File

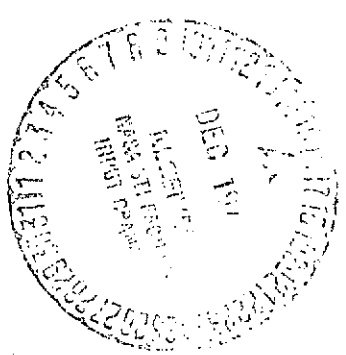
R - 721

DESIGN OF THE SOFTWARE DEVELOPMENT AND VERIFICATION SYSTEM (SWDVS) FOR SHUTTLE NASA STUDY TASK 35 - S

by

Lance W. Drane
Bruce J. McCoy
Leonard W. Silver

August 1972



CHARLES STARK DRAPER LABORATORY

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

CAMBRIDGE, MASSACHUSETTS, 02139

R - 721

DESIGN OF THE SOFTWARE DEVELOPMENT AND VERIFICATION SYSTEM (SWDVS) FOR SHUTTLE NASA STUDY TASK 35 - S

by

Lance W. Drane
Bruce J. McCoy
Léonard W. Silver

August 1972

THE CHARLES STARK DRAPER LABORATORY
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
CAMBRIDGE, MASSACHUSETTS 02139

Approved: J. H. Laning Date: 8/24/72
J. H. LANING, DIRECTOR, DIGITAL COMPUTATION
CHARLES STARK DRAPER LABORATORY

Approved: David G. Hoag Date: 24 Aug 72
D. G. HOAG, DIRECTOR
APOLLO GUIDANCE AND NAVIGATION PROGRAM

Approved: Ralph R. Ragan Date: 24 Aug 72
R. R. RAGAN, DEPUTY DIRECTOR
CHARLES STARK DRAPER LABORATORY

ACKNOWLEDGEMENTS

This report was prepared under DSR Project 55-23890, sponsored by the Manned Spacecraft Center of the National Aeronautics and Space Administration through Contract NAS 9-4065. The study is authorized by the NASA/MSC Task Review Integration Panel (TRIP) as Task 35-S.

Gunter R. Sabionski served as the NASA/MSC technical monitor of this study.

The authors acknowledge the contribution of Daniel W. Corwin, Robert A. D'Angelo and Julia Hsia who actively participated in this study and wrote sections of this report. Numerous other MIT/DL personnel, particularly those in Group 23-B, contributed ideas, wrote working papers, and participated in design sessions. H. D. Nayar was the technical editor for the report.

The publication of this report does not constitute approval by the National Aeronautics and Space Administration of the findings or the conclusions contained therein. It is published only for the exchange and stimulation of ideas.

PREFACE

This report consists of four sections. Each section contains a description of the SWDVS from a different viewpoint. Reading each additional section will increase the reader's insight into the design proposed by this report. The purpose of each section is explained below.

Section 1 is an overview of the SWDVS, including the SWDVS design goals, assumptions, considerations, and a review of the major features of the design. This section serves as a management summary.

Section 2 is a scenario that shows three persons involved in flight software development using the SWDVS in response to a Program Change Request (PCR). The scenario consists of a series of figures representing interactive terminal display screens. It illustrates the various ways in which users interact with the SWDVS in the course of solving a particular problem.

Section 3 describes the SWDVS as approached by different groups of people with different responsibilities in the SHUTTLE program. It thus describes the functional requirements that influenced the SWDVS design.

Section 4 describes the software elements of the SWDVS that satisfy the requirements of the different groups described in Section 3. This section may be used as a reference section for the specific programs mentioned in the previous three sections.

In order to make this report more readable, names were created to represent some of the systems and concepts described. The SWDVS management information retrieval system is called SNOOPY, for example, and the SWDVS test run control and monitor system is called SUPERCREW.

Other names were adopted from systems and concepts currently existing and documented at MIT/DL, and used for the development and verification of the APOLLO software. The specific name MARSROT, for example, is used in place of generalized descriptive substitutes such as "snapshot-rollback management", "post-run edit management", and "tape I/O management" because none of these names conveys the desired concept. The list of acronyms and definitions includes all the specific names that are defined only in this report.

Where possible, this report does not assume an APOLLO software background. In many examples, however, the APOLLO Display Keyboard (DSKY) language is used.

ACRONYMS AND DEFINITIONS

CRT	Cathode ray tube
CTS	Crew Training Simulator
Configuration Control	For flight software development, the management process that limits program changes to only those necessary to ensure mission success and crew safety.
Data Base	All the collected data and programs of the SWDVS.
Design Phase	The flight software development phase using the engineering simulators.
DOD	Department of Defense.
Edit	The process of analyzing data collected during a simulation.
EDIT	A program to edit simulation output.
ESIM	External-environment Simulator (also known as engineering simulator, driver).
FSIM	Functional-computer Simulator (also known as functional flight code simulator).
Flight-code Compiler	A tool of the SWDVS which provides a high-order language for structured programming; the input source language is translated through code generators into code for the various flight computers and the SWDVS host computer.
Generic Software Failure	A failure due to inherent limitations of the flight software.

HOL	High-order language.
ICS	Interpretive-computer Simulator (also known as bit-by-bit simulator, instruction-by-instruction simulator, instruction functional simulator).
I/O	Input/Output.
Implementation Phase	The flight software development and verification phase using the SWDVS.
Keyword	A non-executable program tag assigned at compile-time that indicates the function of a coding block; it is used by the data retrieval system to connect interdependent sections of code.
LIPSVC	List Processing Service. A data management system that allows easy updating of source listings and easy retrieval of previous versions of any source listing.
LOADER	The part of the Interpretive-computer Simulator that preprocesses flight-computer code for efficient simulation by the instruction routines.
MARSROT	MARS Rough-output tape—A system incorporating snapshot-rollback, editing, and rough-output tape management.
MIT/DL	Massachusetts Institute of Technology, Draper Laboratory.
MSC	Manned Spacecraft Center.
NASA	National Aeronautics and Space Administration.
OS	Operating system of the family of computers chosen to host the Software Development and Verification System.
PANSIM	Combination of External-environment Simulator, Functional-computer Simulator, and Interpretive-computer Simulator.

PCN	Program Change Notice, a flight-software control document.
PCR	Program Change Request, a flight software control document.
PCS	Program Control Supervisor; reviews and approves code to be implemented on the flight program.
Plot	The process of generating graphic output.
Redline	The upper and lower extremes of acceptable values of a flight-computer variable; produced by the code-generation process.
Rollback	The name given to the exact bit-by-bit reproducible restarting of a run at a snapshot.
SCB	NASA Software Control Board.
SIL	Systems Integration Laboratory.
SIMSETUP	A program that initializes and ensures compatibility between the Functional-computer Simulator, External-environment Simulator, and Interpretive-computer Simulator variables.
Snapshot	A set of data on a rough-output tape consisting of simulation-program variables complete enough to allow an exact restart of the simulation run to be made at that point.
SNOOPY	A data retrieval system used to keep track of software development for SWDVS.
Special Request	Runtime diagnostic or a flight-program data patch used during a simulation.
Special Request Processor	The program that processes special requests for each simulation run.
STE	Software Test Engineer.

SUPERCREW	The language and program that controls a simulation run.
SWDVS	Sotfware Development and Verification System.
Runtime Package	An executive and set of dynamic diagnostics used to support the execution of a flight program on the host computer.
System Verification Phase	The software development phase using the Systems Integration Lab, Crew Training Simulator, and the Shuttle Mission Simulator.
TDS	Television Display System.
Test Data File	Those sets of data, diagnostics, and directives available for initializing and controlling a simulation.
Test Plan	Data base containing test cases for simulation.
XPANDER	A program that sorts, structures, expands, and translates simulation input.

TABLE OF CONTENTS

TABLE OF CONTENTS (CONT)

TABLE OF CONTENTS (CONT)

ILLUSTRATIONS

SECTION 1

OVERVIEW

1. OVERVIEW

Section 1 is divided into two subsections. The first essentially presents a history of the development of the Software Development and Verification System (SWDVS) design, describes the goals, assumptions, and special considerations of the design effort, and, finally, explains how the design satisfies the stated design goals. The second subsection presents a summary of the major features of the SWDVS design.

1.1 HISTORY OF DEVELOPMENT OF SWDVS DESIGN

1.1.1 Goals

> The primary goal is to design a system for use as a tool to develop and verify flight software from flight software requirements and formulations. The verification process must develop a confidence in the flight software that guarantees very high probability of crew safety and very high probability of mission success.

Because of the amount and complexity of the SHUTTLE software, achieving the desired quality requires automating many functions within the development and verification system. Automated functions emphasize those steps in the verification process in which engineering judgements are made by de-emphasizing the mechanical tasks. This is important because making engineering judgements visible is the best assurance of final confidence in the flight software and the best protection against generic software failure. Confidence in the final flight software is, itself, a judgement which must be built on tiers of previous engineering judgements.

Because the SWDVS must function in a changing environment, the second goal defines the SWDVS's capability to achieve the desired flight software quality in that environment.

> The second goal is to design a SWDVS that is flexible enough to respond to new demands imposed on verification by the evolving flight software by simply adding new control, new diagnostic, and new edit capabilities. Further, uninterrupted use of the SWDVS should be maintained while the SWDVS

itself benefits from expected software tool evolution and expected facility hardware advances during its lifespan.

An effective verification process can generate new insight into the performance of the flight software. That insight can further suggest better techniques for additional software verification to which the SWDVS must be capable of responding. The SWDVS must also respond to the need for a repeated updating of sensor and vehicle models as laboratory test reports and flight test data suggest new models as well as new data for old models.

The flexibility required of the SWDVS is, for the most part, achievable by the design itself without additional cost in runtime efficiency. In some cases, however, a trade-off exists between flexibility and runtime efficiency. Since flexible response offers a major cost savings, as well as an increase in quality, the trade-off is weighted heavily toward flexibility.

In order for the SWDVS design to reflect the goals of the SHUTTLE program, the obligation of cost consideration must be accepted at the design level.

> Thus, the third goal is to design into SWDVS those features that enhance cost-effectiveness by introducing quality into flight software as early as possible in the development chain, by increasing the capabilities of each software test engineer, and by using the facility hardware in an efficient manner.

Any design aspect that gives a software test engineer better insight into each step in the development process, and that step's relationship to the whole, introduces early quality. This same insight into the relationship of each task to the whole development process increases efficiency by allowing a test engineer to grasp a larger problem all at once instead of in smaller steps. In addition, any design aspect that decreases the time required to learn to use the SWDVS is cost-effective. Finally, efficient use of facility hardware is obtained by designing the SWDVS for greatest runtime efficiency when the system is running in its most often used mode. (See Section 3.)

1.1.2 Assumptions

Fulfillment of the design required making certain assumptions. They are as follows:

For the purposes of this design, the NASA/MSC "Green Report", "Space Shuttle Program Avionics Systems Recommendations" (Reference 1), was used. This report is not considered final.

Responsibility for achieving final confidence in flight software is divided among the six simulators defined in the NASA/MSC report, "Space Shuttle Simulation Program" (Reference 2).

All SHUTTLE flight software will be verified to the quality required to achieve very high probability of crew safety and very high probability of mission success.

1.1.3 Considerations

In addition to the assumptions listed above, concern focused on the following broad basic considerations which have, in turn, influenced this design.

1. During the course of the SHUTTLE program, the state-of-the-art for avionics systems will advance. In addition, some respecification and broadening of mission goals can be expected as operational experience is gained. These observations reinforce the emphasis on flexibility in the design of the SWDVS.

2. During the projected lifespan of the SHUTTLE program, progress will be made in the development of general-purpose computer hardware. The SWDVS must, therefore, be designed to accommodate these hardware developments without disrupting smooth software development.

3. Because many people will be involved with SHUTTLE software over this period, the techniques for transferring knowledge of the software systems must be automatic and guaranteed. The SWDVS must be a system that can be operated and maintained by groups of people other than its originators.

4. The amount and complexity of the flight software for SHUTTLE exceeds that for APOLLO. Although APOLLO experience is relevant to solving SHUTTLE software problems, APOLLO solutions alone are not sufficient to accomplish the SHUTTLE task.

5. Utilization of the same manufacturer-supplied software tools used by other data processing centers would be beneficial to the SWDVS. The

SWDVS should share the fruits of development and debugging efforts by others where possible.

6. The benefit of using proven software tools that have established confidence levels because of their period of use should be weighed heavily.

7. The NASA Software Control Board may choose to announce, "the SWDVS is perfect; there will be no more releases." By that time the SWDVS may be staffed by NASA civil servants at NASA/MSC without contractor support.

8. The software test engineers will learn to use the SWDVS on the SWDVS host computer facility.

9. The SWDVS design must allow a uniform development effort that avoids an expensive peak effort. This reflects the statement of the NASA Steering Group as quoted in Reference 2, ". . . to achieve austerity, projects should evolve through a series of logical steps from concept through development." Further, that schedule must allow the major development and verification of the SWDVS to be completed before flight software development and verification begins.

1.1.4 Conclusions

The following conclusions are reached concerning the design of the SWDVS presented in this report.

1. The SWDVS design presented is a sufficient tool with which to develop and verify SHUTTLE software to the quality desired of the implementation phase.

2. The SWDVS will respond flexibly and easily to the changing environment of flight software development.

3. The SWDVS design offers a wide range of options in selecting facility(ies) suitable to changing computation needs and facility hardware advances.

4. The SWDVS offers an approach to controlling flight software development and verification costs.

5. A new software test engineer begins to gain experience running the SWDVS immediately by using the predefined Test Plans and Test Data File which give him access to the previous work of others.

6. The SWDVS represents an integration of state-of-the-art concepts and techniques which have been proven in existing software systems.

7. Detailed design, development, and integration of the SWDVS can begin immediately.

1.2 FEATURES OF SWDVS, A SUMMARY

1.2.1 Structure of SWDVS

The Software Development and Verification System (SWDVS) is one integrated software package. (See Figure 1-1). As a software package, it is readily executable on any member of the family of host computers for which it is designed. The elements of SWDVS are:

The SWDVS-compatible code generator, preferably a high-order language (HOL) flight equations and sequences compiler

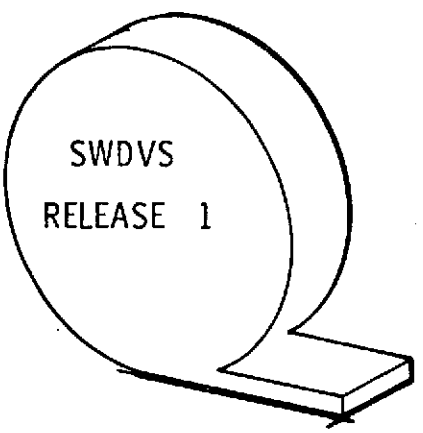
The simulators

The data base with list processing data management control

Diagnostic and analysis programs.

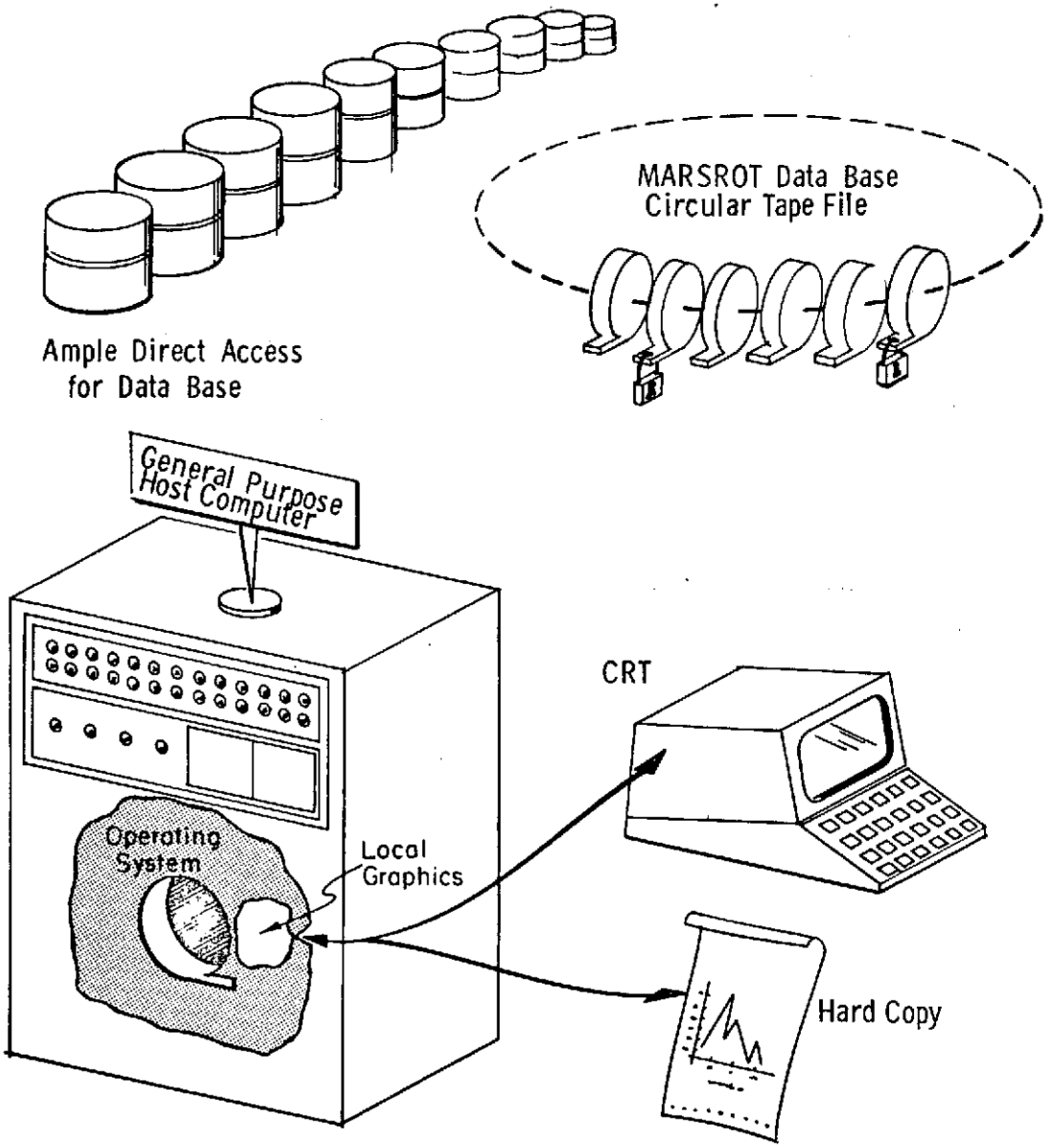
The layout of the primary facility that hosts the SWDVS will be determined at that time in the SHUTTLE program that commitment to a dedicated facility is necessary. At that time current hardware capabilities and computation requirements will determine the best layout. The minimum requirements that a facility must meet to host the SWDVS (called a SWDVS-compatible facility) are designed to be easily attainable by any facility with these features (see Figure 1-2):

A general purpose computer of the same family for which the SWDVS is designed, with its resident, standard manufacturer-supplied operating system, ample direct access storage devices and tape drives



The Software Development and Verification System (SWDVS) is one integrated software package.

Figure 1-1. The Software Development and Verification System (SWDVS)



The SWDVS is hosted on a general purpose computer facility consisting of hardware and the necessary software to interface with that hardware.

Figure 1-2. The SWDVS Facility

Output devices for text and graphics, such as printers and plotters, but also including interactive terminals with hard-copy capability

Software (called SWDVS-compatible graphics) that drives the local graphic display devices from the output of the SWDVS graphics program called GRAPHICS.

The SWDVS is developed on the one prime facility which, if deemed necessary, provides NASA Software Control Board approved releases to secondary facilities. A secondary facility may be a disaster backup facility, additional software subcontractors' facilities, a classified Department of Defense facility, the independent verifier's facility, or a facility rented during peak verification periods. Use of additional facilities will depend upon changing computational requirements. (See Figure 1-3.)

In addition, at some point in the SWDVS's development cycle, after all the pieces are together and the data base contains flight programs and useful information, the SWDVS may be given by NASA to any facility to develop a SWDVS-compatible facility, to develop SWDVS-compatible graphics, to develop a SWDVS-compatible code-generator, to develop a SWDVS-compatible emulator (as a part of one of the Interpretive-computer Simulators), to use a realistic, proven system for further development of software tools for NASA, or to study software development and verification so that Shuttle experience contributes to the state-of-the-art.

1.2.2 Flight-Code Generation

To provide an orderly and controlled flight software production effort and to increase flight software reliability while lowering verification costs, one common HOL flight equations and sequences compiler is recommended for the flight code generation process. However, an assembler may be more appropriate for certain types of systems programs. Assembly language coding might also be used if the compiler is not available when flight software development begins. The compiler should thus be designed to interface with assembly language programs.

The features specified herein are requirements for whatever collection of compiler and/or assemblers is selected for the flight-code generation process. The collection is defined as the SWDVS-compatible code generator, and has the following features:

1. It provides one common source language that is easy to learn, use, debug, modify, and read.

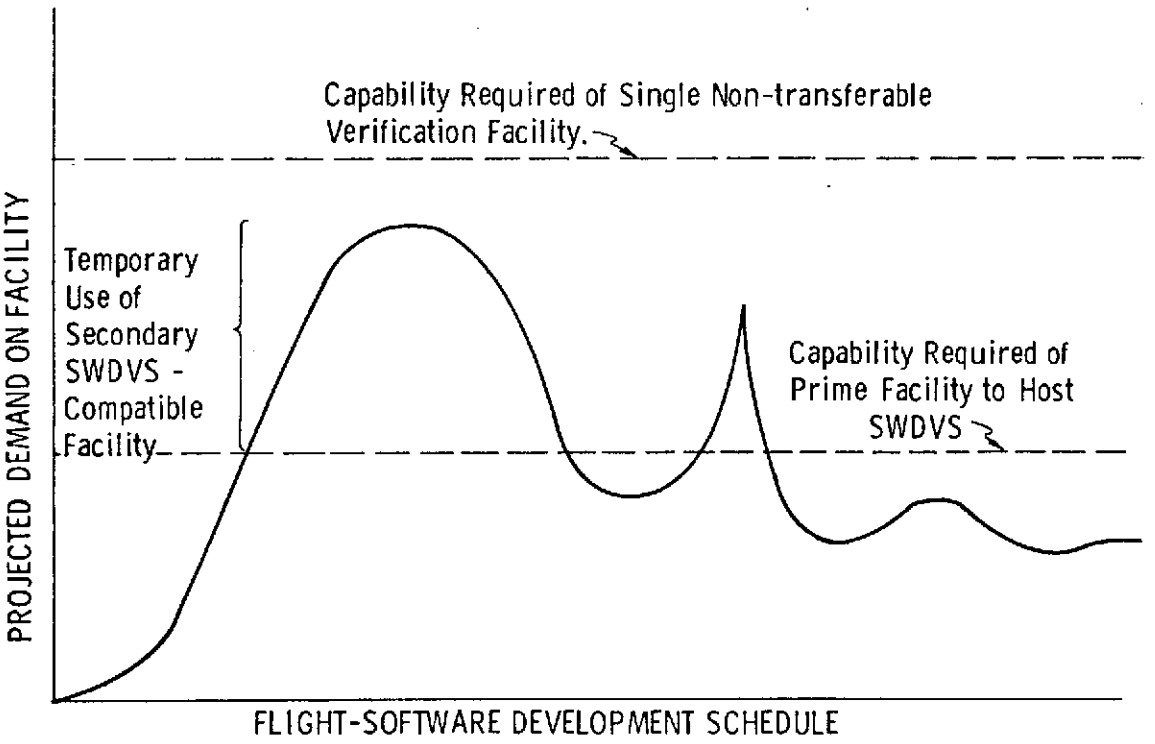


Figure 1-3. Alternate Methods of Responding to Projected Demand on the SWDVS Facility.

2. It provides a translator and code-generators for converting the equations and sequences in the source language into machine code for the flight computers and the SWDVS host computer.
3. It operates in four different modes:
   a. engineering and systems design mode
   b. program module compile mode
   c. integration of program modules mode
   d. simulation and verification (patch) mode.
4. It responds to the requirements of both systems and applications programming.
5. It enforces programming standards and conventions.
6. It provides extensive automatic checking.
7. It provides supervisory control features.

1.2.3 Simulators

The overall structure of the Flight-computer Simulator (PANSIM) permits testing of one flight computer program by means of its Interpretive-computer Simulator (ICS), either alone or in conjunction with the Functional-computer Simulator (FSIM) and the External-environment Simulator (ESIM). The FSIM models one or more additional flight computers interacting with the program being interpretively simulated. The ESIM contains math models of the flight computers' environment, consisting of the avionics subsystems which interact with the flight computers and the universal phenomena.

The Interpretive-computer Simulator (ICS) simulates, on the host computer, the instruction-by-instruction execution of code by a flight computer. It is used by systems and applications programmers to verify the proper functioning of individual program modules and the interfaces between modules. The ICS is also used by software test engineers (STEs) to verify the performance of a complete flight program at the instruction level. The structure of the ICS isolates the non-I/O synthetic instruction processing to allow an orderly transition from the initial all-software synthetic instruction processing techniques in the ICS to firmware (micro-programming) or to hardware (emulation), if desired to achieve runtime efficiency. The SWDVS is so structured that the use of micro-programming or emulation in the ICS is entirely compatible with the remaining software elements.

The Functional-computer Simulator (FSIM) is written in the same high-order language used to write programs for the flight computers but is compiled to run on the SWDVS host machine. Analysts and systems designers compile code modules into the FSIM

in order to examine algorithm performance and interface compatibility. During flight-program verification, the FSIM models the interaction of other flight computers with the program being run on the ICS. The FSIM is also a useful tool for investigating the characteristics of redundant computer systems.

The External-environment Simulator provides avionics subsystem and natural environment models with various levels of fidelity. The simpler ESIM models are used in conjunction with the FSIM as a tool for systems design and analysis of SHUTTLE flight software. More detailed models are available to STEs for use during the flight software verification process. Many of the math models incorporated in the ESIM will be obtained from the Systems Development Simulation Facilities; the resulting commonality between the design phase and implementation phase simulators should aid in reducing SWDVS development costs.

1.2.4 Data Base

The SWDVS is one integrated software package, and it may be accurately described from several viewpoints. One appropriate viewpoint describes the SWDVS as a massive collection of stored source listings, alphanumeric input and output data, and software-management aids. This collection is referred to as the SWDVS symbolic data base.

The complex and changing nature of the data base can be discerned from a partial list of its contents:

1. All developmental and active versions of flight program source listings;
2. All developmental and active versions of PANSIM program source listings;
3. All data files used or previously used for SWDVS programs; all initialization data sets used by PANSIM;
4. Test plans and their status, test case results, and figures of merit;
5. Other selected simulation output or rough output for post-run edits, rollbacks, or simple storage, formatted where necessary to the SWDVS-compatible-graphics level;
6. Program change requests (PCR), program change notices (PCN), and their status;
7. Software-demanded documentation and program management data input by programmers at each simulation run and at each program (re)compilation;

8. Symbolic descriptions of the function and form of each element contained in the SWDVS data base, allowing the manipulation of descriptions of data as well as of data itself.

Two mutually-supporting software systems accept the bulk of the high-level responsibility for managing this data base efficiently and making its contents available for revision, addition or inspection:

- LIPSVC (List Processing Service) maintains all sequential source listings of programs on data sets in a manner suitable for nearly indefinite expansion. The full history of a program's development, and the differences between any two versions are readily available.
- SNOOPY—maintains an extensive and accurate set of descriptive information on all items in the SWDVS data base. Designed as an evolving system, SNOOPY acts as a front end of a collective memory for facts on the SWDVS and its development. It provides for the easy input of new information by scanning all (re)compilations and by directly accepting data from terminals. It provides for the rapid retrieval of information at the request of any engineer, programmer or SWDVS manager, accepting commands and questions in limited-grammar, limited-vocabulary English.

1.2.5 PANSIM Initialization

The STE sets up a simulation run by selecting a standard, predefined initialization and test sequence from the inventory of test cases in the test plan source file. These test cases are constructed with the aid of predefined data sets contained in the test data file. The test engineer modifies the standard test case as necessary to yield the test case specification for his particular run. The types of predefinable inputs that make up a test case specification include:

- Total test specification (test case)
- Simulation control inputs, crew actions (flight plans, SUPERCREW directives)
- Diagnostic and analysis packages (special requests, edit programs)
- Dynamic diagnostics associated with compiled flight computer programs (redlines, code definitions)

These inputs are sorted and structured by the XPANDER program, and then processed by several specialized processors to yield a complete initialization, diagnostic, and

control package for PANSIM. All initialization features are automatically available at runtime; however, the STE may choose to ignore any or all of these features whenever desired.

Predefined initialization and test sequence data sets are defined by STEs specifically assigned that responsibility because of their knowledge of a particular phase of SHUTTLE operations. In this way, the experience of everyone who uses the SWDVS is made available to the individual STE. The number of times each predefined data set has been used is indicated at runtime to provide STEs with a measure of confidence in the data sets through accumulated use. This confidence, coupled with the responsible test engineer's recommendations, allows a manager to promote a predefined data set to an official status in the test data file.

The test case specification for a given run is formulated via interactive terminal, and is submitted in the same manner for batch-mode initialization. This initialization is processed at the highest priority of the operating system in order to give fastest turnaround possible to the waiting STE. Results of the initialization are available for interactive viewing after initialization terminates. (This process for the current APOLLO All-Digital Simulator at MIT/DL would take about two minutes.) The STE then has the option of recycling the initialization, storing the new test case specification for future use, or submitting the run to the batch mode job queue.

1.2.6 PANSIM Test Run Control

The test control program, SUPERCREW, provides the STE with the techniques for controlling and monitoring a test run. It allows the STE to generate test run directives for four general classes of activity:

1. performing crew activities
2. monitoring the test run
3. initiating hardware failures
4. controlling other directives to the simulators.

The ability to generate these test directives in a manner that closely ressembles the test run description is cost effective for the flight software verification process. SUPERCREW provides each STE with easy access to the nominal crew sequences as defined for the test data file by the engineer responsible for each mission segment. It also allows intuitive specification of hardware failures and non-nominal crew activity which is a necessary part of manipulating flight software for the verification process.

Directives to SUPERCREW range from such a general directive as executing a mission segment to the detailed directive to change a single switch position. Directives can activate capabilities of responding in a closed-loop fashion to the simulator response. SUPERCREW can engage the avionics system in a conversational manner according to the predefined conventions. SUPERCREW can maneuver the vehicle via the handcontroller and maneuver the navigation sensors. It causes hardware failures and inserts spurious data onto a data bus. Finally, it monitors continuous functions, such as vehicle state, that would be monitored by a flight controller at an interactive terminal.

1.2.7 MARSROT System

The MARSROT System performs three major functions for the PANSIM: the dump-edit function, the snapshot-rollback function, and the tape-management function.

The dump-edit function writes the output of diagnostic special requests onto a rough-output tape and reads this information back at a later time for selective editing and graphical display. This technique allows the STE to edit the output of a simulation in different ways as many times as he wishes.

The snapshot-rollback function gives the PANSIM a dynamic restart capability that guarantees bit-for-bit repeatability. The STE is allowed to add diagnostic inputs to the PANSIM at rollback time; if no new input is included, the rollback run exactly reproduces the original simulation from the chosen snapshot point. Using the snapshot-rollback system, the STE knows that a long test, run in several short segments with interim analysis of results, will give results identical to one long run. He can add diagnostics via rollback to a simulation that has encountered a problem with the assurance that the rollback run will encounter the identical problem.

The tape-management function maintains a circular file of the rough-output tapes with clerical information concerning each tape.

1.2.8 PANSIM Output

Text and graphical output from a batch-mode simulation is accessible via interactive terminal after the simulation terminates. Hard copy of both text and graphical output is also available, in the same format used for display purposes, at the test engineer's option. In addition, the results of a simulation can be accessed by user-written programs in the SNOOPY system to obtain information for management reports.

Included with the output from a simulation is the LIPSVC name and revision number associated with each program and data set used in that run. This identifying information is sufficient to retrieve, via the LIPSVC source files in the SWDVS data base, all software elements associated with a simulation. In this manner, any run of whatever vintage may be reproduced even though the associated MARSROT with its snapshot points has been deleted due to data base size limitations. Reproducibility by this means (as contrasted with the use of snapshots) is slow, expensive, and tedious, but possible and exact. This capability could be used to analyze very old test cases for phenomena not deemed important at the time, and to limit the number of permanent tapes in the MARSROT library by providing a reasonable alternative to "sanctifying" the MARSROT from a simulation.

A basic set of diagnostic special requests suitable for each ICS should be part of the first delivered SWDVS. This includes at least TRACE, and DUMP. The special request mechanisms are designed so that additional special requests can be implemented easily by the SWDVS staff as the need for them arises.

A basic set of post-run analysis programs, called EDITS, suitable for processing data from test runs should be delivered with the first SWDVS. The major group of EDIT programs should be added to the SWDVS data base during the implementation phase by software developers and software test engineers.

1.2.9 SWDVS Verification

Confidence in flight software depends heavily on confidence in the tools used for verification. Achieving confidence in the SWDVS software is therefore an important step in the flight software development cycle. SWDVS is structured so that a relatively brief test plan can produce the desired level of confidence. The verification of the SWDVS will include generating open-loop responses of the simulator models both to aid the software test engineers in understanding test run results and to aid the SWDVS staff in evaluating the need for simulator model changes as laboratory test results become available.

1.2.10 Interfaces with Other Facilities

For the purpose of reducing development costs, achieving the desired similarities between support software packages, and opening communication between simulation facilities, the SWDVS software package is designed to incorporate some of the common needs of the design phase simulators and the systems verification phase simulators. The major common aspects of these simulators are listed here.

- Each supports the NASA management who ask questions directed to the particular features of each facility.
- Each has a data base that expands with use; these data bases should, therefore, be formatted for expansion and subject to easy high-level manipulation.
- Each needs support in easily communicated areas such as natural environment models. (Called a common math model library in the Shuttle Simulation Planning Committee report, Reference 2).
- Each requires a test run controller. (SIL calls it "Test Director Executive", SWDVS calls it SUPERCREW).
- Each has a potential need to initialize runs specified by another facility. For anomaly search, confirmation, or explanation, a cross-initialization capability is warranted.

SWDVS is designed in a manner which presupposes the exchange of input data, software, and output among these facilities.

1.2.11 Interactive Concepts

The SWDVS data base is structured for efficient interaction via interactive terminals, preferably full screen CRT devices. The SNOOPY data retrieval system formats all graphic output through its own program GRAPHICS, which is designed to interface with the graphics software of any CRT system, plotting system, or TDS system which meets the definition of SWDVS-compatible graphics. GRAPHICS is a device-independent graphics software program.

A real-time, interactive simulation capability is not designed into the SWDVS. Running a real-time, interactive simulation with the features included in the SWDVS would require too many host computer facility resources to be practical. It is felt that such a capability is not a viable solution to the problem of providing flexible monitoring and control of a simulation, fast turnaround, and a feel by the STE for the real time performance of the flight software. Instead, the SWDVS design provides flexible monitoring and control, without real-time interactive simulation, by the use of the SUPERCREW program —whose response is predictable, repeatable, and indefatigable Fast turnaround time can only be obtained by faster (or more) host computers. (e.g., if twenty STEs each want to run a fifteen minute test simultaneously via interaction, they all must sit for five hours–300 minutes–together at their consoles). A feel for the real-time performance of the flight software is better obtained on one of the real-time facilities with actual avionics hardware, where exact reproducibility of a test run is not a consideration.

SECTION 2

SCENARIO

2. SCENARIO

This section leads the reader through a typical use of the SWDVS by three people: a programmer, a program control supervisor, and a software test engineer.

The programmer accepts equations developed by analysts and codes them into meaningful sequences of a high order language using an interactive terminal.

The supervisor reviews and approves the submitted code to ensure consistency, correctness, and fulfillment of implementation requirements. The supervisor also reviews the status of the software development and schedules meetings, presentations, and completion dates accordingly.

The engineer verifies that the compiled program meets software design requirements and that it supports mission plans and techniques.

The code generation process begins with the coding of a program change, which is then verified by eyeball and simulation. Further simulation is performed to ensure flight worthiness. At various points during the development process, the supervisor reviews the coding, test results, and general program status. His main function is to keep the process running efficiently and effectively. He retrieves information stored in the SWDVS data base during the development and verification process. This information aids the supervisor when he makes judgements for software design, use, and delivery.

Much in evidence in this scenario are the comments and promptings of the data base management system, SNOOPY. It should be explained that by specifying certain high-level "modes" of SNOOPY's operation, a user causes the collection of subprograms which automatically lead him into predefined, interactive dialog-trees. These dialog-trees are intended to be flexible, easy-to-use aids, but must also be carefully designed to ensure that nominal software development procedures are followed. Prompted information may be directly necessary to satisfy the user's requests, or it may be necessary purely as a procedural requirement. The reader is invited to be imaginative about the kinds of "behind-the-scenes" data checking and storage that would be taking place.

The Programmer:

T. Symmes has been assigned the job of interpreting Program Change Request (PCR) 423 into flight program SKYVIEW. He has already referenced a printed program listing or has reviewed the code using an interactive terminal display.

To submit a coding change, Symmes must first provide the data base management system, SNOOPY, with certain required "change information", to which he may add anything he feels is necessary. The automatic checking of documents and test titles is routine.

> NOTE: Publication readability and space requirements have made necessary certain idealizations of the dialog stream. The essential flavor, however, is felt to be accurate.

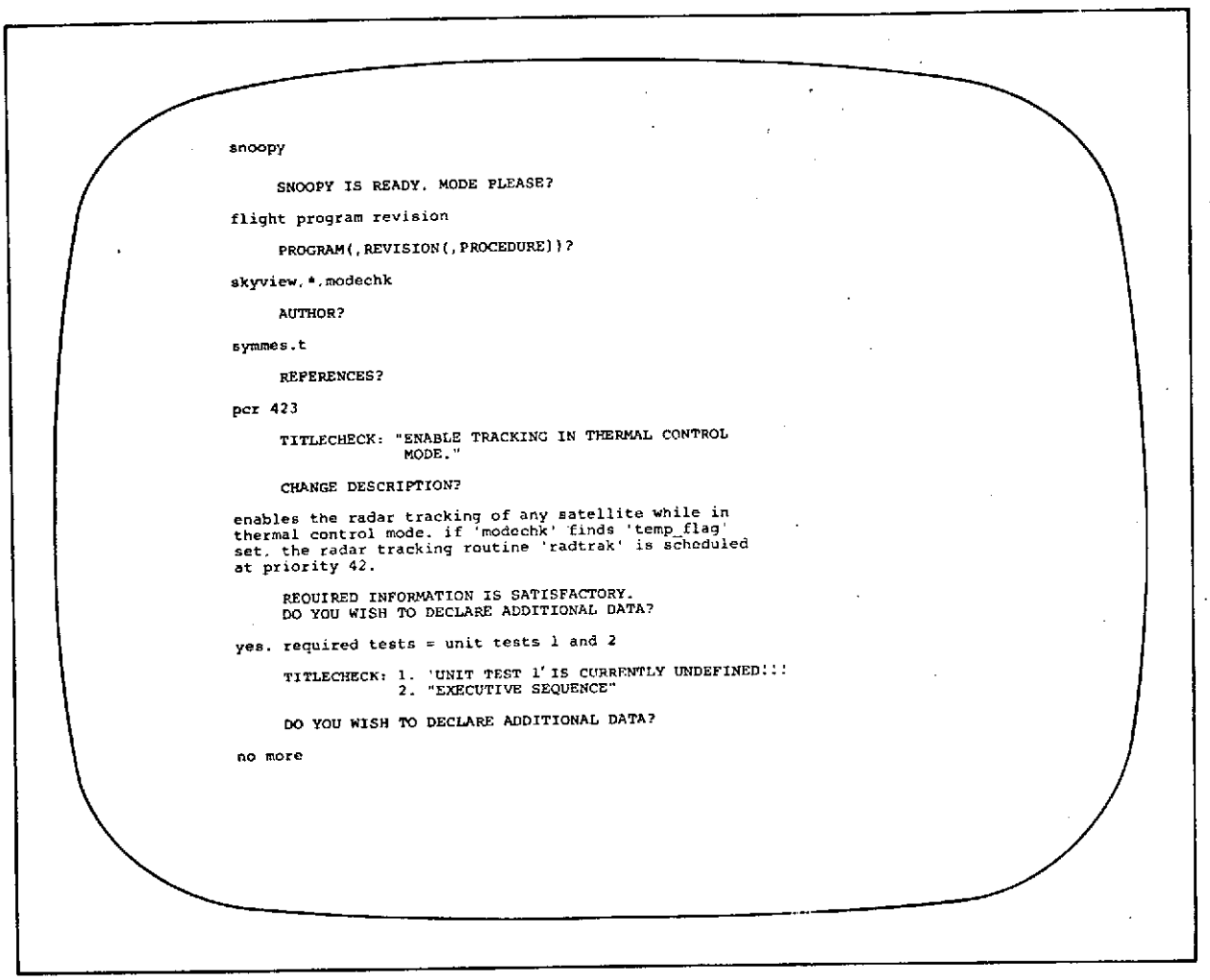

snoopy
SNOOPY IS READY. MODE PLEASE?
flight program revision
PROGRAM(,REVISION(,PROCEDURE))?
skyview,*,modechk
AUTHOR?
symmes.t
REFERENCES?
pcr 423
TITLECHECK: "ENABLE TRACKING IN THERMAL CONTROL MODE."
CHANGE DESCRIPTION?
enables the radar tracking of any satellite while in thermal control mode. if 'modechk' finds 'temp_flag' set, the radar tracking routine 'radtrak' is scheduled at priority 42.
REQUIRED INFORMATION IS SATISFACTORY.
DO YOU WISH TO DECLARE ADDITIONAL DATA?
yes. required tests = unit tests 1 and 2
TITLECHECK: 1. 'UNIT TEST 1' IS CURRENTLY UNDEFINED!!!
2. "EXECUTIVE SEQUENCE"
DO YOU WISH TO DECLARE ADDITIONAL DATA?
no more

Figure 2-1.

SNOOPY describes and displays the section of code T. Symmes wishes to alter. In this case, Symmes inserts new code* at statement 472.

The implemented procedure for changing any part of a flight program includes a keyword** search of the section to be altered. Keyword searches notify the programmer that other sections of coding may be affected by his change.

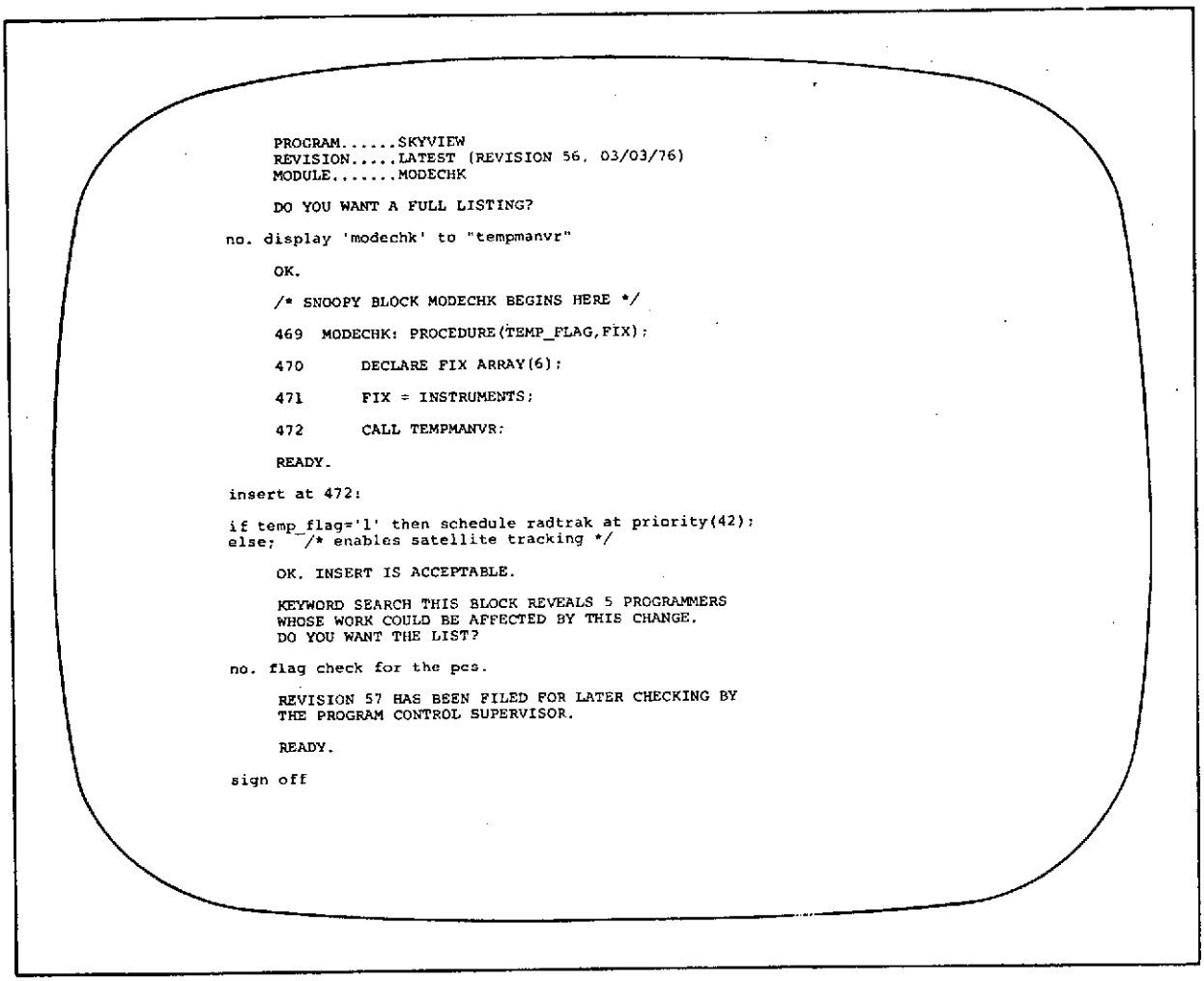
Since only the supervisor has authority to make changes to the main-line assembly, this change is filed for his approval.

Note that the revision #(57) is added automatically preserving all other revisions.

---

*This example makes use of the HAL language.

**See "Acronyms and Definitions."

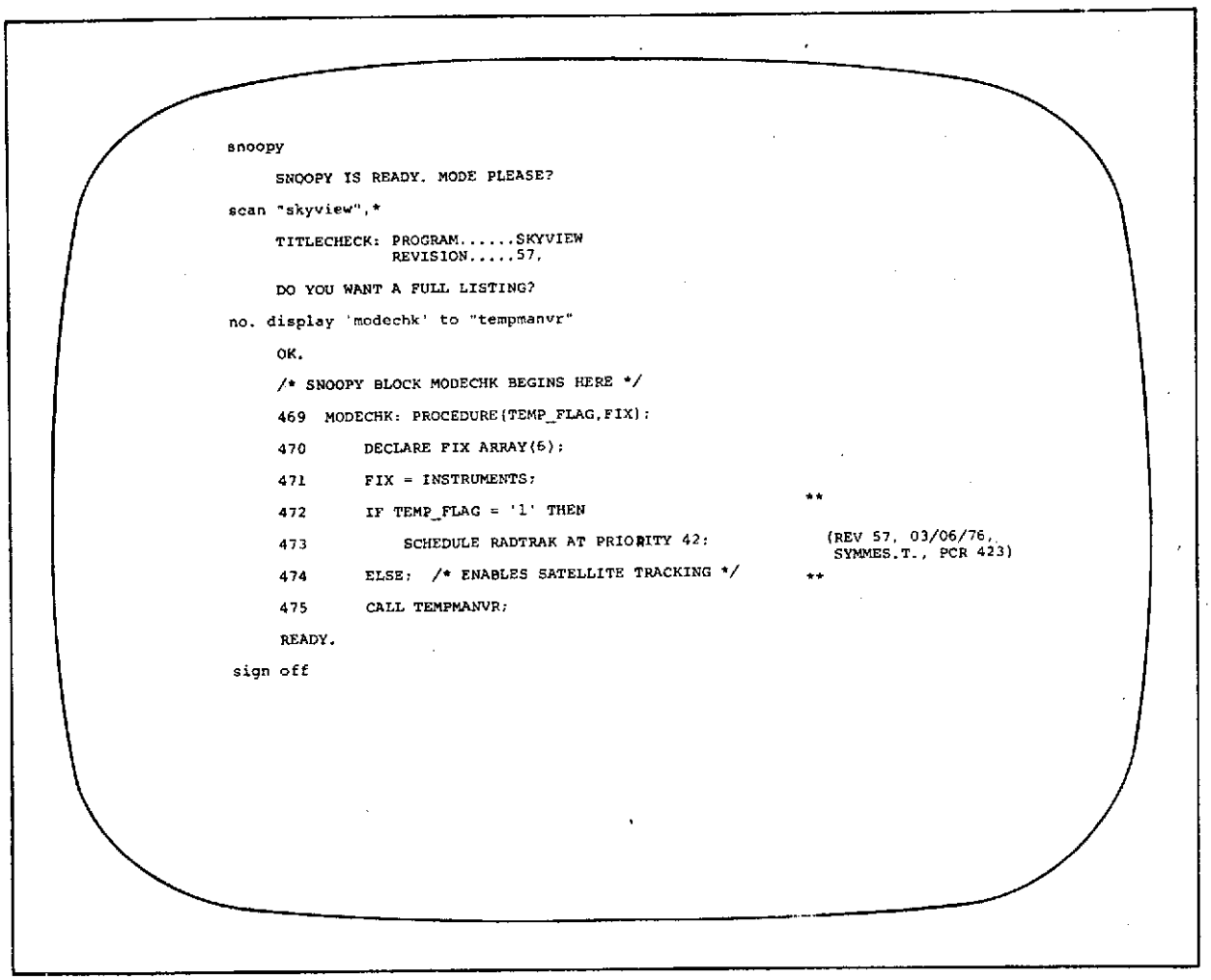


```
     PROGRAM......SKYVIEW
     REVISION.....LATEST (REVISION 56, 03/03/76)
     MODULE.......MODECHK

     DO YOU WANT A FULL LISTING?

no. display 'modechk' to "tempmanvr"

     OK.

     /* SNOOPY BLOCK MODECHK BEGINS HERE */

     469  MODECHK: PROCEDURE(TEMP_FLAG,FIX);

     470       DECLARE FIX ARRAY(6);

     471       FIX = INSTRUMENTS;

     472       CALL TEMPMANVR;

     READY.

insert at 472:

if temp_flag='1' then schedule radtrak at priority(42);
else;   /* enables satellite tracking */

     OK. INSERT IS ACCEPTABLE.

     KEYWORD SEARCH THIS BLOCK REVEALS 5 PROGRAMMERS
     WHOSE WORK COULD BE AFFECTED BY THIS CHANGE.
     DO YOU WANT THE LIST?

no. flag check for the pcs.

     REVISION 57 HAS BEEN FILED FOR LATER CHECKING BY
     THE PROGRAM CONTROL SUPERVISOR.

     READY.

sign off
```

Figure 2-2.

The Program Control Supervisor:

After supervisor approval of Symmes' coding, a new compilation including the change is made. This compilation can then be visually verified on the display terminal by the supervisor as shown. Note that the revision, date, and source are indicated beside the changed statements within the double asterisk sign.



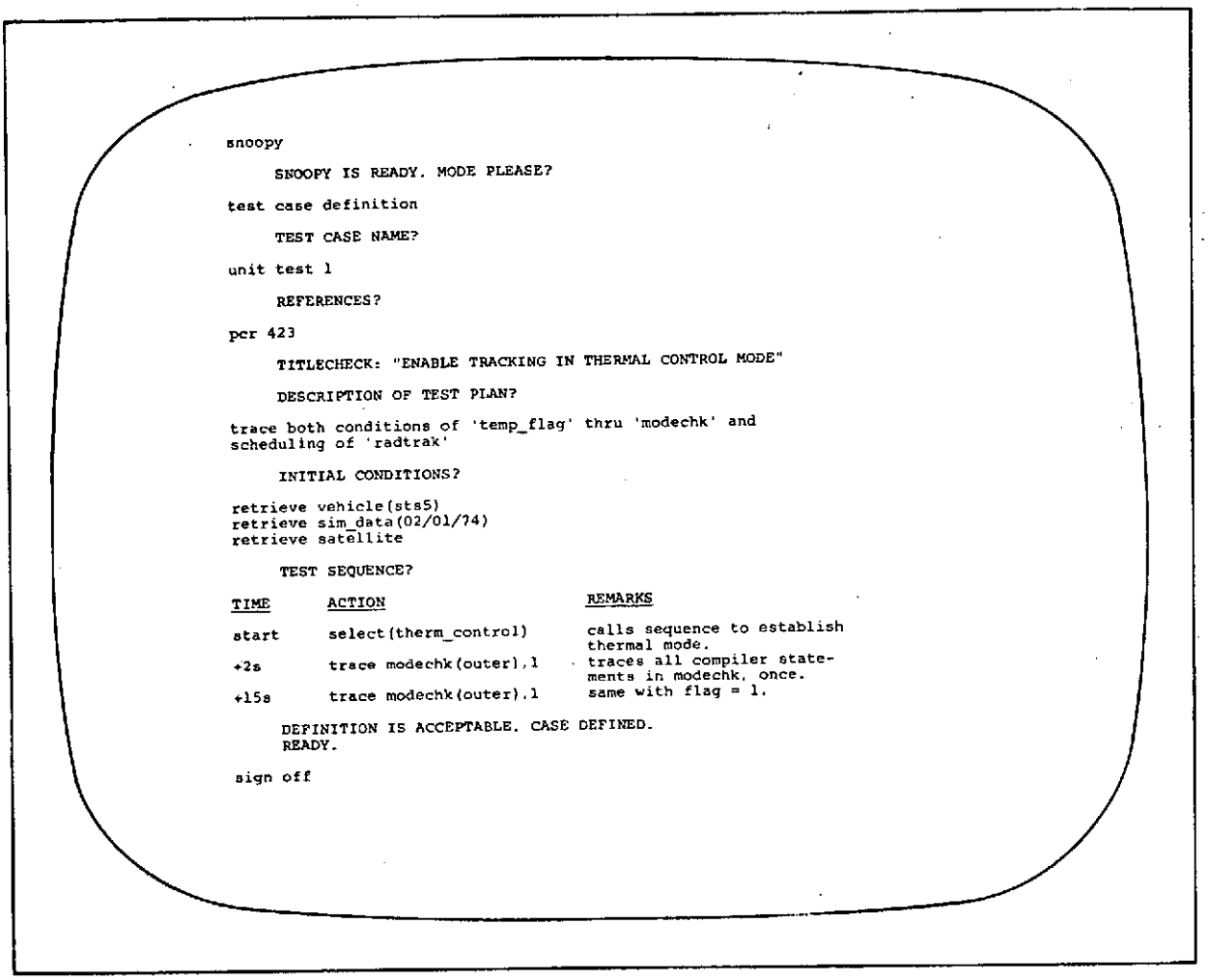
```
snoopy
     SNOOPY IS READY. MODE PLEASE?
scan "skyview",*
     TITLECHECK: PROGRAM......SKYVIEW
                 REVISION.....57,
     DO YOU WANT A FULL LISTING?
no. display 'modechk' to "tempmanvr"
     OK.
     /* SNOOPY BLOCK MODECHK BEGINS HERE */
     469  MODECHK: PROCEDURE(TEMP_FLAG,FIX);
     470       DECLARE FIX ARRAY(6);
     471       FIX = INSTRUMENTS;                                    **
     472       IF TEMP_FLAG = '1' THEN
     473           SCHEDULE RADTRAK AT PRIORITY 42:                    (REV 57, 03/06/76,
                                                                       SYMMES.T., PCR 423)
     474       ELSE;  /* ENABLES SATELLITE TRACKING */              **
     475       CALL TEMPMANVR;
     READY.
sign off
```

Figure 2-3.

The Programmer:

To verify the program change specified by PCR 423, Symmes defined test case UNIT_TEST1 specified in Figure 2-1. Once this test case is defined, it is stored in the Test Plan source file and is retrievable upon request when the test is to be run or reviewed.

In response to SNOOPY's request for INITIAL CONDITIONS, Symmes indicates that predefined data sets will be used for initialization of the simulation. For the TEST SEQUENCE, Symmes indicates that the sequence will begin at the mnemonic time START whose numeric value will be supplied at runtime. At START, the action SELECT will direct the SUPERCREW program to initiate the sequence as prescribed by the data set THERM_CONTROL. At START plus 2 seconds, tracing of only the statements in MODECHK will occur when the code is executed. By START plus 15 seconds, the flag should be equal to 1, so that the other branch can also be traced. (See Figure 2-3.)

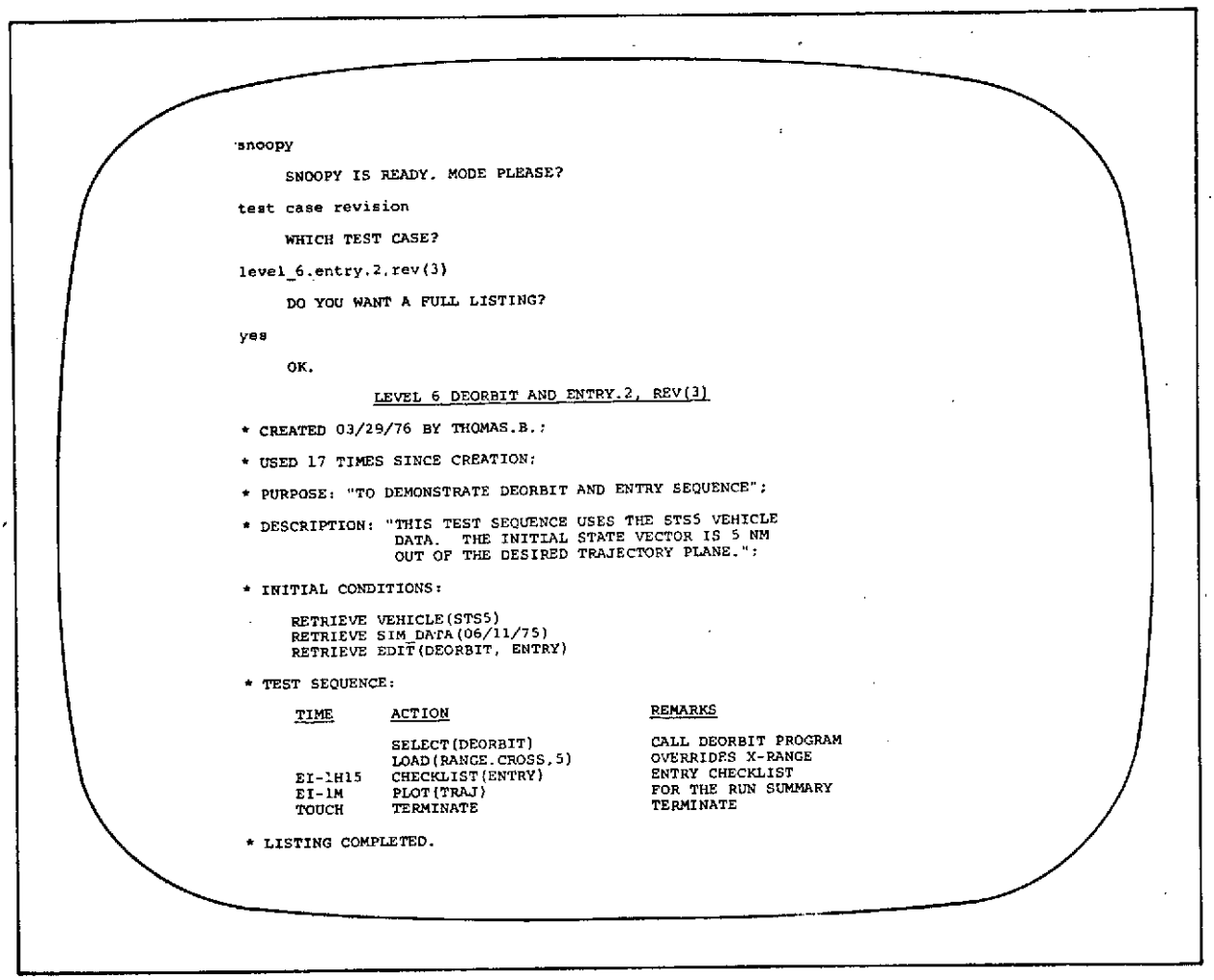


```
snoopy

     SNOOPY IS READY. MODE PLEASE?

test case definition

     TEST CASE NAME?

unit test 1

     REFERENCES?

pcr 423

     TITLECHECK: "ENABLE TRACKING IN THERMAL CONTROL MODE"

     DESCRIPTION OF TEST PLAN?

trace both conditions of 'temp_flag' thru 'modechk' and
scheduling of 'radtrak'

     INITIAL CONDITIONS?

retrieve vehicle(sts5)
retrieve sim_data(02/01/74)
retrieve satellite

     TEST SEQUENCE?

TIME      ACTION                        REMARKS

start     select(therm_control)         calls sequence to establish
                                        thermal mode.
+2s       trace modechk(outer),1        traces all compiler state-
                                        ments in modechk, once.
+15s      trace modechk(outer),1        same with flag = 1.

     DEFINITION IS ACCEPTABLE. CASE DEFINED.
     READY.

sign off
```

Figure 2-4.

The Test Engineer:

After the mission plans have been determined and the flight program has been placed under configuration control, mission performance verification begins. B. Thomas wants to run a test case that she had created on 3/29/76. She requests retrieval of the test case by specifying its title and revision number. SNOOPY loads and displays the page on the interactive terminal, and notes that this data set has been accessed 17 times since its creation. (Figure 2-5 is explained in greater detail in subsection 4.6.)



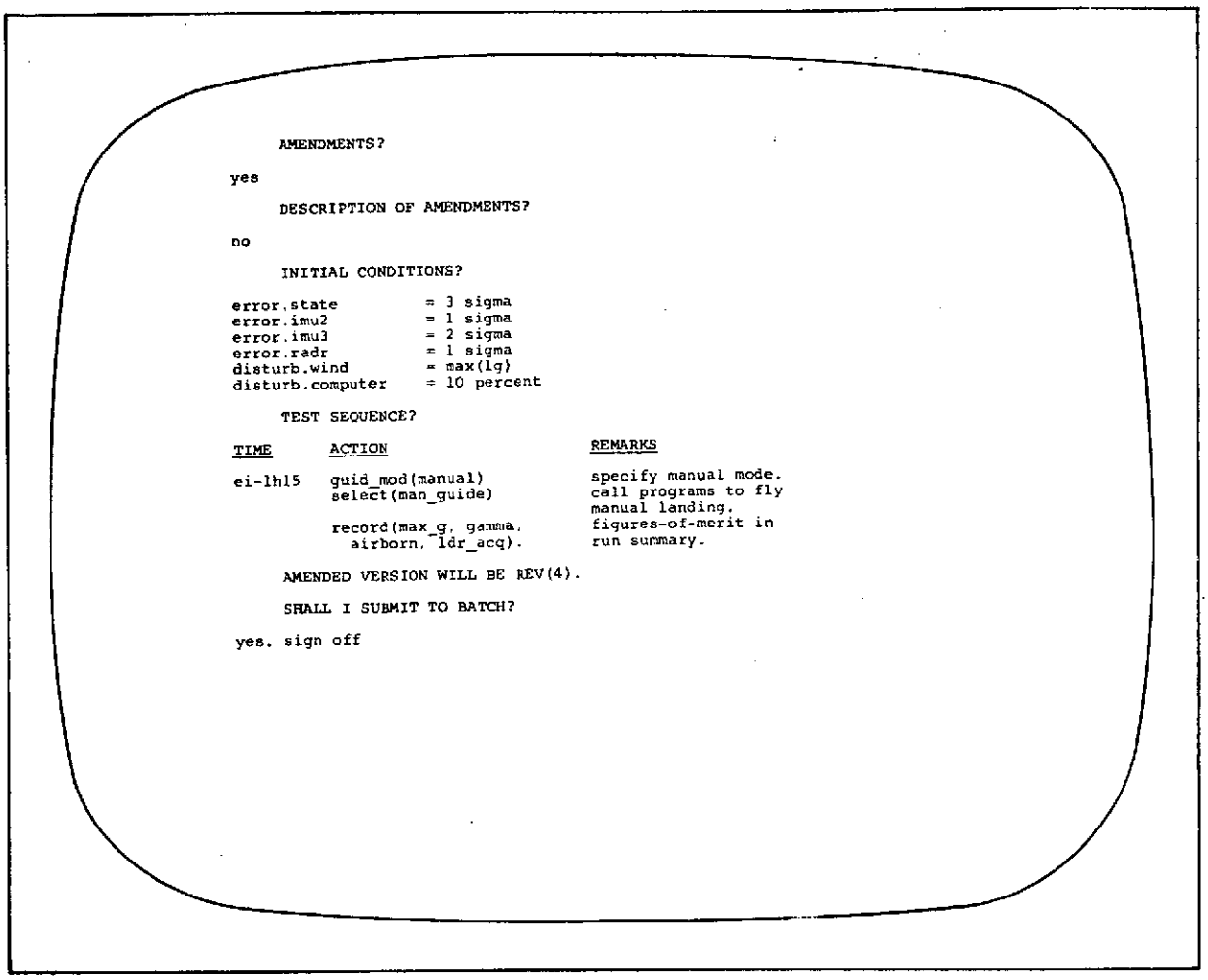
```
snoopy

     SNOOPY IS READY. MODE PLEASE?

test case revision

     WHICH TEST CASE?

level_6.entry.2,rev(3)

     DO YOU WANT A FULL LISTING?

yes

     OK.

              LEVEL 6 DEORBIT AND ENTRY.2, REV(3)

* CREATED 03/29/76 BY THOMAS.B.;

* USED 17 TIMES SINCE CREATION;

* PURPOSE: "TO DEMONSTRATE DEORBIT AND ENTRY SEQUENCE";

* DESCRIPTION: "THIS TEST SEQUENCE USES THE STS5 VEHICLE
                DATA.  THE INITIAL STATE VECTOR IS 5 NM
                OUT OF THE DESIRED TRAJECTORY PLANE.";

* INITIAL CONDITIONS:

    RETRIEVE VEHICLE(STS5)
    RETRIEVE SIM_DATA(06/11/75)
    RETRIEVE EDIT(DEORBIT, ENTRY)

* TEST SEQUENCE:

     TIME       ACTION                      REMARKS

                SELECT(DEORBIT)             CALL DEORBIT PROGRAM
                LOAD(RANGE.CROSS,5)         OVERRIDES X-RANGE
     EI-1H15    CHECKLIST(ENTRY)            ENTRY CHECKLIST
     EI-1M      PLOT(TRAJ)                  FOR THE RUN SUMMARY
     TOUCH      TERMINATE                   TERMINATE

* LISTING COMPLETED.
```

Figure 2-5.

Thomas wishes to amend the initial conditions and test sequence specified in revision 3 of the test case. The initial conditions she desires were set by default to zero in the data set SIM_DATA. She overrides these directly. She then changes the test sequence by instructing the crew to position the GUIDE_MODE switch to MANUAL and by specifying the sequence prescribed in the data set MAN_GUIDE.

The value of the parameters specified in RECORD and the times they occur will be tabulated in the post-run summary. (See Figure 2-7.)

Although the simulation is run in batch mode, the reviewing and specifying of test cases is performed interactively.

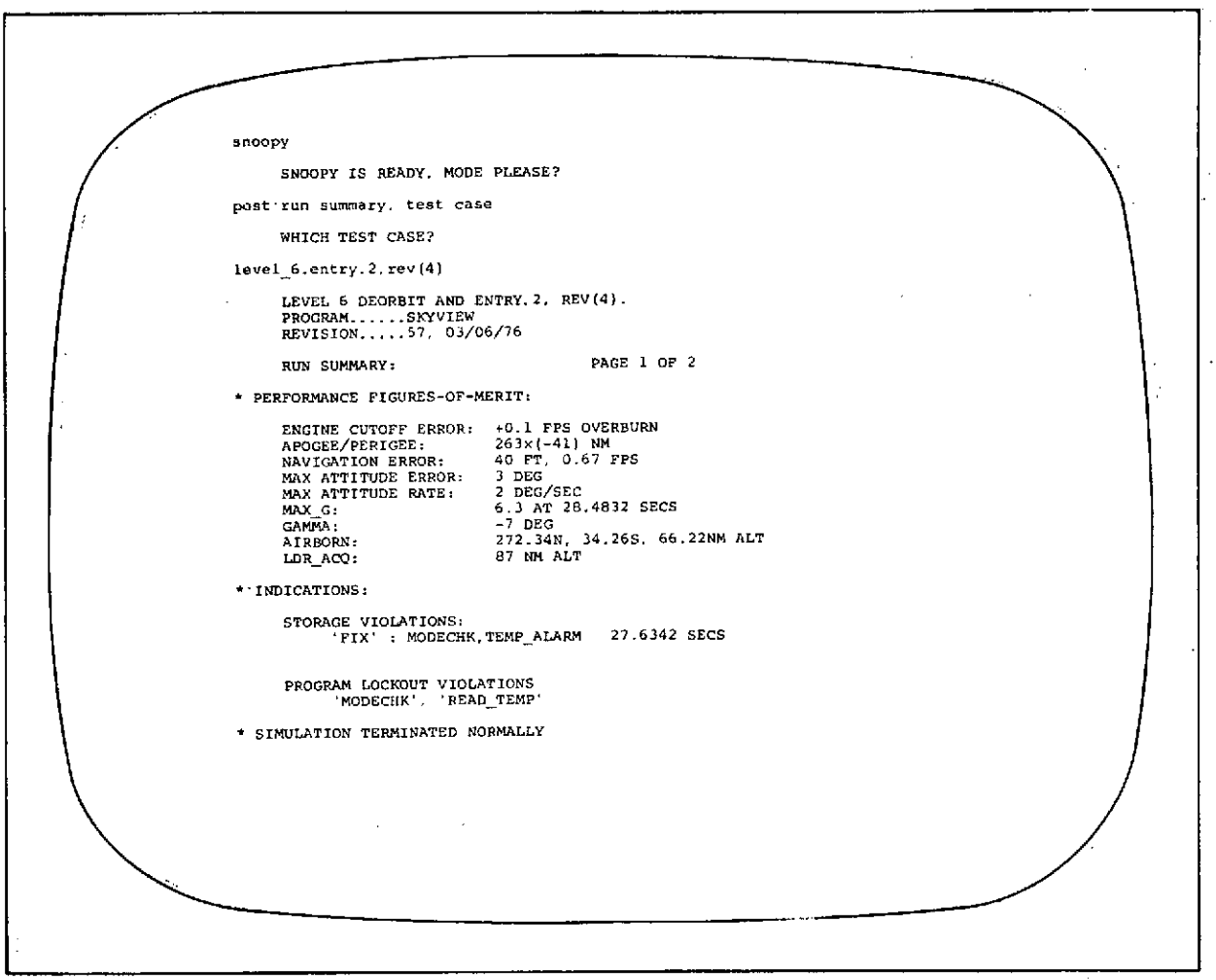


```
     AMENDMENTS?

yes

     DESCRIPTION OF AMENDMENTS?

no

     INITIAL CONDITIONS?

error.state          = 3 sigma
error.imu2           = 1 sigma
error.imu3           = 2 sigma
error.radr           = 1 sigma
disturb.wind         = max(lg)
disturb.computer     = 10 percent

     TEST SEQUENCE?

TIME       ACTION                        REMARKS

ei-1h15    guid_mod(manual)              specify manual mode.
           select(man_guide)             call programs to fly
                                         manual landing.
           record(max_g, gamma,          figures-of-merit in
             airborn, ldr_acq).          run summary.

     AMENDED VERSION WILL BE REV(4).

     SHALL I SUBMIT TO BATCH?

yes. sign off
```

Figure 2-6.

The Test Engineer:

When Thomas is notified that her run is completed, she reviews the post-run summary to determine if the run terminated normally and if error conditions existed during the run. She also examines the figures-of-merit recorded.



```
snoopy

     SNOOPY IS READY, MODE PLEASE?

post run summary, test case

     WHICH TEST CASE?

level_6.entry.2,rev(4)

     LEVEL 6 DEORBIT AND ENTRY.2, REV(4).
     PROGRAM.....SKYVIEW
     REVISION.....57, 03/06/76

     RUN SUMMARY:                          PAGE 1 OF 2

* PERFORMANCE FIGURES-OF-MERIT:

     ENGINE CUTOFF ERROR:   +0.1 FPS OVERBURN
     APOGEE/PERIGEE:        263x(-41) NM
     NAVIGATION ERROR:      40 FT, 0.67 FPS
     MAX ATTITUDE ERROR:    3 DEG
     MAX ATTITUDE RATE:     2 DEG/SEC
     MAX_G:                 6.3 AT 28.4832 SECS
     GAMMA:                 -7 DEG
     AIRBORN:               272.34N, 34.26S, 66.22NM ALT
     LDR_ACQ:               87 NM ALT

* INDICATIONS:

     STORAGE VIOLATIONS:
          'FIX' : MODECHK,TEMP_ALARM   27.6342 SECS


     PROGRAM LOCKOUT VIOLATIONS
          'MODECHK', 'READ_TEMP'

* SIMULATION TERMINATED NORMALLY
```

Figure 2-7.

Page 2 of the post-run summary shows the entry trajectory plot as requested in the test plan.

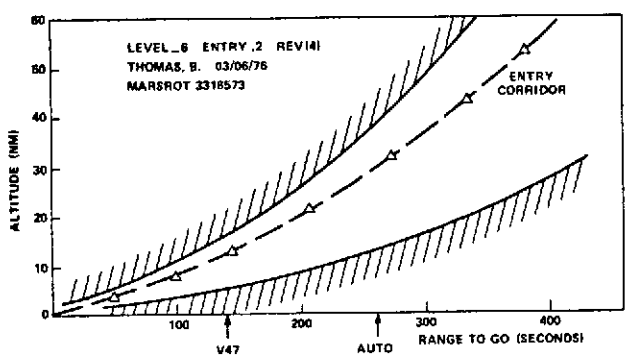
Thomas then makes further requests for:

the output of the PANSIM edit;
the simulation results to be plotted and edited;
the placing of these test results in the Test Reports source file.

LEVEL 6 DEORBIT AND ENTRY.2, REV(4).
PROGRAM.....SKYVIEW
REVISION.....57, 03/06/76

RUN SUMMARY : PAGE 2 OF 2

* REQUESTED PLOTS:



* RUN SUMMARY COMPLETED AS INDICATED SO FAR.
OTHER REQUESTS?

print, plot and edit pansim output
test results to be finalized for report

1. REQUESTS WILL BE SUBMITTED IN BATCH MODE. HARDCOPY WILL BE DELIVERED.
2. SNOOPY NOW SWITCHING TO 'TEST REPORT' MODE. (TOTAL TEST RESULTS OF THE ABOVE RUN HAVE BEEN TEMPORARILY ASSIGNED THE NAME: "RESULTS".)

Figure 2-8.

When SNOOPY has gone to the "update test-report" mode, Thomas requests that the entire post-run summary with edited and plotted data be placed in the Test Reports source file under LEVEL 6, ENTRY.2. She also requests that completion of this test case be recorded in the development plan. (See Figure 2-10.)

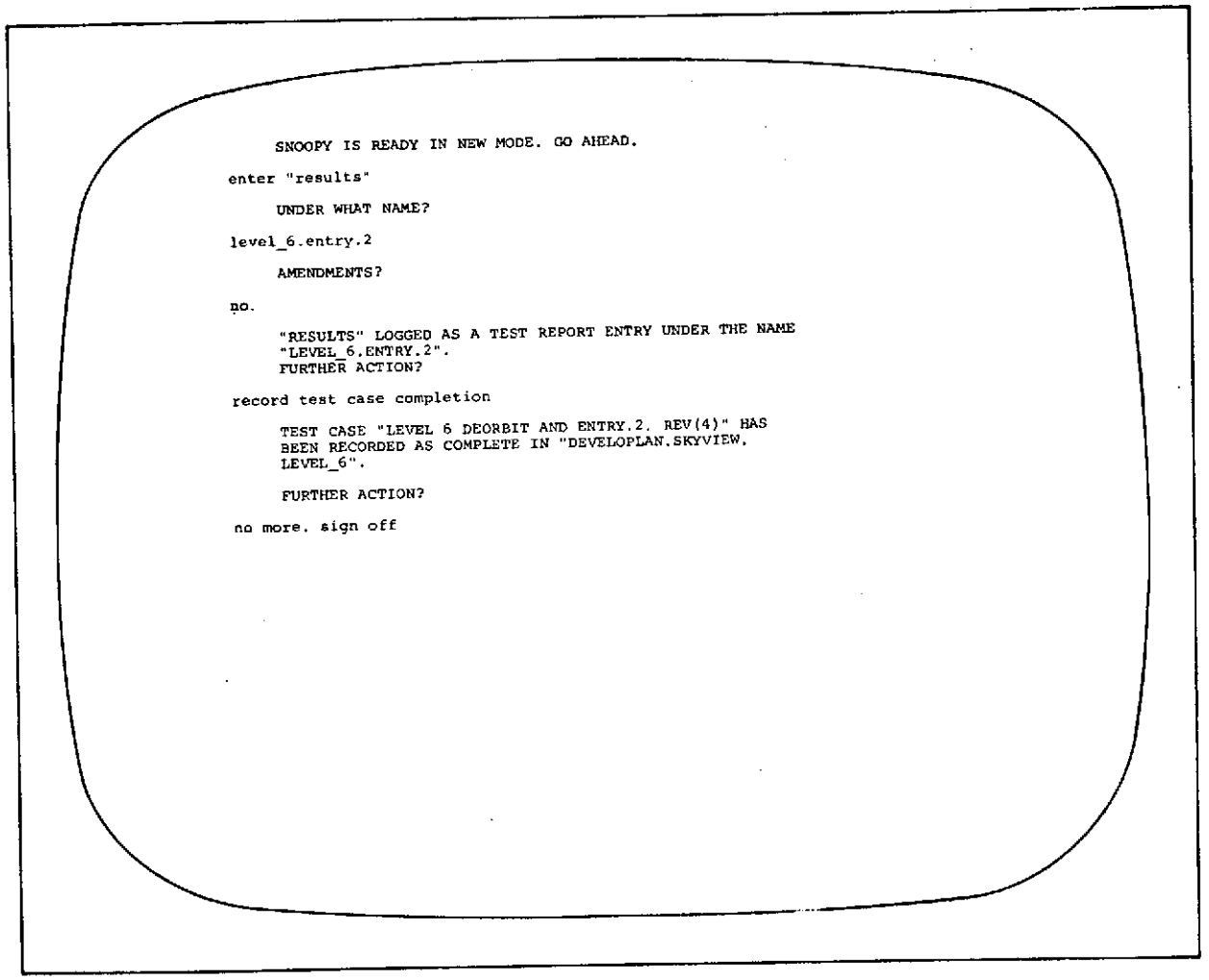
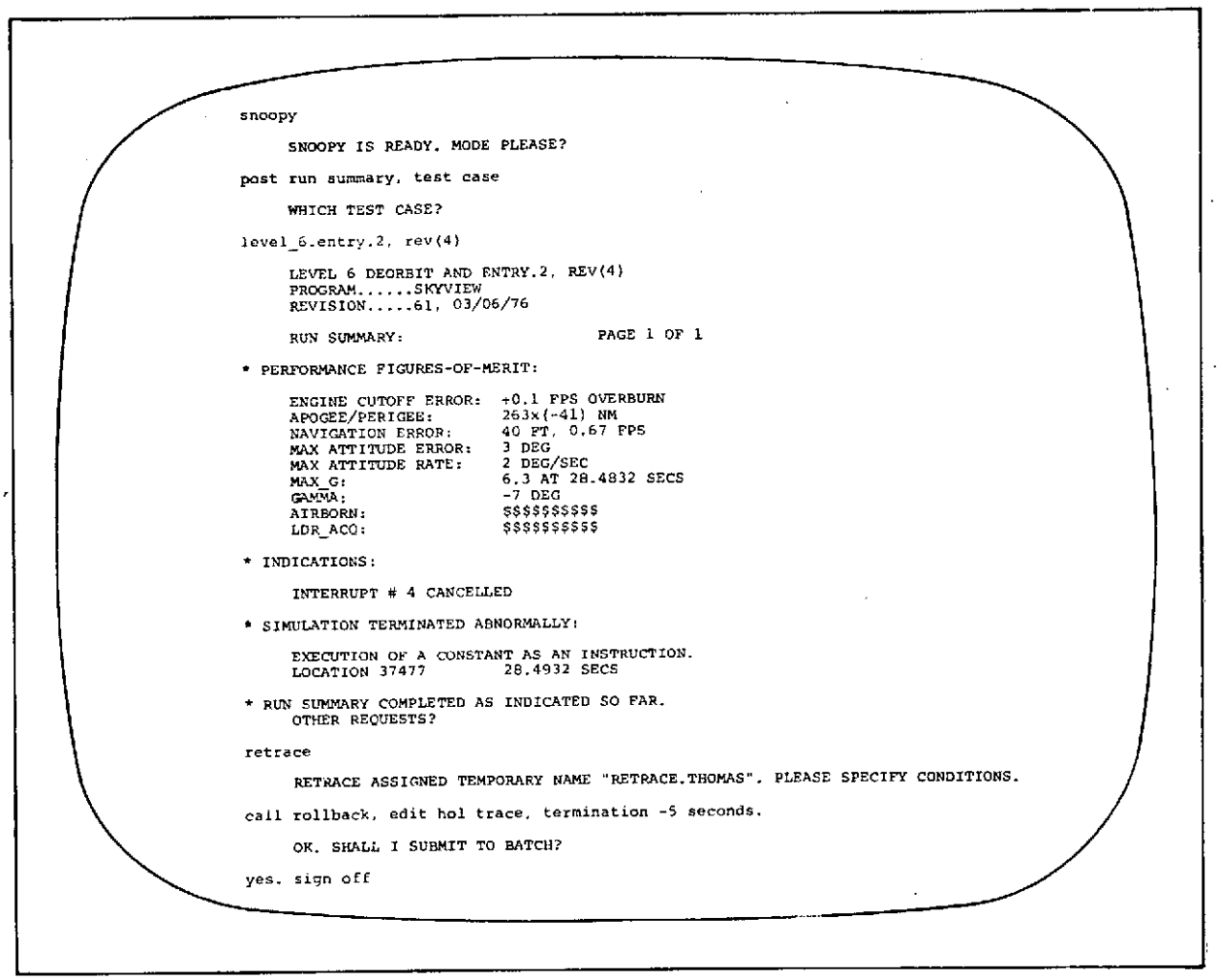

SNOOPY IS READY IN NEW MODE. GO AHEAD.
enter "results"
UNDER WHAT NAME?
level_6.entry.2
AMENDMENTS?
no.
"RESULTS" LOGGED AS A TEST REPORT ENTRY UNDER THE NAME "LEVEL_6.ENTRY.2".
FURTHER ACTION?
record test case completion
TEST CASE "LEVEL 6 DEORBIT AND ENTRY.2. REV(4)" HAS BEEN RECORDED AS COMPLETE IN "DEVELOPLAN.SKYVIEW.LEVEL_6".
FURTHER ACTION?
no more. sign off

Figure 2-9.

The Test Engineer:

Thomas ran this same test case on a later revision (61) of SKYVIEW; the results are shown in Figure 2-11. The run terminated abnormally. The Instruction Simulator aborted the run upon attempting to execute a constant. Note that the events simulating airborn conditions and acquiring radar lock did not occur. Thomas calls for a retrace of the last few seconds of the simulation so that it can be edited for debugging purposes.

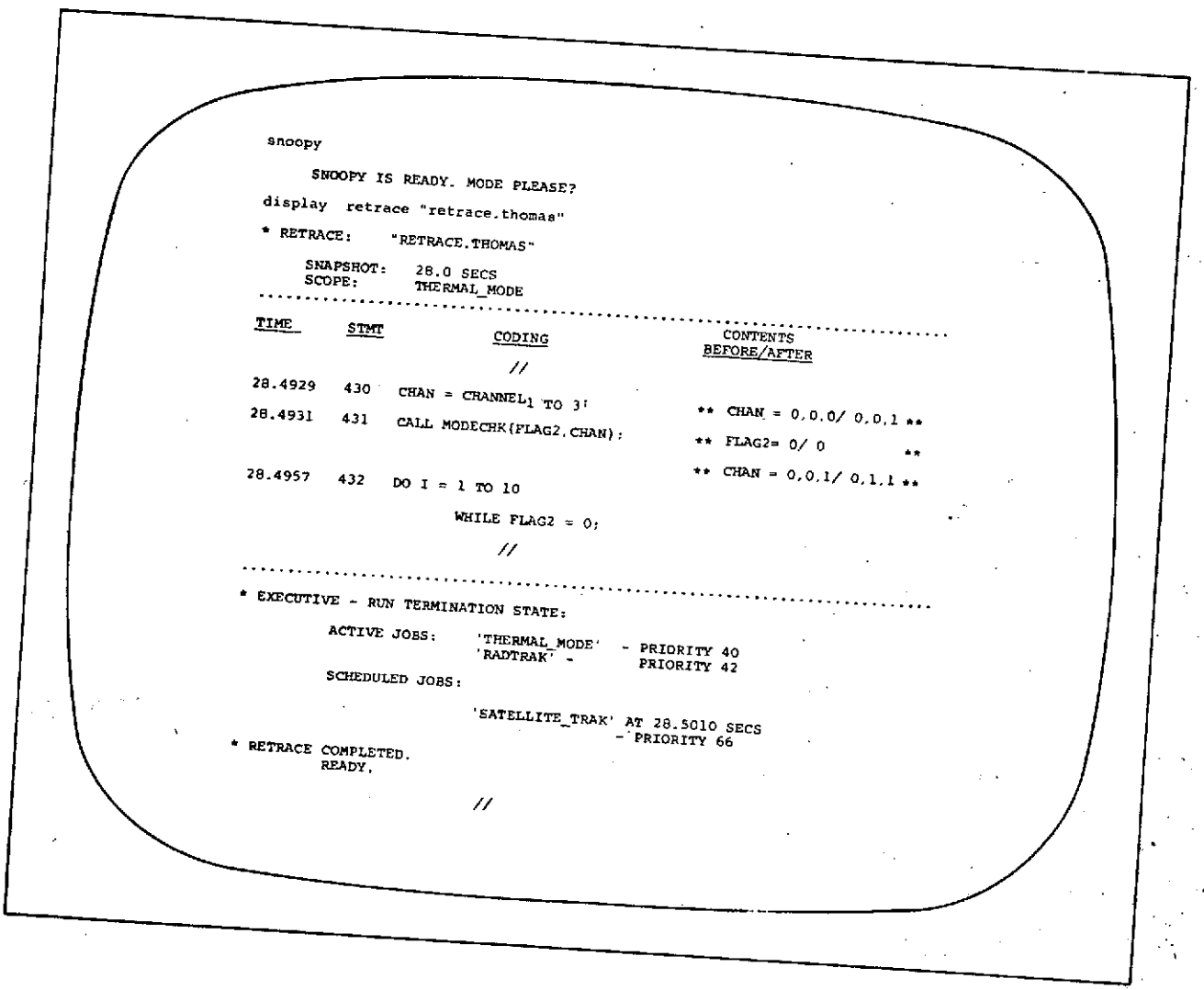


```
snoopy

     SNOOPY IS READY, MODE PLEASE?

post run summary, test case

     WHICH TEST CASE?

level_6.entry.2, rev(4)

     LEVEL 6 DEORBIT AND ENTRY.2, REV(4)
     PROGRAM.....SKYVIEW
     REVISION.....61, 03/06/76

     RUN SUMMARY:                          PAGE 1 OF 1

* PERFORMANCE FIGURES-OF-MERIT:

     ENGINE CUTOFF ERROR:  +0.1 FPS OVERBURN
     APOGEE/PERIGEE:       263x(-41) NM
     NAVIGATION ERROR:     40 FT, 0.67 FPS
     MAX ATTITUDE ERROR:   3 DEG
     MAX ATTITUDE RATE:    2 DEG/SEC
     MAX_G:                6.3 AT 28.4832 SECS
     GAMMA:                -7 DEG
     AIRBORN:              $$$$$$$$$$
     LDR_ACQ:              $$$$$$$$$$

* INDICATIONS:

     INTERRUPT # 4 CANCELLED

* SIMULATION TERMINATED ABNORMALLY:

     EXECUTION OF A CONSTANT AS AN INSTRUCTION.
     LOCATION 37477        28.4932 SECS

* RUN SUMMARY COMPLETED AS INDICATED SO FAR.
     OTHER REQUESTS?

retrace

     RETRACE ASSIGNED TEMPORARY NAME "RETRACE.THOMAS", PLEASE SPECIFY CONDITIONS.

call rollback, edit hol trace, termination -5 seconds.

     OK, SHALL I SUBMIT TO BATCH?

yes, sign off
```

Figure 2-10.

The Test Engineer:

The retrace created a rollback which ran in the batch mode, as any other simulation, using the diagnostic package. The constant was executed during the routine THERMAL_MODE and the tracing of statements of this routine is indicated. Note that the display indicates the time for each statement and the contents of the variables in the statement. An Executive edit is displayed to indicate the activity at the time of the abort. With the information provided here, Thomas can specify further diagnosis to determine the software anomaly.



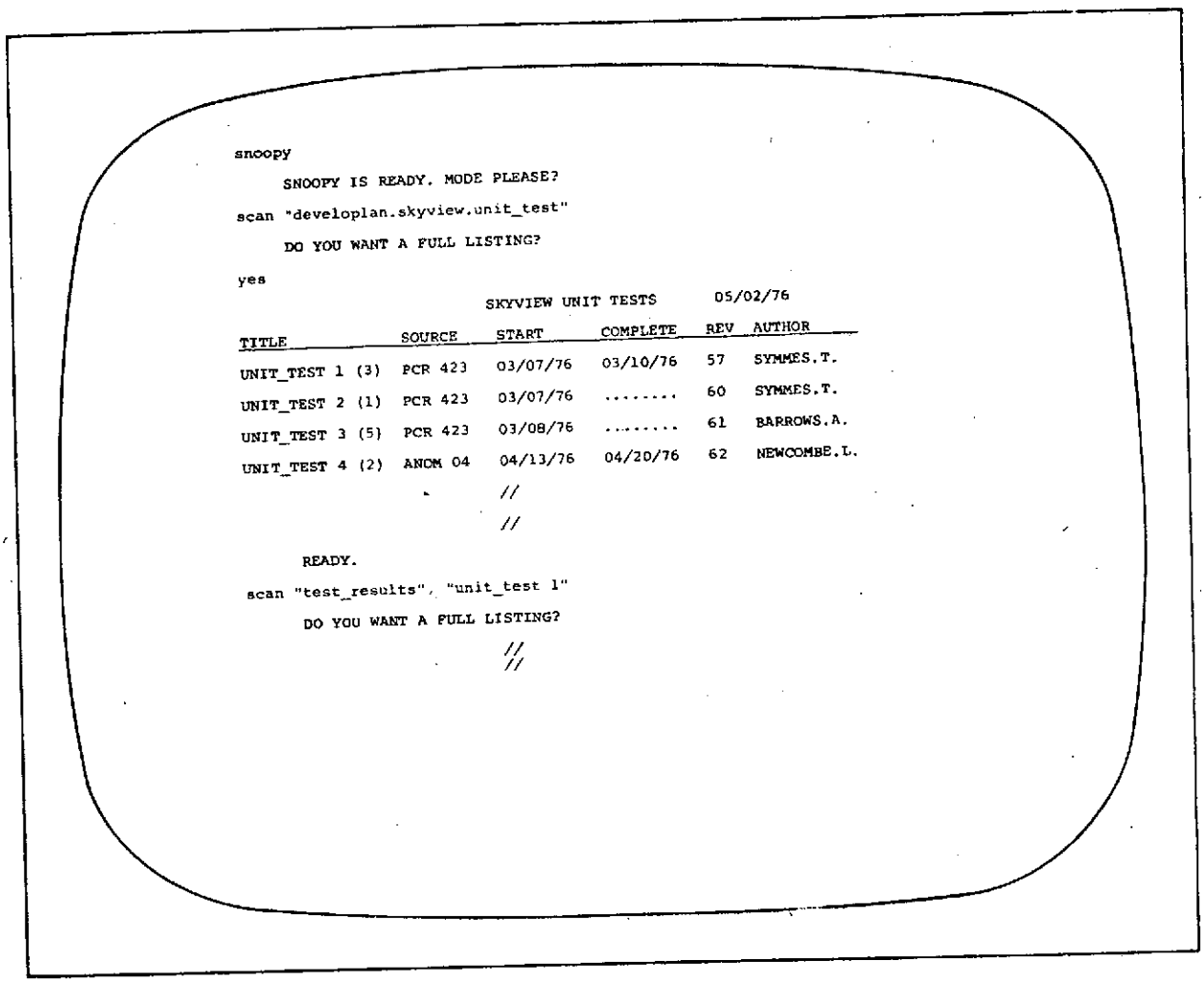
```
snoopy

     SNOOPY IS READY. MODE PLEASE?
display  retrace "retrace.thomas"
* RETRACE:     "RETRACE.THOMAS"

     SNAPSHOT:   28.0 SECS
     SCOPE:      THERMAL_MODE
-------------------------------------------------------------------------
TIME      STMT          CODING                    CONTENTS
                                                  BEFORE/AFTER
                        //
28.4929   430   CHAN = CHANNEL, 1 TO 3;
28.4931   431   CALL MODECHK(FLAG2,CHAN);         ** CHAN = 0,0,0/ 0,0,1 **
                                                  ** FLAG2= 0/ 0          **
28.4957   432   DO I = 1 TO 10                    ** CHAN = 0,0,1/ 0,1,1 **
                    WHILE FLAG2 = 0;
                        //
-------------------------------------------------------------------------
* EXECUTIVE - RUN TERMINATION STATE:
        ACTIVE JOBS:      'THERMAL_MODE'  - PRIORITY 40
                          'RADTRAK'       - PRIORITY 42
        SCHEDULED JOBS:
                          'SATELLITE_TRAK' AT 28.5010 SECS
                                          - PRIORITY 66
* RETRACE COMPLETED.
        READY.
                        //
```

Figure 2-11.

The Program Control Supervisor:

The supervisor can keep abreast of the coding implementation and its verification by reviewing pertinent information stored in the SWDVS data base. This data base is updated by SNOOPY when changes to a flight program are made or when test cases are created or run.

In Figure 2-12, the supervisor asks for the status of unit verifications of all program changes by requesting a display of this development plan. The first entries indicate that one of three tests on PCR 423 have been completed. The supervisor requests more information on UNIT_TEST1 by asking for a display of the test results for this test case.



```
snoopy
     SNOOPY IS READY. MODE PLEASE?
scan "developlan.skyview.unit_test"
     DO YOU WANT A FULL LISTING?
yes
                              SKYVIEW UNIT TESTS        05/02/76

TITLE              SOURCE     START       COMPLETE     REV   AUTHOR
UNIT_TEST 1 (3)    PCR 423    03/07/76    03/10/76     57    SYMMES.T.
UNIT_TEST 2 (1)    PCR 423    03/07/76    ........     60    SYMMES.T.
UNIT_TEST 3 (5)    PCR 423    03/08/76    ........     61    BARROWS.A.
UNIT_TEST 4 (2)    ANOM 04    04/13/76    04/20/76     62    NEWCOMBE.L.
                              //
                              //
      READY.
scan "test_results", "unit_test 1"
      DO YOU WANT A FULL LISTING?
                              //
                              //
```

Figure 2-12

The Program Control Supervisor:

It was found in the Level 6 test results in Figure 2-11 that the scheduling of RADTRAK caused Executive problems; the Software Control Board decided to remove the program change PCR 423. Since the PCR affected other program areas it is necessary for the supervisor to trace through the program revisions for those affected areas.

The supervisor asks SNOOPY to search through the program and finds that added coding was charged to PCR 423 in revisions 57 and 60.

The memo published from the data base for revision 57 is displayed and reviewed by the PCS. All statements affected by the PCR can now be reviewed for revision or deletion.

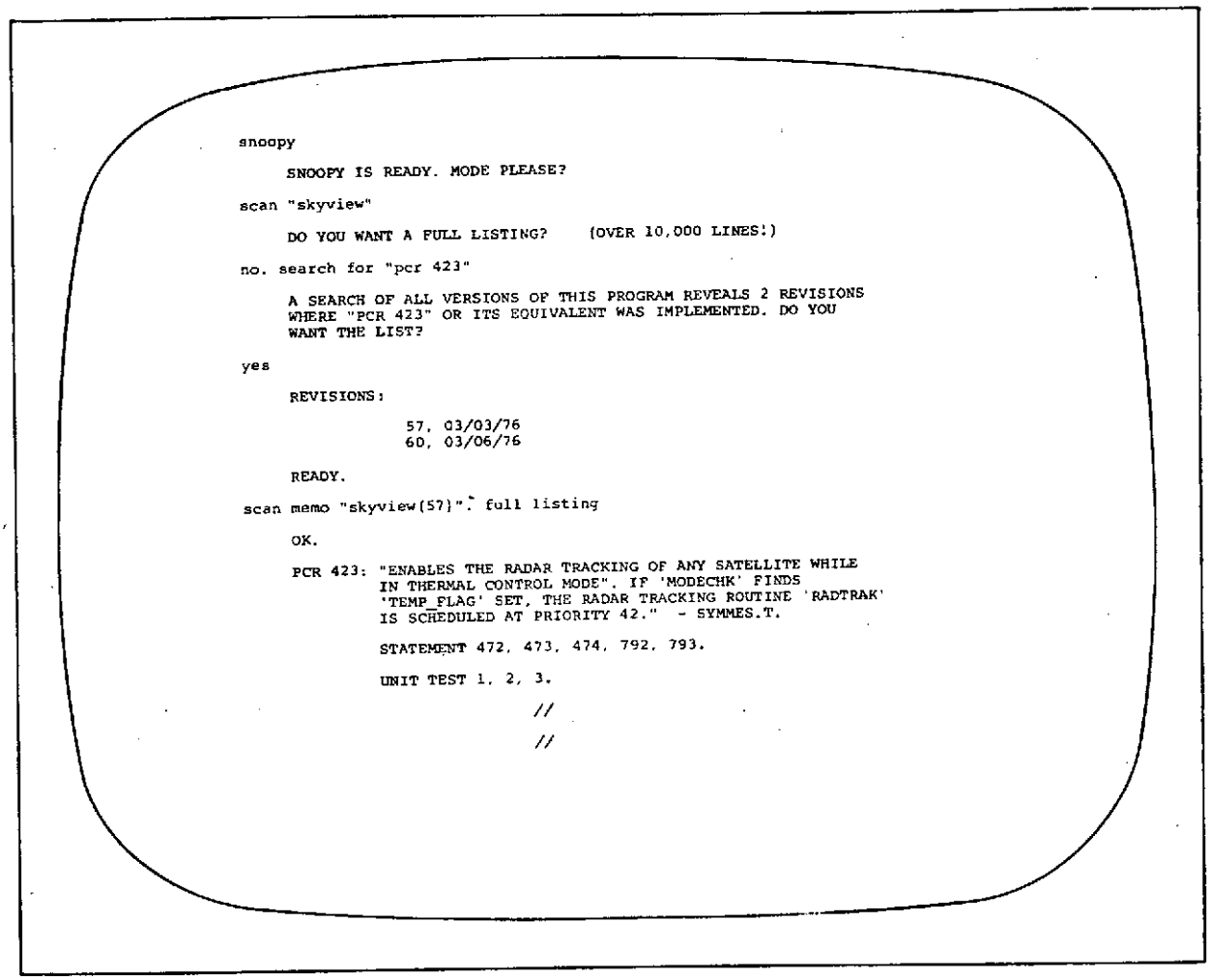

snoopy
SNOOPY IS READY. MODE PLEASE?
scan "skyview"
DO YOU WANT A FULL LISTING? (OVER 10,000 LINES!)
no. search for "pcr 423"
A SEARCH OF ALL VERSIONS OF THIS PROGRAM REVEALS 2 REVISIONS WHERE "PCR 423" OR ITS EQUIVALENT WAS IMPLEMENTED. DO YOU WANT THE LIST?
yes
REVISIONS:
57, 03/03/76
60, 03/06/76
READY.
scan memo "skyview(57)"; full listing
OK.
PCR 423: "ENABLES THE RADAR TRACKING OF ANY SATELLITE WHILE IN THERMAL CONTROL MODE". IF 'MODECHK' FINDS 'TEMP_FLAG' SET, THE RADAR TRACKING ROUTINE 'RADTRAK' IS SCHEDULED AT PRIORITY 42." - SYMMES.T.
STATEMENT 472, 473, 474, 792, 793.
UNIT TEST 1, 2, 3.
//
//

Figure 2-13.

SECTION 3

FUNCTIONAL APPROACHES TO SWDVS

3. FUNCTIONAL APPROACHES TO SWDVS

This section describes the SWDVS in a functional manner from the perspective of the different groups who use it in their work, and also from the perspective of those groups at other facilities who interact with it. The process of SPACE SHUTTLE software development is divided into three phases; the design phase, the implementation phase, and the system-verification phase. As shown in Figure 3-1, the software designers generate program requirements and formulations during the design phase that become the starting point for the implementation phase. In the implementation phase, analysts and systems designers produce a program layout and program-core software, such as the executive routines. Applications and systems programmers next generate individual program modules and their interfaces, and verify them at the individual module level. The modules are then combined to form the complete flight program, which is subject to a thorough testing process by software test engineers. All the functions performed during the implementation phase are accomplished with the SWDVS; the output of this phase is SWDVS-verified software that is passed to the system verification engineers for further testing. The output, in turn, of the system verification phase is flight-rated software ready to support a SPACE SHUTTLE mission.

Thus, it can be seen that each of the groups defined in Figure 3-1 has SPACE SHUTTLE software responsibilities to which a successful SWDVS design must respond. The SWDVS, however, is not intended to be all things to all people; it must be optimized to most efficiently carry out its primary functions of flight software development and verification. This section shows how the resulting SWDVS design meets the requirements of each group, and discusses possible problems that are prevented by the design.

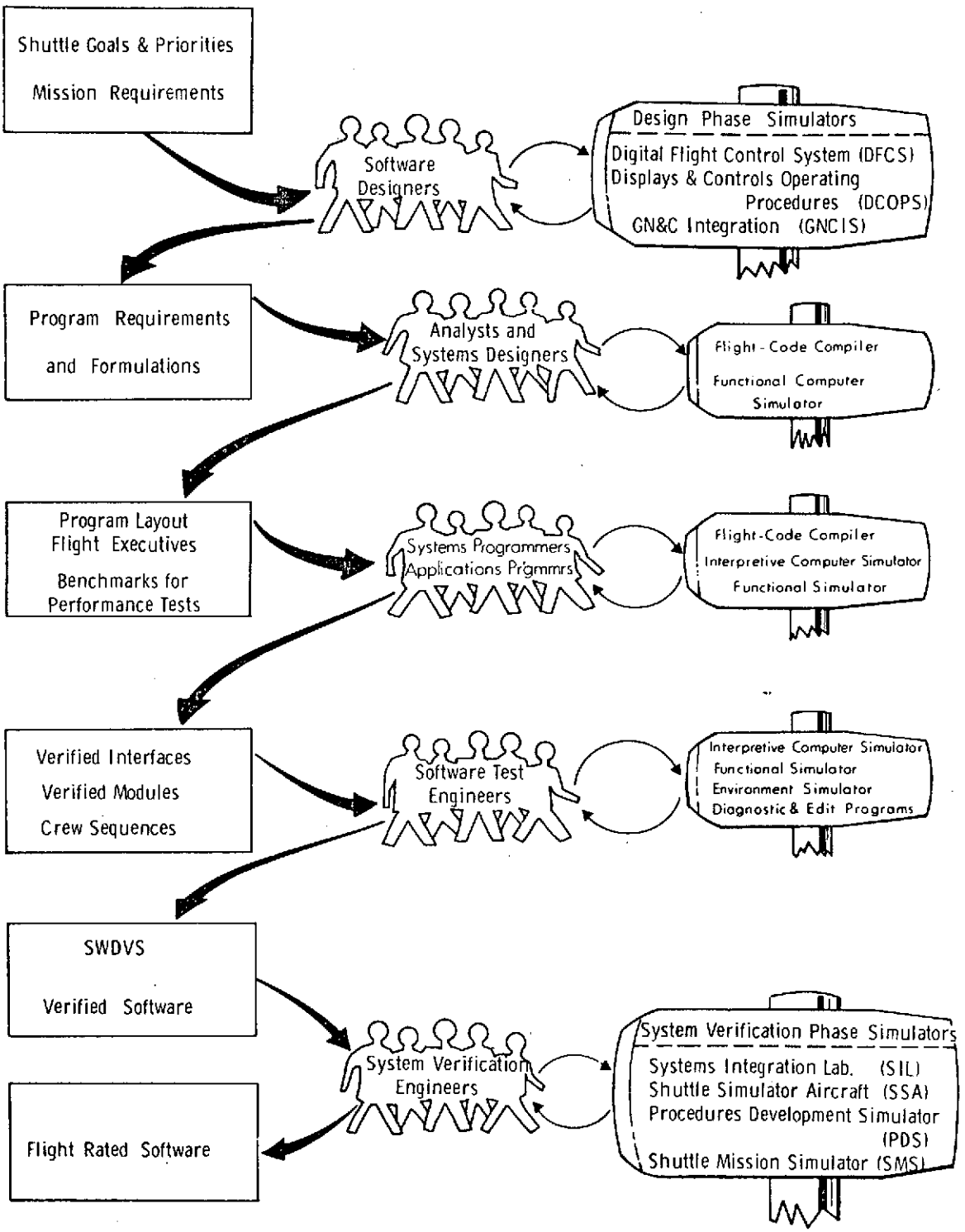


Figure 3-1. Software Development Process

3.1 SWDVS INTERACTION WITH THE DESIGN-PHASE SIMULATORS

The work performed with the SWDVS during the implementation phase of the overall flight software development process is directly dependent on the output of the preceding design phase. The SWDVS accepts software requirements and formulations from the design phase analysts and uses them to develop a program layout for the flight computers. A primary goal of the SWDVS design is, therefore, to facilitate the transfer of information between design-phase and implementation-phase personnel.

An important method of achieving this goal is to provide commonality between the design-phase and implementation-phase software tools, particularly in the simulator sensor models area. For example, the SWDVS can incorporate the math models of the navigation aids used by the Systems Development Simulation facilities into its External-environment Simulator (ESIM) model library. This is advantageous in that it allows transfer of experience gained in generating models for the design-phase simulators to the SWDVS, and prevents needless duplication of effort between the two facilities. It can also contribute to the verification process of the SWDVS, itself, by allowing comparison of SWDVS simulation results with benchmark tests obtained from the design-phase simulators. The capabilities to transfer experience and to supply benchmark runs should aid in reducing SWDVS development costs.

Commonality between the design phase and SWDVS simulators also supports the goal of ultimately replacing SWDVS contractor personnel with NASA personnel.

3.2 SOFTWARE DEVELOPMENT IN THE IMPLEMENTATION PHASE

To clearly illustrate the functions of the SWDVS for the reader, the implementation phase of flight-software development is defined to consist of three basic tasks: flight-software design, flight-software verification, and supervisory control of flight-software development. The following functions are provided by these tasks:

Mathematical and Engineering Design and Analysis— investigating, developing and analyzing NASA-approved software requirements and formulations generated during the design phase of the software development process. (Refer to Figure 3-1.) This function determines, for example, the required data, its sample rate, and its manipulation to accomplish the requirements. (Also, this function could provide Functional-computer Simulator (FSIM) programs to be used as driver models by other analysts and programmers.)

Systems Design and Analysis—laying out the software structure required to meet the requirements of the design phase. This function specifies, for example, I/O processing methods, Executive structure, program-module interfaces, methods for including fault tolerance, and error-recovery techniques.

Systems Programming and Verification—coding and verifying routines specified by systems analysts. Routines are coded into FSIM programs and the actual flight program. This function also includes the segmenting of program functions into modules and establishing the appropriate priorities and timing relationships for the program modules.

Applications Programming and Verification—coding the program modules defined by the system programmers. The integrated program modules and interfaces are verified at different levels of fidelity and detail.

Supervisory Control—establishing and maintaining necessary organization and control procedures to deliver reliable flight software on time.

3.2.1 Flight-software Design

The following aspects of software design require close consideration when choosing a functional tool for developing software.

Familiarization—The new programmer and analyst must begin using the SWDVS as soon as possible. The process of learning the programming language must, therefore, be simple to speed up the early development.

Modularity—To avoid conflicts with other coding, program modules and storage cells must be protected. Further, the use of program modules, constants, and storage cells must be consistent.

Off-line Development—Programmers must be able to develop their program modules without affecting the main-line program.

Programming Aids—The programmers require such information about the program as current structure and the definition of variables and events.

Documentation—Programmers require assistance in recollection of code intent; an orderly method of documenting the program must be available.

3.2.1.1 The Flight-code Compiler

The compiler's high-order language best satisfies the first two considerations because it is easy to understand and it provides readability for ease in visually inspecting (eyeballing) the written code. Further, the compiler's structure provides for modularity and enforces an orderly interface between program modules. The common pool (COMPOOL) source file is maintained by the compiler to ensure consistency in use of constants by analysts and programmers.

The compiler provides many features not available in an assembler, including a source listing that is easily scanned for errors and a detailed set of static and dynamic diagnostics. The static self-diagnostics ensure that the submitted code conforms to the syntax and structure constraints. It can also eliminate certain software errors common in lower level languages; for example, incorrect branching, and errors in addressing.

3.2.1.2 Off-line Development

With an off-line version of a program, the programmer and analyst can investigate and develop new programming techniques in a working copy of the program while the main-line programming effort continues unaffected.

The File List Processing Service (LIPSVC) program illustrated in Figure 3-2 retains program revisions in a descendant, family-tree fashion; LIPSVC makes the preservation capability practical by preserving, in the source language, the changes for each program revision. For both off-line and main-line program development, the compiler listing indicates which statements were added, deleted, or changed, along with the date the compiled revision was made, and the specification source from which the change was derived. A lead-in for each program module gives pertinent information including author, latest modification date, I/O, and other program modules invoked. (See the LIPSVC description in paragraph 4.7.1.)

3.2.1.3 Source-code Information Retrieval

The compiler can be accessed via the Interactive Display System as well as via the batch mode. Thus, information about a compilation can be interactively reviewed and corrected, if necessary, without printing the source listing.

When adding or changing code to the flight program, the programmer specifies his name and the specification reference. The compile-time diagnostics check for legality and update the data-retrieval file with the latest history.

Storage and retrieval of source code information is accomplished by the data base management system, SNOOPY (Figures 3-3 and 3-4).

3.2.1.4 Programming Aids

The compiler generates reference maps for each revision; these maps relate information about a program's structure, its variables, and constants. The structure reference map shows how program modules relate to each other; that is, the program modules called and on what basis—for example, time, priority—they are called. The variables and constants reference map is generated from the COMPOOL and contains parameters associated with each variable and constant. For each parameter such information as scaling, precision, and which program modules use the parameter is listed.

3.2.1.5 Documentation

The programmer provides a description of the program change and a specification source when the coding is submitted for compilation. This description and revision-related data are stored in a documentation data base in a memorandum format. Memos can be published with this data, and information can be retrieved by specifying

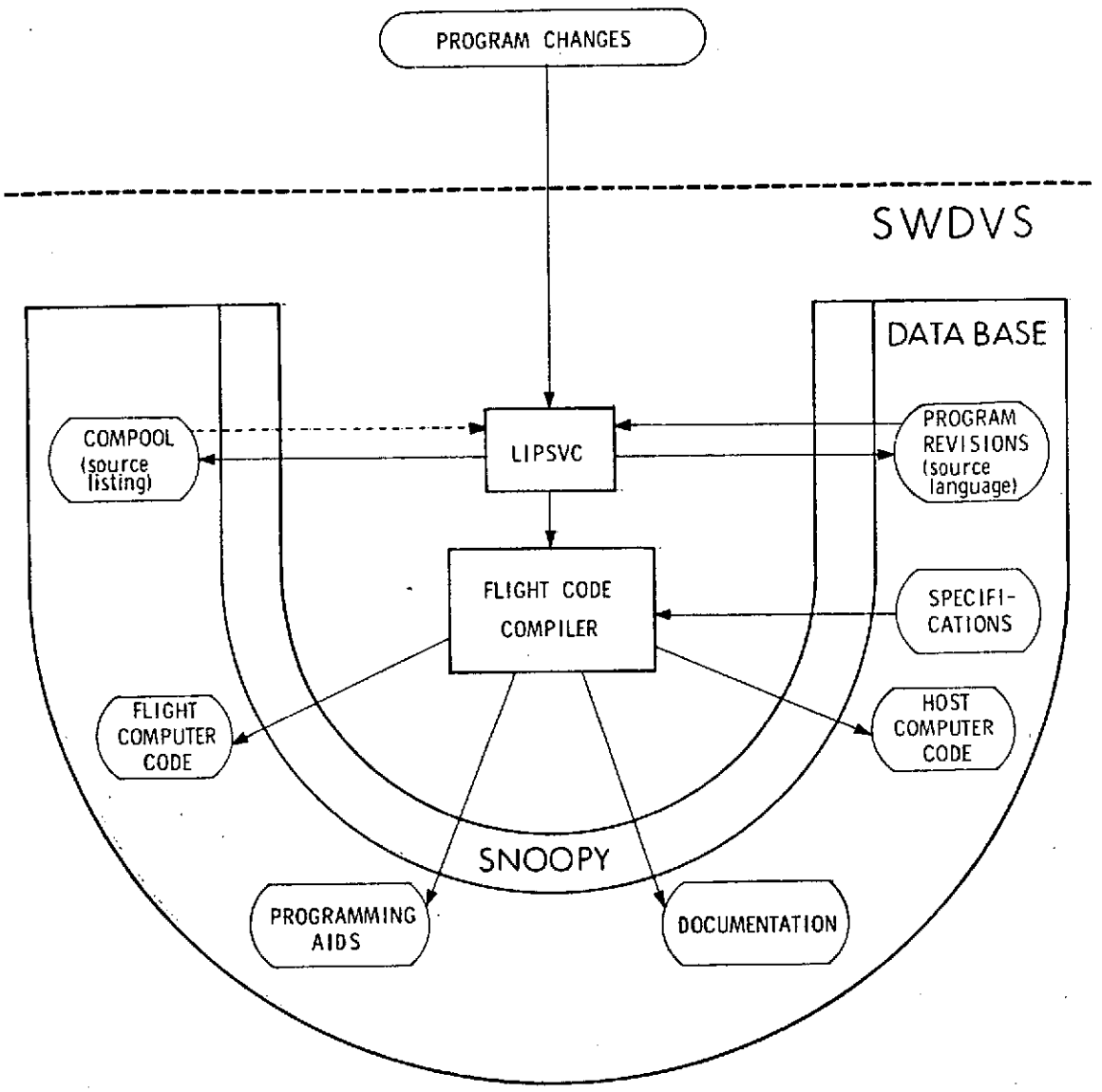
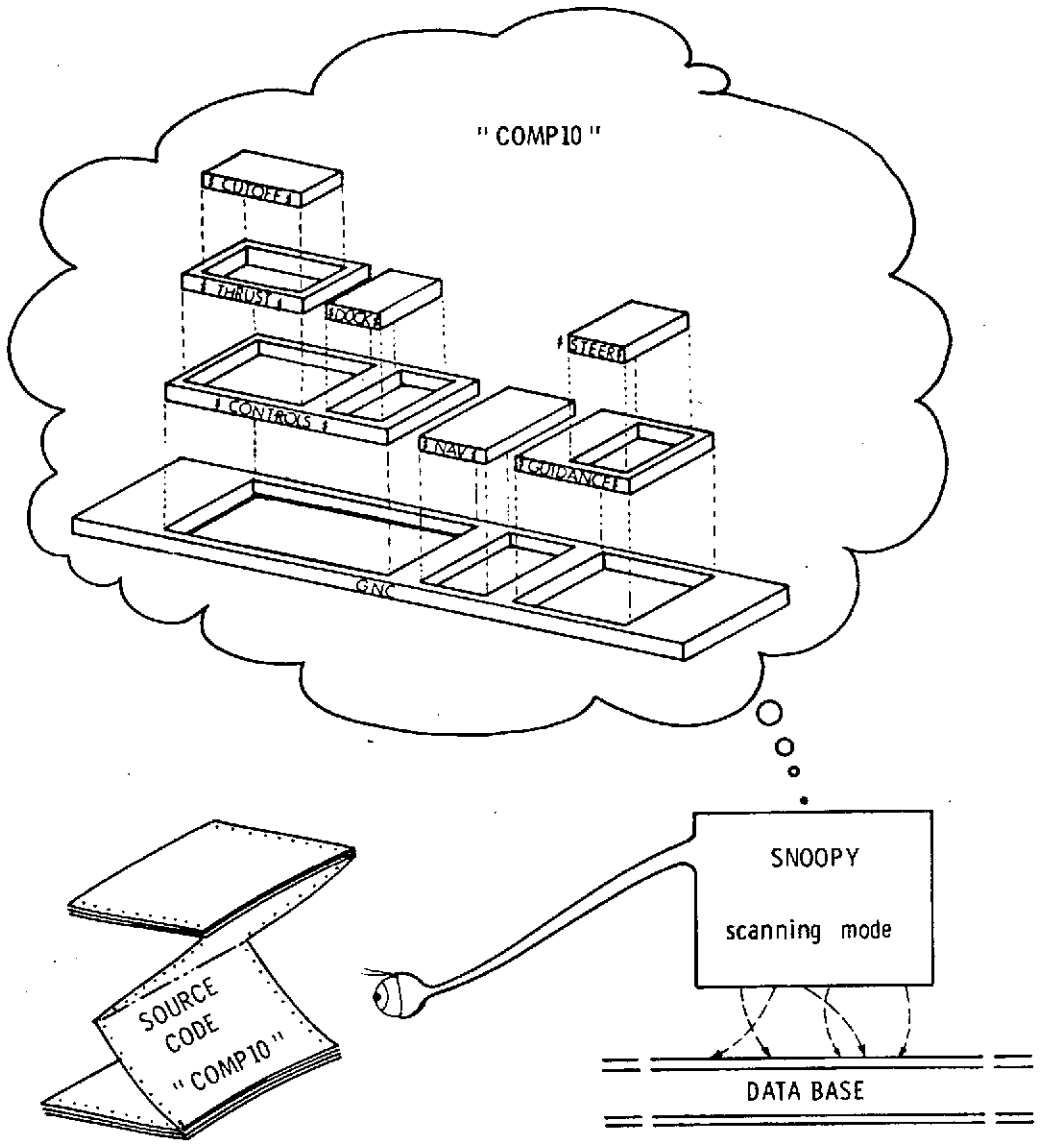
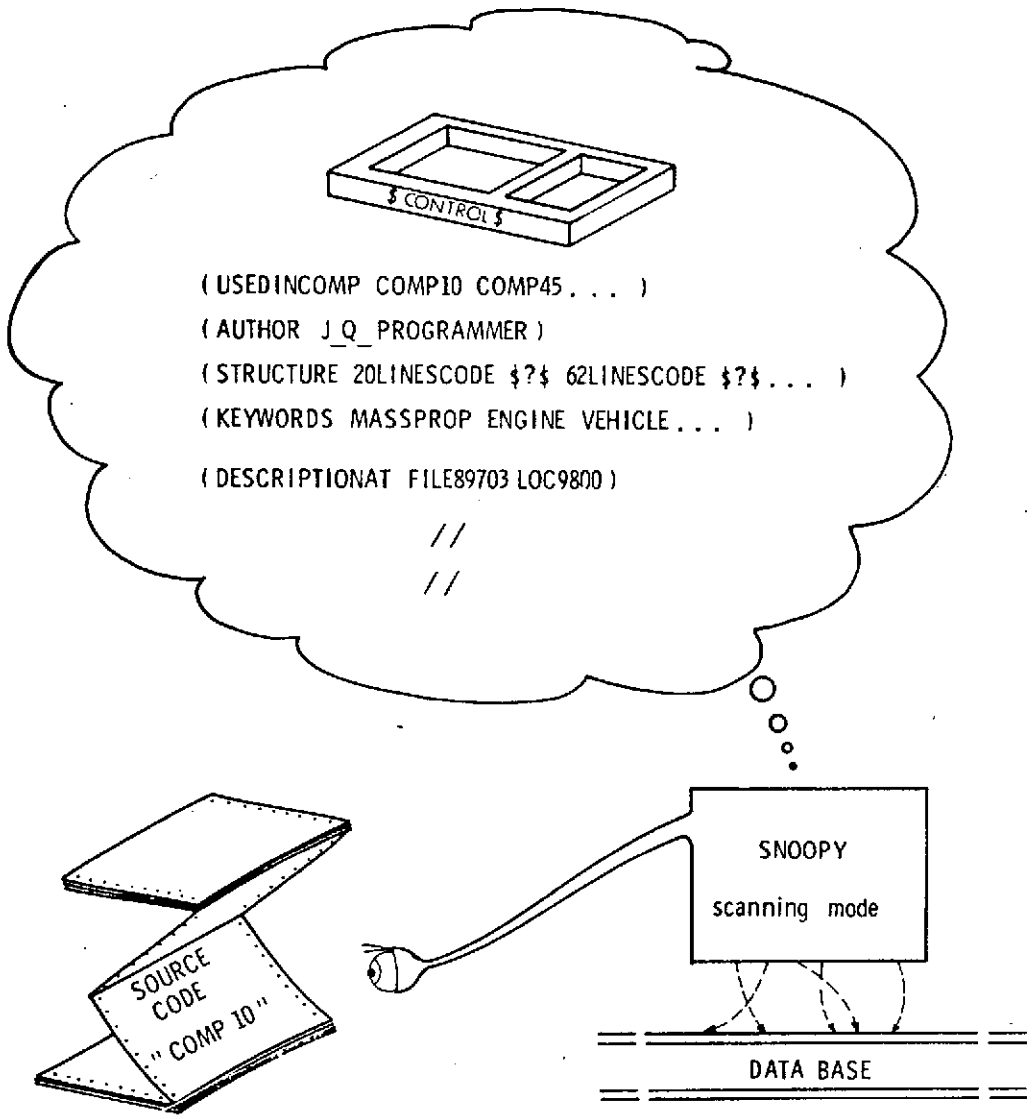


Figure 3-2. Software Design Functional Diagram



SNOOPY scans all source code at compile and recompile time, searching "blocks" as specified by the programmer. Names are assigned to each block (e.g., CONTROL) and the structure within the specific compilation is recorded. (See Figure 2-2 in the Scenario.)

Figure 3-3. Source Code Storage and Retrieval



A number of block-specific, programmer-generated "facts" are scanned, catalogued, and stored in a "descriptor" portion of the SWDVS data base. These facts can be imbedded in the program, itself, transferred from the operating system, or typed in at consoles. To facilitate later retrieval, SNOOPY utilizes the list structure, a high degree of redundancy, and extensive cross-referencing.

Figure 3-4. Source Code Descriptive Information

combinations of such criteria as revision date, revision number, program-change number, and author.

The SWDVS data base is accessible by special documenting programs. For example, an automatic flowcharting program can invoke SNOOPY to return a source listing of a flight-program revision and process it for graphic output.

3.2.2 Flight-software Verification

The SWDVS provides the capability of using flight software under many simulated flight conditions and the capability of collecting information from these simulations to relate program design to verification (Figure 3-5). Verification can be performed by

1. inspection of compiler source language input
2. compiler diagnostics
3. inspection of secondary compiler outputs (e.g. reference maps)
4. simulation (including runtime diagnostics)
5. inspection of post-run simulation output.

The results of simulations are recorded in the SWDVS data base for later inclusion in test reports and development plans. Reports on the testing techniques, the test conditions, and the test results provide visibility into the reliability of the software and its probability of mission success.

In attempting to verify software for the flight computer, the STE and analysts have three requirements:

1. Capability to focus attention on a particular feature of the flight program without necessarily becoming familiar with other program modules and other parts of the flight program simulator (PANSIM) that are required to run the simulation;
2. Capability to test a particular feature at the proper level of detail;
3. Capability to test mission sequences and module interfaces.

Elements of the SWDVS that handle these requirements are

1. PANSIM models
2. Test-plan source file
3. SUPERCREW program

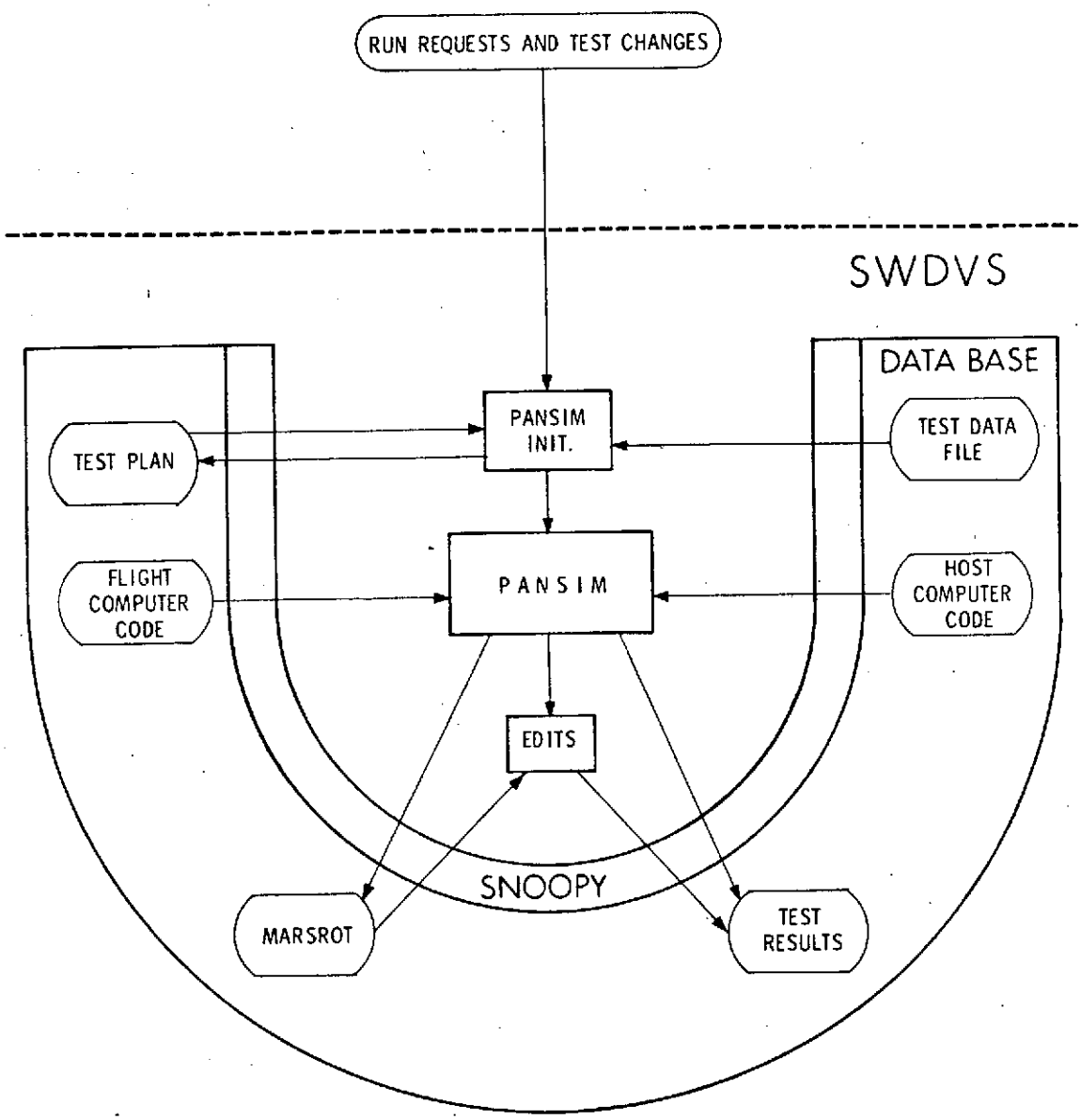


Figure 3-5. Software Verification Functional Diagram

PANSIM Models

PANSIM provides the STE and analysts with a range of fidelities of models and a compatibility with other testing. The fidelity of the model can be chosen to fit the desired level of testing and includes

1. The flight computer interrupt, timing, and interface structure
2. Simple models for rapid execution
3. Sophisticated and simple models for flight computers and environment.

(See paragraph 4.4.1 for discussion in greater detail.)

Test Plan Source File

The STE can select a standard, predefined initialization and test sequence from the inventory of test cases in the test-plan source file. These test cases are constructed of predefined data sets (test-data file) so that additions and replacements can be easily specified by the software test engineer for the feature being tested. (See Figure 3-6.) The unchanged parts of the test case fulfill the remaining simulation requirements.

SUPERCREW Program

The test-sequence data sets are written in a language that enables the STE to easily specify detailed crew sequences to the SUPERCREW program for program manipulation and cockpit control. (See subsection 4.10 for a more detailed description of SUPERCREW.)

3.2.2.1 Test-case Construction

PANSIM is initialized by inputs obtained directly from a test case in the test-plan source file. The test cases contain the following types of data:

1. Heading: title, author, revision number and date, a description of the test case, and, if applicable, a reference to respective program change;

2. Initial condition: commands to the XPANDER program to retrieve and expand initialization data sets which contain specific initialization parameters and models for PANSIM. (See Figure 3-5.);

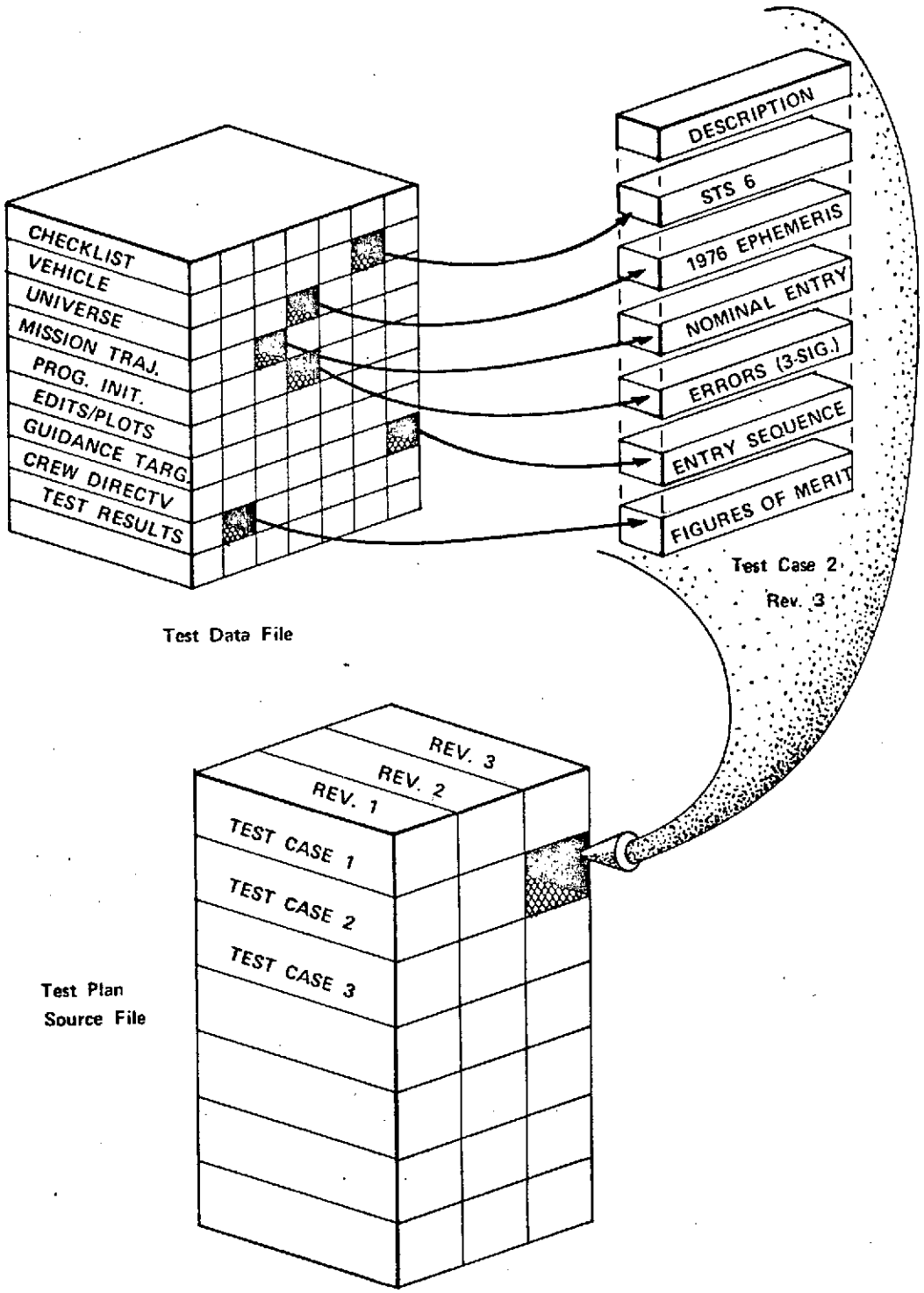


Figure 3-6. Test-Plan Source File

3. Test sequence: commands to the SUPERCREW processor to interpret crew activities directives (responses to checklist and program requests); commands to the special requests processor to set up additional special test conditions, monitors, and dynamic diagnostics.

These test cases are filed in the test-plan source file for immediate use after being created or revised.

Not only can the STE replace the data sets, he can also override statements within the data sets. This capability allows the STE to make small changes to a test sequence while maintaining the bulk of the crew actions.

Also, certain simulator parameters are initialized to "typical values" by the STE when the data sets are originally created or to default values by SIMSETUP when no value is specified. The STE can override these parameters either when better values are available or when the test case warrants it.

Often, a software test engineer must run a test on a moment's notice; the test case must be revised to accommodate special requests or to use different initialization of parameters. Under these circumstances, the STE can use an interactive terminal to revise the test case making it immediately available for simulation. A simple command at this time to run this test case revision on a flight program puts this simulation job in the queue for batch-mode processing.

The STE can create a new revision to the test case when modifications are incorporated. Modifications are categorized as

1. requests to override the specified SUPERCREW directives
2. requests to override the specified initial conditions
3. additions of special requests (TRACE, monitors, etc.)
4. additions of HOL patches to the flight-program code.

The XPANDER program expands the specified inputs into a list of initialization assignments for individual parameters in PANSIM. After the expansion, XPANDER adds to this list the assignment statements that the STE supplied for overriding the specific assignments of the expansion.

3.2.2.2 Simulation

Analysis of software design and initial program checkout is often accomplished by utilizing the compiler runtime package with or without FSIM models. Subsection 4.4 describes this application in great detail.

Other software verification is performed using the test plan source file and combinations of the PANSIM models. The STE requests a simulation by specifying a program revision number, a test case, and/or additional special requests to the test case. The PANSIM initialization(PANSIM INIT.) retrieves the program revision and the predefined inputs of the test case, expands the inputs, and initializes the parameters and models of PANSIM.

The complete simulation including the flight program, test case, snapshots of the simulator's status, and the tests' special request data are placed on a rough-output tape (MARSROT). This tape can be used as input to PANSIM INIT. for continuing or rolling back the simulation later if more details are needed than were provided by the on-line edit or post-run edits and plots. If additions or modifications to the run are desired, XPANDER processes the requests; the notification that changes to the test case were made is added to the test results data base.

If a different program revision or a different PANSIM model is requested, a new MARSROT tape is assigned. Different initializations of the same program or model are allowed, however, on the old MARSROT tape.

Program patching to a compiled version used in a simulation is a cost-saving feature. Thus, when an anomaly is found by a flight-program test, patching in the compiler language can allow checkout of the proposed fix on the same test using the snapshot-rollback simulation feature.

Dynamic diagnostics are provided by the compiler runtime package (on the host computer) and by the Interpretive-computer Simulator (ICS). These diagnostics include the tracing of compiled code, timing notations on occurrence of events, dumping of selected variables, and monitoring the usage of program modules and storage cells. These specifications have no effect on the machine code for the flight computer. At compile-time, programmers can specify error conditions and redlines for variable values or event times that, when violated, will abort the run and/or be recorded in the post-run summary. For each program revision, the compiler generates symbol tables, flags, and other indicators.

3.2.2.3 Test Results

After a run is terminated, the STE reviews a post-run summary edit that provides information about the success of the run. With this information, the STE can determine whether a detailed printout of the simulation activities is required, whether plots and edits are necessary, or whether further testing is required.

The post-run summary is comprised of default information augmented by test-specific information. Figures-of-merit show qualitative and time-oriented information about the simulation, e.g., the value of the maximum range and its time of occurrence. Also, indications of error conditions, which do not compromise the simulation, but which must be recognized, notify the STE of possible problem areas. For example, consider losing one telemetry cycle. The cycle may have been cancelled by a busy executive, i.e., telemetry task had too low a priority to be accomplished. Certainly, the mission can continue if one telemetry cycle is lost, but the fact that the executive is so busy that a task is cancelled may be the symptom of a potential problem.

Also indicated in the post-run summary is the cause of termination of the run. Other than by normal termination, the run can be terminated by aborts conditional on

1. maximum runtime allowable
2. specification provided by the STE (e.g., ABORT IF ALTITUDE < 85 NAUTICAL MILES)
3. specification provided by PANSIM (e.g., SUPERCREW cannot answer program request).

After reviewing the post-run summary, the STE can specify that a trace of the outer level of the HOL statements be made starting at a time just before the abort. The active jobs at the time of the abort are also given. Thus, the STE can see not only what caused the abort, but also can be given sufficient information that further investigation of the abort is not necessary. (See the example in Figures 2-11 and 2-12 of the Scenario.) The STE can add to, or modify, the special request initial conditions or crew activities to debug the run or to correct the error so the simulation can continue normally. These abort-condition monitors can be disabled if they hinder the pursuance of the investigation.

If the run is successful, but more data is needed to analyze details of the performance, data on the MARSROT tape can be plotted and/or edited for review. These plots

and edits can be filed under the test-results source file and are cross-referenced to the corresponding test case. Selections from these data files can be chosen to form test reports.

The data available, then, for test reports are

1. test-case information
2. summary of results
3. selectable edit pages and plots
4. revision of program and date of test.

For each revision of test case simulated, there is a corresponding file in the test results source file so there is a history of the number of tests run on each program revision and on a particular program change.

3.2.3 Supervisory Control of Flight Software Development

Program control supervisors have the responsibility of developing reliable flight software. In the course of performing this task, they are called on to identify software requirements; establish milestones in the development cycle; specify software tools, testing procedures and documentation; and, generally, to provide the necessary control to ensure that the operation proceeds in an efficient and effective manner.

The information storage and retrieval capabilities of the SWDVS are designed to assist program supervisors in meeting their responsibilities in the following areas:

Planning—the program supervisor must have information available on past development for special studies in software development. He must be able to backtrace through the development process to gain insight into the cause (program changes) and the effect (schedule changes). Immediate information on the current status of the software is necessary for committal to implementing further changes to the program without schedule slippage.

Reliability versus efficiency—due to such traditional problems as ambiguous or incorrect specification documents, the uncertainty of the hardware design, and the complexity of a software program, it was difficult to measure the correctness of the software. The program supervisor must still decide how much testing is warranted for the cost incurred.

Visibility—the program supervisor must be familiar enough with the flight programs to effectively make decisions on estimates, design and reliability.

Organization and control—large software systems present numerous problems in controlling input data to the flight program and in setting up and maintaining development and verification procedures to ensure timely delivery of the verified flight software.

Documentation—documentation of the software must be correct, sufficient, up-to-date, and delivered on time.

3.2.3.1 Flight Code Compiler

For each program revision, the compiler illustrated in Figure 3-2 provides information to the SWDVS data base. (See Figure 3-7.) Special programs can access this data base and present the following types of information:

- status and history of program changes;
- updates to scheduling charts for development plans;
- reference maps for program structure, variables, flags, alarm codes, events, and program lockout;
- cross-reference to program changes, specification documents, test plans and test reports;
- up-to-date descriptions of programs;
- publishable descriptions of changes made and a list of sources impacted for each revision.

The compiler language and structure is very readable so that the program supervisor can review the actual code in his familiarization with the program. Equations can be coded as they appear in the specification documents. The program can be structured into program modules; the logical statements are easily understood and the paths easily followed.

To facilitate the implementation of coding into the main-line program, the compiler can assist in configuration control over the use of subroutines, constants, and variables by referring to the COMPOOL and specification source files. These source files are maintained to ensure consistencies not only within the flight program, but also through out other parts of the SWDVS and between the SWDVS and other NASA facilities.

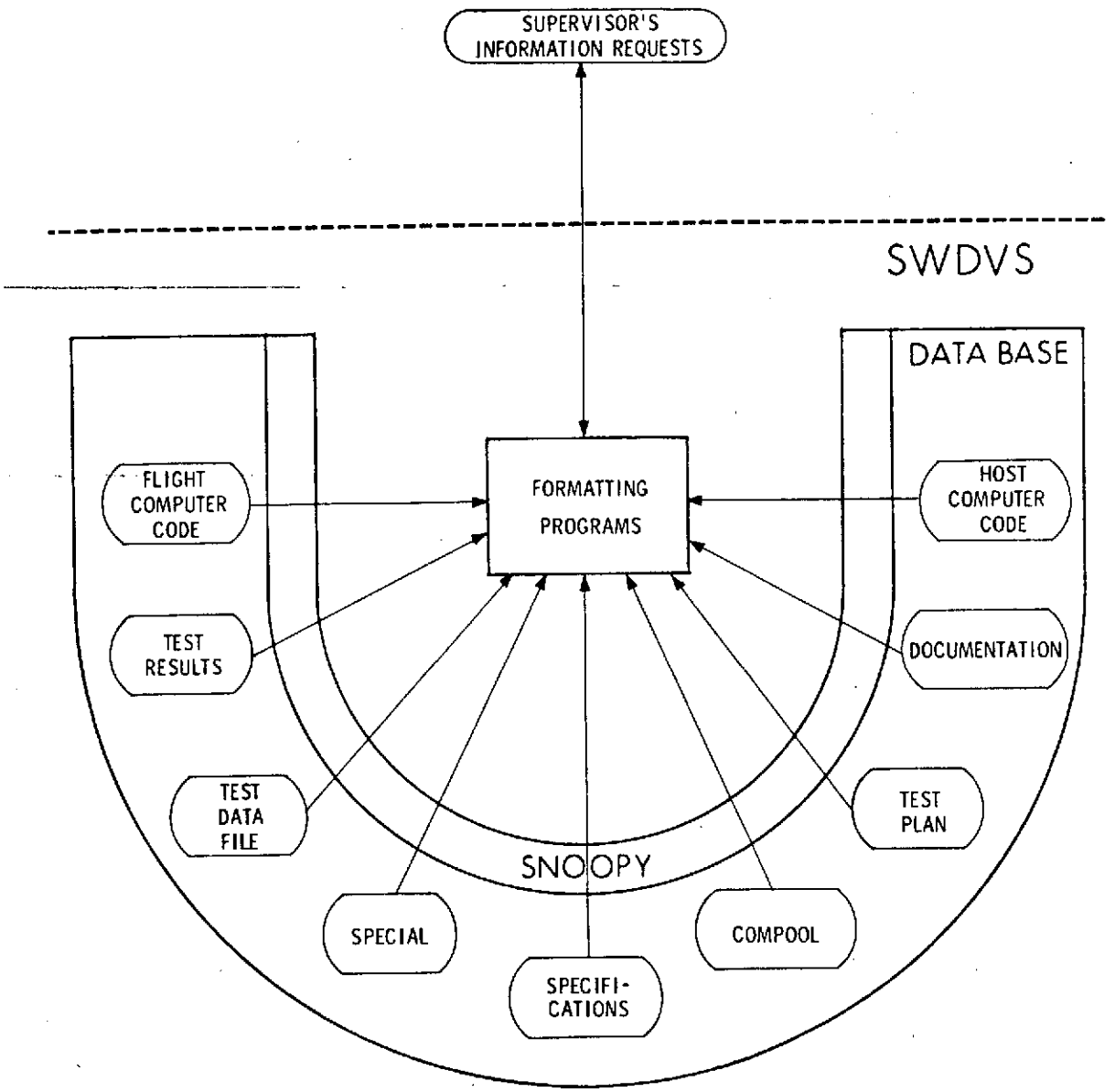


Figure 3-7. Supervisory Information Functional Diagram

The SWDVS can offer profitable possibilities depending on management decisions. For example, if the same high order language is used to code the flight program and its corresponding FSIM model, much of the programming effort need not be duplicated.

3.2.3.2 Organization and Control

The SWDVS provides a facility that will assist in many supervisory control techniques. The SWDVS will only assist, however, if the proper control and cataloging procedures are performed. One supervisory control requirement is to prohibit unauthorized persons from making changes to the main-line program, while allowing development of off-line programs.

The program supervisor requires the capability to preserve all program compilations. The uncertainty of the effect of program changes on the other parts of the program is high enough to warrant implementation and initial testing in an off-line version of the current main-line program. With off-line programs, development and verification can continue concurrently on many program modules. This approach also has the advantages that verification runs are not necessarily repeated when a module is brought on-line, and that the chance of an error rippling through the program is reduced. (The off-line programs are kept for future reference as well. Even though testing is done in an off-line version it does add to the testing history of a program.)

Occasionally an earlier revision of a previously compiled main-line program is chosen to be released for a mission. It could be very difficult to recreate the program had it not been preserved. Further, a preservation capability aids the program supervisor in following the program change status and retrieving historical information. Thus, he can determine whether the simulations were performed on consistent input data, program revisions and simulator models.

3.2.3.3 Retrieval of Information

When verification tests have been completed with satisfactory results, edited and plotted run data are stored in the SWDVS data base. (See Figure 3-5.) Information on various categories can be retrieved to provide the program supervisor with such facts as the number of tests run on a program revision, the test conditions and results, the number of times a particular test was run, the total number of runs on a program change or a flight program, and the dates a test case was started.

The host-computer time consumed is recorded with each compilation and simulation. The associated costs mentioned above are easily retrieved. This information is helpful not only in planning additional software development, but also helpful with such supervisory considerations as:

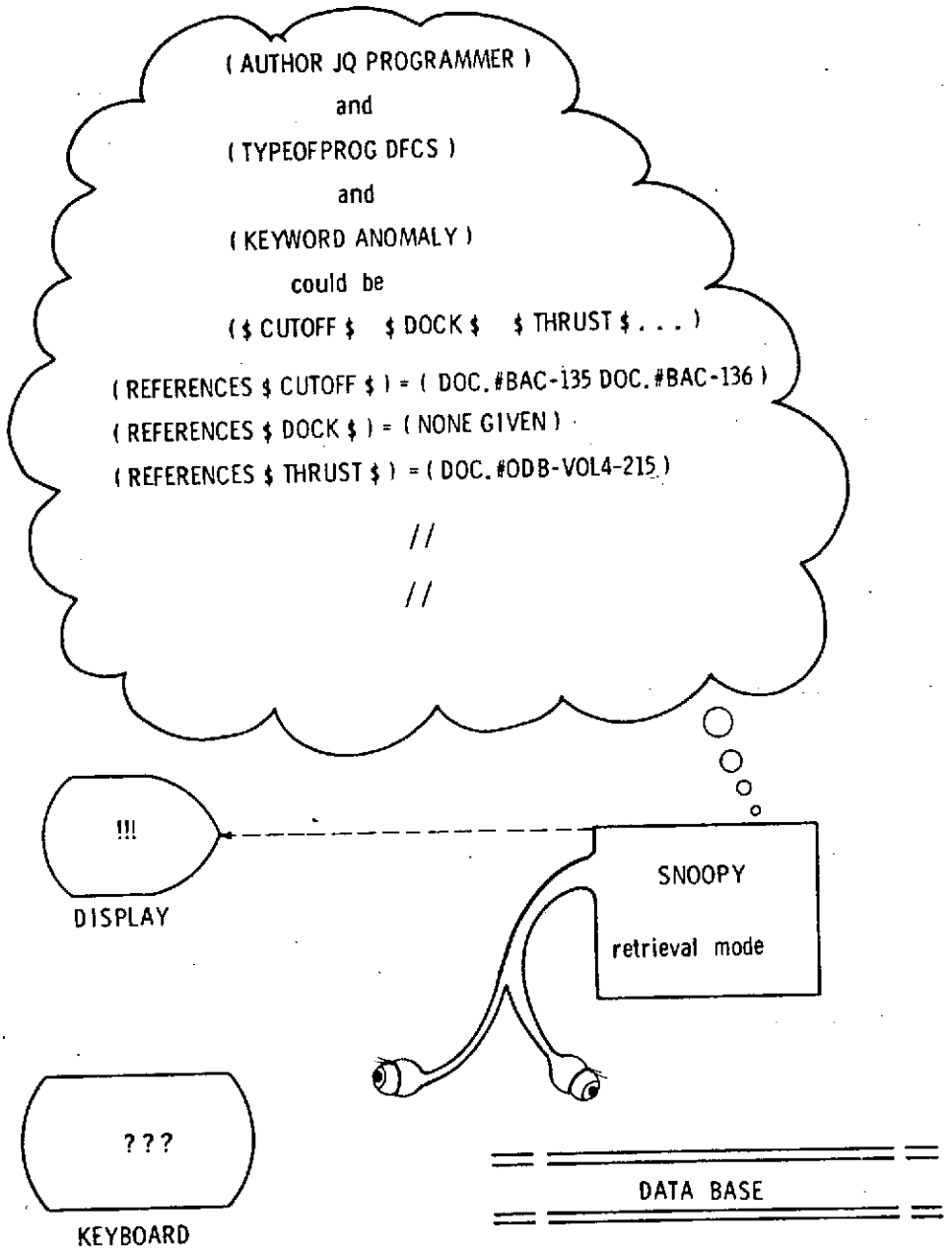
1. The program supervisor may wish to use the collection of data for special studies. The record of each program change and the specific reason (new code or a repair to old code) for the change is kept in a data file. A search program (SNOOPY) can retrieve the changes which were implemented to repair incorrect code, and thus give an indication of how many errors can be expected per line of code for future software development.

2. In order to determine some measure of reliability the program supervisor may wish to know how many simulations were made using a particular test case and on what program revisions the tests were run. SNOOPY is able to retrieve this information with few criteria specified.

3. The program supervisor can retrieve information about the number of coding changes made to completely implement a particular program specification. The associated costs to implement and verify this specification gives the program supervisor an indication of the efficiency of the development effort.

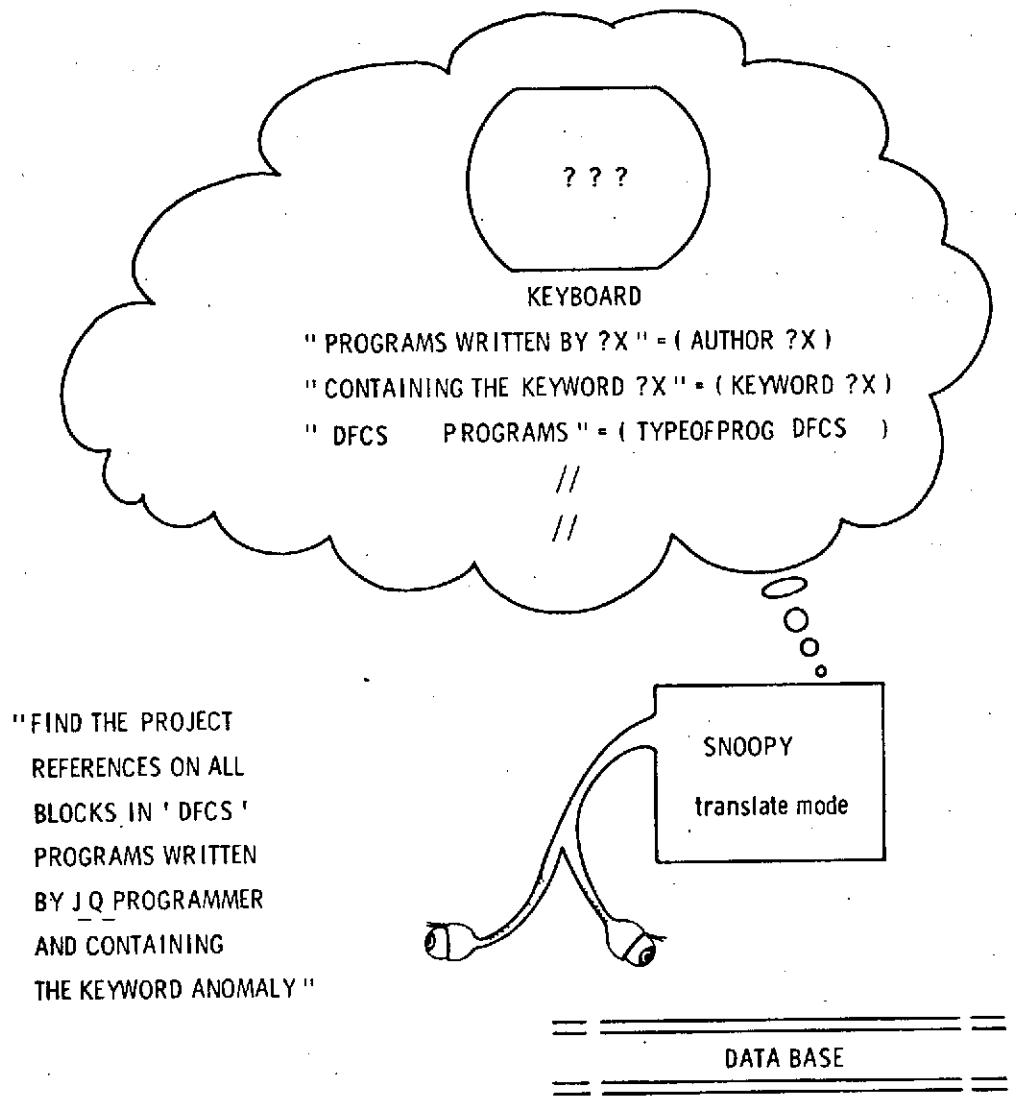
In order to retrieve information from the SWDVS, the information must first be recorded by category. Thus, if it is expected that information about changes due to anomalies will be requested, each program change resulting from an anomaly must be categorized as an anomaly change. (Se Figures 3-8 and 3-9.)

Using the SNOOPY system, information from the compilation can be used to update such documentation for the program supervisor as development plans and scheduling charts. The status of each program-module (or approved program module change), its verification, associated anomalies and fixes, pertinent test plan information, and test reports are stored for data retrieval programs used by supervisor services.



In retrieval mode, SNOOPY will accept symbolic "descriptions" and use them to direct a search through its data base looking for blocks that satisfy them. When all such blocks have been found, SNOOPY returns any previously stored data associated with them.

Figure 3-8. SWDVS Information Retrieval



Operation in retrieval mode can become complex as the "descriptions" and "facts" become numerous and diverse. For this reason, SNOOPY assists users by providing a translation (preprocessing) capability. The input to SNOOPY is a limited-vocabulary, simple-grammar English; the output is the equivalent "descriptions", formatted for direct execution by the retrieval program.

Figure 3-9. Conversation with SNOOPY

3.2.3.4 Documentation

Documentation on the development of flight software can be in such forms as memorandums, specifications, development plans, and flowcharts. The documentation is kept up-to-date by recording information about program changes (descriptions, status, etc.) as they occur. The documentation can be reviewed by calling formatting programs to present information retrieved by SNOOPY. Similarly, the information can be printed for publication when desired. (See Figure 3-7.)

3.3 SWDVS INTERACTION WITH THE SYSTEM VERIFICATION PHASE SIMULATORS

3.3.1 Feedback of Data

Flight software verified on the SWDVS is not considered flight rated until it has undergone additional testing on the System Verification-phase Simulators. These simulators provide the additional verification that can only be achieved by tests run in real-time with human interaction using real avionics hardware. An important design consideration for SWDVS thus involves the processes of communication between it and these facilities: how can feedback channels be set up? How can we ensure that they can lead to an effective tracing and debugging of anomalies? As stated in the Task-5 report (Reference-3), every code executor should have "the potential not only for revealing anomalies, but providing for their diagnosis and correction. Thus, every instance of code being executed increases the confidence in the quality of the flight software." In the case of SHUTTLE flight software, diagnosis and correction of anomalies is closely linked to the capability to coordinate testing with the other facilities involved, and to efficiently transmit explicit data of several kinds.

3.3.2 Areas of Interaction

The specific capabilities deemed most useful to a free and easy exchange of data between the SWDVS and the System Verification Phase Simulators can be identified as follows:

Cross Initialization

If a Systems Integration Laboratory (SIL) test run could be initialized from a standard SWDVS rough-output tape, suspected software-hardware interface problems detected on the SWDVS could be analyzed on real equipment. Alternatively, if SWDVS could initialize from a SIL rough-output tape, the diagnostic capability of its batch mode should lead to insights into a SIL-detected problem.

A mutual cross-initialization capability would also provide a means of specifying a test run for the purpose of a detailed comparison. Confirmation or explanation by one facility of a phenomenon observed at another would be more likely. Dual-facility tests run with good agreement would increase confidence in those areas not directly comparable.

Hardware Models Checking

Good communications would allow detailed performance comparisons of the SWDVS software models and the real hardware they describe. Such comparisons would yield insights into both system performance at the hardware level, and software model limitations. With some data reduction of the output of SIL runs, the SWDVS models could be improved. Hardware performance could be compared to the early "benchmark" test results, with the possible result of tracing SIL hardware malfunctions.

Crew Procedures Checking

Flight software must be designed around realistic crew capabilities. Unworkable procedures not readily apparent to SWDVS' SUPERCREW are best detected through human performance testing on real-time simulators. Feeding back detailed crew performance histories would assist in the development of workable procedures with the right "feel".

3.3.3 Commonality between the SWDVS and Systems Verification Phase Simulators

Software

In the following areas, an emphasis should be placed on incorporating commonality or at least compatability into the support software of the facilities:

Simulation Control Program (called SUPERCREW by SWDVS; "Test Director Executive" by SIL).

Sensor Models, Aerodynamic and Astrodynamic Routines (called External-environment programs by SWDVS; "Common Math Model Library" in Reference 2).

General Analysis Routines (called EDITs by SWDVS).

Data Base Management System (called SNOOPY by SWDVS).

Hardware

At least two possible approaches should be examined for hardware:

Emulators

SWDVS could benefit from an emulator of the Interpretive-computer Simulator which is somewhat akin to a "test cooperative" version of the flight computer. The SIL, Procedures Development Simulators, and Shuttle Mission Simulator require emulators of the flight computers.

The similarities between these two requirements should be examined in search for a common emulator for all simulator facilities.

General Purpose Host Computer

If the family of general-purpose host computers is common between the SWDVS and other facilities, the capability to share common software is enhanced.

3.4 SWDVS VERIFICATION

The primary concept of verification in relation to the SWDVS applies to its role as a tool for verifying flight software. There are, however, two other ideas concerning verification that are important to the SWDVS: first, the need for the SWDVS developers to verify the software of the SWDVS itself, and, second, the possibility of conducting an independent verification of the SPACE SHUTTLE flight software.

Because of budgetary and development schedule constraints, it is important that the SWDVS software be verified in a cost-effective manner. To this end, the SWDVS must be structured so that a relatively brief test plan can produce the desired level of confidence in SWDVS reliability. This is essentially the same requirement that is placed on flight software to facilitate its verification. The verification procedures are designed to ensure that the SWDVS performs in accordance with its specification documents. They will, by means of thorough stress testing, expose SWDVS anomalies early enough to significantly reduce their impact on the flight software development cycle and thereby reduce the time and cost associated with developing SPACE SHUTTLE software.

An additional task can be performed that will increase confidence in SWDVS performance. When analyzing test results, the test engineer often wishes to know the open loop step response of the external environment engineering models contained within SWDVS. The SWDVS verification process can generate for the SWDVS data base that set of open loop step response and verification results likely to be desired by the flight software verifier. In addition, these open loop step responses can be periodically compared to results from the SIL, from test flight data, and from the hardware manufacturers test labs to evaluate the need for External-environment Simulator model changes.

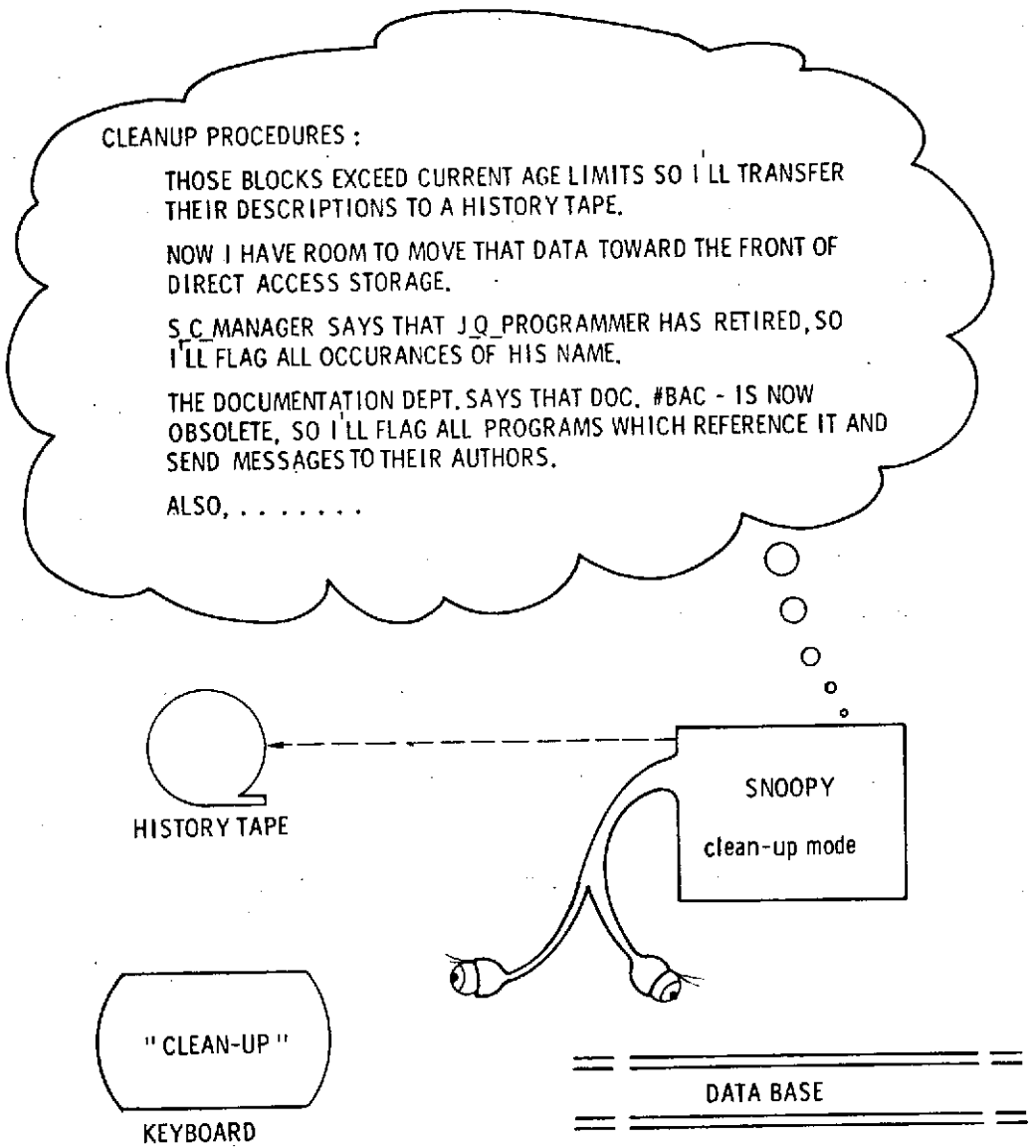
The second concept of SWDVS verification concerns the possibility of an independent verification of flight software. An independent SWDVS might be considered too costly because of the amount and complexity of the SPACE SHUTTLE software. Because of the verification of the SWDVS itself, however, the independent flight software verification could be performed with the same SWDVS used to perform the primary verification (possibly, of course, at a secondary facility).

3.5 ANALYSIS OF THE FLIGHT-SOFTWARE DEVELOPMENT PROCESS

In addition to its primary function of developing and verifying SHUTTLE flight software, the SWDVS should provide a useful case study of the software development and verification process itself. The SWDVS design embodies innovative concepts in such areas as code generation, simulation techniques, data management, and program source listing control that are applicable to a wide range of large, complex software systems. The experience gained from employing these concepts in the SWDVS should be made available so that other software projects, both inside and outside NASA, can benefit. This goal implies a major effort to chronicle the software development and verification process, as carried out with the SWDVS, from both an historical and an analytic point of view.

The key to documenting and analyzing the performance of the SWDVS lies in preserving the SWDVS data base which contains information about the evolving flight software, the SWDVS itself, and the SPACE SHUTTLE hardware data used by SWDVS.

As this data base evolves, obsolete data selected by the project managers is transferred to magnetic tape by the SNOOPY data management and retrieval system. (See Figure 3-10.) These tapes form a SWDVS historical archive. The data are entered in sequential form and structured for ease of retrieval and manipulation at a later time by programs added to the SNOOPY system. These special-purpose programs can be designed at that time to perform historical or analytical studies of the flight-software development process. The entire cumulative data base must be saved in the SWDVS archives to ensure that no information of potential use to future examiners is lost. Therefore, an ultimate collection of several thousand tapes can be envisioned; this number should not be considered excessive in view of the potential benefits to be derived.



Finite direct-access storage plus a rapid turnover in relevant "facts" will require periodic "cleaning" of the data base. Obsolete data will be dumped onto history tapes in a manner suitable for later recovery if necessary. Additional mechanizable procedures of system management may be implemented.

The history tapes should eventually comprise a fully detailed history of SWDVS development from its early stages of implementation.

Figure 3-10. Data Base Cleanup

3.6 INFLUENCE OF FUNCTIONAL APPROACH ON SWDVS DESIGN

The approaches to the SWDVS, as expressed in the preceding parts of this section, with the goals, assumptions and considerations of Section 1, influenced the trade-offs that determined the SWDVS structure.

Two additional design concepts were influential. First, the SWDVS must provide tools powerful enough so that one person's breadth of insight can be large enough to cover a problem associated with flight software – not a problem associated with manipulating the SWDVS (i.e., SWDVS should be used by flight software experts, not SWDVS experts). Second, the SWDVS must be capable of addressing the most unusual and complex problems although it is tuned to operate most efficiently in its most frequently used manner.

The design reflects the trade-offs that must be made if the needs of each functional group are to be satisfied. The cost of developing and using the SWDVS over a period of time was considered, based on experiences with APOLLO and other large systems. Each proposed feature of the SWDVS was analyzed to determine the extent to which it could be incorporated into the design and still remain cost-effective. The analysis focused primarily on the cost to the SWDVS of each feature in terms of runtime efficiency, ease of development and maintenance, and overall size (including the implied paging of program blocks in and out of core at runtime).

The following list of features are deemed to be cost-effective within the overall SWDVS design:

A structure that allows a degree of flexible response to SHUTTLE avionics changes and SHUTTLE mission respecifications.

A structure that allows verification of SWDVS software to be less costly.

Capabilities that allow ease of learning to use SWDVS and ease of continued use.

A structure that allows facility independence among the host computer family.

Input/output capabilities that allow ease of results comparison with other SHUTTLE simulation facilities.

A structure that allows a smooth development process.

Upgradeability to incorporate features deemed too costly now that are candidates for future investment.

The following features are not incorporated into the SWDVS design, despite their beneficial aspects, because their cost appears out of proportion to their benefit.

Capability of running two or more flight computer programs in the Interpretive-computer Simulator simultaneously.
Capability of running real-time simulations.
Use of one language for all SWDVS programs.
Complete host-computer independence (the SWDVS is host-computer independent within the one computer family for which it is designed)

SECTION 4

ELEMENTS OF THE SWDVS SOFTWARE

4. ELEMENTS OF THE SWDVS SOFTWARE

4.1 FLIGHT-CODE COMPILER

The development of flight software includes the generation of code modules for the flight computer and the necessary amount of bench testing to ensure that the individual modules satisfy the intended software design. A compiler supports these functions.

In addition to the translation of its source language into machine code, the compiler provides the following functions:

1. compile-time-self-diagnosis
2. cross-reference source within SWDVS and to NASA facilities through the SNOOPY system for simulation, documentation, and program supervisory purposes
3. communication and control between users
4. code intent information for dynamic verification
5. Runtime package for checkout on the host computer.

The compiler must be flexible enough to support the diverse requirements of the Implementation Phase. It must be able to translate its source language into machine code for the flight computer, but it must also accommodate code-generating modules to translate the language into the machine code of the SWDVS host computer for support of engineering analysis and flight software systems design. It supports the I/O structure of the SWDVS host computer and is compatible with the interactive display system's terminals. The compiler's design is also compatible with the SHUTTLE flight computer hardware and software (Executive) architecture.

The compiler passes information other than the machine code to the flight computer. Some variables, for example, have initial values that must be set up for the flight computer; the compiler assigns a prespecified or random number to the unused computer words that contain no data or instructions. In addition, there may be (1) memory protect features in the flight computer that require information from the compiler, and (2) program lockout features in the Executive that require information from the compiler pertaining to compatibility between programs. The compiler provides this information.

4.1.1 Comparison of Compilers and Assemblers

The process of transforming flight-program equations into the machine code of the flight computer can be accomplished either by a compiler or by an assembler. A compiler, with its associated high-order language, offers the following advantages over assemblers:

1. Faster initial development of flight code.

    a. An assembler requires more lines of code than a compiler for a given set of equations. Since the time spent on the initial programming effort is a function of the number of lines of code to be generated, a compiler reduces the time required to initially code each software module.
    b. Programmers new to the software development effort can learn a compiler language more easily than an assembly language.
    c. A compiler permits easier coding of engineering and mathematical functions since they can be written exactly as they appear in the specification documents.

2. Greater visibility and self-documentation—the program listing written in a compiler language is more easily readable than that written in assembly language.

3. Superior error detection and control.

    a. Software errors are limited mainly to logical errors since, for example, low-level data manipulation and looping are handled by the compiler rather than the programmer.
    b. A compiler can (and should) be designed to limit entry and exit points in program blocks and modules, thus eliminating errors due to incorrect branching.
    c. A compiler eliminates errors in addressing, since memory locations must be referenced by name rather than by relative addressing. (Use of pointers and label variables negates this advantage somewhat; these features should be carefully controlled or perhaps limited to systems programming.)
    d. For a fixed-point machine, a compiler eliminates scaling errors.

e. The sophisticated diagnostics associated with a compiler speed up initial program checkout and significantly reduce the reliance on dynamic simulation to detect programming mistakes—a process that is time-consuming and expensive.

4. Improved Supervisory control.

a. A compiler can incorporate features that disallow certain dangerous programming practices. (Some practices cannot be detected by an assembler.)

b. A compiler can produce meaningful cross referencing and mapping of relationships between program modules and subroutines.

c. A compiler can control access to system variables and produce a summary of the program's usage of constants, variables, and subroutines.

d. Memory allocation for system variables and constants can be controlled by use of the compiler, avoiding costly errors due to invalid memory overlays.

5. Easier transferability—programs can be run and tested on various computers more readily.

4.1.2 Compiler Modes of Operation

The compiler operates in four different modes in support of the following software development functions:

Engineering and Software Systems Design Mode—Before actual flight code can be generated, the flight computer must have been selected. If the same high order language is used for the various flight and host computers, however, the programming can begin as soon as the necessary equations, data management techniques, and desired program sequences have been determined. Thus, the programs can be written and partially verified on any computer for which the compiler can translate code. When the flight computers are determined, translation of code into their machine languages by the compiler and formal verification can begin. A compiler that can easily translate its high-order language into more than one computer's machine instructions, therefore, will permit initial development of software if hardware design and selection difficulties arise.

The compiler's runtime package allows programs to be run on the host computer in conjunction with the SWDVS flight program simulator (PANSIM) models. (Subsection 4.4 describes this functional relationship in detail.) FSIM models created from the flight code, written in the same language, and possibly sharing code written for the flight programs can offer greater confidence during software verification.

Program Module Compile Mode—Early in the development of software, emphasis is placed on individual program module design and checkout. For this mode, the compiler allows constants to be defined outside the common pool (COMPOOL) level and does not restrict the assignment of subroutines and variable names. A TRACE feature is provided with the runtime package since the Interpretive-computer Simulator (ICS) may not be available at this stage.

Integration of Program Modules Mode—When the programs are compiled together, consistencies among variables, subroutines, and constants must be maintained. Configuration control over the programs can be maintained if constants are defined in the COMPOOL and, thus, can be referenced to source documentation.

Simulation and Verification Mode—Program patching to a compiled version for simulations is a cost saving feature. Thus, when an anomaly is found in the flight program, patching around the anomaly in the compiler language can allow checkout of the fixes without the need to compile a new version for each fix. A new compilation is thus not necessary for each anomaly.

4.1.3 Diagnostics

Compile-time checks are made on the submitted program modules to determine if the language requirements have been met. Such checks include

1. variable name consistency with COMPOOL
2. label name consistency with other programs on file
3. subscript limit violation
4. syntax adherence
5. constants cross-reference to other SWDVS systems
6. memory lockout checks
7. single entry and exit points for subroutines
8. precision consistency.

Before the full PANSIM is available with its powerful diagnostic tools, the dynamic diagnostic features offered by the compiler's runtime package can be used. Although these diagnostics are mostly language oriented, they offer enough flexibility to give the programmer a high level of confidence in the written program module and sufficient assistance in troubleshooting when the program module does not function properly.

Compile-time requests of dynamic diagnostics include tracing of compiled code and dumping of selected variables. The tracing of code from the statement level to the program module level is allowed. When another module is entered, the name of the program module invoked and the formal parameters of the module are printed. A program variable dump at selected statement numbers is provided as a further option.

Dynamic diagnostic checks include the following:

1. overflow/underflow
2. divide by zero
3. negative square root argument
4. arcsine argument greater than 1
5. subscript out of range
6. subscript limits
7. monitoring program module lockout
8. monitoring storage lockout
9. execution of a constant
10. executing or referencing undefined registers
11. parameter hard and soft redline violations.

4.1.4 Programming Aids

An aid to programmers and program supervisors is the Program Structure Reference Map. For each program module this map shows the other routines invoked as well as the callers of the current module. This map can be obtained at compile time and can be produced by the compiler or by a utility program that scans all programs in the file to construct a cross-reference table. Similar mapping of runtime program and module usage is provided by the runtime package.

Similar to the Program Structure Reference Map is the COMPOOL-level Variable and System Constants Map. It is produced both on a static (compile-time) and dynamic (runtime) basis.

This cross-reference table includes

1. type (integer, scalar, matrix)
2. class (variable, constant)
3. length (number of components, e.g., 9 x 9 matrix)
4. precision (number of bits representing data word)
5. references (where and how the item is used)
6. scale factor (LSB worth for data word)
7. engineering units (seconds, foot-pounds).

The language includes the capability of specifying program or task lockout. Using this information, analysis of multiprogrammed systems is aided both by static and by runtime reference mapping of program-to-program relationships, showing which program modules are allowed to run together or to interrupt each other. Runtime reference tables for programs and tasks include:

1. all program modules applied or invoked
2. all users of the program module
3. for each invocation, the conditions associated with the scheduling (priority, timing, events, or signals).

A table of references to each item is maintained by the compiler. This table requires that all documentation and simulation programs be accessible for this function. A list of events, flags, and software alarm codes used in the flight program is generated for each compilation. The labels that are not referenced are so noted. Compiler directives are provided, which specify compiler options for format and optional checking facilities.

L - 2

4.2 OVERALL STRUCTURE OF FLIGHT-COMPUTER SIMULATOR (PANSIM)

The Flight-computer Simulator (PANSIM) containing the elements shown in Figure 4-1 has been designed to satisfy the simulation requirements of the Flight Software Development and Verification System (SWDVS). The functional blocks in the figure represent the computer programs to be operated on the SWDVS general-purpose host computer. PANSIM is an all-digital simulator composed entirely of software; it includes no analog elements or SPACE SHUTTLE hardware. An all-digital simulator has been chosen for the following reasons:

- Has the flexibility to respond quickly to changes in SHUTTLE hardware and mission requirements, as they evolve, without costly modifications to simulator equipment.
- Guarantees absolute repeatability of test work.
- Provides freedom from real-time constraints, and thus greatly improves the efficiency of the simulator during periods of reduced flight-computer activity.
- Frees the user from transient failure.
- Readily produces detailed timing studies.
- Is available to multiple users for both interactive and batch-mode operation on the SWDVS general-purpose-host-computer facility.

Interpretive-computer Simulator (ICS)

Verification of flight software at the instruction level requires using an Interpretative-computer Simulator (ICS) that can reproduce flight-computer operation at the instruction level in the general-purpose host machine. Functionally, the ICS consists of three major parts:

1. Loader
2. Instruction-execution routines
3. Diagnostic routines.

The loader preprocesses assembled flight code for efficient simulation by the instruction routines. The instruction-execution routines then simulate the sequenced execution of flight-program instructions. Instruction simulation is performed such that the state of the ICS at the end of each instruction is identical to the state of the actual flight computer. Such operations as truncation, round-off, overflow, arithmetic operations, and timing exhibit the same behavior on the simulated computer as they do on the flight computer.

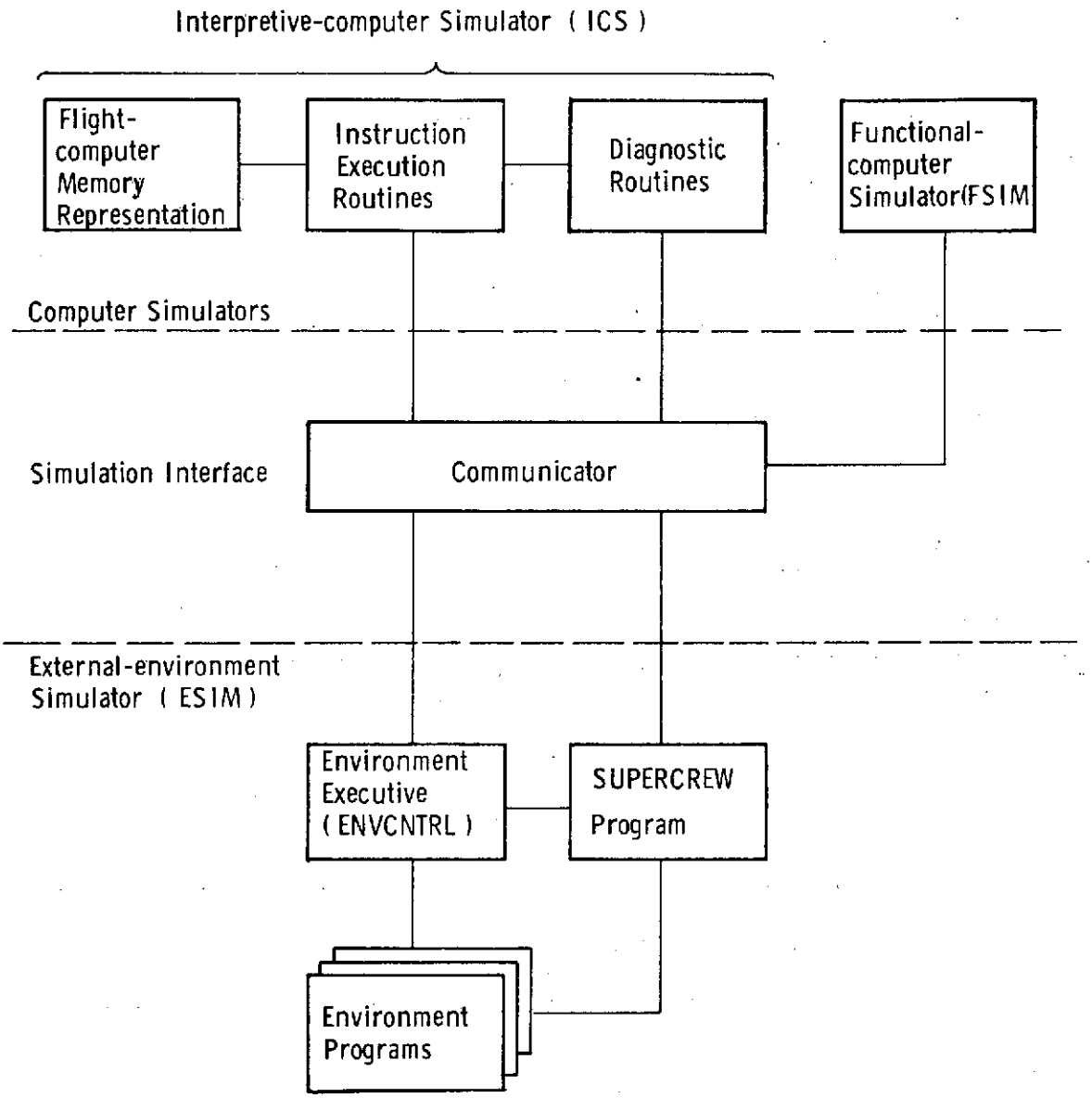


Figure 4-1. Structure of the Flight-computer Simulator (PANSIM)

The diagnostic routines do not affect the state of the simulation. Rather, these routines perform various types of special processing specified by the user as part of the input to the simulation. These include initialization of flight-computer memory locations, temporary flight-program modifications, requests for various environment options, and inclusion of various diagnostic tools.

Functional-computer Simulator (FSIM)

A Functional-computer Simulator (FSIM) is provided for those phases of flight-software development not requiring instruction-by-instruction simulation of flight code. This simulator consists of programs written in the same high-order language used to generate source code for the flight computers but compiled to run on the SWDVS host computer. These programs perform the same functions as the software of the actual flight computer. The FSIM is particularly useful during the early stages of flight-software development for (1) analyzing the interactions of the various routines comprising a flight-computer program, and (2) facilitating development of flight-software executive routines.

The functional-computer simulation capability is mandatory for developing and verifying flight software to be used in the multi-computer avionics system envisioned for the SPACE SHUTTLE. The existence of redundancy in the avionics system—and the need for cross-strapping some types of redundant control systems—defines the requirement of modeling this redundancy. An FSIM can effectively meet this requirement, and provides a powerful method of analyzing both nominal and failure-mode performance of the system.

In addition, the FSIM performs a valuable service during the verification of a flight-computer program by making possible the interpretive simulation of one flight computer and the simultaneous functional simulation of one or more additional computers. The capability to model the interactions of these computers with the flight program being verified is essential to attaining the level of software confidence desired for SHUTTLE.

Communicator

The Communicator provides an interface between the computer simulators and the External-environment Simulator (ESIM). It coordinates data transfers among the simulated flight computers, as well as between the computers and the ESIM. In addition, the Communicator determines the sequencing of the flight computers during multi-computer simulations, and initiates ESIM updates whenever necessary. The

Communicator generates text output for the FSIM programs and files their data for graphical output.

During a simulation, the Communicator processes all interactions between a simulated flight computer and any external hardware or software. These interactions include requests to input data from, or output data to, another computer. Past and present values of data are also stored by the Communicator to ensure that computer read requests are satisfied by data appropriate to the time of the read. The Communicator also coordinates output commands to the ESIM and requests for sensor data from the ESIM. Whenever possible, a sensor-read request is satisfied by means of an extrapolation formula termed a recipe. These recipes are used in lieu of calling the ESIM to update at every read request. Only if the recipe has expired or has been invalidated is the ESIM called to update. Use of extrapolation formulas effectively decouples the simulations of the flight computers and the environment—which have basically different incremental time steps—and greatly improves the efficiency of the simulator by minimizing the number of control transfers between the flight computers and the ESIM.

External-environment Simulator (ESIM)

The External-environment Simulator (ESIM) comprises a package of semi-autonomous- compiler-language subroutines controlled by an executive routine (ENVCNTRL). The entire package represents the SHUTTLE vehicle hardware and flight environment within which the flight computers operate. Using mathematical models, each subroutine simulates a relatively independent portion of the environment; for example, the inertial reference unit hardware, the SHUTTLE vehicle, the radar systems, pilot actions, and the Earth's gravitational field and atmosphere.

While executing flight software, the computer simulators generate such control stimuli as vehicle orientation commands, moding discretes, and throttle commands. The ESIM closes the control loop by responding dynamically to these stimuli, thereby producing the inputs to the simulated flight computers. In addition, the ESIM is capable of responding to a wide variety of non-computer-generated external events. These events can represent hardware malfunctions or any other event that can provide insight into flight software behavior. Finally, the ESIM generates a history of the response of the simulated vehicle hardware to flight software commands.

4.3 INTERPRETIVE-COMPUTER SIMULATOR (ICS)

The Interpretive-computer Simulator (ICS) performs two functions. It simulates, on a host computer, the execution of code by a flight computer at the instruction level, and it performs diagnostics on the state of the computer being simulated—without affecting that state. It can also perform diagnostics on its own operations. The ICS is written in an assembly language—rather than a compiler language—in order to generate more efficient code since it is executed more than any other program in the PANSIM.

The ICS consists of the three major parts shown in Figure 4-2, the loader, the instruction-execution routines, and the diagnostic routines.

4.3.1 Loader

The loader performs four major functions for the simulation:

- Initializes the ICS
- Interprets flags that the flight-program compiler has created for dynamic verification
- Preprocesses flight code to minimize overhead for instruction processing
- Allows for an unlimited number of special requests to be attached to any location of flight-computer memory.

These functions are described in detail below.

Upon entry, the loader calls the MARSROT system (see subsection 4.8) to complete initialization and then calls the special-request processor to generate the tables of translated ICS special requests. The loader then (1) performs the immediate-action special requests, (2) chains together sets of special requests attached to the same locations, and (3) initializes flight-code variables according to STE specification.

The loader next performs its primary function which is to preprocess flight-computer machine code such that interpretive-instruction breakdown is performed only once wherever possible. Flight-code locations flagged by the compiler as instructions are changed to the host-computer addresses of their respective instruction-execution routines followed by the host-computer addresses of their operands. Those instructions with an attached special request or special-request chain are preprocessed differently. They are changed to the address of the special-request entry, or first of a chain of entries, in the table of special requests, followed by

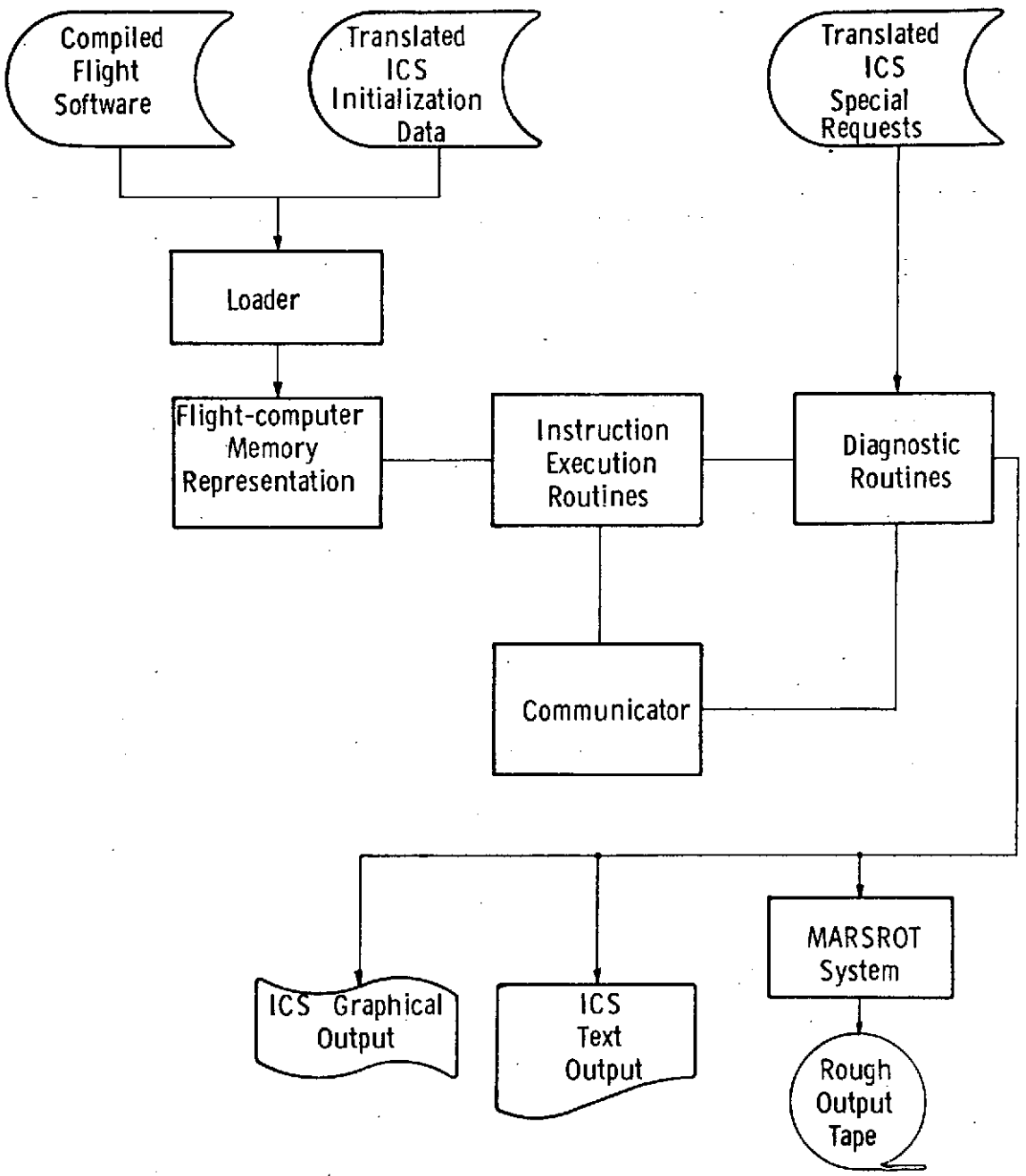


Figure 4-2. Interpretive-computer Simulator (ICS)

the original machine code for the instruction. At execution time, the instruction is decoded in the manner described previously. It is important to note that this expansion of flight code into a series of addresses must be done proportionately. If, for example, the expansion is two to one, i.e., a 16-bit-flight-code word becomes a 32-bit string of addresses in the host computer, then translating any flight-computer address to a host-machine address requires multiplying the flight-computer address by two and adding it to the host-machine address of the beginning of the flight code. If the expansion were not always two to one, then translation of addresses, and therefore branching, would become extremely difficult. For this reason, instructions with attached special requests cannot merely have the address of the special-request entry tacked onto the normal expansion. Instructions that have special requests attached to them, however, are very few; they are the only instructions requiring more than one decoding process.

Flight code flagged by the compiler as data (constants and variables) is also preprocessed, and special requests are attached in a similar manner. The data word, itself, is preprocessed so that an attempt to execute data terminates the run and triggers a diagnostic edit dump. If this safety check is overridden by the STE, the data word is interpreted and executed as if it were an instruction.

The output of the loader is translated flight code ready for fast execution. This output is divided so that paging from the host computer to direct access can be done resulting in a much smaller reserved host computer core size during the simulation.

4.3.2 Instruction-execution Routines

The ICS has two types of instruction routines, I/O instructions and regular instructions. This distinction is made because, in general, I/O instructions have timing and priority algorithms associated with them. The beginning and completion of an I/O instruction is dependent upon the current state of the computer, including the priority of other I/O occuring simultaneously. Since I/O activity interacts with the ESIM, FSIMs, and data busses, a call to the Communicator must be made for any I/O operation. The other instruction routines are written to be as efficient as possible and to take full advantage of the preprocessing performed by the loader. No attempt is made to simulate the microcode of the flight computer; but at the completion of each instruction, the state of the simulator is the same as the state of the actual flight computer. These non-I/O instruction routines are isolated and form a package that could benefit from microprogramming or hardware synthetic processing.

At the end of each instruction, simulation time is incremented and compared to the time of the next timed event, for example, a timed special request. If the event time has been reached or exceeded, the event is performed immediately.

4.3.3 Diagnostic Routines

The functions of the diagnostic routines are (1) to generate diagnostic information, (2) to modify variables in the simulation, and (3) to dump variables onto the MARSROT tape for later editing. Diagnostic requests triggered by the ICS can interact with the ESIM and FSIMs. For example, ESIM modeling of fuel slosh can be turned off for reasons of greater efficiency after the DFCS program in the flight computer terminates.

At termination, the ICS automatically edits the state of those portions of flight-program memory flagged as data and the status of interrupts. If the simulation ended because of a flight-code problem, additional diagnostics information is generated to ascertain the state of the flight computer. This includes, for example, a list of the jobs waiting to be executed and a list of the values of important registers and clocks. If the flight software maintains a chain of program status words, then a backward job chain will be included in the edit.

4.4 FUNCTIONAL-COMPUTER SIMULATOR (FSIM)

A functional-computer simulator (FSIM) is defined as a set of programs, written in a high-order language (HOL) and executed on a general-purpose computer, which models the functions performed by the software modules of a flight computer. It can be run in an open-loop fashion, or in closed-loop in conjunction with a simulation of the flight computer's external environment. The structure of such a simulator is shown in Figure 4-3.

A FSIM for the SWDVS can be implemented most efficiently if it is written in the same high-order language used to generate source code for all SHUTTLE flight computers. This permits the use of one set of source statements for the ICS, FSIM and actual flight computer. One code generator of the HOL compiler generates flight-computer code for the flight computer and the ICS. In addition, it generates the descriptive information about the code necessary for ICS dynamic diagnostics. Another code generator of the HOL compiler generates SWDVS host-machine code from the same source statements so that the instructions can be executed on the host machine directly rather than interpretively; the execution of this code constitutes a functional simulation of the flight-computer program.

4.4.1 FSIM Applications

The FSIM plays an important role in both the design and verification tasks within the flight software development process:

1. Design task — at this stage of program development, systems analysts are concerned with transforming software requirements and formulations into a program layout, including the segmentation of program functions into modules and their interfaces. Systems and applications programmers use the program layout as the basis for the program specifications from which the program is coded. They are concerned with such general problems as timing between subroutines, interface compatability, and suitability of algorithm mechanizations. They need a simulation tool that allows them to code and execute modules quickly and easily, and hence should find the FSIM more suitable to their requirements than an ICS.

Actual execution of flight program modules within the FSIM is greatly simplified by the inclusion of the runtime package. I/O requirements are satisfied easily by the runtime package alone or, in the case of simulations that include the ESIM, by interfacing the runtime package with the Communicator. In this simulation mode, each computer's external I/O requests are transmitted via the runtime package to

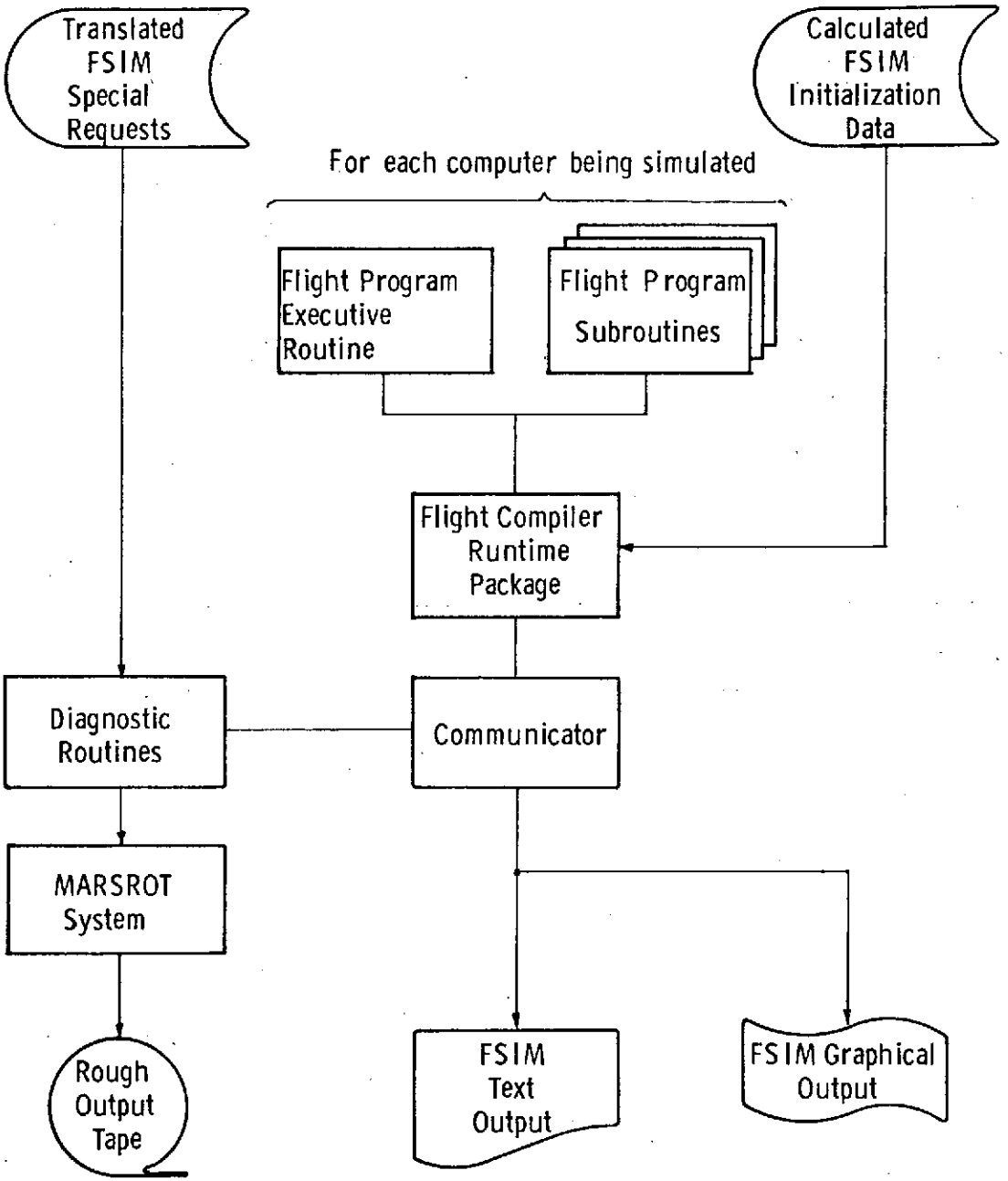


Figure 4-3. Functional-computer Simulator (FSIM)

the Communicator for processing. (For a description of how the Communicator responds to various types of flight computer external I/O activity, see paragraph 4.6.1.) The runtime package is also capable of performing a wide variety of diagnostic functions for the simulated program modules, such as tracing and monitoring (See subsection 4.1 for a description of the diagnostic capabilities of the runtime package.)

Initialization of FSIM programs is simplified since program constants and certain initialization data, such as control-system gains, are specified at compile time via COMPOOL declarations; these data can be overridden at simulation runtime, however, by the use of FSIM special requests. (These are identical to the ones used by that computer's ICS.) Special requests are also used to specify the remaining FSIM initialization parameters, such as the computer's representation of the vehicle state vector, which are run-dependent and therefore obtained from the Test Data File. Requests for text and graphical outputs from the program modules are handled by the Communicator which locates the data to be output by using the variables and system constants reference map generated by the compiler.

2. Verification task — to properly verify the performance of the flight software for one flight computer by means of an ICS, the STE needs a model of the other computers interacting with the flight software being tested. PANSIM can provide this model in three ways:

a. As open-loop specifications by the STE, using the SUPERCREW program (See the GREMLIN feature of SUPERCREW in subsection 4.10).

b. By treating the other computers as though they were closed-loop sensors, and incorporating them into the ESIM (An air-data computer, for example, would be amenable to this type of modeling.)

c. By using an FSIM to represent interactions between computers.

Although the first two techniques listed above can be used to solve the computer modeling problem for certain types of flight computers and for a limited set of test requirements, an FSIM is the most general and powerful solution, primarily because it provides the most comprehensive model of the other computers' flight software. An FSIM is sufficiently flexible, nonetheless, to allow a range of fidelities for each flight program being simulated. To increase simulator runtime efficiency, simplified versions of the flight software can be employed which, although written in the HOL of the flight computers, are intended for simulation use only. These programs represent baseline configurations of their respective flight programs; they are

functionally correct, but do not contain the level of detail of the actual flight programs. For example, a simplified version of the program for the DFCS computer need not contain logic to detect and isolate failed RCS jets. Such programs can remain relatively static since they need not reflect the many revisions being made to each flight program during its development cycle, as long as the revisions do not affect the functional characteristics being simulated.

There is a limit to the use of simplified flight program models in the verification process, however, since they may not be able to provide the level of accuracy required for certain types of program interactions. More important, management may require that specific revisions of each flight program simulated by the FSIM be used rather than generic versions. This requirement implies that, rather than remaining relatively static, the FSIM would have to be continually updated as the various flight programs are revised. If the FSIM is to be a viable tool under these circumstances, it must be capable of incorporating new flight program revisions quickly and automatically, and in a manner that is highly visible to the STEs who perform the verification testing.

The above requirements can be met if the FSIM programs consist of the same source statements that are written for the flight computers. Each flight software revision is then automatically available to the SWDVS for compilation into host computer machine code and inclusion in the FSIM. The SWDVS mechanisms for maintaining configuration control of source listings is thus applied to the FSIM source listings to ensure that any desired flight software revision is available for functional simulation.

4.4.2 FSIM Implementation

The concept of functional simulation discussed in the previous section is most easily implemented if the entire flight program for a given flight computer is written in the high-order language. If, however, certain parts of the program, such as the executive routine, are written directly in the machine language of the flight computer, the process of implementing the program in the FSIM becomes somewhat more complicated. When the source code is machine dependent and need exist only for the flight computer, a useful approach is to interpretively execute the assembled flight program modules in the FSIM, rather than assemble it for the FSIM's host computer as well. A trade-off exists here between the duplication of effort implied by two different assemblies and the reduction in runtime efficiency that may result from interpretively executing code in the FSIM.

For other code modules not written in the HOL, no such trade-off exists, since they would be written in machine language for both the flight computer and the host computer. This category includes well-defined, built-in functions that are either mathematical in nature—such as sine and cosine routines—or are needed to implement a primitive function in the HOL—such as the SCHEDULE or the TERMINATE statement. These functions must exist for each machine that can run a program written originally in the HOL; the FSIM could therefore employ the host-computer versions directly, since they are functionally identical to the flight computer versions. This procedure takes advantage of a necessary duplication of effort.

4.4.3 Use of MARSROT System and Special Requests with FSIM

The snapshot-rollback and special-request features normally controlled by the ICS portion of the PANSIM are also available during functional simulations without the ICS, although a slightly different organizational structure is required in order to perform the ICS functions related to the MARSROT system. During an interpretative simulation, the ICS is responsible for processing special requests, and interacts with the MARSROT system as necessary to execute dump and shapshot requests; it requires the Communicator to process only those special requests that affect another part of PANSIM. In the functional simulation alone, a diagnostic package is directly responsible for processing special requests; it informs the Communicator, in advance, of the time of the next special request, and is called at that time to execute it. These procedural differences are transparent to the software analyst, and allow the FSIM and ICS to use the same predefined data sets of diagnostic special requests, with the FSIM ignoring those special requests that are valid only during an interpretative simulation.

4.5 EXTERNAL-ENVIRONMENT SIMULATOR (ESIM)

The External-environment Simulator (ESIM) of the SWDVS includes a wide variety of simulated effects, ranging from crew actions, to hardware-switching delays, to gravitational fields. Despite this variety, the function of the ESIM is basically cohesive:

1. It closes the loop between flight-computer commands and flight-computer sensor inputs;
2. It responds to non-computer-generated "external events" of a wide variety;
3. It generates a record of the simulated vehicle's performance under the control of the flight computer design being tested.

The ESIM shown in Figure 4-4 exists as a large collection of programmed, deterministic, mathematical models, the combined "solution" of which fixes the vehicle state, and hence the sensor inputs to the flight computers. This solution depends upon a set of initial conditions, a knowledge of all discontinuities (flight-computer control commands plus external events), a choice of the particular models to be used in any simulation run, and a fairly massive amount of numerical data (mass properties of the vehicle, characteristics of the atmosphere, measurement errors associated with the rendezvous radar system etc.). Naturally, the sequencing of the execution of all models within the ESIM is important (the history of dynamics up to vehicle time T, for instance, must be found before the IMU output at time T can be calculated). Also, the management of data (both raw and calculated) becomes a significant problem. The remainder of this section deals with the details of the ESIM organization and the performance of its individual pieces.

4.5.1 Organization

In designing an ESIM for SPACE SHUTTLE, it was sensed that a simple extension of the philosophies and techniques successfully used in Apollo would be effective only if the resulting coding were adequately transparent.

Unless the structural design is properly conceived, programmers cannot approach the models on the necessary one-at-a-time basis. ESIM models are required to be functionally interdependent, and it is unfortunately convenient to program them in a homogenized, non-modular form. The lack of transparency in such a situation would be intolerable for the ESIM. Modularity must be tightly enforced.

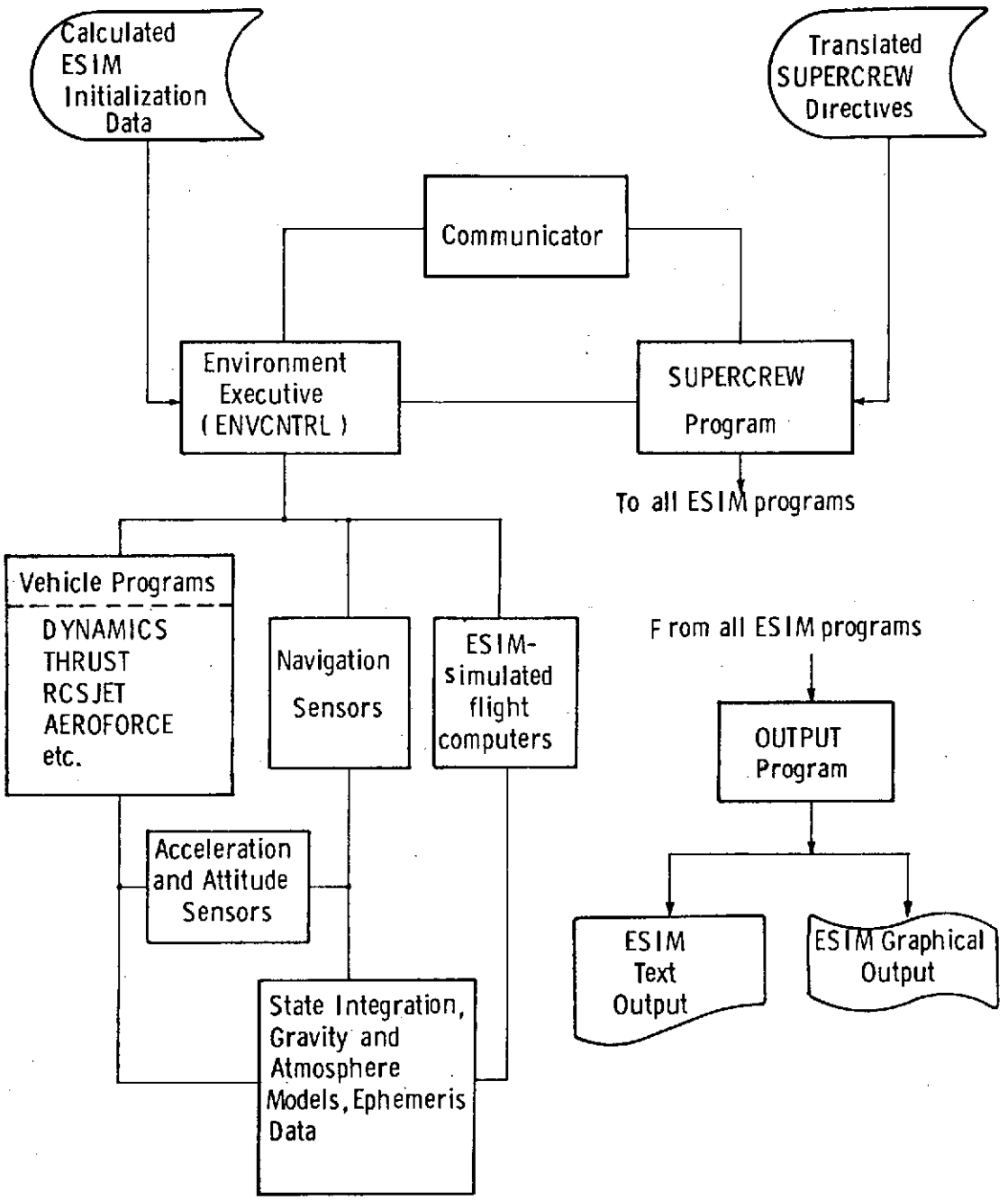


Figure 4-4. External-environment Simulator (ESIM)

Such a restriction was not necessary in previous projects only because they were of short duration (simulation facility personnel changes were minimal) and because the flights were few and highly structured (flexibility requirements were low). In the case of the SPACE SHUTTLE, there will be many flights over many years, and many variations in hardware, payload, flight plan, and flight-computer design must be anticipated. Therefore, the ESIM must be highly flexible, and transparent enough to be easily maintained and modified by new personnel:

1. Wherever possible, each "model" will be placed in its own program, complete with its own data I/O and initialization logic, its own documentation, and its own data file. The number and degree of specialization of programs will greatly increase relative to Apollo, while their individual complexity and length will be greatly reduced.
2. The interface structure between ESIM programs must be defined with modularity in mind. Data will pass automatically by means of common areas, and it will be unnecessary for a program to "know" how its input data are calculated, or why its output data are needed.
3. Programs will reside in a hierarchy that is dependent upon the particular simulation run, or even the particular run situation.

The third point is difficult to describe in general terms. A specific, but oversimplified example, will serve to illustrate the principle:

> Assume that ENVCNTRL calls the VEHICLE to update. The VEHICLE must first determine the contact forces acting on it; it does so by calling either RCSJET or AEROFORCE, depending upon whether the flight regime is orbital or atmospheric. For orbital flight, RCSJET, in turn, may call the FUELSTATE program; for atmospheric flight, AEROFORCE may call ATMOSPHR and ELEVONS. Eventually, in this manner, all the appropriate models for the specific flight situation are contacted, and data begin to flow back upwards through the program hierarchy. Ultimately, the VEHICLE program receives the contact forces it requested. It then calls one of the GRAVITY routines (the routine called depends on the level of accuracy required for the run), and a MASSPROP program to determine the current center of gravity, mass, and inertia of the vehicle. When all necessary information has been assembled, VEHICLE calls its DYNAMICS subroutine to perform the actual integration of the equations of motion, which will advance the state of the vehicle a small distance in time. Eventually, after the process is repeated the required number of times, control returns to ENVCNTRL, with the requested VEHICLE update accomplished.

One should note several points about this example:

1. It is not intended to be completely accurate, but is an illustration of general organization only. The actual choice of programs for the ESIM cannot realistically be made until programming actually begins. Even then, the choice will be far from final. New models, or variations on models, may be added at any time and old models may be deleted or combined.
2. While ENVCNTRL, as in past simulators, remains an executive for the entire ESIM, individual programs within the ESIM can act as executives over still lower-level ESIM routines. This will not confuse ENVCNTRL, for it is aware only of those programs it directly calls or those to which it directly responds.
3. This organization is ideal from the point of view of efficient programming, debugging, and maintenance of the ESIM, but penalizes run speed. The actual tradeoff point for maximizing total facility efficiency remains to be determined.
4. The complex task of specifying the correct set of ESIM models for a given simulation is handled by the use of prestored data sets containing the necessary specifications.

4.5.2 Environment Executive Program (ENVCNTRL)

The Environment Executive Program (ENVCNTRL) provides an interface between the Communicator and the External Environment programs. It aids in the ESIM initialization process, and is responsible for program sequencing during an ESIM update. ENVCNTRL also performs auxiliary functions related to simulation diagnostics and run termination summaries.

ENVCNTRL is the first program to be executed during a simulation. It is responsible for initializing the various ESIM common areas, either with data contained within itself, or with data placed in the data file during the preprocessing phase of the simulation. The latter data include information about the various ESIM model options to be used in the simulation, and the initial ESIM state (initial time, vehicle position, velocity, attitude, angular velocity, etc.). ENVCNTRL calls each ESIM program to initialize, and then calls the Communicator to initialize the flight-computer simulators.

During the simulation, ENVCNTRL is responsible for performing ESIM updates when called by the Communicator. It notes the time to which the ESIM is to be updated (TCALL) and the reason for the update (REASON). Update reasons include such

things as requests for the output of sensors whose extrapolation recipes have expired, return requests previously issued by ESIM programs, and dumping of selected simulation variables onto the Rough Output Tape. ENVCNTRL first reads all commands to the ESIM from the master command list, determines which programs should respond to each, and places them in the appropriate programs' command lists.

Once all commands have been processed, ENVCNTRL proceeds to update the ESIM to TCALL in accordance with the reason for the update. Some reasons, such as an ENVSUM (environment summary) special request, may require an update of the entire ESIM; others, such as a sensor read request, require only that the specific program which supplies the required sensor output be updated. In the latter case, other ESIM programs will be called only as necessary to supply data to the primary program being updated.

ENVCNTRL also provides the following auxiliary functions:

1. Debug options—for purposes of checking out the sequencing of programs within the ESIM, as well as the logical interaction between the simulated flight computers and the ESIM; one or more of the following optional features may be invoked by the STE during a simulation:

    a. Output of the simulation time and program name each time ENVCNTRL calls an ESIM program.
    b. Output of all commands in the master command list each time ENVCNTRL is called by the Communicator.
    c. Output of the recipe and invalidation time whenever a command to the ESIM causes recipe invalidation.
    d. Output of TCALL and REASON each time ENVCNTRL is called by the Communicator.

2. Run termination edit—at the end of a simulation, ENVCNTRL automatically generates a summary of the performance of the ESIM containing the following items:

    a. A sequence of major discrete events during the simulation.
    b. All occurences of off-nominal or illegal activities, including any "redline" violations.
    c. The performance of variables subject to monitoring during the simulation, including such items as a summary of reaction control system jet data and thermal protection system data during entry.

3. Run efficiency and management function monitor—for each simulation, ENVCNTRL automatically generates data on run-time efficiency and stores the information in the simulator efficiency data set. It also stores management-related information for each run to help provide management with a condensed overview of work progress and problems.

4.5.3 Models

General discussion of ESIM models is difficult because they differ so widely in function, complexity, and importance.* The basic approach used in generating ESIM models, however, is easily stated:

1. On the basis of engineering analysis and judgement, compile a list of all hardware or natural effects which might directly or indirectly influence the performance of the real vehicle;
2. For each entry on this list, find or create a mathematical model of sufficient accuracy to verify the flight software;
3. Translate this model into executable code, and imbed it in the ESIM, allowing it to share critical computed data with other models as necessary.

The key phrase is "of sufficient accuracy" (Step 2). On its interpretation hinges much of the value and computational expense of the ESIM portion of the simulator. Examples of real decisions affecting model accuracy (or more properly, inaccuracy) in previous environments include:

1. Modeling the thrust of an RCS jet as a simple delayed step rather than a complex dynamic curve;
2. Using a linear model for fuel slosh instead of a non-linear one, or neglecting slosh effects entirely;
3. Neglecting the small switching time of a hardware relay;
4. Ignoring solar or lunar gravitational effects near the Earth's surface.

It is usually not practical to use the most accurate model available for a simulator, because the required programming and computation time becomes excessive. A simple model of the Earth, for instance might assume:

---

*Even models of certain flight computers, such as the air-data computer, may be included in the ESIM as an efficient means of testing the interface between such relatively simple computers and the other flight computers.

Option I

A spherical geometry
A spherical gravitational potential
A constant rotation rate
An exponentially decaying, static atmosphere

While a more accurate and complex model might use:

Option II

The Fischer ellipsoid geometry.

A gravitational potential expressed as spherical harmonics with latitude and longitude dependance, and perturbed by the sun and moon positions for that date and time.

A time-varying rotation rate including the effects of polar axis precession and nutation.

The 1966 U.S. Standard Atmosphere Supplements dynamics atmosphere model, perturbed by random windgusts with specified statistical properties.

It is obvious that the latter model will be more costly computationally by several orders of magnitude.

Despite the expense of Option II, it will be necessary to include it for SPACE SHUTTLE, although certainly it would not see much use until the later stages of the project. Even then, the vast majority of STEs would not require it (or all of it) for their test programs. A provision must, therefore, be made to easily replace it partially or totally with Option I. One of the key subtleties in generating an ESIM, therefore, is to provide not merely a single model for each effect, but an interchangeable set, capable of providing a wide range of accuracies at widely varying costs.

ENVCNTRL is charged with the task of translating test engineer specifications into a selection of the particular models and model options the ESIM will use for any specific run. A considerable amount of effort must go into setting up prestored "packages" of model options which are self-consistent for various levels of accuracy.

Nominally, model accuracy is then selected at initialization time by a single input specifying the package desired. Test engineers are then able to amend this package in any manner they see fit, with some software checking to ensure that no gross incompatabilities occur. (High frequency vehicle bending, for instance, would be inappropriate in conjunction with a highly simplified model of the IMU, since its effect on the sensor outputs would be lost. Such a case would be pointed out during an interactive initialization. Other more severe inconsistencies would be prohibited entirely.)

Some limited provision should be made to allow changing model options part way into a run. This can be a costly feature, however, and it should be restricted to only those models where the returns warrant the expense.

In all cases, ENVCNTRL responds to the model selection process by generating an expanded list of the models to be used. This list is as complete and readable as possible to ensure clarity of a run's purpose.

4.5.4 Output from the ESIM

One main ESIM task is to provide an accurate, concise history of simulated vehicle performance. This history contains data of three main classes:

1. Discrete events
   (e.g., jet firings, hardware switchings, illegal computer commands, red-line crossings.)
2. Sampled data
   (e.g., angular velocity, acceleration, fuel consumed, altitude.)
3. Environment state summaries
   (e.g., total number of jet firings, maximum re-entry heating, records of off-nominal performance.)

It has been found convenient in the past to present this data in the following forms:

1. Discrete events:
   a. Messages keyed to a time and imbedded sequentially in the text output from a simulation run;
   b. Ordered collections of the above sorted by type into lists at the end of the text output. These allow users to quickly identify the major ESIM events of a run.

2. Sampled data:
   a. Annotated lists of key variables output periodically (or nearly so) during the run (e.g. every simulated half second);
   b. Same as a. but output at the occurence of specific discrete events (e.g., a jet firing).
   c. Scaled, annotated graphical output generated off-line from data periodically filed during the run.
   d. Text or graphical output resulting from post-run edits on such filed data. Such editing is often done by programs written by the STEs for specific test purposes.
3. Environment state summaries:
   a. Annotated text output of major events, running totals, or extrema of selected parameters for a simulation. These are usually the result of on-line edits, and are given at the end of the text output for a run.
   b. Similar to the above, but given periodically throughout the run.

Baseline requirements for any system generating such output are principally the capabilities to:

1. Access and output sampled variables on-line
2. Access and file sampled variables on-line for various off-line uses
3. Output on-line messages describing discrete events
4. Output ordered, sorted lists of such messages at run termination
5. Call on-line edit routines.

Additionally, other features are so useful as to be virtual requirements. The system must permit STEs to:

6. Control discrete message output so that they receive an indication of exactly what they want without being deluged;
7. Design, write, and command the calling of their own on-line edit routines;
8. Control the format of sampled data output with regard to:
   a. Heading, spacing
   b. Order
   c. Period between output
   d. Linkage to discrete events
   e. Units
   f. Coordinate systems
   g. Number of significant digits
   h. Start time for output
   i. Stop time for output

9. Do all of the above for an unlimited number of independant blocks of specified data of convenient sizes.
10. Control on-line filing of data in similarly powerful ways
11. Specify an exact filing period, possibly very small, so as to force ultrafrequent program updates. While very costly, such procedures are sometimes necessary to obtain smooth plots of high frequency variables.

Finally, in addition, the output system must:

12. Provide a sufficient number of packages, defaults and over-rides so that maximum use can be made of the flexibility provided. STE requests must be nominally simple (e.g. "use output package "ENTRY-16") but expandable enough to fill any real need;
13. Be compatible with CRT displays (e.g., 80-column, 40-line formats should be standard);
14. Be flexible enough so that new output features can be easily added as required by future circumstances.

An ESIM output mechanism has been envisioned which provides all the above capabilities, including the most difficult final one. The key concepts of its design are:

1. Assign virtually all output tasks to a single, special-purpose ESIM program, OUTPUT.
2. OUTPUT will maintain a set of "instruction matrices", which contain a full specification of all output requested by the STE, divided by class. The generation of these matrices will occur at initialization time, but they may be subsequently altered during the run as necessary.
3. When called, OUTPUT responds to these matrices as though they were a computer program written in a special language. It "executes" the matrices by routing through a large number of self-contained macro instructions using the matrix elements as data, flags, etc. These macros then perform such actions as filing, printing, changing units, adding headings, calling edits, etc.
4. OUTPUT is called to perform this execution by the other programs of the ESIM. They will call OUTPUT, however, only when a set of conditions is satisfied. OUTPUT itself is responsible for setting these conditions, and for updating them as the run progresses. It does this on the basis of the instruction matrices.

5. Only in the case of discrete message printing do the individual programs actually execute the output instruction. Even in this case, however, the conditions mentioned above must be first satisfied, and additionally could direct the individual program to take some special action. (e.g., calling the program that remembers messages for termination-time output.

6. The communication of data or of the conditions mentioned above between OUTPUT and the rest of the ESIM will occur through the extensive use of common areas.

7. The translation of user-supplied output requests (and default options, packages, etc.) into the instruction matrices can be fairly easily accomplished by requiring the STE to submit his requests in the form of "options" and "amendments".

   a. Options may be recursively defined as:
      i. An ordered list of one or more "primitive" output requests;
      or
      ii. an ordered list of one or more defined options, possibly with amendments.

   b. Amendments are defined as slight modifications to an option, and are designed specifically for interactive initialization efficiency.

8. The use of options and amendments allows an STE to easily design any format. Moreover, once designed, the output format can be given a unique name and stored as a new option. In this way, STEs can greatly influence the design of the input structure to suit their own needs.

4.6 COMMUNICATOR

The Communicator program provides the interface between the simulated flight computers and the ESIM. Its interface functions are divided into three major areas and are described in paragraphs 4.6.1, 4.6.2, and 4.6.3.

4.6.1 Inter-program Input/Output Monitoring

The Communicator coordinates the transfer of data among the simulated flight computers, as well as input-output activities between those programs and the ESIM. Data transfers are of three types: sensor-read requests, flight-computer output commands, and data transfer between computers.

1. Sensor-read requests—computer requests to read data from external sensors are processed by the Communicator, which attempts to supply the data by using an extrapolation formula, called a recipe, provided by the ESIM. If the recipe has expired, or has been invalidated by any change to the physical or logical state of the ESIM, the Communicator calls the ESIM to update the data. The update process also re-establishes the sensor recipe. The use of recipes reduces the frequency of ESIM updates, with a resulting increase in PANSIM efficiency.

2. Flight-computer output commands—commands to the ESIM (for example, sensor-moding discretes and engine on/off commands) are read by the Communicator and placed in a master command list for processing by the appropriate programs during the next ESIM update. If a command would cause a discontinuity in the state of the ESIM, and thus invalidate previously calculated recipes, the Communicator sets the invalidation time of the affected recipes to the time associated with that command.

3. Data transfer between computers—all flight computer outputs intended for another computer are passed to the Communicator by the ICS or the runtime package associated with the FSIM and stored in a common area. Previous values of the data are also stored. When a simulated flight computer requests data generated by another computer, the Communicator chooses the value stored in the common area that is valid at the time of the read request.

4.6.2 PANSIM Sequence Monitoring

The simulation of multiple flight computers is made possible by transforming the parallel operation of the actual computers into a serial progression from one simulated computer to another. During a simulation, each computer's code is executed

until an external interaction point is reached; these external interactions are any of the data-transfer activities discussed previously. Control is then returned to the Communicator, along with a statement of the computer's internal estimate of the incremental amount of simulation time that it has used in executing code. Scale factors are applied to this time estimate to allow for possible biases in the execution speed of the routines within a computer, relative to each other, or in the overall execution speed of the computer itself. The scaled amount of time is added to that computer's clock in the Communicator to indicate its position in time relative to the other computers being simulated.

With the return of control to the Communicator comes a reason for the return specifying the type of interaction to be performed. Because of the serial nature of the simulation, the interaction is not acted on immediately, since, in the case of a request to read data from another computer, for example, the desired data may not be current until the other computer is updated beyond the time of the read request. Instead, the Communicator saves the reason for the return and determines which simulated flight computer is to run next. The algorithm for choosing the next computer selects the one which is farthest behind relative to the other simulated computers. The Communicator first processes the reason for the previous return from that computer, and then calls it to run.

4.6.3 Events Monitoring

The Communicator's central position in the PANSIM allows it to monitor a list of auxiliary items to be acted upon at specified times. These items include:

1. Requests to output FSIM variables in text or graphical form.
2. The next special request to be executed (this function is performed by the Communicator during functional simulations only; if an ICS is included, it is responsible for monitoring and executing special requests).
3. Requests from ESIM programs to be returned to at specific times.
4. The time at which the simulation is to end.

These events are placed in a time-ordered-push-down list. As the time of each item in the list is reached, it is executed and deleted; additional events can be added to the list as the simulation progresses.

4.7 CONTROL OF SOURCE LISTINGS AND DOCUMENTATION

The extreme complexity of the SWDVS constitutes a major threat to its own successful implementation and represents a fundamental design constraint. Without adequate controls, the system could easily fail—not directly because of software failure, but because of accumulated inefficiencies and errors in the routine actions of the personnel managing, revising, and maintaining the system programs. An enormous amount of detail is contained in the SWDVS, and accuracy requirements are strict. It is therefore necessary for the SWDVS to have an adequate number of software aids designed to help in the tasks of program management, program revision, and program documentation. Although these aids are complicated in practice, they can be described in terms of two semi-autonomous programs:

LIPSVC—provides an efficient capability to recreate exactly any source compilation in the history of SWDVS development.

SNOOPY—provides an efficient, semi-automatic mechanism for inputting, storing, and easily retrieving symbolic data (generally other than source listings) useful to SWDVS personnel. The amount and kind of information stored is limited primarily by management decisions on requirements and cost effectiveness.

4.7.1 LIPSVC Program

A filing method known as LIPSVC (from List Processing Service and pronounced as lip service) has been developed for the storage, update, and retrieval of sequential source data in which the updates are considered to be small in size in relation to the data base as a whole. By using a list structure to thread the updates through the original data, LIPSVC avoids the generation of an almost duplicate data base with each revision, and yet allows access to both the original and updated data bases in their entireties. The data base is considered to be divided into members or subfiles, each of which may have more than one reference; updates under different references occur independently of one another. The price paid for these facilities is an increase in I/O time over that required by some standard filing systems. For the SWDVS this price is modest and reasonable.

The user places his source data in his LIPSVC file and assigns the data a name and a revision number. LIPSVC can handle a large number of such named collections referred to as "members" or "subfiles". For storage purposes, a LIPSVC file can be placed on tape, but it must be restored before being accessed again. LIPSVC can only process files stored on a direct access device.

When a user makes an update to a member, LIPSVC does not create a new copy of his file. Instead, LIPSVC adds the new records and the accompanying control statements to the user's file, and revises the file's list structure data, which LIPSVC uses to control the threading of the updates. The space required for the list structure data grows slowly in comparison to that required by standard systems that reproduce the member. Several attractive capabilities are byproducts of the copyless feature. Two of the more useful are the named version and the revision memory.

The named version capability permits the user to manipulate a member under another name without creating a copy of the original member and without interfering with the manipulations performed under the original name. In projects that involve several programming efforts on the same source data, named versions are particularly useful because they allow each project member to perform his own experimentation without risk of interfering with others. The number of named versions created is limited only by storage capacity.

The revision memory capability ensures that the user can retrieve, at any time, any revision of a member or any version of a member. This mechanism has one very important characteristic, that is, backup copies of the file are needed to recover from machine failures only, and not to recover from user failures to update properly. For improper user updates, the user requests LIPSVC to FORGET the bad update in a run subsequent to the one improperly updated. Although this capability greatly reduces backup requirements, eventually the inactive information reaches a stage where the user wants to retrieve only his latest revision of each member and recreate the file. Revision memory permits him to keep only his most recent backup copy of the file in order to have a retrievable record of all his work since he created the file. A short sequence of backup tapes can hold several years of developmental history.

4.7.2 SNOOPY System

Although functionally similar to some data retrieval systems common in the business world, SNOOPY represents an entirely new feature for systems such as the SWDVS, and approaches the current state-of-the-art limits for data management systems.

Rather than forcing hardcopy documentation and human memory to "keep track of" the state of the SWDVS,

> SNOOPY will maintain an evolving set of data files containing much basic descriptive information on past or current programs of the SWDVS. It will allow a manager, programmer, or STE to easily retrieve this

information merely by requesting it at a console. It will provide an equally convenient method of updating the files by adding new information to them. In short, SNOOPY will act as the system's collective memory for key data on the SWDVS itself.

To be workable at all, the SNOOPY program must provide highly reliable and highly complete information within its limited scope. For this reason,

> SNOOPY will "learn" new facts by directly examining the source code of programs. To ensure completeness, all SWDVS programs will be fed through SNOOPY automatically at compile time.

SNOOPY will be capable of scanning the symbolic source code of programs. It will be able to search out variable names, compile cross-reference lists, and perform similar tasks useful to programmers writing or debugging code.

More important, it will be capable of reading and remembering non-executable (but formatted) comments imbedded within the source listing. These comments will contain a large percentage of the descriptive information SNOOPY must maintain, and programmers will be required to keep them accurate to all program variations. In addition, SNOOPY will know and remember such things as the author of a program (or program revision), and the time and date of its (re)compilation. It can also be given additional information via consoles on program contents or function.

The function of the "information input" portion of the SNOOPY program is to recognize key symbolic data (using a set of conventions) imbedded in the source code or input directly from consoles or the operating system itself. It catalogues this information appropriately and stores it in its files. By allowing SNOOPY to interact with the LIPSVC files, however, it is possible to greatly reduce the amount of data SNOOPY actually has to remember. All that is necessary is to make sure SNOOPY knows which LIPSVC file contains the original source listing. Then, if required, it can call it up for re-examination. Another advantage of this interaction is that SNOOPY will be able to manipulate information not only on current simulator programs, but on all older versions as well. The historical development of a program thus is easily traced.

The "information output" portion of SNOOPY is somewhat more advanced than the input portion, and will take advantage of several recent developments in symbol-manipulation languages and heuristic processing. The difficult portion of any data retrieval system is not the retrieval mechanisms themselves, but the conventions by which the user specifies what he wants.

SNOOPY will respond to questions about the SWDVS by displaying annotated lists, pre-packaged text, or (partial) program source listings. To ensure an easy specification of the data required, SNOOPY will communicate directly with the user in limited-vocabulary simple-grammar English.

Programs capable of an interactive discourse in natural language have already been written, and while the technology remains embryonic, it is developing rapidly. Enough progress has been made at this time to allow SNOOPY to be written.

The key ideas imbedded in a natural language program are that

1. There is a defineable syntactic structure for all legitimate sentences in the language. (At first SNOOPY will recognize only very simple sentence structures.)
2. The "meaning" of any word in the language is programmed as an executable block of code, or "macro-instruction". (To keep the number of such blocks down, SNOOPY's vocabulary will be limited, at first, to only a few hundred words, e.g., "print", "program", "environment", "variable-name", etc.)
3. "Understanding" can be equated with an ordered (by the syntax) routing of execution through the individual word "meanings";
4. Semantic ambiguities are resolved by the use of heuristics, a limited "knowledge" of the subject matter, and/or a request to the user for further clarification;
5. The vast bulk of the subtlety of such programs is in their organization, which has been adequately developed for our purposes. The vast bulk of the effort in writing such programs, on the other hand, is programming the word "definitions". (By expanding the scope of SNOOPY's activities gradually, as needed, this effort can be held to a workable level.)

SNOOPY, therefore, will be an evolving program, whose value increases with time. Although it is presently unclear just which capabilities should be developed first, the following partial list gives an idea of what is considered appropriate:

The ability to identify and display any stored data on past or current programs which:

1. Were written by any specific programmer.
2. Were revised between any two dates.

3. Belong to any predefined class (e.g., "vehicle" programs).
4. Contain particular imbedded "keywords" (e.g., "MASSPROPERTIES").
5. Contain particular references to project documents (e.g., to last month's obsolete ODB).
6. Logical combinations of the above or similar features.

The data which might be displayed for such programs would include at the very least:

1. Their author.
2. The history of their development.
3. Packaged descriptions of their form, function, modeling assumptions, etc.
4. All or any specific references to project documentation.
5. Some or all other SWDVS programs sharing particular keywords (or other defined features) with them (for "impact-of-a-change" analysis, etc.).
6. Variable lists, under various organizations.
7. Common areas, subprograms used, exit points, etc.

In addition, there are several simple methods of subdividing full programs into logical, recursively imbedded sub-blocks of code. By giving these a programmer-assigned name, SNOOPY should be able to:

1. Treat these just as if they were actually full programs, handling these sub-blocks as described above.
2. Display the actual source code, scan it for particular variables, etc.
3. Reproduce packaged descriptions on the details of the coding, including model explanations, data files to be used, etc.
4. Display an annotated structure of the sub-blocks within a program.

Since SWDVS programmers, STEs, and managers would all have equal and rapid access to this information, little difficulty would be expected in keeping ll the data for SNOOPY current and accurate. A great deal of work is implied, but it must be done in any event. SNOOPY would allow this work to be divided naturally among virtually all programmers working on the Simulator. It would free them almost completely from the overhead task of coalescing it.

Naturally, the SNOOPY program will be very useful on those occasions when hardcopy documentation is required for extra-facility use. With just a few additional "word definitions", it should be possible to interactively design and edit hardcopy, and so obtain it quickly. It is expected that most of the hard documentation on the SWDVS Simulator will be generated with SNOOPY'S direct assistance.

Once a basic version of SNOOPY is available, its capabilities will be (relatively) easy to extend. As a central software depository for information about the SWDVS, its symbolic data base is potentially a very powerful foundation upon which to build other special-purpose routines. These can be designed to fill particular needs as such needs arise, and need not be fully anticipated at this time.

4.8 MARSROT SYSTEM

The MARSROT system performs three major functions for the PANSIM:

1. The dump-edit function writes the output of diagnostic special requests onto a rough-output tape and reads this information back at a later time for selective editing and graphical display.
2. The snapshot-rollback function gives the PANSIM a dynamic restart capability that guarantees bit-for-bit repeatability.
3. The tape-management function maintains a circular file of the rough-output tapes with clerical information concerning each tape.

4.8.1 Dump-edit Function

The rough-output tape provides a medium for storing the large amounts of data generated by diagnostic special requests during a simulation. The MARSROT system is responsible for writing the data onto the tape; it writes in the order in which the data is passed to it by the simulation. Since the resulting data set is not sorted by special request, the MARSROT system assigns a unique ID to each special request encountered during the simulation, and tags the output of each special request with the ID of that request. These IDs are transparent to the STE. When a post-run edit program edits the output of a specified set of special requests, the MARSROT system translates each special request into its ID number and retrieves the corresponding data from the rough-output tape.

4.8.2 Snapshot-rollback Function

There are two concepts of shapshot-rollback besides the one contained in the MARSROT system; they will be discussed before showing how the MARSROT system concept works.

The first concept is the brute force method: all the core and direct access used in the simulation is put on tape at specified intervals. The advantages of this method are (1) that it allows bit-for-bit repeatability of the simulation, (2) test runs are immediately restored after host-computer failures with controlled loss of computation time, and (3) it is commonly implemented by the host-machine operating system. Its disadvantages, however, are fairly severe. The rollback must be run on the same host machine and with the same operating system configuration. No changes are allowed at rollback except patches to host machine addresses from the operator's console. Finally, it can be very expensive for large systems. For instance, an Apollo all-digital simulation would require a 4-5 million byte brute force snap.

The second method of snapshot-rollback provides dynamic restart points by using a selective simulator state method. This method is commonly implemented by simulator systems; it is very inexpensive, and it can allow changes at rollback. Later runs, however, are not bit-for-bit repeatable since the program code is not stored.

The MARSROT system's snapshot-rollback provides both bit-for-bit repeatability and documented changes at rollback. The snapshots store sufficient simulation data for a rollback at regular intervals onto a test run's unique rough-output tape. The snapshot capability is implemented by first writing a complete copy of all the programs used during the simulation on the beginning of the tape. Then, snapshots are taken at a time interval specified by the STE. These snapshots consist of only the program variables that have changed since the last snapshot. The first snapshot taken contains any initialized program variables. Using this procedure, the average snapshot taken of the Apollo all-digital simulation is only 20-30,000 bytes in contrast to the 4-5 million byte figure mentioned for the brute-force method.

Rollbacks continue a test run in an exactly repeatable manner from any snapshot point on any tape in the library. The STE can continue the run on the same rough-output tape or use the option of branch-off rollback. When the branch-off rollback option is used, the MARSROT system copies the original rough-output tape onto a second one until the desired snapshot is reached. The run is then continued on the new rough-output tape. In this way, the STE can preserve the original simulation while making changes to it.

Rollbacks are implemented as follows. First, the programs on the beginning of the tape are copied into a temporary data set in an identical form as they were in the system before the first run. They are then loaded from this data set so that the operating system can take full charge of the actual loading of the programs and resolving of address constants. In this way, rollbacks are independent of the core region of the original simulation and even of the original host computer, as long as the host computer is of the same computer family with a compatible operating system. Next, the blocks of snapped variables are cumulatively rolled in overlaying each other until the desired snapshot is reached.

At rollback, the STE is allowed symbolic patching of flight-computer code, addition and deletion of diagnostic special requests, changing of ESIM variables, changing of flight-code variables, patching of ICS code, and recompilation of SUPERCREW program inputs. The initial special requests are reprinted at rollback and insertions and deletions are clearly marked. The advantages to the STE of the snapshot-rollback

system are that long tests, run in short segments with interim analysis of results, give identical results as one long run; diagnostics can be added via rollback to a simulation that has encountered a problem, with assurance that when the simulation is rerun the identical problem will be encountered: a rollback with only one variable changed is guaranteed to show the effect of only that change.

Figure 4-5 illustrates using the MARSROT snapshot-rollback system to run a simulation in steps. In the example, the STE wishes to simulate an engine burn sequence from 8475 to 9025 seconds with snapshots taken every 100 seconds. The STE inputs a MAXTIME of 8725 during the first run. Note that a snapshot is taken at the beginning of the run and also at MAXTIME. He examines the intermediate results of the simulation and finds that he wishes to add some diagnostics just after ENGINE ON. So the STE rolls back to 8575 (SNAP 2), adds diagnostic special requests, and changes MAXTIME to 9025. The run aborts, however, just after 8800 seconds due to a flight software problem. In order to preserve the conditions of the abort for diagnosis by the appropriate flight software system specialists, the STE performs a branch-off rollback to SNAP4 onto a new rough output tape. He adds more diagnostics and a flight software patch to work around the flight software problem. He also changes MAXTIME to 8525 to examine the results of his flight software patch before letting the run go to completion. The STE discovers that the patch worked to his satisfaction, so he rolls back to the last snapshot on the new rough output tape and completes the run. The final result is that the STE has two rough output tapes, one containing the simulation from 8475 to the abort and the other containing the simulation from 8475 to completion at 9025 via a software patch at 8775.

4.8.3 Tape-management Function

The MARSROT system automatically maintains a circular tape file (the Apollo file contained 1000 tapes) and a list of each tape's last use date, selecting the oldest used tape for each new run from those tapes not "sanctified" as permanent members. It "cleans up" any tape and its entry in the file not properly closed out by a previous job due to computer failure or PANSIM error. The MARSROT runtime system works as part of the transferable SWDVS structure and is not part of the operating system.

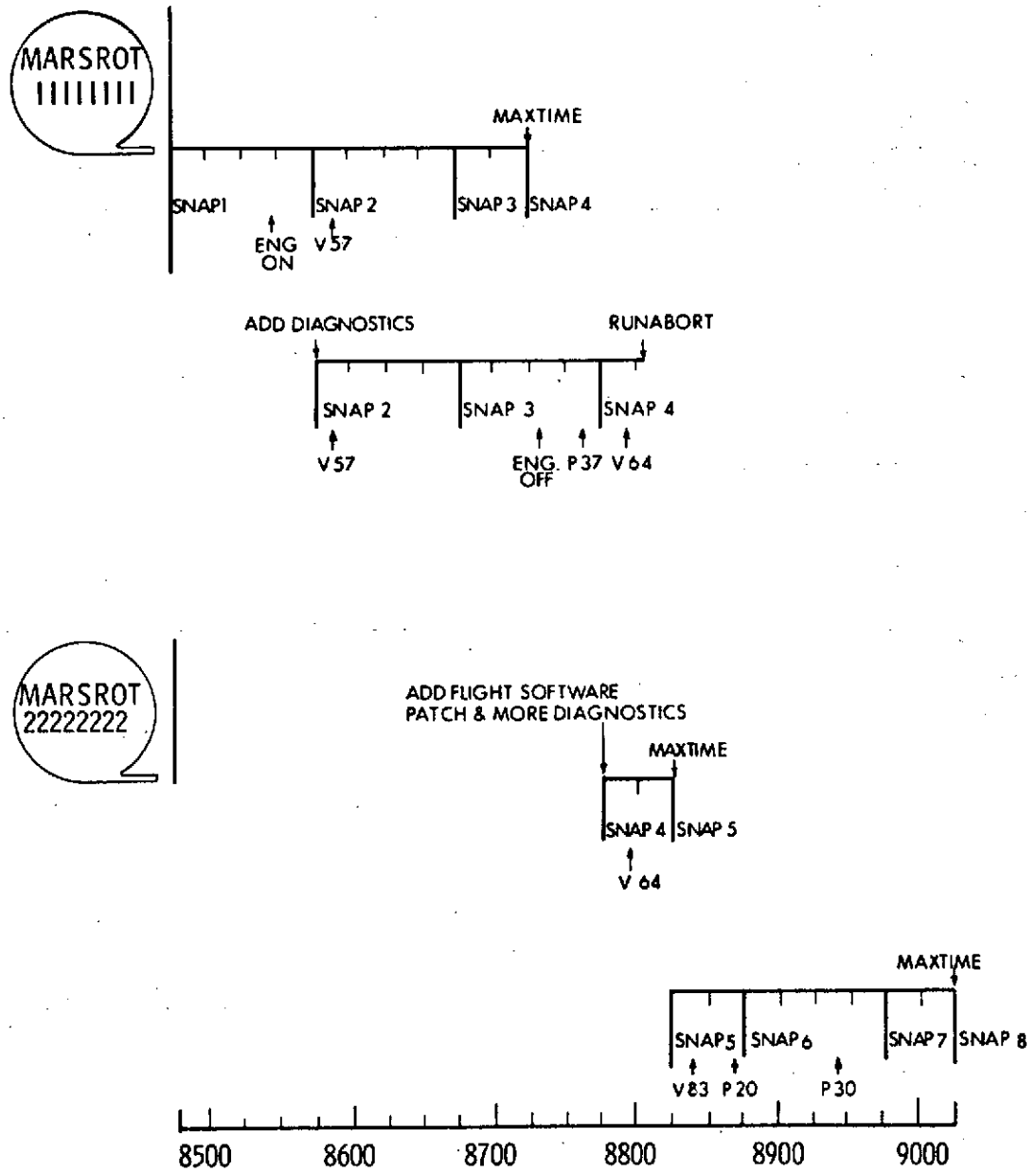


Figure 4-5. Using the MARSROT System to Run a Simulation in Steps

4.9 PANSIM INITIALIZATION

The PANSIM initialization scheme gives the STE a simple, yet highly flexible, means of constructing the input stream for any desired simulation. The STE has access to a wide variety of predefined initialization data sets, consisting of test cases from the official test plan, nominal flight plans, diagnostic and analysis packages, and ESIM input data. The STE chooses the data sets appropriate to his simulation and includes any modifications needed to meet his particular requirements. He carries out the entire process interactively via a display console, and also uses the console to submit the resulting simulation input for batch-mode processing.

Actual processing of the PANSIM input stream is performed by the XPANDER program and specialized processors shown in Figure 4-6. XPANDER expands, sorts, and structures the input stream submitted by the STE, and passes the resulting sets of data to the specialized processors for final processing. The SUPERCREW input processor translates commands to the SUPERCREW program into the necessary form for execution by the program. The SIMSETUP program accepts its inputs in a form that is readily understandable by the STE, and generates a consistent set of initialization data for the ICS, FSIM, and ESIM. The special-request processor translates the diagnostic special requests and initialization data and organizes the resulting information for later use by the PANSIM. The patch mode of the flight compiler is used in interpretive simulations to modify and recompile the flight computer program to be simulated. The output of these specialized processors yields a complete initialization, diagnostic, and control package for the PANSIM.

4.9.1 XPANDER Program

The XPANDER program simplifies the creation of input to the PANSIM by allowing the user to specify input to a simulation in a high-order language. Utilizing the methods described below, XPANDER translates this high-level input into the particular inputs required by the PANSIM.

XPANDER performs three operations on the input to the simulation: sorting, translating, and expanding. Before explaining how XPANDER performs these operations, it is important to note that the program knows the possible structures for an input stream to the PANSIM. It also has access to test plans, the library of initialization data sets, and the dictionary of correct input words. XPANDER views the STE-supplied input as a set of directions to fill in the selected input stream structure.

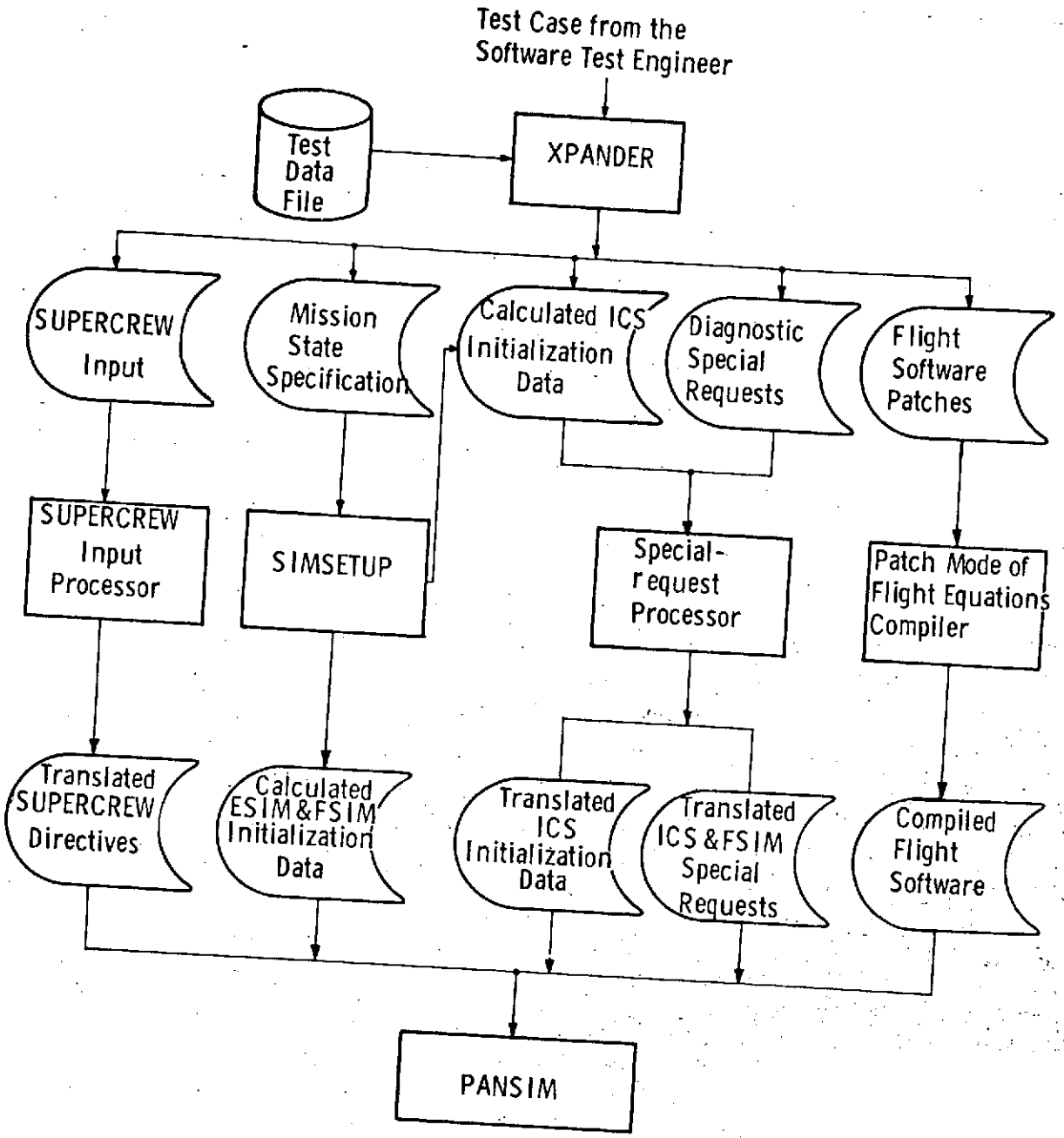


Figure 4-6. PANSIM Initialization

The various processors involved in a simulation, such as the special-request processor and the SUPERCREW input processor read their own input. XPANDER sorts the input stream to form a structure in which all the inputs to each processor are placed together in the proper order. Hence, the STE does not have to concern himself with presorting his input, and each initialization program finds its input at a single point within the input structure.

STE-supplied input to SIMSETUP is in a high order language. XPANDER translates this input into the proper format for SIMSETUP, which uses it to calculate the detailed input data for the ESIM, ISIM, and FSIM. The input to SIMSETUP is in the form NAME=VALUE. NAME appears in the dictionary of input code words and its definition is a location in a data set of inputs to SIMSETUP and the units expected by SIMSETUP. VALUE can be a code word (e.g., VEHICLE=STS127), a number followed by symbolic units (e.g. VEL=122 M/SEC.), or just a number (e.g. TIG=123).

If VALUE is a code word, it will appear in the dictionary and its definition will be a number meaningful to SIMSETUP. If value is a number followed by symbolic units XPANDER will convert the number from these indicated units to the units expected by SIMSETUP. If VALUE is just a number, XPANDER assumes it is already in the proper units. XPANDER places the translation of VALUE into the location in the SIMSETUP input data set defined by NAME.

When the STE includes a macro type input, known as a data set input, XPANDER locates the specified data set in the library chosen for the run and adds its contents to the input stream. XPANDER sorts and translates the input contained in each data set in the manner defined above. The data set feature allows the STE to create and store complex sets of PANSIM initialization inputs and to include an entire set via a single input to XPANDER.

4.9.2 Special-request Processor

In addition to specifying the flight program to be loaded into the ICS, the programmer may wish to set initial conditions, make temporary program modifications, and request special processing. This can be accomplished via inputs, which are referred to as special requests. During the initialization of the simulator, these inputs are processed by a set of routines referred to collectively as the Special-request Processor.

Special requests are used primarily to start or stop a run at specified locations, to initialize or modify the contents of selected locations in memory, and to select a

variety of diagnostic tools. The versatility and flexibility of these options are exemplified in the following list of frequently-used requests:

1. Start the simulation at a specifed flight computer instruction.
2. Stop or abort the simulation at a specified flight computer instruction.
3. Exit when the simulation exceeds a specified time.
4. Modify flight computer instructions or set contents of selected locations.
5. Request flight computer variable or ESIM state variable dumps.
6. Request basic or interpretive tracing.
7. Request periodic snapshots of host computer memory.
8. Advance the flight computer's clock.
9. Initiate hardware restarts.
10. Initiate Environment summary output.
11. Set or alter ESIM models.
12. Set initial contents of input-output channels.
13. Control text and graphical output of flight computer variables.
14. Control text and graphical output of ESIM variables.
15. Output the value of the flight computer clock.
16. Output a specified message.
17. Monitor specified ESIM variables during the simulation.

Special requests can be initiated upon accessing the memory location to which they are attached, or upon reaching a specified time. The requests can be made conditional upon the cumulative number of accesses of the location to which they are attached, the contents of a flight computer register, or a flight-computer time. More than one special request can specify a particular location or time if desired.

Initial conditions and software patches are incorporated directly into the flight computer memory. Special requests associated with a particular location cause a flag to be attached to the location in order to initiate special processing when the location is accessed. Time dependent special requests and conditional statements require more elaborate schemes to ensure their proper execution.

The previous discussion of special requests has described their use in conjunction with the ICS. The use of special requests, however, is not limited to interpretive simulations. They can be used during functional simulations as well, with certain restrictions. Since the concept of flight-computer memory addresses does not apply to a functional-computer simulation, special requests cannot refer to specific memory locations. Similarly, any special requests that operate on the contents of specified locations in flight-computer memory, such as traces, cannot be executed in the

functional mode. Those special requests that call for printing, plotting, dumping or monitoring of selected variables, and that specify the choice of ESIM models, however, are available during functional simulations.

The special-request processor performs the following functions:

1. It processes the special-request input cards; it interprets them, and organizes the information for later use by the simulator.

2. It outputs diagnostic messages for errors that can be detected during the initialization phase so that the STE can correct them and continue the run.

The special-request processor is designed with these objectives in mind:

1. It is as independent of the flight computer as possible, so that a large part of it can be used for several computers with a minimum of reprogramming. To modify the special-request processor for use with a computer simulator other than the one it is originally written for, a programmer has only to change those blocks of code that are clearly marked machine-dependent.

2. The program is divided into a series of logical blocks for clarity as well as for ease of alteration.

3. Symbols are used wherever possible—instead of numbers—so that the sizes of fields do not have to be fixed and can be changed easily without exhaustive searching through the coding.

4. It anticipates the addition or deletion of special requests by making its tables easily extendable.

4.9.3 SIMSETUP Program

SIMSETUP, written in a compiler language, is designed to be a generalized conversion program in the PANSIM initialization scheme. It transforms STE inputs into the initialization data required by both the ESIM and the simulated flight computers. The inputs to SIMSETUP are first processed by the XPANDER program which stores them as floating-point numbers in a data set named ENVDATA.

SIMSETUP generates a complete set of ESIM and flight-computer state variables from a generalized specification of time, and vehicle position, velocity, and orientation. The STE can input values of position and velocity with respect to a predefined rectangular coordinate system, in terms of circular or elliptical orbits, or, for simulations starting on the launch pad, by specifying the latitude, longitude, and altitude of the launch site. SIMSETUP can perform these calculations for both an active and passive vehicle (e.g. a satellite). If a time is specified for either vehicle that differs from the start time of the simulation, then that vehicle's state vector is extrapolated to the start time in order to obtain the correct initial state vector.

The state vectors generated for the environment and the flight computers are consistent with each other; however, the STE can specify error offsets, relative to the environment variables, for any flight-computer quantities. In addition, the STE can override any outputs generated by SIMSETUP. Any output to the ESIM or FSIM can be overridden by assigning a value to the same variable anywhere in the input stream, for example as part of the STE-supplied inputs or in a prestored data set. This will become the final value assigned to the variable, no matter what calculation SIMSETUP makes. An output to the ICS can be overridden by including a special request in the input stream which assigns a value to that same variable.

In addition to the input data required to generate the ESIM and flight-computer state variables, all other inputs to the ESIM and FSIM pass through SIMSETUP. These inputs include specifications of the various ESIM models to be employed during the simulation, and FSIM program parameters, such as autopilot gains or flagword settings.

4.9.4 Interactive Initialization

An interactive initialization capability can greatly improve the efficiency of the PANSIM by permitting STEs to examine the initialization phase of a simulation and make corrections before continuing the run. Many of the errors that occur during a simulation are related to faulty initialization. If STEs can stop their simulations after the initialization phase, examine the results, and make any corrections at that point, the number of runs and the amount of computer time required to perform a series of tests will be greatly reduced.

Interactive initializations are performed in the following manner: the STE generates the input for his simulation via a display terminal which gives him access to files of commonly used initialization data sets. He selects the data sets for his type of

simulation, and makes whatever modifications are necessary for his particular run by means of the terminal keyboard. The STE then submits his input to initialize the PANSIM. The initialization process takes less than two minutes, at which time the PANSIM takes a snapshot of the results and terminates the run. Output to the STE consists of any initialization error messages and a detailed listing of both his original inputs and the resulting initial state of the PANSIM. The STE views his output on the display screen, diagnoses any errors that have been detected, and makes the necessary corrections to the input. In addition, he can compare the initialized state of the PANSIM to his original input to ensure that any modifications he has made to the standard data sets have been done properly. The corrected input can then be submitted for another trial initialization or for running of the complete simulation via batch-mode processing. If the STE is satisfied with the original initialization results, he submits a run for batch processing which performs a rollback to the initialization snapshot and continues the simulation.

4.9.5 Test Case Specification and Simulation Interrelationship

Efficient and credible flight software verification requires coordination of the inputs that initialize all SHUTTLE simulation facilities. During APOLLO, this coordination was accomplished by means of an official test plan and data sets constructed from NASA-provided simulator data packages. Similar procedures are necessary for SHUTTLE software verification.

The official test plan consists of a series of customer-specified tests designed to ensure that the flight software satisfactorily meets its mission requirements. It lists the specific simulations to be run for each mission phase, including both nominal and off-nominal cases, the sequence of events to be performed during each simulation, and the figures of merit to be generated. The data sets contain initial values of flight-computer variables, crew sequences, mission profile reset points, and simulator initial conditions for those reset points. The collection of data sets provides a family of flight conditions from which simulations can be run.

To produce a wider range of simulation test cases, test engineers can generate additional data sets to be combined with the data sets from the official test plan. Testing for customer inspection, however, is performed only in accordance with the official test plan and uses the customer-provided data sets exclusively.

4.9.5.1 Test Plan Language

To ensure that the simulation test runs conform to the test plan, and to minimize initialization errors, the test plans should not only specify how the tests are to be

conducted, but should also provide the inputs necessary to initialize and control the simulation. The latter requirement can be met by using prestored initialization data sets in conjunction with a test plan format identical to the format of the PANSIM initialization input. Thus, a line in the test plan becomes an input to the simulation. Two types of inputs are available: data set lines and SUPERCREW input lines. The former accesses a prestored data set that contains all the information required to perform the action specified by that line in the test plan. The latter is translated by the SUPERCREW input processor into the detailed sequence of crew actions necessary to fulfill the test plan. At run time, the STE may supply additional inputs to override any information specified by the test plan; these overrides will explicitly appear as such in the run initialization output.

To retrieve a test case from the test plan source file, specify:

RETRIEVE TESTCASE LEVEL 6.ENTRY.2,REVISION 3

The test case is displayed on the STE's terminal as follows:

LEVEL 6 DEORBIT AND ENTRY.2, REV. 3;
CREATED 03/29/75 BY B. THOMAS; USE # 117

I. PURPOSE:
TO DEMONSTRATE DEORBIT AND ENTRY SEQUENCE

II. TEST DESCRIPTION:
THIS TEST SEQUENCE USES THE STS5 DATA FILE AND FLIGHT PROGRAM REVISION 57. THE INITIAL STATE VECTOR IS 5 NM OUT OF THE DESIRED TRAJECTORY PLANE.

INITIAL CONDITIONS:
RETRIEVE DATA FILE (STS5)
RETRIEVE DATA PAK (6.11.75)
RETRIEVE EDIT (DEORBIT, ENTRY)

TEST SEQUENCE:

TIME	ACTION	REMARKS
	SELECT (DEORBIT)	CALLS DEORBIT PROGRAM
	LOAD (RANGE.CROSS,5)	OVERRIDES NORMAL CROSS RANGE TARGET.

TIME	ACTION	REMARKS
EI-1H15	CHECKLIST (ENTRY)	BEGIN CHECKLIST AT ENTRY INTERFACE-1-1/4 HR.
TOUCH	TEST OVER	TERMINATE SIMULATION

To override initial conditions, the following types of statements can be added:

```
ERROR.STATE   = 3 SIGMA
ERROR.IMU2    = 1 SIGMA
ERROR.IMU3    = 2 SIGMA
ERROR.RADR    = 1 SIGMA
DISTURB.WIND  = MAX (1G)
DISTURB.COMP  = 10 PERCENT
```

To override test sequence specifications, the following types of statements can be added:

EI-1H15	GUID MODE (MANUAL)	SPECIFY MANUAL MODE
	SELECT (MAN GUIDE)	CALL PROGRAM TO FLY MANUAL LANDING.

In its input form, the test plan is maintained in the retrievable test-plan source file. A cross reference is made to the test results while the testing is being conducted, indicating the status of each test (date begun, date completed). Whenever the test plan changes, the test input is updated by changing the corresponding input lines.

The simulation is initiated via an interactive terminal by fetching the desired test case, in its input form, from the test plan source file. The test case data, plus any additional input supplied by the STE, form the input stream read by the XPANDER program. This program decodes the names of prestored data sets and adds the information contained in the sets to the other initialization data. Each test case data set contains remarks which specify test number and title, author, test objective, test description, LIPSVC data, and use number. It also contains the names of the ESIM programs to be used in this simulation, and the appropriate ESIM initialization parameters, for example, the various model fidelities to be used.

The test case input includes the test sequence to be performed during the simulation. Event times can be specified either in terms of Ground Elapsed Time or in terms of predefined mnemonics representing events whose times have previously been specified (e.g., LIFTOFF for nominal liftoff time). Times relative to these events can also be specified. In addition, mnemonics can be used for events whose times are not predetermined; these times are determined dynamically during the simulation by a MONITOR in the SUPERCREW program.

Test case entries under the heading "Action" in the above example invoke prestored data sets containing the detailed set of procedures required to perform each action.

These data sets include special requests to generate data related to the test events for output during the simulation or post-run editing. Other test case entries generate crew actions; for example, selecting specific flight-computer routines, entering data, responding to displays. Each detail of this input can be overriden; one reason for overriding would be to generate artificial situations as an aid to failure testing and the analysis of off-nominal system behavior.

The test case also prescribes the test report to be generated at the end of the simulation and stored in the test plan source file. Included are the test case number, title and author, the test objective and description, the test sequence actually performed (whether as a result of test case entries or of SUPERCREW inputs supplied at run time), a summary edit, and plots of selected run data.

4.10 SUPERCREW

The SUPERCREW (literally, beyond the crew) program provides control of a simulation by means of a conversational language oriented toward the use of the predefined Test Data File. SUPERCREW has four classes of activity:

a. crew actions
b. flight controllers monitoring the test at terminals
c. hardware failures
d. control of other directives to the simulators.

To achieve this control, SUPERCREW is conceived as an assortment of crew members who interact with the flight computer programs and avionics hardware, and with each other, according to a set of pre-defined conventions. A one-to-one correspondence does not exist between the members of SUPERCREW and the actual Space Shuttle flight crew. Instead the SUPERCREW members represent a convenient division of the various functions which must be performed in the course of exercising flight programs for software verification purposes. Included in SUPERCREW is the capability of simulating non-nominal and unexpected situations as well as normal crew activity.

Directives to SUPERCREW range from a general directive to execute a mission segment as specified by part of the Test Data File, to a detailed directive to change a switch position. These directives are input without reference to a specific crew member; SUPERCREW is responsible for interpreting them and assigning the crew members necessary to accomplish them.

Although the crew members actually represent specific functions within the SUPERCREW program, they are personified in this report to make their different activities more distinct. Hence, each capability of the program is described in terms of the hypothetical crew member responsible for providing that capability.

The SUPERCREW program represents this crew, all working simultaneously.

COMMANDER	Performs nominal mission segments defined in the Test Data File, by coordinating the actions of the OPERATOR, PILOT, NAVIGATOR, and TARGETEER.
OPERATOR	Engages the avionics system in a conversational manner via the onboard CRT.

PILOT	Maneuvers the vehicle with the handcontrollers.
NAVIGATOR	Maneuvers the optics and other navigation sensors.
TARGETEER	Provides guidance and targeting for the PILOT and NAVIGATOR.
SWITCHERS	Push buttons and flip switches in an open-loop manner.
REPORTER	Controls the output of SUPERCREW and the run termination diagnostics.
GREMLINS	Fail the hardware, insert spurious data onto the data bus, and change ESIM state.
MONITORS	Monitor events in the ESIM, FSIM, or ICS to schedule SUPERCREW activities, including other MONITORS. They are directed by the STE.
SPECIAL MONITORS	MGT.MONITOR and SWDVS.MONITOR - MONITORS as above, but directed by software managers and SWDVS personnel, respectively.

These crew members begin to perform their activities when triggered by a discrete event. These discrete events can be trapped by the ICS diagnostic package - special requests, the Communicator, the FSIM diagnostic package or the ESIM.

The class of discrete events which trigger a crew member to begin an activity are:

General Class	Apollo Example
Time	(Time)
Flight computer major alarm notice	(Program Alarm, Operator Error)
CRT requests attention	(DSKY FLASHING)
CRT display complete	(DSKY rewritten)
Flight software record of alarm notice	(Priority display bit set, alarm information stored)
Flight code accesses a specific memory location	(An AGC instruction or erasable location)

A description of the class of activities of each crew member follows.

COMMANDER

The COMMANDER has access to the predefined Test Data File which specifies, in part, the complete crew directives for each mission phase (see Figure 3-4). When a Test Case calls for a segment of the predefined Test Data File to be executed, the COMMANDER responds by coordinating the tasks of the OPERATOR, PILOT, NAVIGATOR, and TARGETEER. Two examples of directives to the COMMANDER follow.

Example 1: Perform a nominal mission program.

The input

```
PERFORM PROGRAM (X)(USING DATA (A,B,C))
      (UNTIL CONDITIONS (Y,Z));
```

yields the SUPERCREW directives to exercise flight computer program "X" using the predefined Test Data File procedures. The optional data (A,B,C) replace the default parameters (see example under PILOT), and the optional termination conditions (Y,Z) replace the default termination conditions.

Example 2: Establish a checkpoint (i.e., establish the predefined conditions associated with a point in a mission sequence).

The input

```
ESTABLISH CHECKPOINT GSOP14A.ENTRY.17;
```

yields the necessary directives to the COMMANDER to establish those conditions specified by the CHECKPOINT in the document GSOP14A for revision number 17 of the mission phase ENTRY. The CHECKPOINT consists of panel switch settings (e.g., SCSMODE AUTO) and initialization parameters for the flight computers (e.g., DAPBOOLS = (HIGHRATE, DOCKED)).

If the software test engineer specifies an individual task for the OPERATOR, the PILOT, the NAVIGATOR, or the TARGETEER, that task is performed instead of the corresponding directives for that crew member in the Test Data File. The functions performed by these four crew members are explained next.

OPERATOR

The OPERATOR interacts in a conversational manner with the onboard CRT and other hardware of the avionics system.

Test Data File directives or he may be directed by the STE. The major functions of the OPERATOR are:

a. LOAD flight program parameters via the CRT using appropriate crew procedures (such as LOAD, VERIFY, PROCEED) and using correct parameter scaling and engineering units (gleaned from the output of the flight code generation process).

b. Initiate a flight program.

c. Remember data displayed on the CRT for later use in conjunction with reporting Figures of Merit or logical decision making by other crew members.

Example 3: Load flight program parameters.

The input

LOAD VG WITH (100,0, - 100) METERS/SECOND;

yields the following OPERATOR sequence (using Apollo nomenclature):

VERB 25 NOUN 81 ENTER
+ 3048 ENTER +0 ENTER -3048 ENTER
WHEN DISPLAY FLASHES WAIT 4 SECONDS
THEN VERIFY VERB 06 NOUN 81
R1 = +3048 R2 = +0 R3 = -3048
THEN WAIT 2 SECONDS THEN PROCEED;

where VG has been translated into the decimal valued NOUN 81, scaled in deci-feet-per-second. This scaling information is part of the output of the flight code generation process for the flight program revision used in this test run.

Example 4: Initiate a Flight Program.

The input

INITIATE P22;

yields the sequence

VERB 37 NOUN 22 ENTER
VERIFY WITHIN (5) SECONDS THAT
MODE = 22 OTHERWISE RECYCLE (0)
TIMES BEFORE (30) SECONDS;

where the default numbers in brackets may be changed as part of the input (e.g., INITIATE P22 (WITHIN = 12).

Example 5: Remember a CRT display.

The input

REMEMBER NAMEX;

yields

WHEN CRT DISPLAY COMPLETE
REMEMBER AS NAMEX (NOUN, R1, R2, R3);

The portion of the CRT to be remembered may be specified by the software test engineer (e.g., REMEMBER NAMEX (R2, PROGRAM ALARM, IMU TEMP);).

PILOT

The PILOT maneuvers the vehicle in a manner that is responsive to the closed loop reaction between the handcontrollers, the Digital Flight Control System, and the vehicle motion.

The PILOT has the SUPERCREW's most sophisticated control algorithms at his disposal, because maneuvering the vehicle is a difficult response to model. But the vehicle control algorithms are simple control systems - not an attempt to model real human response. They benefit from perfect knowledge of the vehicle's state via the ESIM (a privilege rare to control systems).

The PILOT chooses the gains of his control system by looking at the current vehicle state and mass properties. These gains are subject to change by those software test engineers who are studying control.

The PILOT gets the default DFCS initialization data from the Test Data File.

Example 6: Maneuver the vehicle.

The input

MANEUVER PITCH +4 DEG/SEC;

yields the sequence

(VERIFY DFCS INITIALIZED, ELSE INITIALIZE)
(VERIFY RCS FUEL GREATER THAN (63) POUNDS)
(CHECKPOINT GSOP001.RCSMANEUVER.1)
HANDCONTROLLER PITCH +21 PULSES
VERIFY WITHIN (3) SECONDS RATE WITHIN (80) PERCENT
AVOIDING IMU GIMBAL LIMIT (85) DEGREES,
AUTOPILOT (RCSMANEUVER) WITH
RATEFEEDBACK (1) POSITION FEEDBACK (1)
ACCELERATION FEEDBACK (1);

where 21 PULSES is derived from the input 4 DEG/SEC because the Test Data File specifies that the DFCS will use the equation:

$$\text{RATE} = \text{FACTOR}\,(\text{ALPHA PULSES} + \text{BETA PULSES}^2).$$

All of the numbers in brackets are default values which may be overwritten by the input specification (e.g., MANEUVER PITCH + 4 DEG/SEC (RATE FEEDBACK = 0.5);).

NAVIGATOR

The NAVIGATOR maneuvers the navigation aids, such as the optics sextant, via control systems similar to the PILOT.

The NAVIGATOR maneuvers these navigation systems to line-of-sight targets calculated by the crew member TARGETEER. When the maneuver satisfies the

requirements of the TARGETEER, the NAVIGATOR MARKS. The NAVIGATOR adds a class of errors for each MARK or for each group of MARKS as specified. The class of errors includes:

Absolute marking errors (e.g., HORIZON DELTA = -3000 FEET)
Random error deviations (e.g., TRUNNION ERROR = 3 SIGMA)
Sensor biases, errors (e.g., SHAFT BIAS = -0.3 DEG)
Random errors in the sensor analog to digital converters (e.g.
CDU ERROR = 2 SIGMA).

The NAVIGATOR allows the software test engineer to input alterations to the control loops by which he maneuvers the navigation aids.

Example 7: Mark a star.

The input

```
MARK STAR (10) (OPTICS.TRUNNION.BIAS = -.5 DEG)
      (OPTICS SIGMA = 3) (CDU SIGMA = 1);
```

yields those SUPERCREW directives that cause the NAVIGATOR, the TARGETEER, and the PILOT to interact in a way that represents a crew member with one hand on the HAND CONTROLLER and one hand on the OPTICS CONTROLLER maneuvering the vehicle and the optics in order to mark a star.

TARGETEER

The TARGETEER performs targeting calculations for the NAVIGATOR and the PILOT. A target for PILOT could be the desired vehicle attitude matrix in preparation for the Entry mission phase. The TARGETEER would also direct the maneuver to reach the new attitude without causing the IMU to go into gimbal lock. A target for NAVIGATOR could be the shaft and trunnion angles to point the optics at a specified landmark. A complex directive to the TARGETEER would be to select two stars in the appropriate optics field of view which also satisfy certain geometric and analytic constraints.

Example 8: Select and mark a star superimposed on the horizon with the optics.

The input

MARK STAR X WITH NEAR HORIZON;

yields a selection by the TARGETEER of an appropriate vehicle attitude to which the PILOT maneuvers the vehicle. At this attitude the NAVIGATOR will find the star selected by the TARGETEER in the optics field of view.

SWITCHERS

There are many crew members named SWITCHER. Each one pushes buttons, flips switches, and operates handcontrollers in an open-loop manner. They are limited to actions which do not depend upon an avionics system response. (e.g., pushing the button PROCEED alone is an appropriate action for a SWITCHER, but PROCEED as part of a parameter LOAD is unique to the OPERATOR.)

A SWITCHER performs its actions in a reasonably timed manner unless unusual timing is specified by the software test engineer.

More than one SWITCHER may be active at a given time; for example, one may be pushing a button on one control panel while another is simultaneously flipping a switch on another panel.

Some avionics subsystems have constraints on the way that they are exercised. (e.g., TURN ON IMU includes a 90 second procedural delay.) The OPERATOR obeys these procedures and constraints and hence is used to carry out nominal avionics procedures. In contrast, the testing of abnormal procedures is performed with the aid of a SWITCHER, since he obeys directives without question.

Example 9: Action - push a button with normal timing.

The input

ERROR RESET;

yields

ERROR RESET DOWN WAIT 0.250 SECONDS
ERROR RESET UP WAIT 1.000 SECONDS;

Example 10: Action - push buttons with specified timing

The input

```
ERROR RESET DOWN MARK REJECT DOWN
WAIT 1 SECOND
ERROR RESET UP MARK REJECT UP
WAIT 2 SECONDS;
```

yields exactly the specified action - namely, the two buttons depressed simultaneously for 1 second.

REPORTER

The crew member REPORTER controls the output of SUPERCREW and can send output control commands to the OUTPUT program of the External Environment Simulator. The REPORTER can also trigger the run termination diagnostic edits. An STE may input directives to the REPORTER but default directives are derived from the other Test Case information. A mission performance test, for example, receives general overview output by default, and a unit test receives more detailed output by default. The REPORTER also has access to a set of diagnostic responses to some of the anomalous conditions discovered by other crew members.

Examples of Anomalous Conditions

CRT doesn't respond to OPERATOR
Vehicle doesn't respond to PILOT
Conditions are inappropriate for the COMMANDER to perform the specified mission sequence.
An action is inconsistent or redundant.

Examples of Diagnostic Responses

Record anomaly and continue test
Record event in management data set
Terminate test and request run termination edit.
When terminate-with-edit is appropriate, these choices exist:

Output the circular trace of the last 25 flight code branch instructions, if this feature is active.

- Trigger one or more aprropriate diagnostic run termination edits as defined in paragraph 4.5.2.
- Automatically submit a ROLLBACK test run with additional diagnostics.

(The auto ROLLBACK implies an extensive design effort to define, in advance, a useful set of diagnostics for a given error. However, it should be attempted and its results evaluated.)

Example 11: Specify a diagnostic ROLLBACK.

The input

```
WHEN PROGRAM ALARM = ON THEN ROLLBACK TRACE P40;
```

yields the directives to submit a ROLLBACK Test Case of this test run beginning at the last SNAPSHOT point when the CRT alarm called PROGRAM ALARM goes on.

Example 12: Specify less output.

The input

```
PRINT LEVEL 6;
```

yields the directive to limit the output of the SUPERCREW program to the output normal to a mission performance Test Case.

GREMLINS

The GREMLINS cause hardware to fail, spurious data to appear on the data bus, and discontinuities to occur in the ESIM state.

Example 13: Spurious data appears on a data bus.

The input

```
SPURIOUS TO G&N X'37FF';
```

would cause the hexadecimal bit string '37FF' to appear on the data bus to the G&N flight computer.

Example 14: Hardware discontinuity by GREMLIN.

The input

STEP LEFT AILERON UP 1 DEGREE

yields

ACTUATOR 21 POSITIVE 1 DEGREE.

MONITORS

There are many MONITORS during a simulation.

The MONITORS can represent the observations of all of the other crew members and can perform the tasks that would be performed by software test engineers at interactive terminals in a real time simulation facility. They can represent a software manager observing all test runs while looking for a special case or SWDVS personnel monitoring the use of the SWDVS.

A MONITOR samples the conditions of state of the PANSIM at intervals during a simulation. These samples differ from the discrete events that trigger crew members to begin an activity as discussed earlier. Such discrete events also trigger MONITORS. Sampling of the conditions of state is an appropriate way to monitor continuous functions such as fuel depletion, vehicle rate or distance to a satellite.

An interval range for MONITOR sampling is preferable to a fixed interval so that monitoring occurs when the ICS naturally requires an ESIM update to satisfy a flight computer input or output. The MONITOR then does not affect the run by forcing extra updates of the ESIM state.

The interval range between sampling can be specified by the STE who knows to what degree the function being monitored is continuous.

When a MONITOR finds the condition being monitored satisfied, his only activity is to schedule another crew member (possibly another MONITOR) who performs the actions specified.

The concept of scheduling another crew member immediately raises the question - What if he's busy? The crew member must first perform the task with higher priority, and if a low priority task is interrupted, must know how to return to it.

The functions of scheduling crew members, distinguishing tasks of different priority and the selecting of different crew members all fall within the category of functions performed by an executive program in a flight computer. It is not surprising that the SUPERCREW program which, in its own world, is a real time controller of the simulation, should require an executive structure similar to the flight computer software that, in real time, controls the spacecraft.

Asynchronous activity of different priorities will probably occur within SUPERCREW only during abnormal testing when the MONITORS detect unusual conditions. Nominal simulations will benefit from the knowledge that the monitoring took place without the occurrence of asynchronous scheduling and selection by the crew of various tasks.

Example 15: Terminate the test run with an edit of the Digital Flight Control System (DFCS) activity if vehicle rate of rotation is excessive.

The input

```
MONITOR RATE>5 DEG/SEC
THEN TERMINATE WITH EDIT=DFCS;
```

yields the directives to sample vehicle angular rotation rate until it exceeds 5 degrees/second, when the run will be terminated with the specified edit. Because a sampling interval wasn't specified, the default interval of 2 to 2.5 seconds will be used. Because conditions to stop the monitoring weren't specified, the entire run will be monitored.

Example 16: Turn off the SHUTTLE braking engine manually during a rendezvous when the desired distance (DR) to the satellite is achieved.

The input

```
MONITOR DR.SATELLITE<0.5 NAUTICAL MILES
THEN ENGINE OFF;
```

yields the directives to do as requested.

SPECIAL MONITORS

There are two special types of monitors, the MGT.MONITOR and the SWDVS.MONITOR, which have all the characteristics of the regular MONITORS described previously, but are not subject to control by the STE. The SUPERCREW program receives directives to these MONITORS from the Plan-of-the-Day data set controlled by the flight software management team and the SWDVS personnel.

The management team could institute a MGT.MONITOR for some period to

- Monitor a flight program EXECUTIVE resource, e.g., VACAREA (vector accumulator areas)
- Monitor for a specific use of the flight code.

In a similar manner, SWDVS personnel may use SWDVS.MONITOR's to

- Isolate an unusual Test Case
- Determine current methods of use of the SWDVS.

REFERENCES

1. "NASA MSC Space Shuttle Program Avionics System Recommendations", Manned Spacecraft Center, Houston, Texas, 29 November 1971.

2. "Space Shuttle Simulation Program", Manned Spacecraft Center, Houston, Texas, MSC Internal Note EG-71-37, 1 December 1971.

3. "STS Software Development (Study Task 5)", M.I.T. Charles Stark Draper Laboratory, Cambridge, Massachusetts, E-2519, July 1970.

4. "Users Guide to the Apollo Digital Simulator", M.I.T. Charles Stark Draper Laboratory, Cambridge, Massachusetts, April 1972.

5. Johnson, Madeline S. and Donald R. Giller, "M.I.T.'s Role in Project Apollo, Volume 5", M.I.T. Charles Stark Draper Laboratory, Cambridge, Massachusetts, R-700, July 1971.

6. Glick, F.K. and S.R. Femino, " A Comprehensive Digital Simulation for the Verification of Apollo Flight Software", M.I.T. Charles Stark Draper Laboratory, Cambridge, Massachusetts, E-2475, January 1970.

7. "Space Shuttle Systems Integration Laboratory Development Study", Manned Spacecraft Center, Houston, Texas, MSC-05115, October 1971.

8. "Space Shuttle Functional Simulator", Manned Spacecraft Center, Houston, Texas, MSC Internal Note 72-FD-010, May 1972.

R-721 DISTRIBUTION

External

MSC:
National Aeronautics and Space Administration (64 &1 R)
Manned Spacecraft Center
Houston, Texas 77058
ATTN: Apollo Document Control Group (BM 86) (18 & 1R)

C. Bradford (ESIO)
J. Brown (ESIO)
I. Burtzlaff (EB5)
D. Cheatham (EG)
E. Chevers
A. Cohen (PA)
D. Cole (ESIO)
K. Cox (EG2)
L. Dunseith (FA12)
R. Everett (FD)
C. Frasier (EG)
J. Garman (FS6)
T. Gibson (FS)
A. Hambleton (10) FD7
M. Holley (EG9)
T. Keeton (FD7)
M. Keathley (FD7)
S. Mann (FM)
J. Mayer (FM)
C. McCullough (EJ)
R. Nobles (FM7)
A. Nolting (CE3)
R. Parten (FD)
G. Sabionski (10) FS6
J. Satterfield (FS)
W. Sullivan (FM9)
W. Tindall (FA)
J. Williams (FS2)

KSC: (1R)
National Aeronautics and Space Administration
J.F. Kennedy Space Center
J.F. Kennedy Space Center, Florida 32899
ATTN: Technical Document Control Office

LRC: (2)
National Aeronautics and Space Adminstration
Langley Research Center
Hampton, Virginia
ATTN: Mr. A.T. Mattson

R-721

DISTRIBUTION LIST

Internal

P. Adler
S. Albert
R. Battin
C. Beals
I. Beilin
L. Berman
H. Blair-Smith
T. Brand
N. Brodeur
S. Bryant
A. Cook, MIT/MSC
S. Copps
D. Corwin
M. Cramer
R. D'Angelo
S. David
D. Densmore
S. Deutch
L. Drane (100)
G. Edmonds
A. Engel
A. Elias
D. Eyles
P. Felleman
S. Femino
R. Filene
T. Fitzgibbon
D. Fraser
J. Gilmore
K. Glick
J. Goode
B. Goodman
K. Goodwin, MIT/MSC
E. Grace
A. Green
K. Green
E. Hall
M. Hamilton
D. Hamilton
J. Harrison
R. Haslam
A. Hathaway
D. Hoag
P. Howard
J. Hsia
D. Hsiung
I. Johnson
M. Johnston
J. Kernan
A. Klumpp
G. Kossuth
B. Kriegsman
J.H. Laning
L. Larson
R. Larson
T. Lawton, MIT/MSC
G. Levine
D. Lollar
F. Marcus
B. McCoy
R. McKern
D. Millard
R. Millard
P. Mimno
E. Muller
H. Nayar
N. Neville
J. Nevins
E. Olssen, MIT/MSC
W. Ostanek
N. Rppenger
W. Robertson
R. Ragan
R. Russell
P. Rye
C. Schulenberg
N. Sears
G. Silver, MIT/MSC
L. Silver
N. Smith
R. Stengel
P. Stasiowski
J. Vella
K. Vincent
P. Volante
R. Weatherbee
P. Weissman
P. White
R. Whittredge
S. Zeldin
APOLLO Library (2)
MIT/DL TDC (10)