N87-23162

2GCHAS—A HIGH PRODUCTIVITY SOFTWARE DEVELOPMENT ENVIRONMENT

Larry Babb
Computer Sciences Corporation
Systems Sciences Division

2GCHAS - A High Productivity Software Development Environment

Larry Babb
Computer Sciences Corporation
System Sciences Division

To the user, the most visible feature of TAE is its very powerful user interface. To the programmer, TAE's user interface, proc concept, standardized interface definitions, and hierarchy search provide a set of tools for rapidly prototyping or developing production software. The 2GCHAS (pronounced TWO GEE CHARLIE, Second Generation Comprehensive Helicopter Analysis System) project has extended and enhanced these mechanisms, creating a powerful and high productivity programming environment where 2GCHAS' development environment is 2GCHAS itself and where a sustained rate for certified, documented, and tested software above 30 delivered source instructions per programmer day has been achieved. The 2GCHAS environment is not limited to helicopter analysis, but is applicable to other disciplines where software development is important.

BACKGROUND

Predicting the characteristics and performance of helicopters is not a mature discipline; the theory is still developing and the computational tools based on the immature discipline are relatively undeveloped. The 2GCHAS project was established by the U.S. Army to develop a system which can predict the flight characteristics of a helicopter from a physical description of the vehicle. The objectives pertinent to the discussion in this article include;

1. Providing a standard set of helicopter analysis tools based on current theory,

2. Providing an environment which can assist developing new computational tools,

3. Providing the computational framework into which new or modified analysis tools can be inserted.

Three major considerations have driven 2GCHAS' design. The first is the requirement that 2GCHAS be operating system independent. With a rich set of functional requirements and initial implementations on VAX/VMS and IBM's MVS operating systems, 'pragmatic considerations have replaced operating system independence with operating system transportability. To achieve transportability, the user and programmatic interfaces have been defined to be constant across operating systems. Host dependencies have been restricted to the smallest number of units -- operating system procedures, 2GCHAS procedures, and software -- possible. The result is that most of 2GCHAS is and will be operating system

independent.

The second major consideration is that both end users and developers of new 2GCHAS analyses are helicopter engineers and scientists, not programmers or computer scientists. Their language of choice is FORTRAN. Their interest is in developing new analysis tools, modifying existing analysis tools, and using those tools to predict helicopter performance in a user friendly environment. 2GCHAS chose TAE to meet the user friendliness requirements.

The third major consideration is that every 2GCHAS Module (analogous to a TAE Process) has to be invokable both directly by the user and by other Modules and must be replacable at run time. Meeting the run time replacability requirement turned out to be the major contributor to the productivity of the 2GCHAS environment. As background to further discussion, the top level organization of 2GCHAS and the rationale for the replacabilty requirement and its implementaion (run time linking) will be described in some detail.

TOP LEVEL 2GCHAS ORGANIZATION

2GCHAS is divided into the Executive complex and the Technology complex. The Technology Complex will consist of a large number of FORTRAN 77 Modules which perform the helicopter analysis. The Executive Complex provides services (e.g., user interface, data management, Module substitution) required by the Technology Modules and isolates them from the host operating system. When 2GCHAS is transported to a new operating system, only the Executive Complex is required to change.

MODULE REPLACABILITY AND RUN TIME LINKING

The replacability requirement for 2GCHAS Modules derives from the following considerations:

1. There will be a large number of Technology Modules in 2GCHAS. By design, as new helicopter analysis techniques are developed, Modules will be added to or modified within 2GCHAS.

2. A typical analysis will use relatively few of the available Technology Modules. Available Modules might supply alternative approximation methods, apply different flight simulation techniques, or compute different outputs.

3. It is difficult to predict in advance which Modules are required for an analysis. Modules are called as required by a combination of processing options, output results desired, and the description of the helicopter and its flight conditions.

4. Any Module linked into an executable image uses resources even though it is never executed. For example, all Modules in an executable image are assigned virtual memory when the image is run. Modules which are not executed still consume virtual memory quota.

5. Multiple developers from different organizations will be developing new Modules throughout 2GCHAS' life. To provide a framework for developing new analysis techniques, the system structure must be quite open. Maintaining and distributing a standard set of object Modules is difficult for an open system.

The solution to the replacability and transportability requirements is to have run time linking, to have a Module call look exactly like a subroutine call, and to define a Module to be a FORTRAN subroutine. There are only four minor differences between a Module and any other FORTRAN subroutine:

1. The last argument is the completion status of the Module;

2. The ENTRY statement is not permitted;

3. COMMON is not permitted for inter Module communication;

4. A special set of comments, the Preamble, is required to provide information about the Module and its arguments.

Suppose, for example, Module A calls Module B. Each Module is compiled, producing an object file. The object files are then linked by the Linker into an executable image which can be executed using the RUN command. (There must be a main program linked with the subroutines, but it has been omitted here to show the Module linkage process more clearly.) Figure 1 shows how the Modules are linked using the standard linking procedure.

Figure 2 shows the run time linking technique used by 2GCHAS. From the Module source, DEFMOD (DEFine MODule) produces a Module caller source file and then compiles both source files to produce object files. In this example, Module B is defined first. DEFMOD produces the source file for B Caller from the Module B source file, then compiles both source files to produce object files for B Caller and the body of Module B. The object file for B Caller is put in an object library containing the Module callers for all defined Modules. The object file for the body of B is linked to produce a shareable image of B. Similarly, when Module A is defined, the object file for A Caller is put in the Module caller object library and the object file for the body of A is linked with the library of Module callers to produce a shareable image for Module A. Since Module A calls Module B, B Caller is linked into the shareable image for Module A.

When Module A executes the "CALL B" statement, it actually executes a subroutine call on B Caller which has been linked into the image in place of Module B. The B Caller subroutine, created by DEFMOD, calls an Executive service, Module Execution Control (XMEC) to activate the shareable image for Module B. When XMEC is called, it determines if the image has already been activated. If the image has not been activated, XMEC calls an operating system service to activate it. After XMEC activates the shareable image of Module B (or finds it already activated), XMEC executes a subroutine call on Module B and passes the argument list from the subroutine call executed by Module A. When Module B completes its execution, it returns to XMEC, which returns to B Caller, which returns to Module A. On VAX/VMS systems, XMEC uses the Library Service LIB$FIND_IMAGE_SYMBOL to activate shareable images.

From Module A's point of view, a standard subroutine call has been executed. From 2GCHAS' point of view, the Module is assigned virtual memory and other resources only if it is executed.

Run time linking, like all solutions to difficult problems, contains tradeoffs. The advantages of run time linking are;

1. Virtual memory and other resources are allocated to Modules only if the Modules are executed.

2. Enhancements to Modules can be tested by Module substitution at run time.

3. The option of linking a Module directly using the standard Linker remains available with no change in the Module source, since the source is standard FORTRAN 77.

The disadvantages of run time linking are;

1. Program execution time is increased. The first time a Module is called on a VAX 11/785, an elapsed time of about 0.2 seconds is required to activate the Module. Subsequent calls on the Module take considerably less time, although more than subroutine call. Because helicopter analysis runs will be computationally intensive, the overhead time required to activate the analysis Modules will be a small portion of the total job.

2. FORTRAN COMMON blocks cannot be used to communicate between Modules linked at run time. However, subroutines which are contained within a single Module may communicate with each other through COMMON blocks as usual. In 2GCHAS, the Executive contains data management services which provide a data structure intended to replace FORTRAN COMMON blocks in communicating between Modules linked at run time.

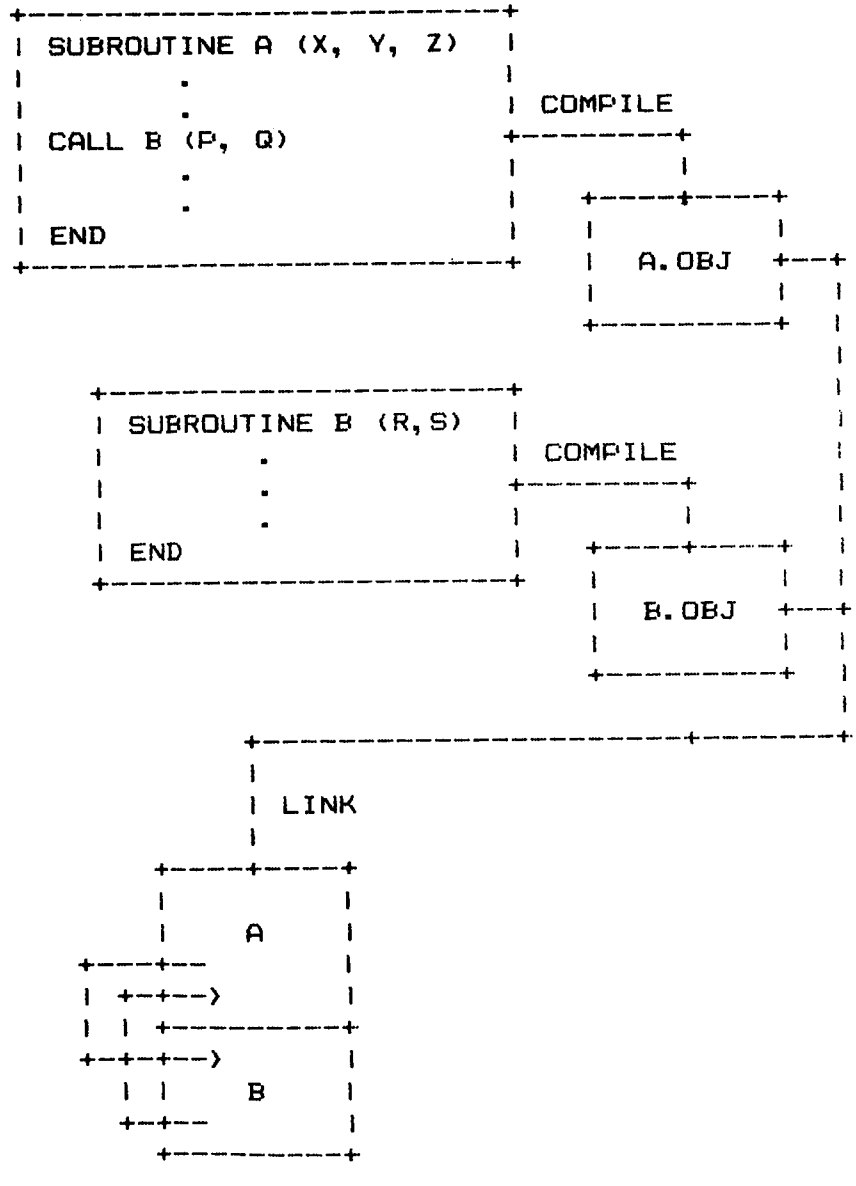

Figure 1. Standard FORTRAN Subroutine Linkage

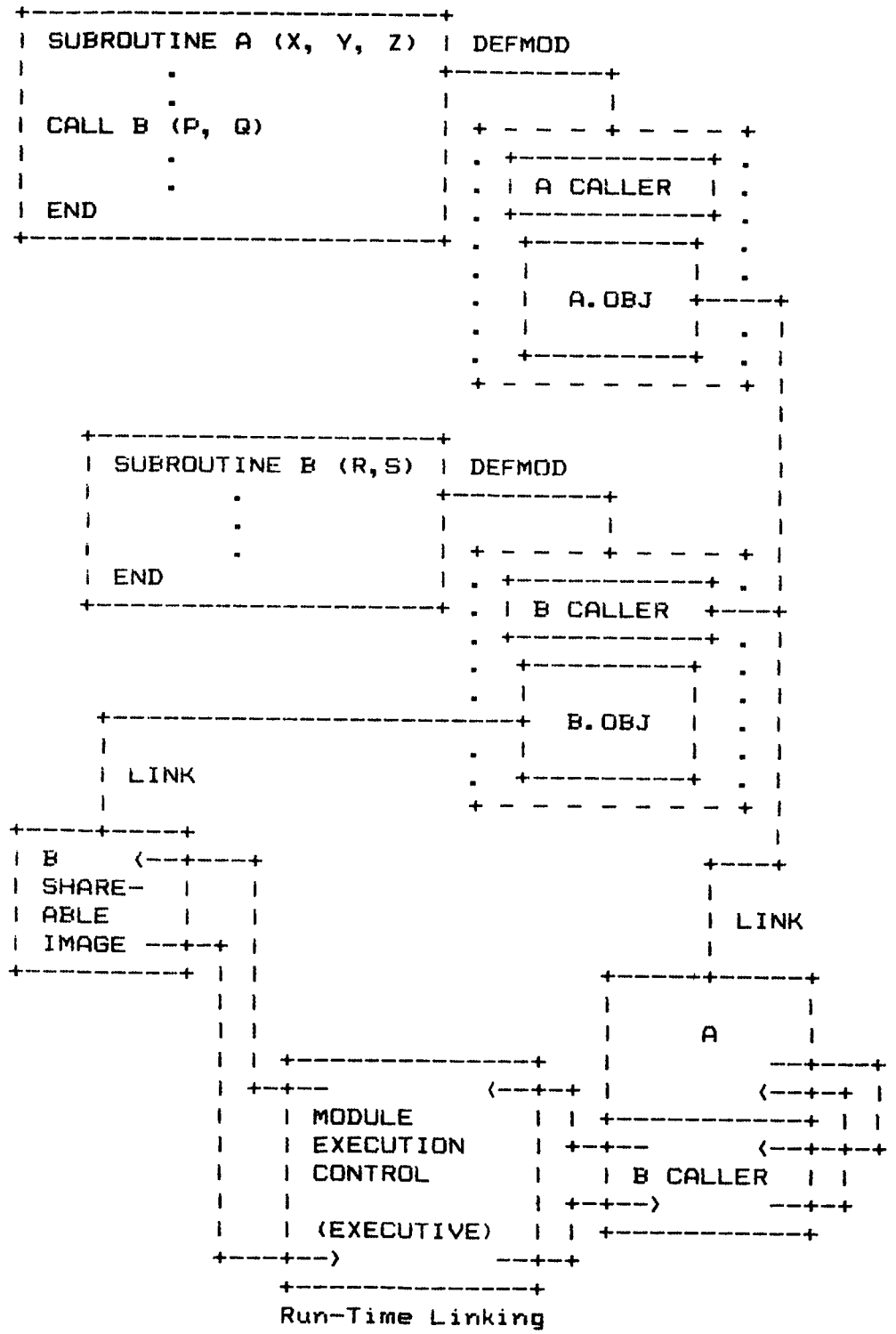

Figure 2. 2GCHAS Module Run Time Linking

DEFMOD TAKES THE DRUDGERY OUT OF MODULE CREATION

DEFMOD's role in creating Modules for run time linking has already been described. DEFMOD provides another feature for developing and unit testing FORTRAN subroutines. From the Module preamble, DEFMOD creates the process PDF and all of the VBLOCK references necessary for the user to invoke the Module directly.

To appreciate the amount of effort that DEFMOD saves, consider a TAE process. It consists of at least two separate files -- the process itself and the process PDF. In the process PDF, information about each parameter can be in as many as three disjoint places -- the PARM statement, the LEVEL1 description, and the LEVEL2 description. The information in each place within the process PDF must be consistent. The process PDF, in turn, must be consistent with the VBLOCK references in the process itself. Consistency of physicaly separated information is hard to achieve and the requirement for it can lead to increased development and maintenance costs. In the case where the help text is separate from the process PDF, there are three files which must be consistent.

In contrast, a 2GCHAS Module source file contains all of the information needed by DEFMOD (DEFine MODule) to create the components necessary to execute the Module. The Module Preamble, that special set of comments at the beginning of a Module, contains all of the information about the Module and its parameters. The preamble format rules are less restrictive thar the rules for a process PDF. The parameter information which becomes the PARM statement, the TAE LEVEL1 text, and the LEVEL2 text is contiguous, not separated. The input which becomes the TAE LEVEL1 parameter information is broken automatically (and reasonably) to fit into the 32 column width restriction.

To be invoked directly by the user, the Module must have, in addition to a process PDF, VBLOCK references for the parameters. But a Module is a FORTRAN subroutine, it contains no VBLOCK references. And the Module's parameters can be of any FORTRAN data type (i.e., INTEGER, REAL, CHARACTER, DOUBLE PRECISION, COMPLEX, or LOGICAL) not just REAL, INTEGER, and STRING.

From the preamble in the Module source file, DEFMOD constructs the necessary files. The process PDF contains the parameter definitions and HELP information. DEFMOD constructs a FORTRAN program (called the Module Main) which contains the VBLOCK references, the necessary conversion code to go between TAE data types and the larger set of FORTRAN data types, and a FORTRAN CALL to invoke the Module.

LOGICAL parameters are an especially good example of the processing provided by DEFMOD. The Module preamble states that the parameter is of type LOGICAL. In the process PDF, the parameter's type is (STRING, 1) and its valid values are the letters "T" and "F". In the Module main, the letter "T" from the input parameter is converted to a FORTRAN logical .TRUE.; the letter "F" to FALSE. The Module main CALLs the Module with a FORTRAN LOGICAL parameter. After successful return from the Module, LOGICAL .TRUE. is converted to "T" and .FALSE. to "F" if the parameter is OUT or INOUT (TAE parameter type NAME).

Creating a Module with DEFMOD saves much of the effort normally required to implement a TAE process and yields three major benefits. The first is the ease with which changes are made to the calling sequences of Modules. It is easy to add, delete, or change a parameter or the parameter's attributes by editing the preamble and FORTRAN statements. It is similarly easy to change the help information for the Module or for individual parameters.

The second benefit is the ease with which small pieces of software (e.g., finding the roots of a polynomial using Newton's method) can be quickly prototyped as Modules and tested from command or tutor mode. Although there is some effort required to put a preamble in a Module, that effort is small compared to the effort of either creating a test driver for the Module or for implementing VBLOCK calls.

The third benefit is that packaging decisions can occur very late in the development cycle. Because the preamble information consists of FORTRAN comments, code developed as a Module can remain a Module or can be made into a subroutine linked in to a larger Module. Such a packaging decision requires no change to the source. Thus, the decision to make a unit into a Module or not is not critical. Not only can the decision can be deferred, but it is easily changed if made incorrectly.

LOOSE COUPLING OF MODULES CONTRIBUTES TO PRODUCTIVITY

Edward Yourdon and Larry Constantine in Structured Design define coupling as the degree to which one software unit depends on knowledge of another software unit in performing its task. They argue that loose coupling leads to a design that is easier to develop and easier to maintain and that close coupling leads to a design which is more difficult to develop and maintain. One of the working definitions for a loosely coupled system is that the software units are black boxes; that is, the only information other software units know about the black box are its name, inputs, and outputs. Any blackbox unit can be replaced with a different unit having the same name, inputs, and outputs with no effect on the system.

2GCHAS Modules have three attributes that make them a loosely coupled system:

1. 2GCHAS Modules are FORTRAN subroutines with names and specific calling sequences; i.e., Modules are black box software units;

2. COMMON is not allowed as a communication mechanism between Modules, thus removing the greatest source of data coupling from 2GCHAS Modules;

3. Modules are linked at execution time, not at compile time or linkage edit time.

So how do Modules help improve productivity? The primary benefit is that any Module can be changed with little concern for other Modules in the system. As long as the new version of the Module has the same name and calling sequence, no changes are required elsewhere. For software development, top down implementation is easy. As they are developed, the functional Modules replace the limited function stub Modules. The replacement only requires that the functional Module be processed by DEFMOD.

The second benefit arises from the fact that Modules are located and loaded at run time via an extension of the TAE hierarchy search. A developer's personal version of any Module can be invoked at run time instead of the "official Module" in the 2GCHAS system library. Thus, a developer can have corrected versions or completely new versions of existing Modules in a private account and can test those Modules as part of a complete system. This means that new Modules to be integrated have been already been tested as part of a complete system.

CHANGE CONTROL HELPS PRODUCTIVITY

2GCHAS has created a set of configuration management support tools which enhance the productivity of all developers. These support tools provide formal change control, automatic system generation from source changes, and continuing operation of the previous version while a new system is being generated.

TAE is delivered for VAX/VMS systems, the System Manager's Guide describes how to set up the TAE version tree so that only the TAE system manager has write access. Change by anyone other than the system manager is effectively prevented. VAX/VMS file protection and the TAE hierarchy search provide both control and flexibility. 2GCHAS uses these basic TAE concepts.

It was recognized early that a mechanism to quickly and reliably introduce changes to 2GCHAS in a controlled manner was required. The mechanism, CHASGEN, consists of seven manual steps, each either a DCL procedure or a 2GCHAS procedure. The CHASGEN process is started when the Configuration Management officer is notified that new source units are being delivered. The Configuration Management officer begins the CHASGEN by copying the new source units -- not object files, libraries, executables, or message help file indices -- into the 2GCHAS version tree. Based on each source unit's file type and update date, the appropriate actions -- compile, library update, link, MSGBLD, etc. -- are performed. Within a short time, the Configuration Management officer, not a senior programmer, has constructed a new 2GCHAS Executive.

CHASGEN has proven to be quick and reliable. However, interruptions to users or developers is costly. Therefore, CHASGEN does not operate against the current version tree, but against a newly created version tree. The previous version remains untouched and operational. When the newly created version tree has been tested and proven to be good, it is then available for use.

STANDARD TAE FEATURES AS PRODUCTIVITY AIDS

The focus thus far has been on the 2GCHAS extensions to TAE and the benefits derived from those extensions. In large measure, however, those extensions have been possible within the real constraints of time and money because TAE provided such a stable platform upon which to build. This last section provides a brief summary and, in some cases, a recapitulation of the TAE features which have directly contributed to a high level of productivity within the 2GCHAS project.

For the programmer at a terminal, HELP and TUTOR, not paper documents, provide 2GCHAS documentation without work flow interruptions of referring to a manual. Fewer interruptions means that the programmer more effective uses time.

The hierarchy search and its extension to Modules provide a simple and easy mechanism to check out and test new software. Little effort is required to have a private version of some part of 2GCHAS and, so, there can be more effort available for other tasks.

TAE provides many features for tailoring the user interface. So far, every 2GCHAS developer has used ULOGON to tailor the user interface. There is a great deal of idiosyncratic tailoring, but two characteristics are common. The first is to use DEFCMD to make commonly used VAX/VMS DCL commands available inside 2GCHAS. The second is having PDFs which allow various editors to be invoked from inside 2GCHAS. The DEFCMDs and editor PDFs means that the 2GCHAS and DCL environments are the same in many respects, making the transition from one environment to the other smoother for the developer.

2GCHAS chose to use message help files as a primary means of providing online diagnostic information. The simple and powerful mechanisms of message help files and the MSGBLD utility contribute to productivity in three ways. The first is that messages and their corresponding help can be grouped. Having the help for possible messages produced by related software units -- a TAE facility -- together in a single place makes it easier to get consistency of information across messages. The second is that the 2GCHAS project experience indicates that message keys tend to get reused for similar or identical situations. When this happens, there is less help text to generate and the message text can be used to provide occurrance specific information. The third property of message files which acts to reduce effort is that they are located outside of the software units to which they apply. This means that the continual activity of making clarifications and additions to message help does not appect the software units themselves.

TAE features have significantly reduced the testing effort for 2GCHAS. Formal testing, the use of predefined and documented test scenarios generating output for comparison against expected results, is a 2GCHAS project requirement. The contents of each new system generation or build are quickly catalogued with script files containing TUTOR and HELP commands for each 2GCHAS service. If the script completes without stopping, then all of the services are present. A missing service stops the script -- $MESSAGE is set to "ATTN" -- allowing the tester to log the missing service. The script files provide for a level of speed and repeatability that a person at a terminal with a written test scenario can only dream of approaching.

2GCHAS test procedures have been designed to be both self verifying and self reporting. Basically this means that the test determines whether or not it succeeded and then reports the success to the test conductor. The effort to perform and document formal testing is reduced because of these test procedures. In addition to scripts, tests are implemented as procedures and processes. $SFI, the success/fail indicator, is set by every test and reported in the session log. The STDOUT command qualifier is used to retain output generated by individual tests. Listings of session logs and test output provide most of the formal test documentation, further reducing the testing effort.

2GCHAS - A High Productivity Software Development Environment

CURRENT 2GCHAS STATUS

The Executive Complex of 2GCHAS is being developed under contract by Computer Sciences Corporation (CSC) at Ames Research Center in a series of five builds. CSC delivered Build 3 of the Executive to the Army in September, 1986. Technology Module development began in early 1986. Technology Module developers will access Executive Build 3 (and Build 4 later) either by telecommunicating with AMES or by having a copy of the 2GCHAS on their own VAX. Run time Linking was included in Builds 1, 2, and 3 of the Executive; was tested by the Army as part of normal build testing; and was used by CSC for developing Builds 2 and 3.

SUMMARY

TAE provides both tools and concepts for achieving high productivity software development. Additional requirements have led the 2GCHAS project to extend TAE's tools and concepts with resulting additional increases in productivity.

ACKNOWLEDGEMENT

I would like to thank the staff of the 2GCHAS project for their support and for their diligence in creating the products and results reported here. Special thanks to Clark Oliphint for providing text and illustrations for the discussion of run time linking.