USAISEC

US Army Information Systems Engineering Command
Fort Huachuca, AZ 85613-5300

ᴵ'.S. ARMY INSTITUTE FOR RESEARCH
IN MANAGEMENT INFORMATION,
ᑌNICATIONS, AND COMPUTER SCIENCES

AIRMICS

A Dynamic Help Generator for U.S. Army Installation-Level Software

ASQB-GM-92-001
September 1991

AIRMICS
115 O'Keefe Building
Georgia Institute of Technology
Atlanta, GA 30332-0800

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188
Exp. Date: Jun 30, 1986

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS NONE
2a. SECURITY CLASSIFICAION AUTHORITY N/A	3. DISTRIBUTION/AVAILIBILTY OF REPORT N/A
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) ASQB-GM-92-001	5. MONITORING ORGANIZATION REPORT NUMBER(S) N/A
6a. NAME OF PERFORMING ORGANIZATION AIRMICS 6b. OFFICE SYMBOL *(If applicable)* ASQB-GM	7a. NAME OF MONITORING ORGANIZATION N/A
6c. ADDRESS (City, State, and Zip Code) 115 O'Keefe Building Georgia Institute of Technology Atlanta, Georgia 30332-0800	7b. ADDRESS *(City, State, and ZIP Code)* N/A
8b. NAME OF FUNDING/SPONSORING ORGANIZATION AIRMICS 8b. OFFICE SYMBOL *(If applicable)* ASQB-GM	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS *(City, State, and ZIP Code)* 115 O'Keefe Bldg. Georgia Institute of Technology Atlanta, GA 30332-0800	10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.
62783A	DY10	05	

11. TITLE *(Include Security Classification)*
A Dynamic Help Generator for US Army Installation-Level Software

12. PERSONAL AUTHOR(S)
Mark G. Washechek

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (*Year, Month, Day*)	15. PAGE COUNT
	FROM______ TO______		88

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
FIELD	GROUP	SUBGROUP	

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

The purpose of this project is to develop a Dynamic Help Generator. This paper describes that Dynamic Help Generator, a programmer's development tool intended to be used by someone who seeks to design and implement a Dynamic Help Module (a module that can be added to interactive programs to produce Dynamic Help messages). Dynamic Help messages are non-interactive, state-specific, and context-specific messages that attempt to answer every question that a user might have at a specific point in a program.

20. DISTRIBUTIONN/AVAILIBILTY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION	
22a. NAME OF RESPONSIBLE INDIVIDUAL LTC Michael E. Mizell	22b. TELEPHONE *(Include Area Code)* (404) 894-3107	22c. OFFICE SYMBOL ASQB-GM

DD FORM 1473, 84 MAR   83 APR edition may be used until exhausted.
All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

THIS REPORT HAS BEEN REVIEWED AND IS APPROVED

s/ James Gantt
James Gantt
Division Chief
MISD

s/ [illegible]
John R. Mitchell
Director
AIRMICS

A Dynamic Help Generator for U.S. Army Installation-Level Software

A Technical Paper

by

Mark G. Washechek
Captain, United States Army

Project Course ISYE 8704
School of Industrial and Systems Engineering
Georgia Institute of Technology
Atlanta, Georgia 30332-0205

Course Advisor:
Donovan Young

September 6, 1991

Accepted by advisor: Donovan Young

A Dynamic Help Generator for U.S. Army Installation-Level Software

A Technical Paper

by

Mark G. Washechek
Captain, United States Army

Project Course ISYE 8704
School of Industrial and Systems Engineering
Georgia Institute of Technology
Atlanta, Georgia 30332-0205

Course Advisor:
Donovan Young

Table of Contents

A Dynamic Help Generator for Installation-Level Software

1. Introduction

The Army Institute for Research in Management Information Communications and Computer Science (AIRMICS) undertook a project to develop a Dynamic Help Generator during the summer of 1991. This paper describes that Dynamic Help Generator, a programmer's development tool intended to be used by someone who seeks to design and implement a Dynamic Help module.

A Dynamic Help module is software that can be added to an interactive program to produce Dynamic Help messages. Dynamic Help messages are non-interactive, state-specific, and context-specific messages that attempt to answer every question that a user might have at a specific point in a program.

Section 1.1 traces the background of the Dynamic Help concept and the background of the U.S. Army research that led to the Dynamic Help Generator, the development reported by this paper. Section 1.2 introduces the Dynamic Help Generator development project. The first implementation of Dynamic Help sought to improve the interface usability of the Automated Central Issue Facility System (ACIFS), a software package for managing U.S. Army clothing warehouses.

1.1 Background

1.1.1 Background of Dynamic Help

Dynamic Help, an extension of context-sensitive help, was introduced by Dr. Donovan Young (Young 1990). It is the latest in a progression of methods that software engineers can use to provide users of interactive programs with online help.

The simplest help system consists of a reference manual containing an index. The manual is organized by listing each feature and an explanation of how to use it. Generally, the user is looking for a combination of features to perform a specific function. If the user can recognize the vocabulary that he associates with what he is trying to accomplish, he can use the manual. A more advanced approach gives the user an online manual. This automated version suffers the same drawbacks as the hardcopy reference manual. Both require the user to look up commands, not functions.

For example, the user may be thinking of *delete, erase, expunge,* or *destroy*, but the index lists only *purge*. A possible solution would be to list synonyms for each command in the index. The user could look up *delete*, and the index would refer him to the correct command: *purge*. Because the index lists features, not the functions that can be performed with features, the user with a specific function in mind is not well served by an index (Young 1990).

A possible method of overcoming this deficiency is listing, with each feature, the common functions related to that feature. For example, if feature "A" is frequently used in "Formatting a matrix," then that function would be mentioned in the section on feature "A". The index would list "Formatting a matrix" as an entry, indicating all of the page numbers where "Formatting a matrix" appears. This would allow the user to look up all of the features that help perform "Formatting a matrix." A typical attempt to provide such indexing is the task-oriented menus in the Help system of the IBM Virtual

Machine System Product VM/SP; the Help menus list tasks such as "Communicate with other users," not just the commands that accomplish the task (Dorazio 1988). Better indexing is valuable, but it cannot solve the whole problem. Unfortunately, some users tend to think up and guess just too many "wrong" words and too many functions — and names for them. Users could also envision functions the software could perform that the designer did not anticipate.

"Successful systems often combine two or three techniques . . . The Info system from Convex provides three routes to information: menus, topics, and commands" (Horton 1990a). Some software companies publish a tutorial manual containing illustrative examples for the user to work through. These tutorial lessons demonstrate the functions available to the user. A tutorial program — a very expensive option — is the automated version of the hardcopy tutorial manual.

"Online documentation" was the earliest form of user online help. When software engineers enabled users to do more than just scroll through the index or table of contents, note the page that appeared to be relevant, and scroll to that page, it became "online help". To ask for help on a specific subject, users would interact with the computer by specifying what was needed, either by "touching" an item in the index or table of contents, or entering a key word or phrase. The computer would then give an explanation of the key word or phrase. These online documentation systems were the first processes that could honestly be called "help" systems (Young 1991a).

Interactive context-sensitive help is one of the most recent developments:

"The help you get depends on where you are in the program when you request help. If you are executing a particular command, for instance, you would get help on that command. If an error has just occurred, then an explanation of that error will result" (Horton 1990).

Some systems, such as the Integraph family of computer-aided design systems, organize the help messages by object and by command. In an interactive protocol by which there is normally a specific set of objects currently active, a help message can be activated without asking the user to clarify what help is needed. A more common and less elaborate facility that is also called context-sensitive is that of the Macintosh System 7 interface. The computer knows what screen the user is working with; this enables the user to touch or specify the type of help wanted by pointing at the object or command in question after invoking the Help command. The computer gives a short help message explaining the object on the screen. Messages usually have ad-hoc structures; hence, software writers must write a help message for every object the user could point to. "Many help facilities . . . link context-sensitive help with an online users reference manual" (Horton 1990).

Dynamic Help differs from context-sensitive help in two ways: *protocol* and *construction*. The protocol for context-sensitive help requires the user to invoke help, to indicate for which object or command the help is being sought, and to exit from help. The protocol for Dynamic Help, by contrast, is not interactive; the user activates objects or commands *before* invoking help. This allows the Dynamic Help message to be truly specific to the situation, rather than just being a canned blurb about a given object or command. Note, however, that the Dynamic Help protocol presupposes that it is possible to *activate* commands or objects without *invoking* them. Otherwise, if the basic interactive procedure has what Dr. Young calls "premature closure" (i.e., a keyboard entry or screen touch is acted upon instantly without waiting for the user to press <RETURN> or answer a *confirm* question), situations are too vague for Dynamic help to be useful.

Dynamic Help messages not only differ from context-sensitive messages in protocol, but also in construction. A Dynamic Help message is assembled from fixed and variable components rather than retrieved (Young 1990). This makes it automatable, whereas in a context-sensitive help system, someone must specifically write each message verbatim.

Dynamic Help is neither interactive nor does it require hundreds of different structures for help messages. It is applicable to programs in which the context is quite narrowly defined by such information as cursor location, highlighted objects and commands, and so forth. A user can perform a task at each location within the program. Regardless of whether the user is in the right place, that location in the program has a specific task allocated to it. The Dynamic Help Module assembles a message explaining everything about performing that task in the current context. In a narrowly-defined context, the help message needs to contain only three things. When in the right place, the user needs either instructions on how to accomplish a narrowly defined task or an explanation of the task If he is in the wrong place, he needs instructions on how to leave this context to find a more appropriate one to suit his purpose (Young 1991a).

Dynamic Help has a significant advantage over reference and tutorial manuals, because the Dynamic Help Module has a means of deriving all of the questions that a user could ask. The user is not required to narrow the scope of his question and use an index or table of contents. Dynamic Help's main advantage over context-sensitive help is not in user interface. (Context-sensitive programmers could create a help message for every location within a program, and these messages could be just as understandable as Dynamic Help messages. Also, a context-sensitive help program will outperform Dynamic Help in word processing applications where a user is confronted with multiple options for each location within the program.) Dynamic Help's great advantage over context-sensitive help is that it is automatable.

Captains Walter Barge (Barge 1991) and Stanley Haines (Haines 1991) demonstrated that software engineers and programmers can use Dynamic Help to generate Dynamic Help modules using a reusable format. This saves an enormous amount of system development time. There is no requirement to write and store every help message in a database.

Dynamic Help uses generic grammar structures whose "slots" are filled by querying data values and dictionaries of objects, formats, commands, functions, variables, domains, and protocols. The Dynamic Help module can construct help messages from variable parameters, fixed text, and dictionary entries. By writing dictionaries, system designers can avoid writing hundreds or thousands of different help messages (Young 1991a).

1.1.2 Background of the Dynamic Help Generator

At the Georgia Institute of Technology, AIRMICS is conducting research on Dynamic Help in support of the US Army's Installation Support Module (ISM) project. AIRMICS' goal is to improve the resulting system's accessibility to novice and intermittent users. New users have difficulty moving from one program to another without extensive training. Additionally, users who do not use a particular program for a period of time forget parts of the language and protocols of its interface.

AIRMICS is working on the Army's Central Issue Facility Software (ACIFS) to make it more user friendly by adding Dynamic Help to it. The cost to rewrite the ACIFS interface, the alternative to Dynamic Help, could be high. Rewriting would mean changing the commands and procedures that a user must perform to run the program. ACIFS would need to be organized and labeled in such a way that even novice users would not need to seek help. This would be costly. Dynamic Help is a less expensive alternative.

AIRMICS performed the programming while Dr. Young designed the Dynamic Help software shell that fits over the current ACIFS. Georgia Tech software engineer Christopher Smith wrote the Informix-4GL code used for the ACIFS Dynamic Help shell. AIRMICS has another coding system, AT&T's Application Connectivity Engineering (ACE), that could speed the conversion of Dynamic Help to other Army database software. Dr. Jerry McCoyd and Mr. Smith are converting Dynamic Help from Informix GL to ACE.

Dr. Young enlisted several US Army officers in graduate programs at Georgia Tech to assist with the design and testing of

Dynamic Help. Captain Renée S. Wolven tested the first implementation of Dynamic help on ACIFS using supply clerks, and she found that it got the clerks past stumbling blocks and reduced errors for careful clerks. Unfortunately, the ACIFS test software is slow, and the system takes a fairly long time to generate the Dynamic Help messages on the user's monitor screen. Captain Wolven documented these findings in her master's thesis (Wolven 1991).

Captain Walter S. Barge, II, in his "Universal Software Documentation via Dynamic Help" research report, May 1991, investigated the possibility of designing Dynamic Help into future database software. Using the principles of Dynamic Help, he designed a workable Dynamic Help system for student course registration at a hypothetical university. His strategy specified a package of tools to define the generic structure and dictionaries of the Dynamic Help Module. The results show that Dynamic Help messages can be automated, allowing the designer to write dictionaries instead of performing the much greater task of writing all of the possible messages. The strategy serves both the needs of initial documentation and those of Dynamic Help.

For existing software, Dynamic Help is equally attractive. Captain Stanley K. Haines is investigating the possibility of automating Dynamic Help for other US Army interactive software (Haines 1991). Captain Haines is determining if Dynamic Help and related Embedded User Support (EUS) techniques should be built into subsets of the general population of the Army's ISM software. He will propose a cost-effective procedure for implementing EUS add-ons for those ISM programs that will provide the greatest return from EUS; this procedure uses the Dynamic help Generator.

1.2 The Dynamic Help Generator

This paper reports on the development of the Dynamic Help Generator, an automated software designer's utility capable of economically creating Dynamic Help modules for existing U.S. Army installation level software. The Generator defines the architecture and structure of Dynamic Help messages as well as the dictionaries that make up the various help messages. The Dynamic Help Generator enables software engineers and programmers to quickly create Dynamic Help modules that will function on most existing installation software.

Section 2 reviews the work which led up to the Dynamic Help Generator, including the modification of ACIFS under the ACE programming development system. Section 3 reports the design of the Dynamic Help Generator — its aims, its architecture, the structure of Generator-produced messages, the structure of the Generator-produced knowledge bases it provides, and an automated Dynamic Help illustration.

Finally, Section 4 evaluates the Dynamic Help Generator as a set of designer's tools: the Generator's requirements, the ACIFS implementation, and generic Generator specifications.

2. Technical synopsis of previous work

ACIFS, as part of the ISM project, was the first U.S. Army software to incorporate Dynamic Help into its user interface. ACIFS #L08-03-02 had a user interface that was sufficiently complex to fully test the concepts of Dynamic Help. AIRMICS created a prototype Dynamic Help module for ACIFS using Informix-4GL software, and CIF clerks tested the prototype module. When Dynamic Help proved successful, it was incorporated into the production version of ACIFS for use in the Army's Central Issue Facilities (CIFs).

2.1 Dynamic Help in ACIFS (a technical explanation)

2.1.1 General Principles

The "Specification of User Requirements and Dynamic Help System Standards for EUS project and CIF Conversion" explains the the general user requirements for the first Dynamic Help system. It calls for "relatively superficial [requiring minimal changes to an existing system] and inexpensive software improvements to make interactive programs more user-friendly . . . for novice and intermittent users" (Young 1990b). Dynamic Help leaves in place the existing help facility if one exists; the two kinds of documentation are complementary.

Dynamic Help is not forced upon the user; he must invoke it. The user highlights objects and commands to get into a specific context, then he can either press the carriage return to invoke the highlighted command or press the Dynamic Help key to invoke Dynamic Help. Dynamic Help depends upon the current context of the interaction and the current state of data.

In a DBMS-based program such as ACIFS, an interactive session can be viewed as a series of database transactions that the user controls. The user exercises control by navigating via menus and cursor keys to a desired *context* where the system is ready to perform a given transaction. By invoking Dynamic Help while in a context, the user can verify that he is in the right "place" (context) and can receive information about the meaning of the transaction (its consequences if completed), how to complete or abort it, and how to move somewhere else.

Dynamic Help works best when a user can easily get into a context that allows help messages to be as specific or as general as he desires. *Premature closure*, where the user cannot highlight commands and objects without invoking them, provides shortcuts for the expert user, but it prevents detailed Dynamic Help. For example, if a cursor is on a menu screen and the user cannot type "2" without invoking menu item "2" and going into another screen, then there can not be a Dynamic Help message that explains only item "2" in detail.

Lack of a neutral context, where whatever object the designer assumes is wanted by the user is automatically activated upon screen entry, also provides shortcuts for the expert user, but it prevents general Dynamic Help. For example, if a user gets into a menu screen where item "1" is automatically highlighted and there is no neutral item, then there can be no Dynamic help message that explains only the overall group of items.

2.1.2 The Dynamic Help prototype in ACIFS

Dynamic Help, as defined by the specifications for user requirements and Dynamic Help System standards for the EUS Project and CIF conversion (Young 1990a), and the implementation of the prototype Dynamic Help system for ACIFS, was based on ad-hoc definitions of context and ad-hoc message structures. There are basically two general contexts in ACIFS: being in a menu screen or being in a data-entry screen.

In a menu screen, the user can select X (exit) to get to the parent (next higher) menu screen (in the Main menu screen, this action causes exit from the program), or select a menu item to get to a child screen, which is either a next-lower menu screen or a data entry screen. Screens are arranged in a hierarchical tree with the Main Menu screen at the top (root) and the data entry screens at the bottom (leaves). All transactions are accomplished by entering data on a data entry screen.

In a menu screen, the user's concern is *navigation*. The user's goal is either to exit the program or to get to a data entry screen to perform a transaction. When the goal is to get to a data entry screen, the user's specific concern is which, if any of the current choices leads toward that screen; if none of the choices lead to the desired screen, then X is the appropriate selection (to get to a higher menu screen, in which one of the choices leads toward the desired data entry screen). Thus, the main requirement for messages in a menu screen is to give information about the consequence of choices, including choice X.

In a data-entry screen, the user's concern is likely to be either the format or content of a field highlighted by the current cursor position, what will be the consequences of making the current data entry, how to make closure of, or abort, the current entry, how to get to another entry (*local navigation*), or how to get to another screen (*global navigation*).

The designers selected a fixed structure for a Dynamic Help message. It consists of a set of sentences (some possibly null) in a fixed order (Wolven 1990a, Wolven 1990b, Haines 1990a, Haines 1990b, and Haines 1990c):

- Context ("Ready to")
- Global navigation
- Navigation (local)
- Meaning
- Domain
- Format
- Content
- Completion
- Quirk
- Pre-Completion Correction
- Undo
- Key Effect

The Ready-to (first) sentence within the Dynamic Help message tells the user that the system is "Ready to" perform a particular task or function. The user is "Ready to" execute something within the program. The program retrieves this sentence from a dictionary to tell the user what he can do. Time constraints to create a Dynamic

Help module and the limitations on the Informix-4GL forced Mr. Smith to use sentences instead of parts of sentences for the test Dynamic Help module. Entire sentences from the Dynamic Help message were stored in dictionaries in the database rather than having parts of sentences stored in separate dictionaries. The size of each message was limited to two pages.

The *global navigation* sentences tell the user how to get back to the screen he just left or how to proceed to the next screen. (The next screen may be the second page of the message or the screen that the user was working on.) Navigation to each "next screen" is simple: pressing the carriage return moves the individual to the next screen, or moves the user through the message and back to the position where the individual originally needed help.

The specifications call for four more sentences, *meaning*, *domain*, *content*, and *format*, that would be provided for the user only when necessary. For ACIFS, the meaning sentence would describe a Unit Identification Code (UIC) to a user along with an example: "WEOQAA". Captain Wolven provided the entries for the meaning sentences. The domain sentence gives a range of values that a user can enter: a social security number would have a range of values from 1 to 999999999. The content sentence lists of all acceptable entries: when the user enters whether the soldier is male or female, the acceptable entries are either "M" or "F". The format sentence describes an acceptable arrangement for an entry. An acceptable entry may be AA###; the sentence says "Where AA is . . . and ### is . . ." The software engineers determined which of these sentences fit in each message.

The next part of the Dynamic Help message contains the *procedure* sentences: the *completion* sentence tells the user how to complete an entry: "press the return key." An additional *quirk* sentence was added to explain information ACIFS automatically provides to the user when the user inputs data or limitations on the information that ACIFS provides. This could be a dollar value on a report of survey or that ACIFS provides data in monthly increments.

The *pre-completion correction* sentence tells a user how he can correct an incorrect entry: by "pressing the backspace key." The *undo* sentence explains the procedure for a user to go back to a previous entry on a screen and make a correction. Undoing an entry involves pressing a combination of arrow, backspace, overstrike, and return keys. The *key effect* sentence explains certain keys within the ACIFS program: ESC, DEL, BREAK, F1, or any other key that performs a specific function.

2.1.3 Initial Dynamic Help Generator architecture

In the specifications (Wolven 1990a, Wolven 1990b, Haines 1991a, Haines 1991b, Haines 1991c) the identification of context was ad-hoc: the field names in data entry screens. The Dynamic help "module" was not a separate program that consulted a context dictionary, but logic, scattered throughout the program, that captured the current contexts.

The sentences were written in a semiautomated way. Sentences were not completely constructed at run time; rather, they were to be partly constructed and partly retrieved. Actual complete messages were provided to the programmer. The writing of these messages by the designers was semiautomated by the use of templates, one for menu-screen messages and one for data-entry-screen messages. Text macros were defined for frequently-used sentence fragments, clauses and phrases.

Appendix 2 gives a sample of the specification of a menu-screen Dynamic Help message. Appendix 3 gives a sample of the specification of a data-entry-screen Dynamic Help message for both global and local messages. Appendix 4 gives some entries from the dictionary of text macros.

Dynamic Help is most applicable to Data Base Management System (DBMS)-based systems like ACIFS. Because ACIFS is organized by function, software engineers can exploit the forms utilities, transparent data-state query, and display capabilities found in systems rooted in a relational DBMS such as ORACLE or INFORMIX (Young, 1990a).

ACIFS is Informix based. Mr. Smith used Informix-4GL to create the first Dynamic Help module. ACIFS is organized into a series of hierarchical menus. At the lowest level menu, each submenu has a multi-page forms screen. Almost all of the screens are one of two types of menu screens or one type of form.

Dynamic Help asks for the current command for the field and the datum the user is on. A datum has a name, a value, and an address (a place in memory) where a value is stored. The system's "pointer" counts to the right and down to reach a location. Various commands can have various structures, but the Dynamic Help module can locate the commands. There is considerable indirect referencing: "Num" by convention is an integer value.

Mr. Smith began the automation of Dynamic Help with the original version of ACIFS #L08-03-02, created by the Software Development Center in Atlanta. He exploited the use of the window utilities in Informix-4GL. Because ACIFS was a DBMS system, there was no need to have a transparent data-state query and display capabilities in order to add Dynamic Help to ACIFS while it was in Informix. The specifications for system standards for the EUS conversion require context monitoring. The context is the identity and purpose of each screen and field. All screens and fields are kept in dictionaries. The ACIFS system is always in a current state: the variables each have a value. State monitoring is continuous within the ACIFS system.

Informix-4GL has RAM limitations, and Mr. Smith felt that the Informix-4GL system would not have enough memory available to hold all of the Dynamic Help dictionaries. The first Dynamic Help module only partially automated Dynamic Help for the original version of ACIFS. The first Dynamic Help module automated the screen and the data field by having written parts of the message on the computer's disk, it contained 2000 lines of code. It was actually 20 small modules each connected to one of the 20 parts of the ACIFS program. These 20 small modules were connected to the Dynamic Help module. It had an auto-current program state message. Future versions of Dynamic Help will have just one module capable of retrieving information from the different parts of ACIFS (Smith, 1991).

The first Dynamic Help module recalled sentence size pieces of data to create its messages. The module made Standard Query Language (SQL) queries to a database instead of constructing messages from the codes that make up ACIFS. SQL enables a software engineer to store and retrieve information. Future versions of Dynamic Help modules will use parts of sentences to create messages.

The logic behind the organization of the code specifies that the code lines be spread throughout the program. The code lines are added to each form. Each form is generated using the forms utility of Informix-4GL (a subroutine). For example, the active screen will have a function attached to it, and the Dynamic Help module can use the line numbers when recalling information from the code lines. The module can then construct sentences for the Dynamic Help messages from pieces of the code lines.

DBMS-based interactive programs maintain identifiers that keep track of the user's current screen and previous screen. The ACIFS Dynamic Help system needs continual access to these two screens and two other items. Dynamic Help needs to know the next screen (ordinarily-next screen) that the user could move to and the currently-highlighted object on the screen. Dynamic Help then keeps these four items in its own knowledge base. Since the ordinarily-next screens are not accessible from ACIFS, Captain Wolven provided a table of ordinarily-next screens. Dynamic Help must also be able to make (SQL) queries to the ACIFS database to determine the state of the system (Young 1991a).

Once the Dynamic Help module takes control from ACIFS, Dynamic Help has a pop-up display that will temporarily write over the existing screen by creating a window. ACIFS is still operating, but the Dynamic Help module is in temporary control of the user interface. If necessary, Dynamic Help will suppress any existing messages. This suppression is the only programming required outside of the Dynamic Help module.

A user needs a keyboard key that will highlight an item that he needs help with. The keyboard for the Excel VT100 monitor does not have the traditional F10 help key, so Mr. Smith used the F4 key instead. Later, when Captain Haines demonstrated Dynamic Help at The Army Software Development Center in Washington D.C., Mr. Smith selected the Control 'W' key to guarantee a usable key (Smith 1991).

Since the start of the project, the message tone is unchanged. Messages contain only declarative sentences about objects. They are concise and give the facts to the user. Messages are not too personal; they contain no words like "you". "Ready to" messages do not belittle the user by telling him what to do.

The original ACIFS Dynamic Help messages were written in plain English, yet they worked just as well for the user. In fact, the plain English messages may have been even more understandable to the user than the fully automated dynamic help messages (Young 1991a).

The first Dynamic Help Generator was designed to test the feasibility of Dynamic Help because it behaved like the envisioned Dynamic Help module. The original ACIFS dynamic help messages were written in plain English, and the first Dynamic Help Generator did not enjoy the efficiency of a completely automated Dynamic Help, yet it worked just as well for the user.

The Dynamic Help Generator is one step closer to full automation. "Ready to" is only one sentence. It does not have a single structure. The software engineer may need to automate as many as eight sentences for a Dynamic Help message. Each command will have a structure. Currently, "ready to" sentences have only 4-5 structures. (Young 1991a).

2.2 Conversion of ACIFS to ACE

Most U.S. Army databases are separated by type and location; ACIFS is a small subset of these databases. Ideally, the Army would have one master database; unfortunately, an Armywide database would be unmanageable. Instead, the Army can have a standard software language and standardized databases (Young 1991a). The most promising language is C code, and the ILIDB is the standard Database for all ISM programs. One benefit of standardized databases is the learning leverage gained by software engineers from using the same type of database. Programmers would not have to relearn information for each new application that they attempt.

A new Dynamic Help Generator is being constructed that will automate the specification for future Dynamic Help modules. The current SQL routines and database characteristics of the first Dynamic Help module should require few changes. The ideal situation is to keep programmers from specifically importing new code into the Help Message Generator. Unfortunately for the software engineers, Informix-4GL restricts the capabilities of Dynamic Help by limiting what type of information can be gathered by the Dynamic Help Generator. Informix-4GL is a processing language for databases that can perform some logic. Informix-4GL and ACE perform similar functions; unfortunately, Informix-4GL is not easily translated into ACE. ACE uses C language to perform SQL queries (Smith 1991a).

The Army has an abundance of ISM programs written in C code and compatible with ACE. The current specifications require translating the first Dynamic Help Generator's Informix-4GL routines into the C language of ACE. Some conversion is done by hand. The ACIFS program development under ACE is a process whereby the Informix-4GL DBMS code is rewritten into C code so that the ACE utility can be used by software engineers. Mr. Smith is upgrading the Informix based ACIFS into an ACE version of ACIFS based on C code. ACE is front end loaded and has the potential to be platform (hardware) specific. When a software engineer automates a Dynamic Help Generator, he refers to automating the repeatable procedures for a set of tasks.

Once the ACE version of ACIFS #L08-04-01 is created on the AT&T 3B2 mainframe computer, software engineers should be able to use the ACE utility on any C language computer. The programmer who seeks to create a Dynamic Help module for another DBMS type of software has a specific set of rules to follow. New Dynamic Help modules can then be written in C code with the aid of the Dynamic Help Generator (Smith 1991a).

2.3 Automatability of Dynamic Help

Captain Haines participated in the initial work on Dynamic Help automation. Before the automation of ACIFS, Dynamic Help was fully designed for implementation in the ACE programming environment. Captain Haines implemented an automation method by hand for a subset of about 7% of ACIFS. He generated 46 messages and compared them with the prototype Dynamic Help messages. Wherever prototype messages imparted information not available from the automated messages, he added Ad-hoc sentences to the automated messages. Next, he compared the total amount of text that needed to be authored and stored in both the prototype messages and the automated messages. The results show that the automated messages required the authorship and storage of only half of the information required by prototype messages. When compared with the work needed to create dictionaries for the prototype Dynamic Help module, the designer's work was cut in half for automated message dictionaries. Thus, automation of Dynamic Help is possible and can succeed in reducing authoring effort and storage requirements.

To estimate Dynamic Help automation efficiency for the *whole* ACIFS program, Captain Haines examined the proportions of objects and acts of each type that were already defined in the dictionaries for the 46 sample messages. For example, the sample covered only 6 out of 100 data-entry screens, so 94 additional rows would be needed in the screen dictionary. On the other hand, information about clothing items and soldiers, and all generic information, was already in the dictionaries for the 46 messages. Although the growth in authorship and storage would be less than linear for the prototype messages, it would be *far* less than linear for automated messages. Captain Haines estimated that the automated messages for all of ACIFS would occupy about 47K characters, while the prototype messages would occupy about 247K characters, representing a *fivefold* reduction in designer's work and storage.*

A Generator can provide a generic context-vector structure, a set of generic act types and levels, a set of generic object levels (but not types), a generic message structure (a set of sentence structures), a generic set of dictionaries, and a generic logic for performing context identification, message construction, message assembly for display, and I/O control. The Dynamic Help designer must provide the specific context-vector element names and types, identify the variables that track them in the program, provide the object definitions (types and levels), and fill all the dictionaries.

The Dynamic Help system consists of four parts (explained more fully in section 3.1): the input/output and control logic, the context monitor, the knowledge base system, and the message production system. The Generator can partially automate the design of all parts except the context monitor. First, the *input/output and control logic* part prepares the program to work with the Dynamic Help module. It assigns a key to invoke Dynamic Help, and it monitors that key and the control system. It also puts up and takes down messages from the screen. The input/output and control logic is not automatable (Haines 1991).

Second, the *context monitor* must update the context continually from the program as the context changes. It must be part of the running program — not part of the Dynamic Help module. As an

* Not reported in Captain Haines' thesis is the fact that the prototype message files actually occupy 290K characters, but this is because they were not semi-automated to the full extent assumed. Some redundant storage was permitted to avoid processing at run time.

example, the current screen is one of the elements of the context vector. If the program keeps a variable that identifies the current screen, context *monitoring* is already accomplished (the Dynamic Help module will be able to perform this element's part of context *identification* simply by reading the value of the variable). On the other hand, if the program simply erases the display and puts up a new one without bothering to maintain an identifier for it, the Dynamic Help module designer would have to add screen-tracking logic throughout the program to feed the context vector. The context monitor part of the system is not automatable, yet the Dynamic Help Generator's documentation could give software engineers advice to assist them with providing for context monitoring.

Third, the *knowledge base system* part of the Generator is automatable. Part one is the *Dynamic Help dictionaries* that hold the pieces of the Dynamic Help message sentences. The second part is the *knowledge maintenance system*, the complicated part. The challenge is keeping the dictionaries updated. It is possible that a soldier could draw equipment from CIF and turn in some of the equipment before the dictionaries could be updated from the database. If the clerk invoked Dynamic Help, the message would contain old data. In this situation, the same soldier is being entered into the database each visit, yet the clerk would only have the original data stored in the dictionaries from the session initiation. One way to avoid this is the *Database query system*, which depends on DBMS-base tables and uses these tables from the DBMS-base for dictionaries. A change in the database will automatically be reflected in the dictionaries.

Unfortunately, database queries are 1000 times slower than memory queries (Young 1991a).

A faster way to maintain the knowledge base is the knowledge update system. The *knowledge update system* updates the knowledge base each time it changes the database. The *knowledge reconciliation and initiation system* does two things. It initially queries information from the database to fill the empty dictionaries. It also periodically reconciles knowledge-base data with database data.

Finally, there is the *message production system* that identifies contexts and uses context vectors to construct messages. (Section 3.1 describes context vectors.) The message production system contains the *context identification system.* This routine asks the dictionaries for values to fill each blank within the Dynamic Help message. A message may have 14 blanks to fill. The *message construction system* uses fixed sentences and a fixed order. It assembles the parts of the sentences, and it can be automated using ACE.

After simulating the automation process by hand, Captain Haines validated the automation process by measuring the savings in message size achieved by automation. He recommends

> that a Dynamic help Generator be produced by extracting the generic tasks in dictionary design, knowledge maintenance, context identification and message construction for the ACIFS automated Dynamic Help system currently under development (Haines 1991).

3. Design of the Prototype Dynamic Help generator

3.1 Aims of the Prototype Dynamic Help Generator

The Generator built during the Summer of 1991 at AIRMICS is a formalization and parameterization of subroutines and practices, produced in house by AIRMICS as a set of ACE routines written in C, the ACE shell language, and a report. Software engineers can use these subroutines and practices to automate Dynamic Help in the ACE programming environment. The ACIFS program is the first application.

The intent of the prototype Generator is to simplify the addition of automated Dynamic Help to DBMS-based application programs in the ACE environment. Future versions of the Generator could be published as a set of software developer's tools for use in the ACE environment (Smith 1991).

The General Specification for Dynamic Help Generators (Young 1991b) provides a list of software items for generic Dynamic Help systems:

Software items outside the Dynamic Help Module

1. I/O AND CONTROL LOGIC
2. CONTEXT MONITOR

Software items in the Dynamic Help Module

3. KNOWLEDGE BASE SYSTEM
    - 3a. Message and Context Specifications
    - 3b. Dynamic Help Dictionaries
    - 3c. Knowledge Maintenance System
        - 3c1. Database Query System

3c2. Knowledge Update System

3c3. Knowledge Reconciliation and Initiation System

4. MESSAGE PRODUCTION SYSTEM

4a. Context Identification System

4b. Message Construction System

Item 1 is specific to the programming environment and would not appear in a generic Generator; however, the prototype Generator is ACE-specific. An aim of the prototype Generator is *to provide I/O and control logic* for a Dynamic Help system. This involves providing a Help key to invoke Dynamic Help for the Extended Terminal Interface Program (ETIP), providing for a display of messages constructed by Dynamic Help, and providing for exit from the messages to restore the previous screen display, context, and data state.

Item 2 is specific to the application program. In ACIFS, for example, there are about 700 reachable contexts, and they can be accurately described by naming the screen and the highlighted or cursor-resident field in the screen. These fields are almost always a menu item if the current screen is a menu screen, or a data-entry item if the current screen is a data-entry screen. The only exception is a few pop-up fields in which yes/no responses are prompted. Despite the simplicity of a screen/field definition of context, and its generality for DBMS-based applications (almost any application written in the forms-definition utility language of a DBMS-based system will have such easily-described contexts), contexts are *not* described in this way for automated Dynamic Help.

An automated Dynamic Help system must identify contexts in terms of lists of active objects and active acts of defined object types and act types. Some of these listed objects and acts are close to their counterparts in a screen/field definition; for example, if the cursor is on "truck 45" in the "wheeled vehicle" screen, then these two facts directly translate into two values in the context vector. The context can be thought of as a point in the program that the vector describes.

The context vector is a one dimensional list of text fields. Each field is a group of 1 to 14 characters. Automation requires the context vector to functionally determine the data to fill each variable part (slot) in each sentence.* Each slot can be filled either directly or indirectly using a set of rules expressed in the dictionaries and sentence specifications. A slot can be filled with the name of a context element (e.g. screen, soldier, or data entry), the value of a context element (e.g. the turn-ins screen, soldier 472544982, or SSN entry), the text retrieved from the dictionaries, a value queried from the database, or with a combination. The characters retrieved are functionally determined by the values of one or more context elements.

The context vector contains more elements than are minimally required to identify the context. If the lowest-level act (which is always a data entry, a selection, or a response to a question) is part of a higher level act (like a procedure that is a collection of the lowest-level acts), the active procedure is also listed in the context vector. The context can include up to four levels of acts. Similarly, if the lowest-level object (which is always the datum being entered or the

* A functionally determines B (written A $\rightarrow$ B) if there is a procedure to determine the value of B given the value of A.

selection being made) is part of a collection of objects, the active collection is also listed in the context vector. There can be any number of levels of objects. Besides these redundant context-specific active higher-level acts and objects, there are *appositives* in the context vector. For example, if a screen has a numerical identifier, a short name, and a long descriptor, both available in the program, these can be carried in the context vector as appositive elements.

In general, the designer must identify how the application system tracks every fact needed in the context vector. He needs to provide a variable, if one does not already exist, that is updated wherever each fact can change. Since these variables must be provided at various points throughout the application's subroutines, and since the *types* of objects and acts are specific to the application, there is no hope of automating the translations from program situation to context value. Thus, item 2, the context monitor, is outside the scope of the Generator. (However, context identification — the Dynamic Help module's method of interpreting the context vector — is automatable; see items 3a and 4 below.)

The Generator provides a structure for Item 3, the knowledge base system. Dynamic Help will have its standard sentence specifications, standard data structure for context specification, and standard dictionaries regardless of the application (of course the *contents* will vary). One aim of the prototype generator is *to provide message specifications and a structure for context specification* (item 3a); another aim is *to provide templates for the Dynamic Help dictionaries* (item 3b).

Item 3c, the knowledge maintenance system, appears to be too variable for full automation. The original concept included keeping the knowledge base of a DBMS-based application entirely in the database and querying information (from disk through the DBMS engine) at run time (Young 1990a). However, the programmer discovered long delays when the Dynamic Help module must go to the disk to generate messages (Young 1991), and messages must be constructed from a memory-resident knowledge base. Unfortunately, the knowledge base must duplicate some data that is stored in the database, and this duplication makes it necessary to provide a knowledge update system (item 3c2) and a system for reconciling the knowledge base with the database. The reconciling system can also serve as a means to initially fill the dictionaries with database data (item 3c3). If run-time queries are quick enough, all this complexity can be avoided. The delays encountered in the ACIFS system may just be artifacts of poor performance of the version of the Informix DBMS engine used in ACIFS. We omit the knowledge maintenance items (3c, including 3c1, 3c2, and 3c3) from the scope of the Generator.

Item 4, the message production system, can be fully automated. The aims of the Dynamic Help Generator include *to provide a generic context identification system and a generic message construction system.* Context identification is simply the task of reading values of active objects and active acts from the context vector. While this task is automated, the context monitoring that stores those values is not.

3.2 Architecture of the Generator-Produced System

The thoroughness of the system design requirements analysis will determine the ease of incorporating Dynamic Help into a system. Those systems created with operational modeling and implemented through automated code generation are incompatible with Dynamic Help. The code generators create codes in machine-oriented, not human language. The requirements analysis must define the objects in the system and the functions within the system. The ISM systems with their high usage rates usually require a more detailed user interface compatible with Dynamic Help (Barge 1991).

The diagram in Figure 1 depicts the architecture of the Dynamic Help System whose design is to be automated through the Dynamic Help Generator. The following explanation describes the specifics of the architecture, using the Dynamic Help System's run-time activities, from the time that the user presses the Help key to the point where the Dynamic Help module returns control to the application program.

The Dynamic Help System architecture is made up of 4 processes:

- Identify context vector
- Retrieve message parts
- Combine message parts
- Display message

When the user presses the help key (F10), the system interrupt is activated. (An interrupt is a condition whereby a routine program is temporarily suspended). The algorithm interrupts the current program with a procedure call and invokes the Dynamic Help module. The algorithm pushes the Dynamic Help module to the

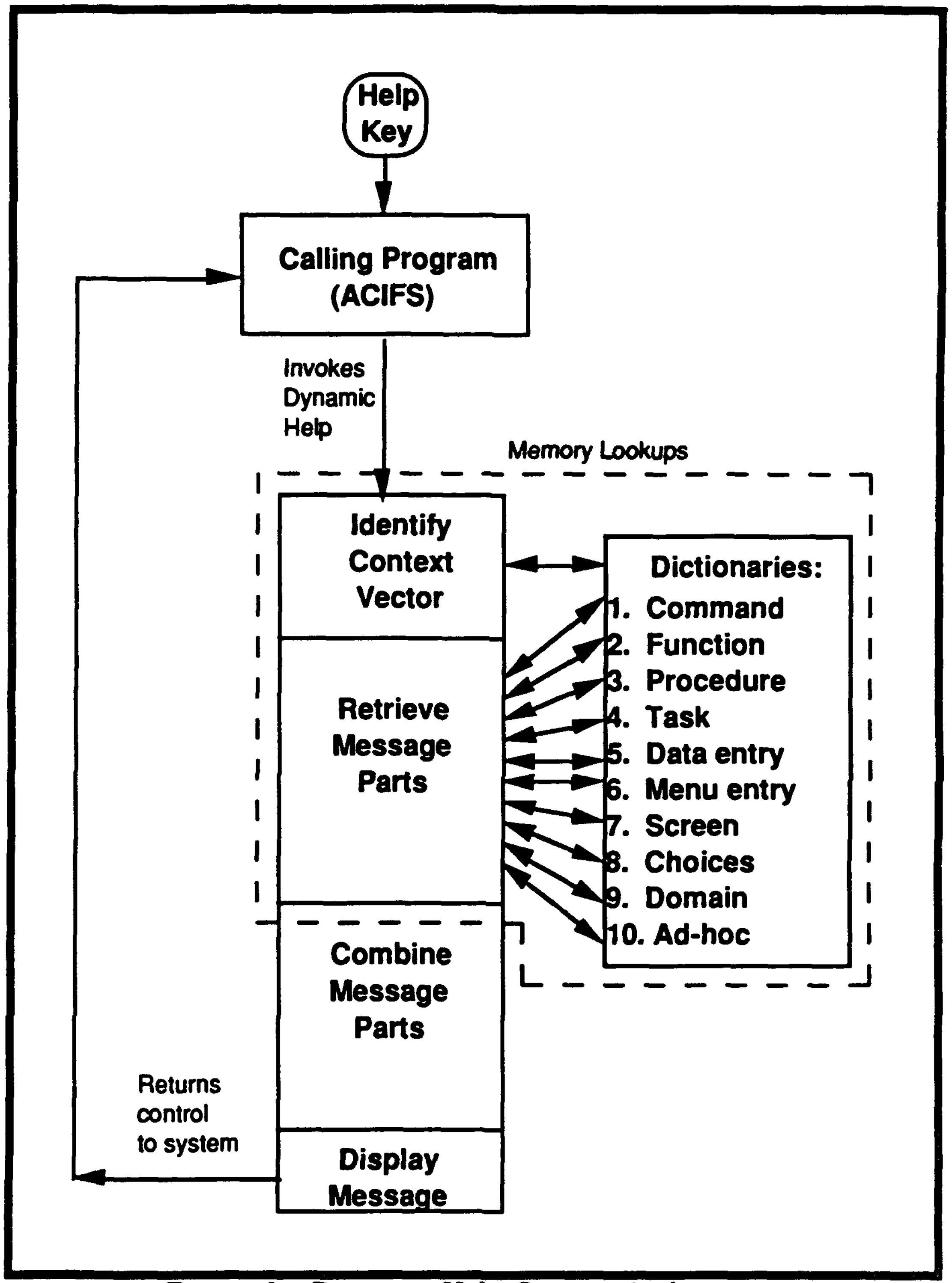

Figure 1: Dynamic Help System Architecture

top of the stack of the system's available program software (Smith 1991a).

Ideally, all Dynamic help will operate within memory. Using the ACE programming tools, Dynamic Help builds arrays for session and screen entries that make information available in advance. When a user logs onto a system, new information is called up from the disk and stored in variable names in arrays. The Dynamic Help module uses algorithms from the Dynamic Help Generator to retrieve information from the arrays (dictionaries). The information could be codes that *identify* the *context vector* or characters contained in each dictionary variable. These variables fill the various parts of the Dynamic Help message sentences.

This means, that with 10 dictionaries, there is no need for disk operations to create help messages. There would be 5 object dictionaries (data entry, menu entry, screen, choices, and domain) and 4 act dictionaries (command, function, procedure, and task). The tenth dictionary is ad-hoc; it holds one or more complete message sentences in each variable (Haines 1991).

Based on the active act and active object (ACIFS has a screen and a field identifier) Dynamic Help identifies the context vector. The active object and active act identifiers are unique numeric parameters that identify the current screen and field (either one or the other may be null). The algorithms within the Dynamic Help Generator use memory lookups to find the two numeric parameters in the dictionaries. The dictionaries serve as tables or memory stores; they are indexed by the numeric parameters corresponding to one or more of the variables of the user/program context.

The *retrieve message parts* routine uses the context vector to identify which message fragments need to be retrieved and how to fill in the message strings. The prototype Dynamic Help module would reach down and retrieve one complete string for each message sentence; the ACE programming tools enabled designers to "expand out" each string by creating one string (sentence) from several small strings of data. The Dynamic Help algorithm reaches down into several dictionaries to create seven of the eight sentences. The Ad-hoc sentence is stored in a dictionary as a complete string (Smith 1991a).

The message variables in ACIFS are stored in dictionaries. Each message string has its own index code. A variable within the dictionary may have several numeric parameters associated with it. ACIFS strings are indexed by screen identification only (for global and menu messages) or by both screen and field identification (local messages). In different messages, some of the sentences may be the same. Dynamic Help can call up the same string for more than one message. Strings are also distinguished by type (Context, etc.) Each string whose index matches the numeric parameter is retrieved.

All but one of the eight sentences that make up the Dynamic Help message is automated (Section 3.3 provides a complete explanation of these eight sentences). The *ad-hoc* sentence allows the software designer to clarify automated Dynamic Help messages. It gives the user more specific information about the screen the user is working on and the field the user highlighted. The ad-hoc sentence must be written by the designer.

The message is retrieved and assembled in memory (RAM). *Combining message parts* includes assigning the variables to particular message strings. Dynamic Help must keep track of each type of sentence and place it in a certain order (e.g. the "Ready to" sentence is always first). The message then becomes an array of strings (Smith 1991a).

To define a Dynamic Help message, the software engineer must specify the context and fill all of the sentence variable parts (slots). The values that specify the context identifiers, of each type of active act and each type of active object, determine the context vector's numeric parameter. When there is more than one active object or active act, programmers use fixed (assigned) types. An entry in the context vector tells which object is the active one of each type. In order to fill the sentence slots from the context vector, a software engineer needs an attribute and a name for each context vector item. The *name* of an item is its value. The *attribute* of an item identifies it. Each part of a *fixed verb phrase* is an attribute of a combination of two items in the context vector, an attribute of a combination of more than two items in the context vector, or a compatible function of values in the context vector (Young 1991a).

Combining implies that when the variables are retrieved from storage, the strings are assigned to an array of program variables (a type of character string). The array is built according to how the message parts will appear ("Ready to" first, etc.). Formatting expands the string using the variables retrieved from the dictionaries; it also inserts blanks between items, periods at sentence ends, linefeeds, carriage returns, and tabs where needed.

The variables in the defined Help screen are initially null. This screen is a "form" in ACE. It is a generic page with empty strings (line widths can be up to 60 characters). ACE combines the assigned variables (those in the context vector). The Dynamic Help module does a considerable amount of formatting. The sentences must be broken down to fit into the Dynamic Help message. A routine assists in determining where the break is made. The routine constantly checks the length of the sentences when software engineers assign the strings to message screens. The *combine message parts* routine then maps them onto the screen (Smith 1991a).

The *display message* algorithm from ACE displays the strings and prompts the user. When the user exits the Dynamic Help environment, system control returns to the previous program.

The knowledge update system from the Dynamic Help Generator captures input data, the values of screen entry fields. The same routines are used to organize strings. Dynamic Help uses updates made when a clerk first enters them (SSN, etc.). Dynamic Help replaces old variable values with the new information about the soldier. The difficulty is identifying the correct dictionary. Dynamic Help uses the context vector to find the correct dictionaries to store information.

The two components of speed heavily influence the design architecture (Young 1991a). The order of sentences will determine how quickly a clerk can react to a help message. Although the user may initially need general information listed first to understand messages, the users quickly master general information and need screen specific information during run time. Once the information is displayed with screen specific information ahead of general information, a delay in the display time may not delay the user because the user can be reading the first sentences generated while the machine is writing the rest of the message.

The second component of speed is the speed at which the computer constructs the help messages. The prototype Dynamic Help messages required the computer to send a request to the Informix engine that required multiple disk operations to do the job. Calling up the SSN meant going outside the system to the disk (the database). A Dynamic Help module should not have to perform disk operations like database queries. Using Random Access Memory (RAM), the computer can pull information up 1000 times faster than having the computer call up the information from memory. Therefore, going to a disk means delays in generating messages (Young 1991).

Dynamic Help consists of three related developments: the design strategy, the generator, and the module. The Dynamic Help design strategy is an analytical method used to create the structure and content of Dynamic Help messages. Programmers use the Dynamic Help Generator to transform these parameters into Dynamic Help Module software.

3.3 Prospective Structure of Generator-produced messages

This section documents the message structure design before it was refined, during the Summer of 1991, both by the final stages of Captain Haines' automatability research and by the actual implementation of the prototype Generator. The optimum number of sentences in a Dynamic Help message is eight. The eight sentences allow sufficient automation without constraining Dynamic Help to an overly narrow application. "Dynamic Help messages are built with sentences that contain clauses that contain phrases (informally we use the word 'phrase' sometimes to mean a set of phrases)" (Young 1990). The sentences have generic names; these are "Ready to", ad-hoc, meanings, choices, format, domain, alternatives, and general help sentences. "A complete Dynamic Help message consists of all of its sentences (including any null sentences)" (Barge 1991). Rarely will a message need all of the sentences to provide help for a user.

The sentences are listed in order from screen-specific help to general information. The "Ready to" sentence is first. It orients the user to the primary act that he is trying to accomplish and the objects that this act will manipulate. A clerk may be ready to "issue" a "bag, sleeping" to a soldier. The context sentence is an imperative verb phrase built from dictionaries. This "Ready to" sentence uses the most detailed act in the context, or the lowest act applicable to what the user is doing.

The sentence reads "Ready to" <verb> <direct argument> <preposition> <indirect argument>. The direct and indirect argument statements are noun phrases. They fill a descriptive field for the active object of whatever type. Example: "Ready to" <issue> <clothing> <to> <soldier>. The direct action object is the clothing piece and the indirect active object is the soldier.

The *ad-hoc* sentence is next. It allows the software designer to give the user more specific information about the screen that the user is working on and the field the user highlighted. For instance, "Once NSN entry is completed, a correction to it cannot be made." A clerk must "delete and reenter NSN to make correction." Because it is not automated, the Ad-hoc sentence can explain "quirks, consequences, special input protocols, etc." (Young 1991b).

The *meanings* sentence explains the relevant active act and relevant active object in the context. The meaning of an object is retrieved from an object dictionary while the meaning of the act is retrieved from an act dictionary. One such sentence states "The End Item Code [active object] informs [active act] the supply system about the major end item that the requisitioned item supports or replaces."

The *choices* sentence is fourth in the order. Although the prototype Dynamic Help System actually assembles lists of choices and includes them in the message, this sentence has been revised so that it merely tells the user how to view a list of choices if it is already on the screen, or how to invoke a list of choices, when one is available.

The *format* sentence states the "format for the direct argument of the primary act" (Young 1991b). "The format sentence provides an example of a correct entry or provides a short description of the correct format, whichever is more informative" (Young 1990). Examples are "EXAMPLE: Hat, hot weather" and "EXAMPLE: 8415011841352."

The *domain* sentence seeks to give the acceptable range of entries "for the direct argument of the primary act" (Young 1991b). It is specific to the current field that the user invoked Dynamic Help for. For instance, "Acceptable entries are 1 to 99999."

The *alternatives* sentence lists applicable alternative acts. Included in this sentence are the undo, abort, commit and closure types of sentences. An undo sentence would say "To make a correction, press UP or DOWN Arrow Keys as needed and perform entry again, or press RIGHT Arrow Key Backspace key (non-destructive), overstrike, and press Return Key."

The last sentence is the *general help* sentence. This sentence tells a user how to access context-independent help. (The prototype Generator as actually implemented, following a suggestion in Captain Barge's study, produces a fixed sentence that tells the user how to invoke General Help. The information described here is retrieved at the user's option rather than always included.) The general sentences usually explain the meaning of keys (ESC, DEL, F1, etc.). The navigation sentences, local and global, are now a part of both the general help sentence and the alternatives sentence. Global navigation involves moving between screens, windows, and menus. Local navigation involves moving within a screen, window, or menu. An example of a global navigation sentence is "Ready to go to" <Active menu selection> "where you can" <verb clause>. A user can navigate backward (downward) or forward (upward). There is a limited number of different methods to move within a D-base program like ACIFS.

One component of the screen display is the speed that the user can react to a message. The message architecture places screen specific information first so that a user familiar with Dynamic Help messages can get help quickly and return to the ACIFS program before the Dynamic Help module completes the message. First-time users or users who have forgotten the general help sentence information can read through the message to find more general information at the end of the message.

In short, the situation that the user finds himself in and the data that the user has entered determines the context. The Dynamic Help module uses this context to find the correct sentence structures for the Dynamic Help message, and the module fills the sentences from Dynamic Help dictionaries. The module passes the message to the application program to display the message for the user.

See Section 3.4.2 below for more detailed and up-to-date message structures (those actually implemented) for the Generator .

3.4 Structure of Generator-produced Knowledge Bases

The application program loads the Dynamic Help module and its knowledge base at the beginning of run time. The structure of the knowledge base is constant; it contains the context vector structure, a message structure, and a group of Dynamic Help dictionaries filled with contents.

3.4.1 Structure of the Context Vector

The context vector has a structure that defines active object types and levels, act types and levels, and appositives. Before run time, the context vector is empty. At run time, the operating program fills the context vector and revises it every time the context changes. (More accurately, the operating program revises variables, and the Dynamic Help module reads their values as elements of the context vector.) Context vector values fill messages directly with values of vector elements or indirectly through the Dynamic Help dictionaries. One or more elements in the context vector serve as a *key entry* to each dictionary. The dictionary returns elements to fill sentence *slots* (Young 1991a).

The context vector is volatile (always changing), but the knowledge base is nonvolatile. The parts of the knowledge base that are *generic* are built into the Dynamic Help Generator. The parts that are *application specific* are supplied by the designer using the Generator's tools. The parts that are *volatile*, the values of elements of the context vector, change during a run to reflect the current context.

The structure of the context vector is part of the knowledge base even before the vector is formed. The context vector structure consists of three elements:

- A list of object types
- A list of act types
- A list of appositives

Elements of the context vector are used as key entries for all dictionary queries. An application program variable or function call provides the value of each element. Figure 2 is an example of the structure and contents of a context vector. The "Object of level 5 (e.g. screen level)" is an example. The value of that element (screen) identifies the current screen.

In ACE, the identifier of the current screen is returned upon invoking the function call: "curr_screen". The function call asks for a value of the "curr_screen". The value would be a value of the variable "screen_id". In message construction, the value of "screen_id" is used as a key entry or as part of a key entry to enter some of the dictionaries for a description of the screen or for other text to fill

Context Vector and Its Associated Levels, Types and Program Variables

Level of object or act type	•••	Act of level 4 (Task level)	Act of level 3 (Procedure level)	Act of level 2 (Function level)	Act of level 1 (Command level)	•••	Object of level 5 (e.g. Screen level)	Object of level 4 (e.g. Docu-ment level)	Object of level 3 (e.g. Soldier level)	Object of level 2 (e.g Inven-tory item)	Object of level 1 (e.g. Screen field level)	Appos.
Element names (object or act type) used in sentence slot definitions or specifications and in dictionary column headings	•••	Task	Procedure	Function	Command	•••	Screen	Soldier	Doc-ument	Inven-tory item	Screen field	la-bel
Program variables or functions that return values	•••	e.g. top-level menu item	e.g. mid-level menu item	e.g. screen function	Field flag	•••	Curr_ screen	SSN	doc_id	LIN, NSN	Curr_ field	—
Context vector value used as a key entry in dictionaries	•••	Adjust-ment trans-actions	Prepare R/S	Clothing issue	enter, goto	•••	e.g. screen_ id	462-54-3333	R/S	7510-00-823-7874	e.g. field_ id	—

Figure 2: The Context Vector

sentence slots. To "read" this element of the context vector, the Dynamic Help module invokes the curr_screen call. At run time, there is a place in a sentence that an attribute of the context is supposed to fit. To represent the "screen" attribute of the context vector, there is a word "screen" (an object of level 5) associated with a program specific variable of the function named "curr_screen". Curr_screen, when invoked, returns a value; let us call it "screen_id". The Dynamic Help module takes "screen_id" and enters dictionaries using it as a key entry to retrieve material to fill message slots.

To specify a sentence slot requires specifying several things: the name of the dictionary, fields in dictionary that fill the slot, and the name of the key entry to the sentence slot. Assume, for example, that the sentence slot will be filled with a description of the screen. If the operating program screen names were sufficiently described by the value of "screen_id", the designer could simply specify that the value of "screen_id" would fill the slot. Failing that, if the operating program maintained a suitable screen description which could be designated by the designer as an appositive element of the context vector, the designer could include that value, say "screen_descrip," as an appositive element of the context vector. He could specify that the slot to be filled with the appositive element.

Failing both of these opportunities to fill the slot directly from the context vector, the designer would need to store the screen descriptions in a screen dictionary, say (in relational notation)

screen (k, descrip)

(This means the name of the dictionary is "screen", its key field is "k", and its other field is "discrip".)

Now let us use this notation for the sentence slot specification:

screen.descrip (k ← screen_id)

This says that the slot is to be filled with the descrip value from the screen directory in the row whose k column contains the value screen_id. The notation is equivalent to the SQL query

```
SELECT   descrip
FROM     screen
WHERE    k = screen_id
```

The message structure is a set of generic sentence structures; it determines the order of message parts in the sentence structures; and it controls how messages are assembled to be sent to the display.

3.4.2 General Message Structure

The knowledge base contains a representation of message structure. It has eight sentence structures that provide structure for the message. The sentence order in the message structure is completely generic. Seven of the eight sentences have generic structures; one dictionary stores all of the ad-hoc (non-generic) sentences. The message structure in the knowledge base has logic for querying dictionaries and the database. The Dynamic Help Generator will provide the code that represents all the generic structures. Additionally, it will be possible to modify the generic message structure to provide additional sentences if a designer needs them.

Sentence slots make up the variable parts of each message. Each slot specifies a context vector element as an object type, act type, or appositive name. It also specifies the name of the dictionary consulted using the value of that element of the context vector as the key entry, and it specifies the dictionary field whose text actually fills the slot.

To retrieve parts of a sentence, the knowledge base uses a *dictionary lookup.* "An SQL query is similar to a dictionary lookup. An SQL query says 'select A from B where C = D'" (Young 1991a). Here, A is the dictionary field whose text actually fills the slot, B is the name of the dictionary consulted using the key entry, and C is the key entry to reach a specified slot containing an object or act type or appositive name. In the notation introduced above, a slot specification is <B.A (C ← D)>.

This sequence is valid when the knowledge base pulls something straight out of a dictionary. *When a slot is to be filled with a context vector element value* (i.e. screen_label), the <screen_label> can fill the slot directly if the screen_label is an appositive in the context vector for the screen. *When a slot is to be filled with a dictionary value using one key,* each slot is filled with a *dictionary field* value from a *dictionary* where the key entry is a *context vector element name.* Thus, a slot is specified by listing the *dictionary field, dictionary, and context vector element name* in the notation introduced above. Informally,

> if the slot is to be filled with the *descriptor* of the *current screen,* and the dictionary is the *screen dictionary,* then the slot would be specified: <descriptor *of* screen *from* screen dictionary> (Young 1991a).

A slot may be filled with an *indirectly specified* text fragment: <e.g. darg>. The variable *darg* is the name of a procedure that returns the fragment possibly from multiple elements of the context vector using multiple or indirect keys of a dictionary. The name "darg" is a procedure that identifies the lowest-level active act, looks up the direct argument of that act, and uses that argument as keys to look up its description.

3.4.3 Sentence Structures

The Dynamic Help Generator should allow sentence specification according to the following generic sentence structures, and it should also allow modification of slot specifications, modification of sentence structures, and addition of new sentence structures. However, support of modifications is not a high priority requirement; the main requirement for sentences is providing for slot specification (Young 1991a).

The following specifications assume, except for the "Ready to" sentence, that each slot will be filled by the value (either literal text or the name of a variable, function, or slot specification that returns literal text) in column A of dictionary B in the unique row where the value in the key column C is the specified value D. This specification has the form <B.A (C ← D)>, which is equivalent to the SQL query SELECT A FROM B WHERE C EQUALS D.

1. "Ready to" sentence.

Ready to <verb><darg><prep><iarg>

where	**<verb>**	is **<PRIDICT.verb (k ← priact)>**
	<darg>	is **<PRIDICT.darg (k ← priact)>**
	<prep>	is **<PRIDICT.prep (k ← priact)>**
	<iarg>	is **<PRIDICT.iarg (k ← priact)>**

where **priact** is the value from the context vector of the lowest-level active act; **PRIDICT** is the act dictionary for act-type **i**, where **i** is the act type of the lowest-level active act; and **k** is the key field column heading in the dictionary. Often **<darg>** and **<iarg>** are themselves not literal text but specifications.

2. Ad-hoc sentence.

<ADDICT.sentence (k ≈ CV)>

wher ADDICT is the ad-hoc dictionary; **k** is its key field, which is a vector of the same structure as the context vector; and **CV** is the vector of values in the current context vector. In **k**, each element has the following possible values:

Value	Meaning: This element of **CV**...
actually null	can be anything
the text string "null"	must be null
the text string "not null"	cannot be null
<A>	must evaluate to **<A>**

where **<A>** is a literal text string or a specification.

The designer can store an ad-hoc sentence in the **sentence** field of the ad-hoc dictionary and put a context or partial context in the **k** field. Every message whose context vector matches **k** for a row in the dictionary will include the sentence from that row.

Any number of rows from the ad-hoc dictionary (grammatically, more than one sentence) can be part of the ad-hoc sentence (Young 1991a).

3. Meanings sentence.

Meaning of <item>: <MDICT.meaning (k = CV)>

where **MDICT** is the meanings dictionary; **k** is a field of two elements, **i** and **r**, where **i** is the index in the context vector identifying which active act, active object or appositive the meaning is for, and **r** is a code that identifies under what condition the meaning is to be included in the message: when the act or object is the **darg**, when it is either the **darg** or the **iarg**, when it is the **iarg**, or when it is active (not null); and where **item** is an act or object or appositive that can appear in element **i** of the context vector.

The designer can store a meaning in the **meaning** field of the meanings dictionary for a given act or object or appositive stored in the dictionary's **item** field; then, according to the code stored in **r**, the message will include a meanings sentence when that act or object or appositive is of suitable status.

Any number of rows from the meanings dictionary (grammatically, more than one sentence) can be part of the meanings sentence.

4. Choices sentence.

<CHDICT.choices (name ← screen_field)>

where **CHDICT** is the choices dictionary; **choices** is a field containing text that instructs the user how to invoke a list of choices or reminds the user to see a list already on display (on the running screen, not in the help message); and **name** is the key field of the dictionary. The designer stores in **name** the value of a particular instance of the object of type "screen_field". Then, whenever the context vector includes that value as the value of the active object of the screen_field type, the sentence is included in the help message.

5. Format sentence.

Format: <FORDICT.format (k ← darg)>

where **FORDICT** is the format dictionary; **format** is an explanation or illustration of the format of a particular screen field; **k** is the value of the screen field; and **darg** is the value of the direct argument for the "Ready to" sentence above. If there is a value of **k** in the dictionary that equals the current **darg**, then the designer has provided a **format** for the **darg**, and the format sentence is included in the message (Young 1991a).

6. Domain sentence.

DOMAIN: <DOMDICT.domain (k ← darg)>

where **DOMDICT** is the domain dictionary; **domain** is an explanation or illustration of the domain of the data item associated with a screen field; **k** is the value of the screen field; and **darg** is the value of the direct argument for the "Ready to" sentence above. If there is a value of **k** in the dictionary that equals the current **darg**, then the designer has provided a **domain** for the **darg**, and the domain sentence is included in the message.

7. Alternatives sentence.

To <verb><adarg><prep><aiarg>,<procedure>

where **<verb>** is **<PRIDICT.verb (k ← altact)>**

<adarg> is **<PRIDICT.darg (k ← altact)>**

<prep> is **<PRIDICT.prep (k ← altact)>**

<aiarg> is **<PRIDICT.iarg (k ← altact)>**

where **altact** is the value

ALTDICT.altact (h = CV)

where **ALTDICT** is the alternatives dictionary that lists alternative acts for contexts or partial contexts; **h** and **CV** are, for this dictionary, defined as are **k** and **CV** for the ad-hoc dictionary above (sentence 2); and **altact** is the field in the alternatives dictionary that identifies the alternative act. There may be more than one alternative act for a given context or partial context: possibly an alternative act of the undo or abort nature, possibly an alternative act of the commit or closure nature, and possibly an alternative act of the navigation nature. However, the designer provides alternative acts sparingly, generally omitting any that are a primary act in a different context (Young 1991a).

Let **j** be the act-type of the alternative act. Then **PRIDICT** is the act dictionary for act-type **j**, having the key-field column heading **k**, so that **verb**, **adarg**, **prep**, and **aiarg** have values identified above (**adarg** is the direct argument of **altact**, and **aiarg** is the indirect argument).

Finally, **<procedure>** is **PRODICT.procedure (k ← altact)** where **PRODICT** is the procedures dictionary, **k** is its key field column heading, and **procedure** is the text that explains the procedure for invoking the alternative act.

If the designer has provided an alternative act for the context, and has also provided a procedure for the alternative act, the message will include the alternatives sentence for that act.

8. General Help sentence.

To see general documentation, <genproc>

where **<genproc>** is a fixed text phrase such as "press F1" provided by the designer. This sentence appears as the last sentence in every Dynamic Help message.

The structure provided *behind* **genproc** depends on the existing on-line documentation. One possibility is for **genproc** to invoke the existing help system. Another is for **genproc** to invoke construction of a message that contains both fixed and variable parts, and is supported by further dictionaries.

The proper place to store basic instructions such as how to complete an entry, which would appear in a high proportion of all Dynamic Help messages, is behind the General Help sentence. These instructions may be somewhat context-specific. For example, in ACIFS, which has only two types of screen — menu screen and data-entry screen — there is a **GENDIC** general help dictionary whose key identifies the screen type, so that the user receives basic instructions for menu screens or for data-entry screens, but not both, upon invoking general documentation (Young 1991a).

3.4.4 Dictionary Structures

The Dynamic Help module for ACIFS needs the following dictionaries to create sentences: object, format, commands, functions, variables, domain, and protocols (Haines 1991). Here we will specify a more general set of dictionaries. Dictionaries have column headings for the key-entry column and for other columns.

The knowledge about an object is contained in dictionaries — not the context vector, unless the running program contains descriptive variables that can be used as appositives in the context vector. The context vector is a means of identifying information that will fit into a Dynamic Help message. In the event the designer failed to define a particular kind of object in a DBMS-based system, software engineers seeking to incorporate Dynamic Help need to define it for the purpose of help messages.

To support the generic sentence structures, the Dynamic Help Generator should provide structures for a generic set of dictionaries, and it should allow modification of dictionary structures and definition of additional dictionaries.

Assuming the sentence structures described in Section 3.4.3, each of four act types requires an act dictionary; each object type requires an object dictionary; the ad-hoc sentence requires a dictionary; there is a meanings dictionary, a choices dictionary, a format dictionary, a domain dictionary, an alternatives dictionary, a procedures dictionary, and a general dictionary. All but the general dictionary have fixed structures.

There are four act dictionaries, one for each act level, with structures as follows:

Act level	Informal name	Dictionary name and structure
1	Command Dictionary	COMMAND (<u>id</u>, verb, prep, darg, iarg)
2	Function Dictionary	FUNCTION (<u>id</u>, verb, prep, default_value, detailed_content)
3	Procedure Dictionary	PROCEDURE (<u>id</u>, name)
4	Task Dictionary	TASK (<u>id</u>, name)

Note that there is no field specifying the act level, since acts of various levels are segregated into different dictionaries. Note that there is no **meaning** field; long descriptive meanings for those acts (and objects) that require them are kept in the Meanings Dictionary.

In each act dictionary the key entry field is **id**, which is an act identifier value that can be read from the context vector. The **verb** or **name** fields give the language for describing what is being done in an act, and the **darg**, **iarg**, **default_value**, and **detailed_content** fields, along with the **prep** field, give the language for describing what objects are being directly and indirectly manipulated by the act.

There can be distinct dictionaries for each type of object. The lowest-level type of object on a data-entry screen is a screen field. Its important attributes are the name of the datum it represents, the current value of that datum, the function-level (level 2) act that is performed upon entry of the datum, the domain of the datum, and the format of the screen-field (which is often regarded as an attribute of the datum).

When the original design of the application program is clean, most of these attributes are unnecessary to track in dictionaries. In ACIFS, for example, there is only one database datum being entered in a given screen field, and the same datum would never have two distinct entry formats (e.g. SSN would have the same format wherever entered). The screen field (in a data-entry screen) can actually be identified by the name of the datum it represents as if the two things — field and database item — were identical. Since acts are tracked separately, it would be redundant to treat them as attributes of objects.

The object dictionaries must provide short descriptors of acts for use in sentences. Their only generic structure is

OBJECT (<u>id</u>, descriptor)

where **id** is the object identifier value read from the context vector, and **short_descriptor** is the noun phrase that describes the object in sentence slots. If objects of all types are put into a single dictionary, a **level** field is also needed. If a long **meaning** is to be stored for most objects, and the logic for when to include meaning is trivial, the Meanings Dictionary can be omitted and a **meaning** field can be added to the object dictionary or dictionaries.

The Meanings Dictionary has the structure

MEANINGS (<u>id</u>, i, r, meaning)

where the key field **id** is the context-vector value that identifies an object or act, **i** is the position in the context vector where this object's or act's type is formed (i.e., the location in the context vector where this object's or act's **id** value can appear), and **r** is the condition code (recall Meanings sentence, Sentence 3 in Section 3.4.3) that governs when the meaning is to be included in the message.

The Ad-hoc Dictionary has the structure

ADDICT (<u>k</u>, sentence)

where **k** is a vector of the same form as the context vector, with the domain listed for Sentence 2 in Section 3.4.3.

The Choices Dictionary has the structure

CHOICES (<u>id</u>, sentence)

where **id** is a screen field identifier and **sentence** is a message fragment telling the user how to see or invoke a list of choices when this screen field is the **darg**. The format Dictionary and the Domain Dictionary also refer to the **darg** and have similar structure

FORMAT (id, sentence)

DOMAIN (id, sentence)

whenever the **darg** is an object whose **id** is listed in one of the dictionaries, the **sentence** is included in the message.

The Alternatives Dictionary lists alternative acts that may be available in addition to the primary act for a context. This dictionary, the Procedures Dictionary, and the various act dictionaries feed the Alternatives sentence. The Alternatives Dictionary has the structure

ALTDICT (h, altact)

where **h** is a vector of the same structure as the context vector and **altact** is the act identifier for the alternative act. The key entry field is **h**, which is a context or partial context interpreted as described for the similar vector **k** for the Ad-hoc sentence in Section 3.4.3.

The Procedures Dictionary gives the user procedures to perform selected acts and has the structure

PRODICT (k, procedure)

where the key field **k** contains an act identifier and **procedure** contains instructions in the form of an imperative verb phrase. Recall that the Alternatives sentence has a structure that returns sentences such as "To delete soldier 462544872 from order 7940, press F3," where "To" is a fixed string, "delete soldier 462544872 from order 7940" is a catenation of the **verb**, the **adarg**, the **prep** and the **aiarg** for the alternative act (that is, the alternative act's **verb**, **darg**, **prep** and **iarg** from the appropriate act dictionary for the alternative act), the comma is fixed text, and **procedure** is from the Procedures Dictionary.

If a procedure sentence (for the primary act) were added to the Dynamic Help message, or if a procedure clause were added to the "Ready to" sentence, the Procedures Dictionary is already in appropriate form to support the added requirements.

Recall that the General Help sentence is fixed text. We assume, for purposes of providing a generic General Help Dictionary, that the available General Help message is a collection of fixed-text sentences, each one associated with a partial context. The structure of the General Help Dictionary is

GENDIC (**k**, sentence)

where **k** is the key entry field and is defined like **k** for the Ad-hoc sentence (Sentence 2) in Section 3.4.3. Alternatively, the entry field can be defined like **k** in the Meanings sentence (Sentence 3) in Section 3.4.3. In either case, **k** determines, according to the context, whether each General Help sentence will be included in the General Help message. In application programs such as ACIFS, having only menu screens and data-entry screens, it is natural to have some General Help sentences appear only for menu screens, some only for data-entry screens, and some for both; thus, a single element from the context vector, rather than a mask of the whole vector, would be appropriate for **k**, since it is necessary only to know the screen type.

3.5 Automated Dynamic Help Illustration

An interactive scheduling program has a Gantt-chart window used in the scheduling process, showing activities in the schedule as bars, each consisting of a stack of horizontal color stripes representing resource consumption rates. When the duration-change command is active, the user can touch (by mouse or other pointer device) either the left (start-time) or right (end-time) end of an activity, then touch *another point to* "drag" that end left or right to crash or lengthen the activity's duration (Young 1991a).

Suppose that the user has reached the scheduling process, has touched the DUR command-menu item (which expresses an intention to change durations of activities), and has touched one end of the bar that depicts the activity named "ACT 37" in the Gantt-chart window. If Dynamic Help is invoked, the context vector is as follows:

CV. CONTEXT VECTOR

1 process	2 command	3 window	4 activity	5 resource	6 data field
SCH	END_DURCH	GANTT	ACT37	(null)	(null)

|◄— acts —►|◄— objects —►|

The Dynamic Help system reads *the context vector* and determines from it that the primary act is that of completing a duration change ("END_DURCH"). This is done by identifying the rightmost (lowest-level) act that is not null in the context vector.

Determination of the *primary* act:

priact $\leftarrow$ cv(i) such that i is the max CV index among those for non-null acts

The Dynamic Help system next begins constructing the message, and the first sentence is the "Ready to" sentence. Because the primary act is a command-level act (i=2), the sentence is filled from the command dictionary according to the structure provided for this case.

Structure of the "ready to" sentence:

Ready to <verb><darg><prep><iarg>

where: <verb> is DICT1.verb (cmd $\leftarrow$ priact)
<darg> is DICT1.darg (cmd $\leftarrow$ priact)
<prep> is DICT1.prep (cmd $\leftarrow$ priact)
<iarg> is DICT1.iarg (cmd $\leftarrow$ priact)

(If the lowest-level active act had been a process-level act, the structure could be different and could be filled from the process dictionary.) The spaces between sentence fragments and the period at the end of the sentence are part of the sentence specification, but they are not shown.

The command dictionary and macros table have entries as illustrated:

DICT1. COMMAND DICTIONARY

cmd	verb	darg	prep	iarg
BEG_DURCH	"begin to crash or lengthen"	"an activity"	"by"	MAC(17)
END_DURCH	"Complete crashing or lengthening"	CV(4)	"by"	MAC(18)

MAC. MACROS TABLE

mac_id	macro
17	"touching one end"
18	"touching to left or right of its active end"

With the illustrated data, the "Ready to" sentence can be constructed (Young 1991a).

Construction of the "Ready to" sentence:

1. priact ← END_DURCH
2. <verb> ← "complete crashing or lengthening"
3. <darg> ← CV(4) ← "ACT37"
4. <prep> ← "by"
5. <larg> ← MAC(18) ← "touching to left or right of active end"

Thus the constructed "ready to" sentence is

Ready to complete crashing or lengthening ACT37 by touching to left or right of its active end.

Simiiarly, given the ad-hoc dictionary, the Ad-hoc sentence can be constructed.

DICT2. AD-HOC DICTIONARY

context						sentence
1	2	3	4	5	6	
	END_*					"Optionally, touch a point on the timescale"

Construction of the Ad-hoc sentence

1. CV obeys CV(2) ← END_*

2. Structure of the Ad-hoc sentence is

   <ad-hoc sentence>

   where <ad-hoc sentence> is DICT2.
   sentence (context ϵ CV)

Thus the message fragment that is provided for all contexts in which the active command is one whose CV value begins with "END_" is retrieved from DICT2:

Optionally, touch a point on the timescale.

In this context, the Meanings sentence, Choices sentence, Format sentence and Domain sentence would be null. The Meanings sentence would be null because the designer did not provide either a meaning for the primary act in the act meanings dictionary nor a meaning for the darg (direct argument of the primary act) in the object meanings dictionary. The Choices sentence would be null because no choices availability was listed for the darg in the choices dictionary. The Format and Domain sentences would be null because the darg is not a data field (Young 1991a).

Assume it has been decided that whenever a multi-command procedure is ready for closure (as indicated by there existing both an active activity and an active command whose value begins "END_"), the user may be confused as to how to abort the procedure. There are several possibilities. The user can touch an end of a different activity to begin changing its duration instead of that of ACT 37, but this would be an expert shortcut irrelevant to a Dynamic Help message. The user can touch the NEU (neutral-state) command menu item, which can always be touched to deactivate both the active command and the active activity, but this is general and would be documented under General Help. Finally, the user can touch a different command menu item and be ready to do something different (such as specify a start time) or the active activity — another expert shortcut. Confusion can be resolved by providing an Alternatives sentence for all contexts like this (Young 1991a). The alternatives dictionary has the same key (context) as the ad-hoc dictionary:

DICT3. ALTERNATIVES DICTIONARY

context						sentence
1	2	3	4	5	6	
	END_*		not null			"To abort, touch another activity or another command or NEU"

Thus, including the Alternatives sentence and the fixed-text General help sentence, the complete Dynamic Help message for the illustrated context would be:

> Ready to complete crashing or lengthening ACT37 by touching to left or right of its active end. Optionally, touch

a point on the timescale. To abort, touch another activity
or another command or NEU.
To see general documentation, press F1.

4. The Dynamic Help Generator: a set of Designer's Tools

4.1 Requirements for the Dynamic Help Generator

There are two basic requirements for the Dynamic Help Generator work undertaken at AIRMICS in the Summer of 1991. The first is to implement a prototype Generator for the specific ACIFS application program in the specific programming environment ACE. This prototype Generator is the collection of ACE routines being prepared by Mr. Smith and due for delivery September 30, 1991, with internal documentation.

The second basic requirement is to specify the requirements for a generic Dynamic Help Generator. This section provides those requirements, which should govern further Generator development. These requirements are compatible with the results of Captain Haines' automatability research and with the "Proposal for Dynamic Help Developer's Tool kit" (Smith, 1991b).

There are 5 requirements for the generic Dynamic Help Generator ("Developer's Tool kit"):

1. The generic Generator shall consist of four tools:
   - Program context definition tool
   - Help sentence specification generator
   - Help dictionary generator
   - Help message production routine

2. Each tool shall consist of routines and documentation fully integrated into the ACE programming environment and

immediately usable for developers of Dynamic Help for existing ISMs whose interfaces are written under ACE and whose general nature is that of DBMS-based application programs.

3. Each tool shall be written so as to be maximally portable from ACE to another environment. In particular, routines that can be written in the **C** language shall not be written in the ACE script language.

4. Those parts of tools that produce code to query an application's database at run time shall be written in a standard dialect of the SQL query language, in such a way that the SQL queries can easily be translated to the **C** language.

5. Each tool will provide flexibility for the developer to accept default structures or modify them. In particular, there must be flexibility for a Dynamic Help developer
    - to add context monitoring logic to the application program
    - to alter the structure of the context vector
    - to change the specifications for filling any sentence slot
    - to add sentences to the message
    - to alter the fixed order of sentences
    - to specify additional dictionaries where needed

The *program context definition tool* will be an interactive subroutine in the ACE environment. It will interact with the developer to allow definition of global variables at points where the context changes.

The *Help sentence specification generator* will be an interactive subroutine in the ACE environment. It will interact with the developer to present each of the eight standard sentence structures, to collect slot specifications for each sentence slot (including acceptance of generic slot specifications), to allow revision of fixed text in sentences, to allow new slots to be added to a sentence, to add new sentence, and to revise the order of sentences.

The *Help dictionary generator* will be an interactive subroutine in the ACE environment. It will interact with the developer to present each of the standard dictionary structures, to collect data from the developer to fill each row of each dictionary, to alter the definition of the key entry field of each dictionary, to add fields, to delete fields, and to define new dictionaries. SQL statements may be used for defining and filling dictionaries (that is, the developer can be allowed to use standard SQL commands such as DEFINE, INSERT, DELETE, UPDATE, COMMIT, etc. in interacting with the tool).

The *Help message production routine* will be totally automated and will not require developer attention. However, the developer will be provided with the interactive capability of producing messages and sending them to a screen or printer destination while in the development environment; this will include the capability of producing an arbitrary developer-defined set of context-vector values to simulate a run-time context.

The technical requirements for generic structures of messages and dictionaries are exactly those given above in Sections 3.4.1, 3.4.2, 3.4.3, and 3.4.4, except as may be modified by future specifications.

4.2 Generator Implementation for ACIFS

The Dynamic Help Generator grew out of a need for software so simple and so easy to field that U.S. Army DBMS-based system users could use the software as if it were off-the-shelf software from a computer store. The sending of 4-5 person teams to every Army post to spend two weeks programming the installation's software, training the users, and troubleshooting is too slow and expensive (Gantt 1991).

The two possibilities that would accomplish the aim of speeding up user proficiency included tutorials and help facilities. Because Dynamic Help would have an immediate application, it was selected for development. Dynamic Help assumes that a user knows a limited amount about turning on a system and can perform some simple user functions, and it means that the users can perform the rudiments of CIF tasks on paper. The Dynamic Help facility was intended to get the users through CIF tasks.

Dr. Donovan Young oversaw the development of the entire Dynamic Help project and provided the necessary guidance for team members. Both Dr. Young and Mr. Christopher Smith designed the initial specifications for a Dynamic Help module (Young 1990a). Captain Renée Wolven and Captain Stanley Haines assisted Mr. Smith by writing the prototype Dynamic Help messages. Captain Wolven tested the usefulness of the Dynamic Help messages using the prototype Dynamic Help module written by Mr. Smith (Wolven 1991).

Captain Walter Barge designed the concept and philosophy for a Dynamic Help Generator (Barge 1991). Captain Haines developed a procedure to generate Dynamic Help messages using a context vector, dictionaries, knowledge bases, and the eight sentence structures (Haines 1991).

Dr. Jerry McCoyd performed the initial Dynamic Help Generator conversion from Informix-4GL to C language. Mr. Smith is creating a working prototype of the tools and software for the Dynamic Help Generator (Smith 1991a).

At present, the U.S. Army software Development Center in Atlanta is rewriting the ACIFS software; it will then be compatible with the ACE C language that the Dynamic Help Generator uses to construct Dynamic Help modules. Unfortunately, the ACIFS #L08-04-01 will reach the field without a Dynamic Help module.

It will take time to add Dynamic Help to existing DBMS-based Army software; it would be smarter to design Dynamic Help into new systems. Fortunately, The Dynamic Help Generator will give the software designers of new software the increased capability to systematically add Dynamic Help to systems when specifying the systems. Additionally, Dynamic Help has the capability to change (Help messages are created for specific contexts) as installation software is upgraded to avoid obsolescence (Gantt 1991).

Once the generator and supporting documentation are complete, the ACIFS #L08-04-01 will be ready to receive the new Dynamic Help modules. AIRMICS will give the software and documentation to the U.S. Army Software Development Center in Washington D.C. and in Atlanta for their review. The two centers are anxious to get the software and documentation and to test it. It will then be sent to the field immediately with the next software upgrade team. The development centers will use the Dynamic Help Generator to incorporate Dynamic Help into all future DBMS-based software.

4.3 Recommendations

A proposal for further Dynamic Help automation is under consideration (Smith 1991b). It provides for preparation of a generic Dynamic Help Generator and application to another ISM. It is recommended that the requirements listed in Section 4.1 above be adopted to guide this work.

It is also recommended that part of this work be the early preparation of a formal specification document for the generic Generator. This would translate the requirements (Section 4.1) into a more ACE-specific set of tasks that can be managed, scheduled and monitored for quality. It would also isolate the early design decisions regarding how to implement the Generator under ACE. Any delay of these decisions could delay the production of a finished Generator.

It is essential to publish the finished Generator as widely as possible, within the Army and DOD, and in the technical literature. The recent AIRMICS accomplishments of producing a help system that makes complex interactive software usable without training, and of showing that the production of the system can be efficiently automated, have tremendous implications for all developers of complex interactive software. AIRMICS has found a cost-effective way of improving software usability without reprogramming, and the results deserve wide dissemination.

Appendix 1
Description of ACRONYMS

ACRONYMS

ACE	Application Connectivity Engineering
ACIFS	Automated Central Issue Facility System
AIRMICS	Army Institution for Research in Management Information, Communications, and Computer Science
DBMS	Data Base Management System
ETIP	Extended Terminal Interface Program
EUS	Embedded User Support
ILIDB	Installation Level Integrated Data Base
ISM	Installation Support Module
RAM	Random Access Memory
SQL	Standard Query Language
UIC	Unit Identification Code

Appendix 2
Sample Specification of a Menu-screen Dynamic Help Message

Date 6 Jan 91 HELP Message Specifications (MENU) Page ___

LOCATION INFORMATION:

<A> Current Screen: Orange Accounting

<X> Current Selection: 1. Accountable to Non-Accountable

B. Previous Screen: Administrative Adjustments Menu

C. Ordinarily Next Screen: <NU>

HEADING: <A>

GLOBAL CONTEXT: Where the user is & what the user can do.

"Ready to <CH> an item's accounting method.

NAVIGATION:

BACKWARD: "To return to B. <HLT> X <E> <CR>

FORWARD: "To go to C. <NU>

HEADING: <X>

LOCAL CONTEXT:

"Ready to <CH> or <ITM> from accountable to non-accountable <100>.

PROCEDURE SENTENCES:

COMPLETION: "To go to A. <CR>.

"To make another selection. <LRAH> <OPT> <E> <CR>

UNDO: "To undo or reverse a selection after it has been pressed <DEL>.

KEY EFFECT:

KEY	EFFECT
ESC	<NU>
DEL	

Appendix 3
Sample Specification of the Specification of a Data-entry-screen Dynamic Help Message for Both Global and Local Messages

Date 25 Jan 91 HELP Message Specifications (INPUT - GLOBAL) Page 35

LOCATION INFORMATION:

A. Current Screen: Turn In to SSA

B. Previous Screen: Turn Ins to SSA/DRMO

C. Ord Next Screen <NU>

HEADING: <A>

GLOBAL CONTEXT: (Where the user is and what the user can do

"Ready to initiate or post a <TI> to SSA. To initiate a <TI> to SSA, <CR> to bypass ~~the~~ <DNR> field. <ENTER> NSN <ξ> <QTY> <TB> <TID> <ξ> <CR>. New <DNR> will be displayed. To post a <TI> to SSA, <ENTER> <DNR> <QTY> <TID> <ξ> condition code. <ξ> <CR.

BACKWARD NAVIGATION: "To return to B. <QUT>, <DEL>. "

FORWARD NAVIGATION: "To go to C. <NU> .

COMPLETION: "To complete an entry. <CR> .

QUIRKS: "NOTE: <BLAK> .

PRE-COMPLETION CORRECTION: "To make a correction while the cursor is still in the entry field. <NU> .

UNDO: "To make a correction. <NU> .

KEY EFFECT:

KEY	EFFECT
ESC	<NU>
DEL	'
F3	'
F4	'

75

COPY AVAILABLE TO DTIC DOES NOT PERMIT FULLY LEGIBLE REPRODUCTION

HELP Message Specifications [illegible] - LOCAL

LOCATION INFORMATION:

[illegible] Current Screen: Automated Turn In

[illegible] Current Data Object: TIQTY

[illegible]: SIZE

[illegible]: <NU>

HEADING: <X>

LOCAL CONTEXT:

[illegible] ENTER> <QTY> of <f06>, <f10> <TID> or <IND>, If <TI> <QTY> <MATCH> <X> <CR>.

[illegible]:

[illegible]: <NU>

[illegible]: <NU>

[illegible]: <NU>

[illegible]: <NU>

[illegible]: <NU>

[illegible]: <NU>

[illegible]

<NU>

[illegible]

[illegible]

<NU>

[illegible]

[illegible] <CR>

[illegible] <Nu>

[illegible]

[illegible] <NU>

[illegible]
<NU>

[illegible]
<NU>

[illegible]

[illegible]	[illegible]
[illegible]	<NU>
[illegible]	/
[illegible]	/
[illegible]	/
[illegible]	/
[illegible]	/
[illegible]	/

Appendix 4
Sample Entries from the Dictionary of Text Macros

Dictionary of HELP Message Codes

< > - Phrase [ ] - Example, List or Format { } - Query

CODE	DEFINITION/MEANING
<&>	and
A	*Current Screen Definition
<ACCEPT>	Acceptable entries are
<AUTH>	authorized
<AUTO>	automated
<B>	*Previous Screen Definition
<BACK>	use BACKSPACE and overstrike incorrect data
<BCKSP>	press BACKSPACE key
<BLAK>	BACKSPACE key does not erase data
<C>	*Ordinarily Next Screen
<CMPD>	completed
<COMP>	complete all
<CR>	press Return
<DEL>	press DEL (Delete) key
<DES>	desiring
<DESD>	is desired
<DN>	press DOWN Arrow key
<DX>	direct exchange
<ENTER>	enter the
<ENTRS>	entries
<ERR>	an error occurs
<ESC>	press ESC key
<EXAM>	**EXAMPLE:
<EXCH>	exchanged
<FIN>	to complete
<G>	to go to
<HLT>	highlight
<IND>	individual
<INDS>	individual's
<INIT>	initial
<INST>	another installation
<ITM>	item
<ITMS>	items
<ISS>	issue
<ISSD>	issued
<LRAH>	use LEFT & RIGHT Arrow keys to highlight
<LT>	press LEFT Arrow key
<MANU>	manual
<MATCH>	matches displayed
<MF>	"M" for male and "F" for female
<NEED>	as needed
<NOTE>	**NOTE:
<NOUSE>	Do not use this option if
<NU>	*NULL entry
<OH>	letter O and 0 are different
<OPT>	desired option number
<OVER>	overstrike
<PERF>	perform entry again
<PR>	press
<PROC>	process

Bibliography

Barge, Walter. "Universal Software Documentation via Dynamic Help." Research report, Georgia Institute of Technology, 1991.

Dorazio, Patricia. "Help Facilities: A Survey of the Literature." Technical Communication, 118-121. Second Quarter, 1988.

Gantt, James. [Dynamic Help Generator implementation within the U.S. Army]. Interview with the author, Georgia Institute of Technology, 19 August 1991.

Haines, Stanley. "Automatable User Support for Existing U.S. Army Installation-Level Software." Masters thesis, Georgia Institute of Technology, expected completion September, 1991.

Haines, Stanley. "Specification of Dynamic Help Menu Messages for EUS Project and CIF Conversion." School of Industrial and Systems Engineering, Georgia Institute of Technology, 14 May 1991a.

Haines, Stanley. "Specification of Dynamic Help Global Messages for EUS Project and CIF Conversion." School of Industrial and Systems Engineering, Georgia Institute of Technology, 16 May 1991b.

Haines, Stanley. "Specification of Dynamic Help Local Messages for EUS Project and CIF Conversion." School of Industrial and Systems Engineering, Georgia Institute of Technology, 30 May 1991c.

Horton, William. "Help Facilities." In Designing & Writing Online Documentation, 253-267. New York, NY: John Wiley & Sons, 1990.

Horton, William. "Making information Accessible." In Designing & Writing Online Documentation, 253-267. New York, NY: John Wiley & Sons, 1990a.

Smith, Christopher, Mike McCracken, Donovan Young. "Embedded User Support Project Plan." School of Industrial and Systems Engineering, Georgia Institute of Technology, 1990 [photocopy].

Smith, Christopher. [A technical explanation of ACIFS and the Dynamic Help Generator]. Interview with the author, Georgia Institute of Technology, 24 July 1991.

Smith, Christopher. [A technical explanation of the architecture of the Dynamic Help Generator]. Interview with the author, Georgia Institute of Technology, 20 August 1991a.

Smith, Christopher and Young, Donovan. "Proposal for Dynamic Help Developer's Toolkit." School of Industrial and Systems Engineering, Georgia Institute of Technology, 1991b.

Wolven, Renée. Effectiveness Testing of Embedded User Support for U.S. Army Installation-Level Software. Masters thesis, Georgia Institute of Technology, 1991.

Wolven, Renée and Young, Donovan. "Specification of Dynamic Help Messages for EUS Project and CIF Conversion." School of Industrial and Systems Engineering, Georgia Institute of Technology, 17 January 1991.

Wolven, Renée and Young, Donovan. "Specification of Dynamic Help Messages for EUS Project and CIF Conversion." School of Industrial and Systems Engineering, Georgia Institute of Technology, 29 January 1991.

Young, Donovan. "Embedded User Support for U.S. Army Installation Software." White Paper, Georgia Institute of Technology, 1990.

Young, Donovan. "Specification of User Requirements and Dynamic Help System Standards for EUS Project and CIF Conversion." School of Industrial and Systems Engineering, Georgia Institute of Technology, 1990a [photocopy].

Young, Donovan. "Dynamic Tutorials for Installation Support Software." School of Industrial and Systems Engineering, Georgia Institute of Technology, 1991 [photocopy].

Young, Donovan. [Dynamic Help and the Dynamic Help Generator]. Unpublished lectures to the Author. Georgia Institute of Technology, 1991a.

Young, Donovan. "General Specifications For Dynamic Help Generators." School of Industrial and Systems Engineering, Georgia Institute of Technology, 1991b.