and Richard S. Soper. "Simulation and object-oriented programming: the development of SERB," Simulation, Vol. 52, September 1988, pp. 93-100.

Lenat, D. B., and J. S. Brown. "Why AM and Eurisko Appear to Work," Artificial Intelligence, Vol. 23, No. 3, August 1984, pp. 269-294.

Newburger, Bruce. "Simulate any size circuit with object-oriented modules," Electronic Design, Vol. 36, March 3 1988, pp. 75-78.

Pascoe, Geoffrey A. "Elements of Object-Oriented Programming," Byte, Vol. 9, August 1986, pp. 139-44.

"PC Scheme User's Guide and Language Reference Manual, Student Edition," (Texas Instruments) The Scientific Press, Redwood City CA, c. 1988.

Stairmand, Malcolm C., and Wolfgang Kreutzer. "POSE: a Process-Oriented Simulation Environment embedded in SCHEME," Simulation, Vol. 46, April 1988, pp. 143-153.

Unpublished Class Notes from "Fundamentals of Artificial Intelligence," Air Force Institute of Technology, Fall 1986.

Wilson, Rŏn. "Object-oriented languages reorient programming techniques," Computer Design, Vol. 47, November 1 1987, pp. 52-62.

APPENDIX

```
;; SIMULATION -- DIGITAL CIRCUIT==========================
;;=======================================================

; Queue Operations ======================================

(define (make-queue) (cons '() '()))

(define (empty-queue? queue) (null? (front-ptr queue)))

(define (front queue)
  (if (empty-queue? queue)
      (error "FRONT called with an empty queue" queue)
      (car (front-ptr queue))))

(define (insert-queue! queue item)
  (let ((new-pair (cons item nil)))
    (cond ((empty-queue? queue)
           (set-front-ptr! queue new-pair)
           (set-rear-ptr! queue new-pair)
           queue)
          (else
           (set-cdr! (rear-ptr queue) new-pair)
           (set-rear-ptr! queue new-pair)
           queue))))

(define (delete-queue! queue)
  (cond ((empty-queue? queue)
         (error "Delete called with an empty queue" queue))
        (else
         (set-front-ptr! queue (cdr (front-ptr queue)))
         queue)))

(define (front-ptr queue) (car queue))
(define (rear-ptr queue) (cdr queue))
(define (set-front-ptr! queue item) (set-car! queue item))
(define (set-rear-ptr! queue item) (set-cdr! queue item))

;; The Agenda ===========================================


(define (make-time-segment time queue) (cons time queue))
(define (segment-time s) (car s))
(define (segment-queue s) (cdr s))

(define (segments agenda) (cdr agenda))
(define (first-segment agenda) (car (segments agenda)))
(define (rest-segments agenda) (cdr (segments agenda)))
(define (set-segments! agenda segments) (set-cdr! agenda segments))
(define (current-time agenda) (segment-time (first-segment agenda)))
```

```
(define (empty-agenda? agenda)
 (and (empty-queue? (segment-queue (first-segment agenda)))
      (null? (rest-segments agenda))))

(define (add-to-agenda! time action agenda)
 (define (add-to-segments! segments)
  (if (= (segment-time (car segments)) time)
      (insert-queue! (segment-queue (car segments)) action)
      (let ((rest (cdr segments)))
       (cond ((null? rest)
              (insert-new-time! time action segments))
             ((> (segment-time (car rest)) time)
              (insert-new-time! time action segments))
             (else (add-to-segments! rest))))))
 (add-to-segments! (segments agenda)))

(define (insert-new-time! time action  segments)
 (let ((q (make-queue)))
  (insert-queue! q action)
  (set-cdr! segments
          (cons (make-time-segment time q)
                (cdr segments)))))

(define (remove-first-agenda-item! agenda)
 (delete-queue! (segment-queue (first-segment agenda))))

(define (first-agenda-item agenda)
 (let ((q (segment-queue (first-segment agenda))))
  (if (empty-queue? q)
      (sequence (set-segments! agenda
                               (rest-segments agenda))
                (first-agenda-item agenda))
      (front q))))

(define (make-agenda)
 (list '*agenda* (make-time-segment 0 (make-queue))))

(define the-agenda (make-agenda))

;; Necessary Precursors
;;==========================================

(define (call-each procedures)
 (if (null? procedures)
     'done
     (sequence
      ((car procedures))
      (call-each (cdr procedures)))))

(define (logical-not s)
 (cond ((= s 0) 1)
       ((= s 1) 0)
       (else (error "Invalid signal" s))))
```

```
(define (logical-and s1 s2)
  (cond ((and (= s1 0) (= s2 0)) 0)
        ((and (= s1 0) (= s2 1)) 0)
        ((and (= s1 1) (= s2 0)) 0)
        ((and (= s1 1) (= s2 1)) 1)
        (else (error "Invalid signals" s1 s2))))

(define (logical-or s1 s2)
  (cond ((and (= s1 0) (= s2 0)) 0)
        ((and (= s1 0) (= s2 1)) 1)
        ((and (= s1 1) (= s2 0)) 1)
        ((and (= s1 1) (= s2 1)) 1)
        (else (error "Invalid signals" s1 s2))))

;; Objects
;;=========================================================

(define (make-wire)
  (let ((signal-value 0) (action-procedures '()))
    (define (set-my-signal! new-value)
      (if (not (= signal-value new-value))
          (sequence (set! signal-value new-value)
                    (call-each action-procedures))
          'done))

    (define (accept-action-procedure proc)
      (set! action-procedures (cons proc action-procedures))
      (proc))

    (define (dispatch m)
      (cond ((eq? m 'get-signal) signal-value)
            ((eq? m 'set-signal!) set-my-signal!)
            ((eq? m 'add-action!) accept-action-procedure)
            ((eq? m 'display-action-procedures) action-procedures)
            (else (error "Unknown operation -- WIRE" m))))

    dispatch))

(define (get-signal wire)
  (wire 'get-signal))

(define (set-signal! wire new-value)
  ((wire 'set-signal!) new-value))

(define (add-action! wire action-procedure)
  ((wire 'add-action!) action-procedure))

(define (display-action-procedures wire)
  (wire 'display-action-procedures))
```

```
(define (inverter input output)
  (define (invert-input)
    (let ((new-value (logical-not (get-signal input))))
      (after-delay inverter-delay
                   (lambda ()
                     (set-signal! output new-value)))))
  (add-action! input invert-input))

(define (and-gate a1 a2 output)
  (define (and-action-procedure)
    (let ((new-value (logical-and (get-signal a1) (get-signal a2))))
      (after-delay and-gate-delay
                   (lambda ()
                     (set-signal! output new-value)))))
  (add-action! a1 and-action-procedure)
  (add-action! a2 and-action-procedure))

(define (or-gate o1 o2 output)
  (define (or-action-procedure)
    (let ((new-value (logical-or (get-signal o1) (get-signal o2))))
      (after-delay or-gate-delay
                   (lambda ()
                     (set-signal! output new-value)))))
  (add-action! o1 or-action-procedure)
  (add-action! o2 or-action-procedure))

(define (half-adder a b s c)
  (let ((d (make-wire)) (e (make-wire)))
    (or-gate a b d)
    (and-gate a b c)
    (inverter c e)
    (and-gate d e s)))

;; Sample Simulation
;;==========================================================

(define (after-delay del action)
  (add-to-agenda! (+ del (current-time the-agenda)) action the-agenda))

(define (propagate)
  (if (empty-agenda? the-agenda)
      'done
      (let ((first-item (first-agenda-item the-agenda)))
        (first-item)
        (remove-first-agenda-item! the-agenda)
        (propagate))))
```

```
(define (probe name wire)
  (add-action! wire
               (lambda ()
                 (princ name)
                 (princ "  At time = ")
                 (princ (current-time the-agenda))
                 (princ " New value = ")
                 (princ (get-signal wire))
                 (newline))))

(define inverter-delay 2)
(define and-gate-delay 3)
(define or-gate-delay 5)

(define input-1 (make-wire))
(define input-2 (make-wire))
(define output (make-wire))
(define carry (make-wire))

(probe 'input-1 input-1)
(probe 'output output)
(probe 'carry carry)

(half-adder input-1 input-2 output carry)
(set-signal! input-1 1)
```

APPENDIX D

The code for the Excel aircrew scheduler is attached. User-specified names and data structures are not included, as the code is intended only to show the visual programming methodology more clearly. Code modularity, control structures used, and placement is apparent, but the code has been expanded to show detail. Ordinarily, columns are much more narrow, and lines of code that are wider than the column not entirely visible. This is shown in the second copy of the same code.

	A
1	make schedule
2	Places pilots as needed using find pilot
3	Move to "Mission" on schedule
4	=SELECT("Missioncolumn")
5	= down()
6	Find last mission
7	=IF(ISBLANK(ACTIVE.CELL()), up(),GOTO(A5))
8	= right(2)
9	Schedule remaining pilots
10	=IF(ISBLANK(ACTIVE.CELL()),find_pilot(),IF(ACTIVE.CELL()="Pilot",GOTO(A13)))
11	= up()
12	=GOTO(A10)
13	=BEEP()
14	=SELECT("datahome")
15	=SELECT("schedule")
16	=RETURN()
17	
18	find pilot
19	=IF(COLUMN(ACTIVE.CELL())<>6,RETURN())
20	=IF(ROW(ACTIVE.CELL())<12+!no. of db rows,RETURN())
21	=define cells()
22	=constrain availability()
23	=constrain flight leaders()
24	=IF(ISBLANK(!current slot),GOTO(A27))
25	=check guy()
26	=IF(A25="find another",GOTO(A27),RETURN())
27	=try P list()
28	=IF(A27=TRUE,RETURN())
29	=try low currency()
30	=IF(A29=TRUE,RETURN())
31	=try hi events()
32	=IF(A31=TRUE,RETURN())
33	error trap -- no pilots found
34	=SELECT("current slot")
35	=FORMULA.FILL("**** NO PILOT ****")
36	=RETURN()
37	
38	try to fly
39	=DATA.FIND()
40	=DEFINE.NAME("current pilot",ACTIVE.CELL())
41	=ip avail or not req()
42	=IF(A41="ip not req",GOTO(A48))
43	=IF(A41="req ip not avail",RETURN(FALSE))
44	=IF(A41="req ip avail",SELECT("current slot"),RETURN(FALSE))
45	=IF(ISBLANK(ABSREF("r[-1]c",!current slot)),inc req qual(),check qual())
46	=SELECT("current slot")
47	=IF(A45=FALSE,RETURN(FALSE))
48	fly the guy
49	=update avail("flying")
50	=SELECT("current slot")
51	=FORMULA(!current pilot)
52	=RETURN(TRUE)

	A
53	inc_req_qual
54	=SELECT(ABSREF("r[-1]c[-1]",!current_slot))
55	=FORMULA(5)
56	=RETURN(TRUE)
57	
58	check_qual
59	=SELECT("possible_IP")
60	=FORMULA(ABSREF("r[-1]c",!current_slot))
61	=IF(!qual_check=5,RETURN(TRUE),RETURN(FALSE))
62	
63	constrain_flight_leaders
64	=IF(ISBLANK(!current_qual),RETURN())
65	=SELECT("qual_criterion")
66	=FORMULA.FILL(!current_qual)
67	=SELECT("curr_criterion")
68	=FORMULA.FILL(">=0")
69	=RETURN()
70	
71	constrain_availability
72	=SELECT("avail_criterion")
73	=FORMULA("Free")
74	=RETURN()
75	
76	update_avail
77	=ARGUMENT("reason",2)
78	=SELECT("current_pilot")
79	= right(!no._of_db_cols+1)
80	=IF(AND(reason="flying",ISBLANK(ACTIVE.CELL())),FORMULA(!current_Land))
81	= right()
82	=ACTIVE.CELL()+1
83	= right(A82)
84	=SELECT("rc:rc[1]")
85	=INSERT(1)
86	=FORMULA(!current_TOT)
87	=SELECT("rc[1]")
88	=FORMULA(!current_Land)
89	=RETURN()
90	
91	define_cells
92	=DEFINE.NAME("current_slot")
93	=DEFINE.NAME("current_qual",ABSREF("rc[-1]",!current_slot))
94	=DEFINE.NAME("current_msn",ABSREF("rc[-2]",!current_slot))
95	=DEFINE.NAME("current_Land",ABSREF("rc[-3]",!current_slot))
96	=DEFINE.NAME("current_TOT",ABSREF("rc[-4]",!current_slot))
97	=DEFINE.NAME("current_line_no.",ABSREF("rc[-5]",!current_slot))
98	=SELECT("current_criteria")
99	=MATCH(!current_msn,!$A$1:$I$1,0)
100	= right(A99)
101	=SELECT(ACTIVE.CELL())
102	=DEFINE.NAME("curr_criterion")
103	=DEFINE.NAME("msn_criterion",ABSREF("rc[-1]",!curr_criterion))
104	=RETURN()

	B
1	try_P_list
2	=save_criteria()
3	=SELECT("P_list_mission_crit")
4	=FORMULA(!current_msn)
5	=set_db_P_list()
6	=SELECT("priority_x_range")
7	=EXTRACT(TRUE)
8	=set_db_con_db()
9	=IF(ISBLANK(ABSREF("r[1]c",!priority_pilot)),restore_criteria(),GOTO(B13))
10	=SELECT("pilot_criterion")
11	=CLEAR(1)
12	=RETURN(FALSE)
13	=SELECT("pilot_criterion")
14	=FORMULA(ABSREF("r[1]c",!priority_pilot))
15	=IF(!possible_pilots>0,GOTO(B19))
16	=SELECT(ABSREF("r[1]c[-1]:r[1]c",!priority_pilot))
17	=EDIT.DELETE(2)
18	=GOTO(B9)
19	=try_to_fly()
20	=IF(B19,comment("priority"),GOTO(B16))
21	=reset_criteria()
22	=update_P_list()
23	=RETURN(TRUE)
24	
25	update_P_list
26	=set_db_P_list()
27	=SELECT("P_list_crit_num")
28	=FORMULA(ABSREF("r[1]c[-1]",!priority_pilot))
29	=DATA.FIND()
30	=SELECT("rc[4]")
31	=DEFINE.NAME("priority_comment",ACTIVE.CELL())
32	=FORMULA(!current_line_no.)
33	=SELECT("P_list_crit_num")
34	=CLEAR(1)
35	=set_db_con_db()
36	=SELECT("current_slot")
37	=RETURN()
38	
39	try_low_currency
40	=save_criteria()
41	=SELECT("curr_criterion")
42	=FORMULA(">=0")
43	=SELECT("r[1]c")
44	=IF(ACTIVE.CELL()>!low_currency,GOTO(B54))
45	=COPY()
46	=SELECT("curr_criterion")
47	=PASTE.SPECIAL(3,1)
48	=IF(!possible_pilots=0,GOTO(B54))
49	=try_to_fly()
50	=IF(B49,comment("currency"))
51	=IF(B49,GOTO(B57),make_temp_unavail())
52	=GOTO(B41)

	B
53	
54	=restore_criteria()
55	=RETURN(FALSE)
56	
57	=reset_criteria()
58	=SELECT("current_slot")
59	=RETURN(TRUE)
60	
61	
62	try_hi_events
63	=save_criteria()
64	=SELECT("msn_criterion")
65	=SELECT("r[1]c")
66	=COPY()
67	=SELECT("msn_criterion")
68	=PASTE.SPECIAL(3,1)
69	=IF(!possible_pilots=0,GOTO(B54))
70	=try_to_fly()
71	=IF(B70,comment("events"))
72	=IF(B70,GOTO(B57),make_temp_unavail())
73	=GOTO(B64)
74	
75	
76	
77	update_unavail
78	=ARGUMENT("reason",2)
79	=SELECT("current_pilot")
80	=_right(!no._of_db_cols+1)
81	=IF(AND(reason="flying",ACTIVE.CELL()=!current_Land),,GOTO(B93))
82	=DEFINE.NAME("last_Land")
83	=SELECT("current_slot")
84	=_up()
85	=IF(ACTIVE.CELL()="PILOT",GOTO(B91),IF(ACTIVE.CELL()=!current_pilot,,GOTO(B84)))
86	=_left(3)
87	=COPY()
88	=SELECT("last_Land")
89	=PASTE()
90	=GOTO(B93)
91	=SELECT("last_Land")
92	=CLEAR(1)
93	=_right()
94	=ACTIVE.CELL()
95	=_right(B94)
96	=SELECT("rc:rc[1]")
97	=IF(reason="dnif",,GOTO(B103))
98	=IF(ACTIVE.CELL()<>!current_TOT,DIALOG.BOX(S50:Y52),GOTO(B100))
99	=RETURN()
100	=_right()
101	=IF(ACTIVE.CELL()<>!current_Land,DIALOG.BOX(S50:Y52),GOTO(B103))
102	=RETURN()
103	=EDIT.DELETE(1)
104	=RETURN()

	C
1	ip avail or not req
2	=save_criteria()
3	=SELECT("pilot criterion")
4	=FORMULA(!current_pilot)
5	=SELECT("qual criterion")
6	=FORMULA(">1")
7	=SELECT("curr criterion")
8	=FORMULA(">=0")
9	=IF(!possible_pilots>0,RETURN("ip not req"))
10	=SELECT("criteria2")
11	=COPY()
12	=SELECT("current criteria")
13	=PASTE()
14	=SELECT("curr criterion")
15	=FORMULA(">=0")
16	=!possible_pilots
17	=restore_criteria()
18	=IF(C16>0,RETURN("req ip avail"),RETURN("req ip not avail"))
19	
20	
21	comment
22	=ARGUMENT("reason",2)
23	=IF(reason="currency",GOTO(C30),IF(reason="events",GOTO(C38),GOTO(C24)))
24	=IF(reason="user specified",GOTO(C43),RETURN())
25	=SELECT(ABSREF("rc[3]",!current_slot))
26	=FORMULA(ABSREF("r[1]c[-1]",!priority_pilot))
27	=SELECT("rc[1]")
28	=FORMULA(" on Priority List")
29	=RETURN()
30	=SELECT("curr criterion")
31	=SELECT("r[1]c")
32	=COPY()
33	=SELECT(ABSREF("rc[3]",!current_slot))
34	=PASTE.SPECIAL(3,1)
35	=SELECT("rc[1]")
36	=FORMULA(" Days of Currency")
37	=RETURN()
38	=SELECT(ABSREF("rc[3]",!current_slot))
39	=FORMULA(!msn_criterion)
40	=SELECT("rc[1]")
41	=FORMULA(" Events Remaining")
42	=RETURN()
43	=SELECT(ABSREF("rc[3]",!current_slot))
44	=FORMULA("OK")
45	=SELECT("rc[1]")
46	=FORMULA(" - User Specified")
47	=RETURN()
48	=RETURN()
49	
50	
51	
52	

	C
53	remove_pilot
54	=DEFINE.NAME("current_slot")
55	=ACTIVE.CELL()
56	=_left(3)
57	=DEFINE.NAME("current_Land")
58	=_left()
59	=DEFINE.NAME("current_TOT")
60	=SELECT("current_slot")
61	=COPY()
62	=SELECT("pilot_criterion")
63	=PASTE()
64	=DATA.FIND()
65	=DEFINE.NAME("current_pilot",ACTIVE.CELL())
66	=update_unavail("flying")
67	=SELECT("pilot_criterion")
68	=CLEAR(1)
69	=SELECT("current_slot")
70	=SELECT("r[-1]c[-1]")
71	=IF(ACTIVE.CELL()=5,,GOTO(C76))
72	=SELECT("rc[12]")
73	=COPY()
74	=SELECT("rc[-12]")
75	=PASTE()
76	=SELECT("current_slot")
77	=_right(3)
78	=DEFINE.NAME("current_comment")
79	=_right()
80	=IF(ACTIVE.CELL()=" on Priority List",,GOTO(C99))
81	=set_db_P_list()
82	=SELECT("r[1]c:r[1]c[4]")
83	=CLEAR(1)
84	=SELECT("current_comment")
85	=COPY()
86	=SELECT("P_list_crit")
87	=SELECT("r[1]c")
88	=PASTE()
89	=DATA.FIND()
90	=_right(4)
91	=SELECT(ACTIVE.CELL())
92	=CLEAR(1)
93	=SELECT("P_list_crit")
94	=SELECT("r[1]c:r[1]c[4]")
95	=CLEAR(1)
96	=_left()
97	=FORMULA("=")
98	=set_db_con_db()
99	=SELECT("current_comment")
100	=SELECT("rc:rc[1]")
101	=CLEAR(1)
102	=SELECT("current_slot")
103	=CLEAR(1)
104	=RETURN()

	D
1	Schedule
2	Make Schedule
3	Place a Pilot
4	Remove a Pilot
5	-
6	DNIF a Pilot
7	Free a DNIFed Pilot
8	-
9	Show Schedule
10	Print Schedule
11	Reset Schedule
12	
13	
14	
15	Pilot data
16	Pilot Qualifications
17	Event Requirements
18	Event Currencies
19	-
20	Add Pilot...
21	Delete Pilot...
22	Propagate Data
23	-
24	Debrief Pilot...
25	Debrief Update
26	-
27	Graph Data...
28	Plot Availability
29	
30	on_open
31	=ACTIVATE.PREV()
32	=ADD.MENU(1,D1:E20)
33	=ADD.MENU(1,D15:E29)
34	=RETURN()
35	
36	on_close
37	=SELECT.LAST.CELL()
38	=SELECT(ACTIVE.CELL():ABSREF("rc[1]",!plot_area))
39	=EDIT.DELETE(2)
40	=DEFINE.NAME("end_plot_area",ABSREF("rc[1]",!plot_area))
41	=SELECT("datahome")
42	=SAVE.AS?()
43	=RETURN()
44	
45	make_temp_unavail
46	=SELECT("current_pilot")
47	= right(!no._of_db_cols-1)
48	=FORMULA("Req IP not avail")
49	=FORMULA("=",!pref_criterion)
50	=SELECT("msn_criterion")
51	=CLEAR(1)
52	=RETURN()

	D
53	dnif
54	=SELECT("nonavail")
55	=DIALOG.BOX(S36:Y44)
56	=IF(D55,,RETURN())
57	=FORMULA(Y39)
58	=DEFINE.NAME("current_TOT")
59	= right()
60	=FORMULA(Y42)
61	=DEFINE.NAME("current_Land")
62	=SELECT("datahome")
63	= down(Y38)
64	=DEFINE.NAME("current_pilot")
65	=SELECT("pilot_criterion")
66	=FORMULA(!current_pilot)
67	=constrain_availability()
68	=!possible_pilots
69	=reset_criteria()
70	=IF(D68=0,DIALOG.BOX(S46:Y48),update_avail("dnif"))
71	=RETURN()
72	
73	undnif
74	=SELECT("nonavail")
75	=DIALOG.BOX(S36:Y44)
76	=IF(D75,,RETURN())
77	=FORMULA(Y39)
78	=DEFINE.NAME("current_TOT")
79	= right()
80	=FORMULA(Y42)
81	=DEFINE.NAME("current_Land")
82	=SELECT("datahome")
83	= down(Y38)
84	=DEFINE.NAME("current_pilot")
85	=update_unavail("dnif")
86	=RETURN()
87	
88	check_guy
89	=SELECT("pilot_criterion")
90	=FORMULA(!current_slot)
91	=IF(!possible_pilots=0,DIALOG.BOX(S21:Y26),GOTO(D95))
92	=SELECT("pilot_criterion")
93	=CLEAR(1)
94	=IF(Y26,RETURN(find_another),RETURN(FALSE))
95	=try_to_fly()
96	=IF(D95=FALSE,DIALOG.BOX(S29:Y34),GOTO(D101))
97	=SELECT("pilot_criterion")
98	=CLEAR(1)
99	=IF(Y34,RETURN(find_another),RETURN(FALSE))
100	
101	=reset_criteria()
102	=comment("user_specified")
103	=RETURN(TRUE)
104	

	E
1	
2	Macro1!make_schedule
3	Macro1!find_pilot
4	Macro1!remove_pilot
5	
6	Macro1!dnif
7	Macro1!undnif
8	
9	Macro1!show_schedule
10	Macro1!print_schedule
11	Macro1!reset_schedule
12	
13	
14	
15	
16	Macro1!qual
17	Macro1!events
18	Macro1!cur
19	
20	Macro1!add_pilot
21	Macro1!del_pilot
22	Macro1!propagate
23	
24	Macro1!debrief
25	Macro1!debrief_update
26	
27	Macro1!plot
28	Macro1!plot_avail
29	
30	
31	qual
32	=SELECT("qualifications")
33	=RETURN()
34	
35	events
36	=SELECT("events")
37	=RETURN()
38	
39	cur
40	=SELECT("currencies")
41	=RETURN()
42	
43	show_schedule
44	=SELECT("datahome")
45	=SELECT("schedule")
46	=RETURN()
47	
48	print_schedule
49	=SELECT("schedule")
50	=SET.PRINT.AREA()
51	=PRINT()
52	=RETURN()

	E
53	Procedures for moving cell:
54	
55	move to mission
56	= right(A99-1)
57	=RETURN()
58	
59	up
60	=ARGUMENT("moves",17)
61	=SELECT(,"r[-1]c")
62	=IF(ISNA(moves),RETURN())
63	=IF(moves=1,RETURN())
64	=SET.VALUE(E66,moves-1)
65	=SELECT(,"r[-1]c")
66	=E66-1
67	=IF(E66<1,RETURN(),GOTO(E65))
68	
69	down
70	=ARGUMENT("moves",17)
71	=SELECT(,"r[1]c")
72	=IF(ISNA(moves),RETURN())
73	=IF(moves=1,RETURN())
74	=SET.VALUE(E76,moves-1)
75	=SELECT(,"r[1]c")
76	=E76-1
77	=IF(E76<1,RETURN(),GOTO(E75))
78	
79	left
80	=ARGUMENT("moves",17)
81	=SELECT(,"rc[-1]")
82	=IF(ISNA(moves),RETURN())
83	=IF(moves=1,RETURN())
84	=SET.VALUE(E86,moves-1)
85	=SELECT(,"rc[-1]")
86	=E86-1
87	=IF(E86<1,RETURN(),GOTO(E85))
88	
89	right
90	=ARGUMENT("moves",25)
91	=SELECT(,"rc[1]")
92	=IF(ISNA(moves),RETURN())
93	=IF(moves=1,RETURN())
94	=SET.VALUE(E96,moves-1)
95	=SELECT(,"rc[1]")
96	=E96-1
97	=IF(E96<1,RETURN(),GOTO(E95))
98	
99	Handy-dandy test program:
100	
101	test
102	=DIALOG.BOX(S53:Y70)
103	=IF(E102,GOTO(E102))
104	=RETURN()

	F
1	reset schedule
2	=SELECT("Missioncolumn")
3	= down()
4	Find last mission
5	=IF(ISBLANK(ACTIVE.CELL()), up(),GOTO(F3))
6	Move to last pilot slot
7	= right(2)
8	Remove remaining pilots
9	=IF(ISBLANK(ACTIVE.CELL()),,IF(ACTIVE.CELL()="Pilot",GOTO(F12),remove_pilot()))
10	= up()
11	=GOTO(F9)
12	=BEEP()
13	=RETURN()
14	
15	reset criteria
16	=SELECT("current_criteria")
17	=CLEAR(1)
18	=RETURN()
19	
20	
21	Criteria, DB Swaps:
22	
23	save criteria
24	=SELECT("current_criteria")
25	=COPY()
26	=SELECT("criteria1")
27	=PASTE()
28	=RETURN()
29	
30	restore criteria
31	=SELECT("criteria1")
32	=COPY()
33	=SELECT("current_criteria")
34	=PASTE()
35	=RETURN()
36	
37	
38	set db P list
39	=SELECT("P list")
40	=SET.DATABASE()
41	=SELECT("P list crit")
42	=SET.CRITERIA()
43	=RETURN()
44	
45	set db con db
46	=SELECT("con db")
47	=SET.DATABASE()
48	=SELECT("con db crit")
49	=SET.CRITERIA()
50	=RETURN()
51	
52	

	F
53	plot
54	=dbox3()
55	=IF(NOT(F54),RETURN())
56	=SELECT("end_plot_area")
57	=IF(ISBLANK(!end_plot_area),GOTO(F61))
58	=SELECT(,"rc[1]")
59	=IF(ISBLANK(ACTIVE.CELL()),DEFINE.NAME("end_plot_area",),GOTO(F58))
60	=GOTO(F57)
61	=IF(ISBLANK(P58),RETURN())
62	=FORMULA(P58)
63	=IF(NOT(ISBLANK(P62)),SELECT("rc[1]"),GOTO(F69))
64	=FORMULA(P62)
65	=IF(NOT(ISBLANK(P65)),SELECT("rc[1]"),GOTO(F69))
66	=FORMULA(P65)
67	=IF(NOT(ISBLANK(P69)),SELECT("rc[1]"),GOTO(F69))
68	=FORMULA(P69)
69	=SELECT(!end_plot_area:ACTIVE.CELL())
70	=EXTRACT(FALSE)
71	=NEW(2)
72	=GALLERY.COLUMN(1)
73	=IF(ISBLANK(P65),LEGEND(FALSE),LEGEND(TRUE))
74	=RETURN()
75	
76	plot_avail
77	=SELECT("avail_plot_formula")
78	=COPY()
79	=SELECT("avail_plot_hours")
80	=PASTE()
81	=SELECT("avail_plot_area")
82	=COPY()
83	=PASTE.SPECIAL(3,1)
84	=NEW(2)
85	=RETURN()
86	
87	
88	propagate
89	=reset_criteria()
90	=SELECT("qual_data")
91	=EXTRACT(FALSE)
92	=DEFINE.NAME("qualifications")
93	=SELECT("event_data")
94	=EXTRACT(FALSE)
95	=DEFINE.NAME("events")
96	=SELECT("cur_data")
97	=EXTRACT(FALSE)
98	=DEFINE.NAME("currencies")
99	=SELECT("datahome")
100	=RETURN()
101	
102	dbox3
103	=DIALOG.BOX(J56:P72)
104	=RETURN(F103)

	G
1	add_pilot
2	=dbox1()
3	=IF(NOT(G2),RETURN())
4	=SELECT(!A3)
5	= right(!no. of db cols)
6	=SELECT("rc[15]")
7	=DEFINE.NAME("end_of_row")
8	=SELECT(!A3:!end_of_row)
9	=INSERT(2)
10	=FORMULA(P5)
11	=FORMULA(P8,"rc[1]")
12	=SELECT("r[-1]c[2]")
13	=COPY()
14	=SELECT("r[1]c")
15	=PASTE()
16	= right(!no. of db cols)
17	=SELECT("r[-1]c")
18	=COPY()
19	=SELECT("r[1]c")
20	=PASTE()
21	=FORMULA(0,"rc[1]")
22	=FORMULA(2400,"rc[2]")
23	=SELECT(!AB26:!AM26)
24	=INSERT(2)
25	=SELECT("con_avail_data")
26	=SORT(1,!A:A,1)
27	=propagate()
28	=SELECT("pilot_list")
29	=COPY()
30	=SELECT("avail_plot_pilots")
31	=PASTE()
32	=SELECT("datahome")
33	=RETURN()
34	
35	
36	dbox1
37	=DIALOG.BOX(J3:P10)
38	=RETURN(G37)
39	
40	
41	debrief
42	=dbox4()
43	=RETURN()
44	
45	dbox4
46	=DIALOG.BOX(S3:Y18)
47	=RETURN(G46)
48	
49	
50	
51	
52	

	G
53	del_pilot
54	=dbox2()
55	=IF(NOT(G54),RETURN())
56	=FORMULA(P22,!pilot_criterion)
57	=DATA.FIND()
58	=SELECT(ACTIVE.CELL())
59	=DEFINE.NAME("current_pilot")
60	= right(!no. of db cols)
61	=SELECT("rc[15]")
62	=DEFINE.NAME("end_of_row")
63	=reset_criteria()
64	=SELECT(!current_pilot:!end_of_row)
65	=EDIT.DELETE(2)
66	=SELECT(!AB26:!AM26)
67	=EDIT.DELETE(2)
68	=SELECT("pilot_list")
69	=COPY()
70	=SELECT("avail_plot_pilots")
71	=PASTE()
72	=propagate()
73	=SELECT("datahome")
74	=RETURN()
75	
76	dbox2
77	=DIALOG.BOX(J20:P25)
78	=RETURN(G77)
79	
80	
81	
82	
83	
84	debrief_update
85	=clear_criteria()
86	=SELECT("criteriahome")
87	=SELECT(,"r[1]c")
88	=FORMULA(Y5)
89	=DATA.FIND()
90	=SELECT(,"rc[3]")
91	=FORMULA(ACTIVE.CELL()-Y10)
92	=IF(ACTIVE.CELL()<0,FORMULA(0))
93	=SELECT(,"rc[3]")
94	=FORMULA(ACTIVE.CELL()-Y13)
95	=IF(ACTIVE.CELL()<0,FORMULA(0))
96	=SELECT(,"rc[3]")
97	=FORMULA(ACTIVE.CELL()-Y16)
98	=IF(ACTIVE.CELL()<0,FORMULA(0))
99	=reset_criteria()
100	=propagate()
101	=SELECT("datahome")
102	=RETURN()
103	
104	

	I	J	K	L	M	N	O	P
1	Add Pilot Box	Item	Horiz	Vert	Item	Item	Text	Initial/Resul
2		Num	Pos	Pos	Hght	Wdth		
3								
4	Static Text	5					Name:	
5	Edit Text	6						Dyer, Doug
6	Static Text	5					Qualification:	
7	Edit text box	6						2FL
8	Combo list box	16					type	3
9	Ok Button	1	200	4			Enter	
10	Cancel Button	2					Cancel	
11								
12								
13								
14								
15								
16								
17								
18	Del Pilot Box	Item	Horiz	Vert	Item	Item	Text	Initial/Resul
19		Num	Pos	Pos	Hght	Wdth		
20								
21	Static Text	5					Pilot:	
22	Edit text box	6						Dyer, Doug
23	Combo list box	16					lpilot list	5
24	Ok Button	1	200	4			Enter	
25	Cancel Button	2					Cancel	
26								
27								
28								
29								
30								
31								
32								
33								
34								
35								
36								
37								
38								
39								
40								
41								
42								
43								
44								
45								
46								
47								
48								
49								
50								
51								
52								

	I	J	K	L	M	N	O	P
53								
54	Plot box	Item	Horiz	Vert	Item	Item	Text	Initial/Resul
55		Num	Pos	Pos	Hght	Wdth		
56								
57	Static Text	5					X-Axis:	
58	Edit text box	6						PILOT
59	Combo list box	16					!db_headings	1
60	Static Text	5						
61	Static Text	5					Attribute 1:	
62	Edit text box	6						DACBT
63	Combo list box	16					!db_headings	8
64	Static Text	5	250	4			Attribute 2:	
65	Edit text box	6						
66	Combo list box	16					!db_headings	9
67	Static Text	5						
68	Static Text	5					Attribute 3:	
69	Edit text box	6						
70	Combo list box	16					!db_headings	8
71	Ok Button	1		350			Enter	
72	Cancel Button	2	340	350			Cancel	
73								
74								
75								
76								
77								
78								
79								
80								
81								
82								
83								
84								
85								
86								
87								
88								
89								
90								
91								
92								
93								
94								
95								
96								
97								
98								
99								
100								
101								
102								
103								
104								

	R	S	T	U	V	W	X	Y
1	Debrief Box	Item	Horiz	Vert	Item	Item	Text	Init/Result
2		Num	Pos	Pos	Hght	Wdth		
3								
4	Static Text	5					Name:	
5	Edit Text	6						Able, Adam
6	Combo list bo	16					!type2	1
7	Static Text	5						
8	Static Text	5					WD	
9	Static Text	5					No. completed:	
10	Edit Text	6						2
11	Static Text	5					ACBT	
12	Static Text	5					No. completed:	
13	Edit Text	6						
14	Static Text	5					DACBT	
15	Static Text	5					No. completed:	
16	Edit Text	6						
17	Ok Button	1	200	4			Enter	
18	Cancel Buttor	2					Cancel	
19								
20								
21	Error boxes:		295	199	294	151	Oh-oh!	
22		1	207	109	64		OK	
23		5	9	9			Because of qualification,	
24		5	10	35			availability, or currency,	
25		5	10	58			you can't fly this pilot.	
26		13	6	113			Place someone else	TRUE
27								
28								
29			295	199	294	151	Oh-oh!	
30		1	207	109	64		OK	
31		5	9	9			This pilot requires an instru	
32		5	10	35			pilot. Either none is availabl	
33		5	10	58			the flight leader slot is fille	
34		13	6	113			Place someone else	TRUE
35								
36	DNIF Box		359	188	189	277	DNIF/UnDNIF a Pilot	
37		5	9	5			Who and When?	
38		15	9	30	171	92	!pilot list	10
39		7	9	156	171			730
40		5	9	189			To:	
41		5	10	135			From:	
42		7	9	208	171			1600
43		2	13	245	64		Cancel	
44		1	111	245	64		OK	
45								
46	Can't DNIF		352	199	217	81	Oh-oh!	
47		5	9	5			That pilot is already busy!	
48		4	40	44	133		Cancel DNIF	
49								
50	Can't UnDNIF		352	199	217	81	Oh-oh!	
51		5	9	5			Time mismatch.	
52		4	40	44	133		Cancel UnDNIF	

	R	S	T	U	V	W	X	Y
53	Debrief box		165	96	323	372	Debrief	
54		6	7	9	171			Dingo, Dave
55		16	9	40	171	114	!pilot list	4
56		5	11	164			Event	
57		15	10	185	129	181	!event list	1
58		1	202	15	105		Store Event	
59		2	202	53	105		Done	
60		11	150	167	156	199	No. Completed	2
61		12	173	183			0	
62		12	173	202			1	
63		12	173	220			2	
64		12	173	238			3	
65		12	173	256			4	
66		12	173	272			5	
67		12	173	290			6	
68		12	173	308			7	
69		12	173	326			8	
70		12	173	344			9	
71								
72								
73								
74								
75								
76								
77								
78								
79								
80								
81								
82								
83								
84								
85								
86								
87								
88								
89								
90								
91								
92								
93								
94								
95								
96								
97								
98								
99								
100								
101								
102								
103								
104								

	A	B	C
1	make schedule	try P list	ip avail or not req
2	Places pilots as needed us	=save criteria()	=save criteria()
3	Move to "Mission" on sched	=SELECT("P list mission	=SELECT("pilot criterion")
4	=SELECT("Missioncolumn")	=FORMULA(!current msn)	=FORMULA(!current pilot)
5	= down()	=set db P list()	=SELECT("qual criterion")
6	Find last mission	=SELECT("priority x range	=FORMULA(">1")
7	=IF(ISBLANK(ACTIVE.CELL(	=EXTRACT(TRUE)	=SELECT("curr criterion")
8	= right(2)	=set db con db()	=FORMULA(">=0")
9	Schedule remaining pilots	=IF(ISBLANK(ABSREF("r[1	=IF(!possible pilots>0,RET
10	=IF(ISBLANK(ACTIVE.CELL	=SELECT("pilot criterion")	=SELECT("criteria2")
11	= up()	=CLEAR(1)	=COPY()
12	=GOTO(A10)	=RETURN(FALSE)	=SELECT("current criteria
13	=BEEP()	=SELECT("pilot criterion")	=PASTE()
14	=SELECT("datahome")	=FORMULA(ABSREF("r[1]c	=SELECT("curr criterion")
15	=SELECT("schedule")	=IF(!possible pilots>0,GOT(	=FORMULA(">=0")
16	=RETURN()	=SELECT(ABSREF("r[1]c[-1	=!possible pilots
17		=EDIT.DELETE(2)	=restore criteria()
18	find pilot	=GOTO(B9)	=IF(C16>0,RETURN("req ip
19	=IF(COLUMN(ACTIVE.CELL())	=try to fly()	
20	=IF(ROW(ACTIVE.CELL())<1	=IF(B19,comment("priority"	
21	=define cells()	=reset criteria()	comment
22	=constrain availability()	=update P list()	=ARGUMENT("reason",2)
23	=constrain flight leaders(	=RETURN(TRUE)	=IF(reason="currency",GOTO
24	=IF(ISBLANK(!current slot),		=IF(reason="user specified"
25	=check guy()	update P list	=SELECT(ABSREF("rc[3]",!c
26	=IF(A25="find another",GOT	=set db P list()	=FORMULA(ABSREF("r[1]c
27	=try P list()	=SELECT("P list crit num	=SELECT("rc[1]")
28	=IF(A27=TRUE,RETURN())	=FORMULA(ABSREF("r[1]c	=FORMULA(" on Priority Lis
29	=try low currency()	=DATA.FIND()	=RETURN()
30	=IF(A29=TRUE,RETURN())	=SELECT("rc[4]")	=SELECT("curr criterion")
31	=try hi events()	=DEFINE.NAME("priority co	=SELECT("r[1]c")
32	=IF(A31=TRUE,RETURN())	=FORMULA(!current line nc	=COPY()
33	error trap -- no pilots foun	=SELECT("P list crit num	=SELECT(ABSREF("rc[3]",!c
34	=SELECT("current slot")	=CLEAR(1)	=PASTE.SPECIAL(3,1)
35	=FORMULA.FILL("**** NO PI	=set db con db()	=SELECT("rc[1]")
36	=RETURN()	=SELECT("current slot")	=FORMULA(" Days of Currer
37		=RETURN()	=RETURN()
38	try to fly		=SELECT(ABSREF("rc[3]",!c
39	=DATA.FIND()	try low currency	=FORMULA(!msn criterion)
40	=DEFINE.NAME("current pil	=save criteria()	=SELECT("rc[1]")
41	=ip avail or not req()	=SELECT("curr criterion")	=FORMULA(" Events Remair
42	=IF(A41="ip not req",GOTO	=FORMULA(">=0")	=RETURN()
43	=IF(A41="req ip not avail",	=SELECT("r[1]c")	=SELECT(ABSREF("rc[3]",!c
44	=IF(A41="req ip avail",SEL	=IF(ACTIVE.CELL()>!low cu	=FORMULA("OK")
45	=IF(ISBLANK(ABSREF("r[-1	=COPY()	=SELECT("rc[1]")
46	=SELECT("current slot")	=SELECT("curr criterion")	=FORMULA(" - User Specifi
47	=IF(A45=FALSE,RETURN(FA	=PASTE.SPECIAL(3,1)	=RETURN()
48	fly the guy	=IF(!possible pilots=0,GOT(	=RETURN()
49	=update avail("flying")	=try to fly()	
50	=SELECT("current slot")	=IF(B49,comment("currenc	
51	=FORMULA(!current pilot)	=IF(B49,GOTO(B57),make	
52	=RETURN(TRUE)	=GOTO(B41)	

	A	B	C
53	inc req qual		remove pilot
54	=SELECT(ABSREF("r[-1]c[-	=restore criteria()	=DEFINE.NAME("current slc
55	=FORMULA(5)	=RETURN(FALSE)	=ACTIVE.CELL()
56	=RETURN(TRUE)		= left(3)
57		=reset criteria()	=DEFINE.NAME("current Lar
58	check qual	=SELECT("current slot")	= left()
59	=SELECT("possible IP")	=RETURN(TRUE)	=DEFINE.NAME("current TO
60	=FORMULA(ABSREF("r[-1]c		=SELECT("current slot")
61	=IF(!qual check=5,RETURN(T		=COPY()
62		try hi events	=SELECT("pilot criterion")
63	constrain flight leaders	=save criteria()	=PASTE()
64	=IF(ISBLANK(!current qual),	=SELECT("msn criterion")	=DATA.FIND()
65	=SELECT("qual criterion")	=SELECT("r[1]c")	=DEFINE.NAME("current pil
66	=FORMULA.FILL(!current qu	=COPY()	=update unavail("flying")
67	=SELECT("curr criterion")	=SELECT("msn criterion")	=SELECT("pilot criterion")
68	=FORMULA.FILL(">=0")	=PASTE.SPECIAL(3,1)	=CLEAR(1)
69	=RETURN()	=IF(!possible pilots=0,GOT	=SELECT("current slot")
70		=try to fly()	=SELECT("r[-1]c[-1]")
71	constrain availability	=IF(B70,comment("events")	=IF(ACTIVE.CELL()=5,,GOTO
72	=SELECT("avail criterion")	=IF(B70,GOTO(B57),make	=SELECT("rc[12]")
73	=FORMULA("Free")	=GOTO(B64)	=COPY()
74	=RETURN()		=SELECT("rc[-12]")
75			=PASTE()
76	update avail		=SELECT("current slot")
77	=ARGUMENT("reason",2)	update unavail	= right(3)
78	=SELECT("current pilot")	=ARGUMENT("reason",2)	=DEFINE.NAME("current co
79	= right(!no. of db cols+1)	=SELECT("current pilot")	= right()
80	=IF(AND(reason="flying",ISB	= right(!no. of db cols+1)	=IF(ACTIVE.CELL()=" on Pr
81	= right()	=IF(AND(reason="flying",AC	=set db P list()
82	=ACTIVE.CELL()+1	=DEFINE.NAME("last Land")	=SELECT("r[1]c:r[1]c[4]")
83	= right(A82)	=SELECT("current slot")	=CLEAR(1)
84	=SELECT("rc:rc[1]")	= up()	=SELECT("current commen
85	=INSERT(1)	=IF(ACTIVE.CELL()="PILOT"	=COPY()
86	=FORMULA(!current TOT)	= left(3)	=SELECT("P list crit")
87	=SELECT("rc[1]")	=COPY()	=SELECT("r[1]c")
88	=FORMULA(!current Land)	=SELECT("last Land")	=PASTE()
89	=RETURN()	=PASTE()	=DATA.FIND()
90		=GOTO(B93)	= right(4)
91	define cells	=SELECT("last Land")	=SELECT(ACTIVE.CELL())
92	=DEFINE.NAME("current slc	=CLEAR(1)	=CLEAR(1)
93	=DEFINE.NAME("current qu	= right()	=SELECT("P list crit")
94	=DEFINE.NAME("current m	=ACTIVE.CELL()	=SELECT("r[1]c:r[1]c[4]")
95	=DEFINE.NAME("current La	= right(B94)	=CLEAR(1)
96	=DEFINE.NAME("current TC	=SELECT("rc:rc[1]")	= left()
97	=DEFINE.NAME("current lir	=IF(reason="dnif",,GOTO(B1	=FORMULA("=")
98	=SELECT("current criteria"	=IF(ACTIVE.CELL()<>!curren	=set db con db()
99	=MATCH(!current msn,!$AS	=RETURN()	=SELECT("current commen
100	= right(A99)	= right()	=SELECT("rc:rc[1]")
101	=SELECT(ACTIVE.CELL())	=IF(ACTIVE.CELL()<>!curren	=CLEAR(1)
102	=DEFINE.NAME("curr criter	=RETURN()	=SELECT("current slot")
103	=DEFINE.NAME("msn crite	=EDIT.DELETE(1)	=CLEAR(1)
104	=RETURN()	=RETURN()	=RETURN()

	D	E	F
1	Schedule		reset schedule
2	Make Schedule	Macro1!make_schedule	=SELECT("Missioncolumn")
3	Place a Pilot	Macro1!find_pilot	= down()
4	Remove a Pilot	Macro1!remove_pilot	Find last mission
5	-		=IF(ISBLANK(ACTIVE.CELL(
6	DNIF a Pilot	Macro1!dnif	Move to last pilot slot
7	Free a DNIFed Pilot	Macro1!undnif	= right(2)
8	-		Remove remaining pilots
9	Show Schedule	Macro1!show_schedule	=IF(ISBLANK(ACTIVE.CELL
10	Print Schedule	Macro1!print_schedule	= up()
11	Reset Schedule	Macro1!reset_schedule	=GOTO(F9)
12			=BEEP()
13			=RETURN()
14			
15	Pilot data		reset criteria
16	Pilot Qualifications	Macro1!qual	=SELECT("current criteria'
17	Event Requirements	Macro1!events	=CLEAR(1)
18	Event Currencies	Macro1!cur	=RETURN()
19	-		
20	Add Pilot...	Macro1!add_pilot	
21	Delete Pilot...	Macro1!del_pilot	Criteria, DB Swaps:
22	Propagate Data	Macro1!propagate	
23	-		save criteria
24	Debrief Pilot...	Macro1!debrief	=SELECT("current criteria'
25	Debrief Update	Macro1!debrief_update	=COPY()
26	-		=SELECT("criteria1")
27	Graph Data...	Macro1!plot	=PASTE()
28	Plot Availability	Macro1!plot_avail	=RETURN()
29			
30	on_open		restore_criteria
31	=ACTIVATE.PREV()	qual	=SELECT("criteria1")
32	=ADD.MENU(1,D1:E20)	=SELECT("qualifications")	=COPY()
33	=ADD.MENU(1,D15:E29)	=RETURN()	=SELECT("current criteria'
34	=RETURN()		=PASTE()
35		events	=RETURN()
36	on_close	=SELECT("events")	
37	=SELECT.LAST.CELL()	=RETURN()	
38	=SELECT(ACTIVE.CELL():A[		set_db_P_list
39	=EDIT.DELETE(2)	cur	=SELECT("P_list")
40	=DEFINE.NAME("end_plot_a	=SELECT("currencies")	=SET.DATABASE()
41	=SELECT("datahome")	=RETURN()	=SELECT("P_list_crit")
42	=SAVE.AS?()		=SET.CRITERIA()
43	=RETURN()	show_schedule	=RETURN()
44		=SELECT("datahome")	
45	make_temp_unavail	=SELECT("schedule")	set_db_con_db
46	=SELECT("current_pilot")	=RETURN()	=SELECT("con_db")
47	= right(!no._of_db_cols-1)		=SET.DATABASE()
48	=FORMULA("Req IP not ava	print_schedule	=SELECT("con_db_crit")
49	=FORMULA("=",!pref_criteri	=SELECT("schedule")	=SET.CRITERIA()
50	=SELECT("msn_criterion")	=SET.PRINT.AREA()	=RETURN()
51	=CLEAR(1)	=PRINT()	
52	=RETURN()	=RETURN()	

	D	E	F
53	dnif	Procedures for moving cell	plot
54	=SELECT("nonavail")		=dbox3()
55	=DIALOG.BOX(S36:Y44)	move_to_mission	=IF(NOT(F54),RETURN())
56	=IF(D55,,RETURN())	= right(A99-1)	=SELECT("end plot area")
57	=FORMULA(Y39)	=RETURN()	=IF(ISBLANK(!end plot area
58	=DEFINE.NAME("current TO		=SELECT(,"rc[1]")
59	= right()	up	=IF(ISBLANK(ACTIVE.CELL(
60	=FORMULA(Y42)	=ARGUMENT("moves",17)	=GOTO(F57)
61	=DEFINE.NAME("current Lar	=SELECT(,"r[-1]c")	=IF(ISBLANK(P58),RETURN(
62	=SELECT("datahome")	=IF(ISNA(moves),RETURN())	=FORMULA(P58)
63	= down(Y38)	=IF(moves=1,RETURN())	=IF(NOT(ISBLANK(P62)),SE
64	=DEFINE.NAME("current pil	=SET.VALUE(E66,moves-1)	=FORMULA(P62)
65	=SELECT("pilot criterion")	=SELECT(,"r[-1]c")	=IF(NOT(ISBLANK(P65)),SE
66	=FORMULA(!current pilot)	=E66-1	=FORMULA(P65)
67	=constrain availability()	=IF(E66<1,RETURN(),GOTO(	=IF(NOT(ISBLANK(P69)),SE
68	=!possible pilots		=FORMULA(P69)
69	=reset criteria()	down	=SELECT(!end plot area:AC
70	=IF(D68=0,DIALOG.BOX(S4	=ARGUMENT("moves",17)	=EXTRACT(FALSE)
71	=RETURN()	=SELECT(,"r[1]c")	=NEW(2)
72		=IF(ISNA(moves),RETURN())	=GALLERY.COLUMN(1)
73	undnif	=IF(moves=1,RETURN())	=IF(ISBLANK(P65),LEGEND(F
74	=SELECT("nonavail")	=SET.VALUE(E76,moves-1)	=RETURN()
75	=DIALOG.BOX(S36:Y44)	=SELECT(,"r[1]c")	
76	=IF(D75,,RETURN())	=E76-1	plot_avail
77	=FORMULA(Y39)	=IF(E76<1,RETURN(),GOTO(	=SELECT("avail plot form
78	=DEFINE.NAME("current TO		=COPY()
79	= right()	left	=SELECT("avail plot hours
80	=FORMULA(Y42)	=ARGUMENT("moves",17)	=PASTE()
81	=DEFINE.NAME("current Lar	=SELECT(,"rc[-1]")	=SELECT("avail plot area"
82	=SELECT("datahome")	=IF(ISNA(moves),RETURN())	=COPY()
83	= down(Y38)	=IF(moves=1,RETURN())	=PASTE.SPECIAL(3,1)
84	=DEFINE.NAME("current pil	=SET.VALUE(E86,moves-1)	=NEW(2)
85	=update_unavail("dnif")	=SELECT(,"rc[-1]")	=RETURN()
86	=RETURN()	=E86-1	
87		=IF(E86<1,RETURN(),GOTO(	
88	check_guy		propagate
89	=SELECT("pilot criterion")	right	=reset criteria()
90	=FORMULA(!current slot)	=ARGUMENT("moves",25)	=SELECT("qual data")
91	=IF(!possible pilots=0,DIALC	=SELECT(,"rc[1]")	=EXTRACT(FALSE)
92	=SELECT("pilot criterion")	=IF(ISNA(moves),RETURN())	=DEFINE.NAME("qualificatic
93	=CLEAR(1)	=IF(moves=1,RETURN())	=SELECT("event data")
94	=IF(Y26,RETURN(find anoth	=SET.VALUE(E96,moves-1)	=EXTRACT(FALSE)
95	=try_to_fly()	=SELECT(,"rc[1]")	=DEFINE.NAME("events")
96	=IF(D95=FALSE,DIALOG.BO	=E96-1	=SELECT("cur data")
97	=SELECT("pilot criterion")	=IF(E96<1,RETURN(),GOTO(	=EXTRACT(FALSE)
98	=CLEAR(1)		=DEFINE.NAME("currencies"
99	=IF(Y34,RETURN(find anoth	Handy-dandy test program:	=SELECT("datahome")
100			=RETURN()
101	=reset criteria()	test	
102	=comment("user specified	=DIALOG.BOX(S53:Y70)	dbox3
103	=RETURN(TRUE)	=IF(E102,GOTO(E102))	=DIALOG.BOX(J56:P72)
104		=RETURN()	=RETURN(F103)

	G
1	add_pilot
2	=dbox1()
3	=IF(NOT(G2),RETURN())
4	=SELECT(!A3)
5	= right(!no. of db cols)
6	=SELECT("rc[15]")
7	=DEFINE.NAME("end of row
8	=SELECT(!A3:!end of row)
9	=INSERT(2)
10	=FORMULA(P5)
11	=FORMULA(P8,"rc[1]")
12	=SELECT("r[-1]c[2]")
13	=COPY()
14	=SELECT("r[1]c")
15	=PASTE()
16	= right(!no. of db cols)
17	=SELECT("r[-1]c")
18	=COPY()
19	=SELECT("r[1]c")
20	=PASTE()
21	=FORMULA(0,"rc[1]")
22	=FORMULA(2400,"rc[2]")
23	=SELECT(!AB26:!AM26)
24	=INSERT(2)
25	=SELECT("con_avail_data")
26	=SORT(1,!A:A,1)
27	=propagate()
28	=SELECT("pilot_list")
29	=COPY()
30	=SELECT("avail_plot_pilots
31	=PASTE()
32	=SELECT("datahome")
33	=RETURN()
34	
35	
36	dbox1
37	=DIALOG.BOX(J3:P10)
38	=RETURN(G37)
39	
40	
41	debrief
42	=dbox4()
43	=RETURN()
44	
45	dbox4
46	=DIALOG.BOX(S3:Y18)
47	=RETURN(G46)
48	
49	
50	
51	
52	

	G
53	del_pilot
54	=dbox2()
55	=IF(NOT(G54),RETURN())
56	=FORMULA(P22,!pilot_crite
57	=DATA.FIND()
58	=SELECT(ACTIVE.CELL())
59	=DEFINE.NAME("current_pi
60	= right(!no. of db cols)
61	=SELECT("rc[15]")
62	=DEFINE.NAME("end of row
63	=reset_criteria()
64	=SELECT(!current_pilot:!en
65	=EDIT.DELETE(2)
66	=SELECT(!AB26:!AM26)
67	=EDIT.DELETE(2)
68	=SELECT("pilot_list")
69	=COPY()
70	=SELECT("avail_plot_pilots
71	=PASTE()
72	=propagate()
73	=SELECT("datahome")
74	=RETURN()
75	
76	dbox2
77	=DIALOG.BOX(J20:P25)
78	=RETURN(G77)
79	
80	
81	
82	
83	
84	debrief_update
85	=clear_criteria()
86	=SELECT("criteriahome")
87	=SELECT(,"r[1]c")
88	=FORMULA(Y5)
89	=DATA.FIND()
90	=SELECT(,"rc[3]")
91	=FORMULA(ACTIVE.CELL()-
92	=IF(ACTIVE.CELL()<0,FORM
93	=SELECT(,"rc[3]")
94	=FORMULA(ACTIVE.CELL()-
95	=IF(ACTIVE.CELL()<0,FORM
96	=SELECT(,"rc[3]")
97	=FORMULA(ACTIVE.CELL()-
98	=IF(ACTIVE.CELL()<0,FORM
99	=reset_criteria()
100	=propagate()
101	=SELECT("datahome")
102	=RETURN()
103	
104	

	I	J	K	L	M	N	O	P
1	Add Pilot Box	Item	Horiz	Vert	Item	Item	Text	Initial/Resul
2		Num	Pos	Pos	Hght	Wdth		
3								
4	Static Text	5					Name:	
5	Edit Text	6						Dyer, Doug
6	Static Text	5					Qualification:	
7	Edit text box	6						2FL
8	Combo list box	16					type	3
9	Ok Button	1	200	4			Enter	
10	Cancel Button	2					Cancel	
11								
12								
13								
14								
15								
16								
17								
18	Del Pilot Box	Item	Horiz	Vert	Item	Item	Text	Initial/Resul
19		Num	Pos	Pos	Hght	Wdth		
20								
21	Static Text	5					Pilot:	
22	Edit text box	6						Dyer, Doug
23	Combo list box	16					!pilot list	5
24	Ok Button	1	200	4			Enter	
25	Cancel Button	2					Cancel	
26								
27								
28								
29								
30								
31								
32								
33								
34								
35								
36								
37								
38								
39								
40								
41								
42								
43								
44								
45								
46								
47								
48								
49								
50								
51								
52								

	I	J	K	L	M	N	O	P
53								
54	Plot box	Item	Horiz	Vert	Item	Item	Text	Initial/Resul
55		Num	Pos	Pos	Hght	Wdth		
56								
57	Static Text	5					X-Axis:	
58	Edit text box	6						PILOT
59	Combo list box	16					!db_headings	1
60	Static Text	5						
61	Static Text	5					Attribute 1:	
62	Edit text box	6						DACBT
63	Combo list box	16					!db_headings	8
64	Static Text	5	250	4			Attribute 2:	
65	Edit text box	6						
66	Combo list box	16					!db_headings	9
67	Static Text	5						
68	Static Text	5					Attribute 3:	
69	Edit text box	6						
70	Combo list box	16					!db_headings	8
71	Ok Button	1		350			Enter	
72	Cancel Button	2	340	350			Cancel	
73								
74								
75								
76								
77								
78								
79								
80								
81								
82								
83								
84								
85								
86								
87								
88								
89								
90								
91								
92								
93								
94								
95								
96								
97								
98								
99								
100								
101								
102								
103								
104								

	R	S	T	U	V	W	X	Y
1	Debrief Box	Item	Horiz	Vert	Item	Item	Text	Init/Result
2		Num	Pos	Pos	Hght	Wdth		
3								
4	Static Text	5					Name:	
5	Edit Text	6						Able, Adam
6	Combo list bo	16					!type2	1
7	Static Text	5						
8	Static Text	5					WD	
9	Static Text	5					No. completed:	
10	Edit Text	6						2
11	Static Text	5					ACBT	
12	Static Text	5					No. completed:	
13	Edit Text	6						
14	Static Text	5					DACBT	
15	Static Text	5					No. completed:	
16	Edit Text	6						
17	Ok Button	1	200	4			Enter	
18	Cancel Button	2					Cancel	
19								
20								
21	Error boxes:		295	199	294	151	Oh-oh!	
22		1	207	109	64		OK	
23		5	9	9			Because of qualification,	
24		5	10	35			availability, or currency,	
25		5	10	58			you can't fly this pilot.	
26		13	6	113			Place someone else	TRUE
27								
28								
29			295	199	294	151	Oh-oh!	
30		1	207	109	64		OK	
31		5	9	9			This pilot requires an instru	
32		5	10	35			pilot. Either none is availabl	
33		5	10	58			the flight leader slot is fille	
34		13	6	113			Place someone else	TRUE
35								
36	DNIF Box		359	188	189	277	DNIF/UnDNIF a Pilot	
37		5	9	5			Who and When?	
38		15	9	30	171	92	!pilot list	10
39		7	9	156	171			730
40		5	9	189			To:	
41		5	10	135			From:	
42		7	9	208	171			1600
43		2	13	245	64		Cancel	
44		1	111	245	64		OK	
45								
46	Can't DNIF		352	199	217	81	Oh-oh!	
47		5	9	5			That pilot is already busy!	
48		4	40	44	133		Cancel DNIF	
49								
50	Can't UnDNIF		352	199	217	81	Oh-oh!	
51		5	9	5			Time mismatch,	
52		4	40	44	133		Cancel UnDNIF	

	R	S	T	U	V	W	X	Y
53	Debrief box		165	96	323	372	Debrief	
54		6	7	9	171			Dingo, Dave
55		16	9	40	171	114	!pilot list	4
56		5	11	164			Event	
57		15	10	185	129	181	!event list	1
58		1	202	15	105		Store Event	
59		2	202	53	105		Done	
60		11	150	167	156	199	No. Completed	2
61		12	173	183			0	
62		12	173	202			1	
63		12	173	220			2	
64		12	173	238			3	
65		12	173	256			4	
66		12	173	272			5	
67		12	173	290			6	
68		12	173	308			7	
69		12	173	326			8	
70		12	173	344			9	
71								
72								
73								
74								
75								
76								
77								
78								
79								
80								
81								
82								
83								
84								
85								
86								
87								
88								
89								
90								
91								
92								
93								
94								
95								
96								
97								
98								
99								
100								
101								
102								
103								
104								

APPENDIX E

The following code is an example of the functional programming used by most spreadsheet programmers. The program uses four inputs to generate and print out data on a form used to justify rental cars for government travel. All parameters are visible to the user and may be edited directly. The only non-automated requirement is a data-of-travel text change on the form. Formulas used to do the calculations are shown.

RADC/CO Capt Doug Dyer 8-May-90

1. No adequate government or public transportation exists between points of arrival, TDY location(s) and lodging/meal facilities.

2. Date of travel 13-14 Jun 90 Number of travelers 3

3. COMMERCIAL TRANSPORTATION (Circle appropriate mode)

Limo/taxi - airport to motel:	$4.00	X Nr of Travelers	3	=	$12.00
Taxi/ - motel to TDY station	$16.00	X Nr of R/T	6	=	$96.00
Limo/taxi - motel to airport:	$4.00	X Nr of Travelers	3	=	$12.00
			Total Cost:		$120.00

4. RENTAL CAR:

$33.00	per day X	2 day		$66.00
$1.00	per gal. of gas X	1	gallon (20 miles)	$1.00
			Total Cost:	$67.00

5. SAVINGS TO THE GOVERNMENT: $53.00

6. Rental Vehicle arrangements completed by SATO on 8-May-90

x2973 DOUGLAS E. DYER, Capt, USAF

Automated Rental Car Justification: TO USE, CHANGE FIRST FOUR ITEMS BELOW			
CHANGE DATES OF TRAVEL, SATO RESERVATION TO LEFT, THEN PRINT			
FIRST PAGE ON FORM 1820, CHECK RENTAL CAR AND SIGN			
The following data are required for calculation			Comments
Distance from Airport to Motel Loca	2	Miles	Change each time
Distance from Hotel to TDY Locatio	8	Miles	Change each time
Number of Days at TDY Location:	2	Days	Change each time
Number of travelers	3	Travelers	Change each time
Common Carrier Costs----------			
Cost/mile of airport carrier (limo/tax	$2.00		Change Occasionally
Solo Travel Cost to Motel from Airp	$4.00		
Solo Travel Cost to Airport from Mo	$4.00		
Total Travel Cost to and from Airpo	$24.00		
Cost/mile of local carrier (taxi):	$2.00		Change Occasionally
Number of trips between hotel and T	2		
Travel to and from Hotel:	$96.00		
Total Common Carrier Costs:	$120.00		
Rental Car Costs----------			
Cost per day of rental car:	$33.00		Change Frequently
Cost of gas (per gallon):	$1.00		Change Occasionally
Estimated miles of travel required:	20		
Estimated Fuel Costs: (20 mpg)	$1.00		
Total Cost of Rental Car:	$67.00		
Cost savings to the Government	$53.00	NOTE: MUST EXCEED $25	

	A	B	C
1	RADC/CO		
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13	1. No adequate governme		
14	TDY location(s) and lodgi		
15			
16	2. Date of travel		13-14 Mar 90
17			
18	3. COMMERCIAL TRAN		
19			
20		Limo/taxi - airport to mote	
21		Taxi/ - motel to TDY stati	
22		Limo/taxi - motel to airpo	
23			
24			
25			
26	4. RENTAL CAR:		
27			
28		=P24	per day X
29		=P25	per gal. of gas X
30			
31			
32			
33	5. SAVINGS TO THE G		
34			
35	6. Rental Vehicle arrange		
36			
37			
38			
39			
40			
41			
42			
43			
44			
45	x2973		DOUGLAS E. DYER, C

	D	E	F
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			
16			
17			
18			
19			
20			=P13
21			=P7*P17
22			=P14
23			
24			
25			
26			
27			
28		=P8	day
29			=P26/20
30			
31			
32			
33			
34			
35			
36			
37			
38			
39			
40			
41			
42			
43			
44			
45			

	G	H	I
1	Capt Doug Dyer		
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			
16	Number of travelers		
17			
18			
19			
20	X Nr of Travelers		
21	X Nr of R/T		
22	X Nr of Travelers		
23			
24			Total Cost:
25			
26			
27			
28			
29	gallon (	=P26	miles)
30			
31			Total Cost:
32			
33	=P30		
34			
35			=NOW()
36			
37			
38			
39			
40			
41			
42			
43			
44			
45			

	J	K	L
1			=NOW()
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			
16	=P9		
17			
18			
19			
20	=P9	=	=F20*J20
21	=P18*P9	=	=F21*J21
22	=P9	=	=F22*J22
23			
24			=L20+L21+L22
25			
26			
27			
28			=B28*E28
29			=B29*F29
30			
31			=L28+L29
32			
33			
34			
35			
36			
37			
38			
39			
40			
41			
42			
43			
44			
45			

	M	N	O
1	Automated Rental Car Jus		
2	CHANGE DATES OF TR		
3	FIRST PAGE ON FORM		
4			
5	The following data are req		
6	Distance from Airport to N		
7	Distance from Hotel to TD		
8	Number of Days at TDY L		
9	Number of travelers		
10			
11	Common Carrier Costs----		
12	Cost/mile of airport carrie		
13	Solo Travel Cost to Motel		
14	Solo Travel Cost to Airpor		
15	Total Travel Cost to and fr		
16			
17	Cost/mile of local carrier (		
18	Number of trips between h		
19	Travel to and from Hotel:		
20			
21	Total Common Carrier Co		
22			
23	Rental Car Costs----------		
24	Cost per day of rental car:		
25	Cost of gas (per gallon):		
26	Estimated miles of travel r		
27	Estimated Fuel Costs: (20		
28	Total Cost of Rental Car:		
29			
30	Cost savings to the Gover		
31			
32			
33			
34			
35			
36			
37			
38			
39			
40			
41			
42			
43			
44			
45			

	P	Q	R
1			
2			
3			
4			
5			Comments
6	2	Miles	Change each time
7	8	Miles	Change each time
8	2	Days	Change each time
9	3	Travelers	Change each time
10			
11			
12	2		Change Occasionally
13	=P6*P12		
14	=P13		
15	=P6*2*P9*P12		
16			
17	2		Change Occasionally
18	=IF(P8>1,(P8-2)*2+2,0)		
19	=P7*P18*P9*P17		
20			
21	=P15+P19		
22			
23			
24	33		Change Frequently
25	1		Change Occasionally
26	=P6*2+P7*P18		
27	=P26/20*P25		
28	=P27+(P8*P24)		
29			
30	=P21-P28	NOTE: MUST EXCEI	
31			
32			
33			
34			
35			
36			
37			
38			
39			
40			
41			
42			
43			
44			
45			

APPENDIX F

The following code is a solution to the 8-puzzle describe in Nilsson's book. Note that the spreadsheet representation closely matches Nilsson's Figure 1.2. Concrete data structures are much easier to work with than invisible ones. During execution, the program explores the possible moves and picks the one which reduces the error most (hill climbing).

	Beginning Eight Puzzle:			
	2	8	3	
	1	6	4	
	7		5	
	Out-of-place matrix:			
	-1	-1	0	
	-1		0	
	0	-1	0	
The value of the Hill Climbing Function is:				-4
(Hill Climbing Function is a summation of the out-of-place matrix,				
or "minus" the number of tiles out of place).				
	Desired puzzle configuration at completion:			
	1	2	3	
	8		4	
	7	6	5	

	A
1	Solve
2	Explore
3	=SET.VALUE(n,North())
4	=South()
5	=SET.VALUE(e,East())
6	=West()
7	=SET.VALUE(s,South())
8	=North()
9	=SET.VALUE(w,West())
10	=East()
11	Test_and_Move
12	=IF(n>=MAX(e,s,w),North(),IF(e>=MAX(s,w),East(),IF(s>=w,South(),West())))
13	Test_Done
14	=IF('Dyer:Public:Current :8 puzzle'!$D11<>0,GOTO(Explore))
15	=BEEP()
16	=RETURN()
17	
18	
19	Macros for moving the cell
20	
21	left
22	=SELECT(,"rc[-1]")
23	=RETURN()
24	
25	right
26	=SELECT(,"rc[1]")
27	=RETURN()
28	
29	up
30	=SELECT(,"r[-1]c")
31	=RETURN()
32	
33	down
34	=SELECT(,"r[1]c")
35	=RETURN()
36	
37	Below is an expansion of one of the exploration macros: (Next page shows layout)
38	
39	North
40	= up()
41	=IF(ISNUMBER(ACTIVE.CELL()),COPY(),RETURN(-10))
42	= down()
43	=PASTE()
44	= up()
45	=CLEAR(1)
46	=RETURN('Dyer:Public:Current :8 puzzle'!$D49)
47	-2
48	m

	B	C	D
1		North	
2		= up()	
3		=IF(ISNUMBER(ACTIVE.CELL	
4		= down()	
5		=PASTE()	
6		= up()	
7		=CLEAR(1)	
8		=RETURN('Dyer:Public:Curre	
9		-2	East
10	West	m^	= right()
11	= left()		=IF(ISNUMBER(ACTIVE.CELL
12	=IF(ISNUMBER(ACTIVE.CELL		= left()
13	= right()		=PASTE()
14	=PASTE()		= right()
15	= left()		=CLEAR(1)
16	=CLEAR(1)		=RETURN('Dyer:Public:Curre
17	=RETURN('Dyer:Public:Curre	South	0
18	-10	= down()	e^
19	w^	=IF(ISNUMBER(ACTIVE.CELL	
20		= up()	
21		=PASTE()	
22		= down()	
23		=CLEAR(1)	
24		=RETURN('Dyer:Public:Curre	
25		-2	
26		s^	
27			
28			
29			
30			
31			
32			
33			
34			
35			
36			
37			
38			
39			
40			
41			
42			
43			
44			
45			
46			
47			
48			

APPENDIX G

The following code is used to record long distance telephone data using a visual input and as much default information as possible. The program uses pull-down menus to start and end a call and paste the called party's data into a database. If the party has not been called before, the data is stored in a table which may be edited or sorted. At the end of the month, the user sends the data file produced by the program across a network to an analysis center. The analysis center has another program which consolidates data into one file and applies the analysis functionality inherent in Excel to flexibly analyze the data and discover errors.

DATE	HOUR	NAME	EXT	CITY	STATE	TELEPHONE N	NAME	COMPANY	DURATION	REASON FOR CALL
2-Apr-90	13:08	Dyer	3529	Pittsburgh	PA	412-268-3842	Dr Bill Wolf	MCU	0:00:56	Darpa/RADC Planning Initiative
30-Mar-90	8:54	Dyer	3529	Arlington	VA	703-695-1766	Lucy Snodgrass	SAIK (DRIPA/INFO)	8:54:36	Darpa/RADC Planning Initiative
30-Mar-90	8:48	Dyer	3529	Arlington	VA	703-276-3530	Peggy	SAIK	0:02:19	Darpa/RADC Planning Initiative
29-Mar-90	11:36	Dyer	3529	Burlington	MA	617-229-6565	Mable	Marriot Burlington	0:04:35	Darpa/RADC Planning Initiative
20-Mar-90	9:38	Dyer	3529	Lubbock	TX	806-742-3904	Dr John Smith	Texas Institute of Techn	0:00:42	aircrew scheduling
20-Mar-90	9:32	Dyer	3529	Unknown	VA	703-522-6221	Bob Speakeasy	Dolphin Systems	0:02:02	Darpa/RADC Planning Initiative
2-Mar-90	11:30	Dyer	3529	Boston	MA	617-494-6600	Claudia	Marriot	0:01:06	Hotel Reservations
2-Mar-90	10:46	Dyer	3529	Boston	MA	617-494-6600	Claudia	Marriot	0:02:00	Hotel Reservations
28-Feb-90	11:30	Dyer	3529	Scott AFB	IL	618-632-2424	Jack White	Miter Machinery	11:30:13	Darpa/RADC Planning Initiative
13-Feb-90	13:19	Dyer	3529	Boston	MA	617-547-4800	Ilene	Sheraton Hotel	0:03:43	Hotel Reservations
12-Feb-90	14:07	Dyer	3529	Boston	MA	617-494-6600	Claudia	Marriot	0:01:00	Hotel Reservations
31-Jan-90	13:42	Dyer	3529	Norfolk	VA	804-451-0000	Jean	Comfort Inn	0:04:50	Hotel Reservations
24-Jan-90	11:07	Dyer	3529	Scott AFB	IL	618-632-2424	Jack White	Miter Machinery	0:05:47	Darpa/RADC Planning Initiative
23-Jan-90	16:09	Dyer	3529	Scott AFB	IL	618-632-2424	Jack White	Miter Machinery	0:00:29	Darpa/RADC Planning Initiative
16-Jan-90	12:50	Dyer	3529	San Diego	CA	619-224-3460	Dr. Jose Cuervo	BBM	0:01:51	Darpa/RADC Planning Initiative
3-Jan-90	11:43	Dyer	3529	Santa Monica	CA	213-393-0404	Dr. Bill Tucker	SAND Corp	0:01:02	Darpa/RADC Planning Initiative
3-Jan-90	11:38	Dyer	3529	Santa Monica	CA	213-451-8711	Tonia	Pacific Shores Hotel	0:04:00	Hotel Reservations
3-Jan-90	11:32	Dyer	3529	Santa Monica	CA	213-393-0404	Dr. Bill Tucker	SAND Corp	0:02:50	Darpa/RADC Planning Initiative
3-Jan-90	10:05	Dyer	3529	Santa Monica	CA	213-393-0404	Dr. Bill Tucker	SAND Corp	0:01:16	Darpa/RADC Planning Initiative
2-Jan-90	16:26	Dyer	3529	Santa Monica	CA	213-393-0404	Dr. Bill Tucker	SAND Corp	0:02:07	Darpa/RADC Planning Initiative
2-Jan-90	16:02	Dyer	3529	San Diego	CA	619-224-3460	Dr. Jose Cuervo	BBM	0:07:31	Darpa/RADC Planning Initiative
2-Jan-90	15:55	Dyer	3529	Santa Monica	CA	213-393-0404	Dr. Bill Tucker	SAND Corp	0:00:29	Darpa/RADC Planning Initiative
2-Jan-90	14:18	Dyer	3529	Santa Monica	CA	213-393-0404	Dr. Bill Tucker	SAND Corp	0:01:00	Darpa/RADC Planning Initiative
TEMPLATE:										
DATE	HOUR	NAME	EXT	CITY	STATE	TELEPHONE N	NAME	COMPANY	DURATION	REASON FOR CALL
11-May-90	8:33	Dyer	3529						-4:34:26	Darpa/RADC Planning Initiative
	Currently Processing a Ca			No						

	Help Info: **********					
	The data to the left is important (Times 9 Font).					
	You may manually input any data, or use Long Distance macros provided.					
	New information about who you're calling is normally stored in the list below.					
	You may edit that list and sort it using some manually assigned frequency.					
	When requested, make a data file and TOPS this file to Mary.					
	(Only a duplicate of the data file will go. You may save it or trash it.)					
	To be safe, don't delete records until they are at least two months old.					
	See Doug Dyer for additional info. This is an experimental application.					
	Last Insert to Frequently Dialed Numbers:					
	Pittsburgh	PA	412-268-3842	Dr Bill Wolf	MCU	
	Frequently dialed numbers ##########					
(Freq.)	CITY	STATE	TELEPHONE NO.	USE DATA TO LEFT	COMPANY CALLED	
	Pittsburgh	PA	412-268-3842	Dr Bill Wolf	MCU	
	Arlington	VA	703-695-1766	Lucy Snodgrass	SAIK (DRIPA/INFO)	
	Arlington	VA	703-276-3530	Peggy	SAIK	
	Burlington	MA	617-229-6565	Mable	Marriot Burlington	
	Lubbock	TX	806-742-3904	Dr John Smith	Texas Institute of Technology	
	Unknown	VA	703-522-6221	Bob Speakeasy	Dolphin Systems	
	Norfolk	VA	804-451-0000	Jean La Greene	Comfort Inn	
	Scott AFB	IL	618-632-2424	Jack White	Miter Machinery	
	Santa Monica	CA	213-393-0404	Dr. Bill Tucker	SAND Corp	
	San Diego	CA	619-224-3460	Dr. Jose Cuervo	BBM	

	A	B
1	start_call	end_call
2	=FORMULA("Yes",!processing_flag)	=SELECT("duration_calc")
3	=SELECT(!A2:K2)	=CALCULATE.NOW()
4	=INSERT(2)	=COPY()
5	=DEFINE.NAME("current_row")	=SELECT("duration")
6	=SELECT(,"rc[1]")	=PASTE.SPECIAL(3,1)
7	=DEFINE.NAME("start_time",ACTIVE.CELL())	=FORMAT.NUMBER("h:mm:ss")
8	=SELECT(,"rc[8]")	=FORMULA("No",!processing_flag)
9	=DEFINE.NAME("duration",ACTIVE.CELL())	=RETURN()
10	=SELECT("rc[1]")	
11	=DEFINE.NAME("reason")	
12	=SELECT("template")	
13	=COPY()	
14	=SELECT("current_row")	
15	=PASTE.SPECIAL(3,1)	
16	=SELECT("rc[4]")	
17	=DEFINE.NAME("city")	
18	=RETURN()	
19		
20		
21	on_open	on_close
22	=ADD.MENU(1,A26:B32)	=DELETE.MENU(1,9)
23	=ADD.MENU(5,A26:B32)	=DELETE.MENU(5,9)
24	=RETURN()	=CLOSE(TRUE)
25		=RETURN()
26	Long Distance	
27	Begin Call	'Log Macros'!start_call
28	Who's Called?...	'Log Macros'!paste_who
29	Comment...	'Log Macros'!comment
30	End Call	'Log Macros'!end_call
31	Cancel Call	'Log Macros'!can_it
32	Make data file	'Log Macros'!make_data_file
33		
34		
35	can_it	
36	=IF(!processing_flag="Yes",GOTO(A39))	
37	=ALERT("No call in process",3)	
38	=RETURN()	
39	=SELECT(!A2:K2)	
40	=EDIT.DELETE(2)	
41	=FORMULA("No",!processing_flag)	
42	=RETURN()	
43		
44		
45		
46		

	C	D
1	paste_who	process_data
2	=DIALOG.BOX($E$2:$K$17)	=SELECT(!M18)
3	=IF(NOT(C2),RETURN())	=FORMULA(K10)
4	=IF(K15=1,GOTO(process_data))	=SELECT(!N18)
5	=K15	=FORMULA(K12)
6	=SELECT("datahome")	=SELECT(!O18)
7	=_up()	=FORMULA(K5)
8	=_down(C5)	=SELECT(!P18)
9	=SELECT("rc:rc[4]")	=FORMULA(K6)
10	=COPY()	=SELECT(!Q18)
11	=SELECT("city")	=FORMULA(K8)
12	=PASTE()	=IF(K16,insert_data())
13	=RETURN()	=SELECT(!M18)
14		=GOTO(C9)
15		
16		insert_data
17		=SELECT("datahome")
18		=SELECT("r[1]c:r[1]c[4]")
19		=INSERT(2)
20		=SELECT(!M18)
21	make_data_file	=SELECT("rc:rc[4]")
22	=SELECT("Database")	=COPY()
23	=COPY()	=SELECT("datahome")
24	=NEW(1)	=SELECT("r[1]c")
25	=PASTE()	=PASTE()
26	=DEFINE.NAME("data")	=RETURN()
27	=SAVE.AS(!$C$2&" Long Distance Data",1)	
28	=CLOSE()	comment
29	=RETURN()	=SELECT("reason")
30		=DIALOG.BOX(E19:K23)
31		=IF(D30,FORMULA(K21))
32		=RETURN()
33		
34		
35		
36		
37		
38		
39	_down	_up
40	=ARGUMENT("moves",17)	=ARGUMENT("moves",17)
41	=SELECT(,"r[1]c")	=SELECT(,"r[-1]c")
42	=IF(ISNA(moves),RETURN())	=IF(ISNA(moves),RETURN())
43	=SET.VALUE(C45,moves-1)	=SET.VALUE(D86,moves-1)
44	=SELECT(,"r[1]c")	=SELECT(,"r[-1]c")
45	=C45-1	=D86-1
46	=IF(C45<1,RETURN(),GOTO(C44))	=IF(D86<1,RETURN(),GOTO(D85))

	E	F	G	H	I	J	K
1	Dialog Box Data					text	result
2	Who are you calling?	149	148	410	297	Who are you calling	
3	3	363	10	34		OK	
4	5	18	21			Telephone #	
5	6	18	44	171			
6	6	18	97	171			
7	5	18	131			Company	
8	6	17	151	171			
9	5	17	187			City	
10	6	16	205	171			
11	5	18	242			State	
12	6	16	263	171			
13	5	19	77			Name	
14	5	212	18			Select one:	
15	15	212	45	189	208	!r21c16:r81c16	2
16	13	212	270			Store Data	TRUE
17	2	330	260	64		Cancel	
18							
19	Why are you calling?	162	102	253	74		
20	5	8	13			Reason	
21	6	7	45	232			
22	1	174	8	64		OK	
23	2	97	9	64		Cancel	
24							
25							
26							
27							
28							
29							
30							
31							
32							
33							
34							
35							
36							
37							
38							
39							
40							
41							
42							
43							
44							
45							
46							

MISSION

OF

ROME LABORATORY

Rome Laboratory plans and executes an interdisciplinary program in research, development, test, and technology transition in support of Air Force Command, Control, Communications and Intelligence (C^3I) activities for all Air Force platforms. It also executes selected acquisition programs in several areas of expertise. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C^3I systems. In addition, Rome Laboratory's technology supports other AFSC Product Divisions, the Air Force user community, and other DOD and non-DOD agencies. Rome Laboratory maintains technical competence and research programs in areas including, but not limited to, communications, command and control, battle management, intelligence information processing, computational sciences and software producibility, wide area surveillance/sensors, signal processing, solid state sciences, photonics, electromagnetic technology, superconductivity, and electronic reliability/maintainability and testability.