

California State University
Northridge

Department of Computer Science

THESIS/PROJECT PROPOSAL

TITLE: Automated Software Test Tool

Eugean Hacoians

File No: 932-1022

Date: June, 96

Approved:_____

Committee Chair: Dr. Shan Barkataki

1 OBJECTIVE

The objective of the Graduate project described below is to introduce the development of a software tool which will be used to automate the testing process of a JPL-specific set of software programs.

2 INTRODUCTION

The Multi-mission Ground System Office (MGSO), which is part of the Jet Propulsion Laboratory (JPL) organization, produces a multiple set of core software programs to assist in the generation of flight sequences that are uplinked to spacecraft through the Deep Space Network (DSN). MGSO develops these software programs by collecting all the common requirements from different JPL/NASA projects. Upon delivery of the MGSO core software, each project modifies the program and tailors it to their specific needs and requirements by manipulating the necessary files.

These programs are inter-linked together. For instance, the output of one software program is an input to another, in addition to passing initialization files such as the command database and flight rules. A Sequence Integration Engineer (SIE) may generate the initial input file by using one of these software tools.

One can look at these programs as an "operating system" of the spacecraft, but with some differences. Consider the following: when a UNIX command directive such as "ls" is entered at the command line, the result is a list of the current working directory. However, the steps involved in executing this command directive occur within the operating system and are transparent to the user. The "ls" command, after some translations, is converted into binary, loaded into CPU memory, and then executed. A similar process occurs on the spacecraft which carries the computer system (spacecraft brain) onboard. Due to size and weight limitations, there is a limited storage (hard disk) on board the spacecraft computer's system. Therefore, only a portion of the operating system is installed onboard, while the remaining portion remains in the ground system. In other words, the command translations and binary conversions

remain in the ground system, and then command bits are uplinked to the spacecraft for command processing and execution.

These programs are large in size and complexity. Many files could be manipulated during the process of adaptation and, therefore, each software program must be tested at the unit level. Also, other initialization files are created during the adaptation phase either manually, by software or combination of both. These files must also be tested for completeness and correctness.

The process of testing these files is painstaking work. Due to the amount of data contained in each file, it is very difficult to test all possibilities. To simplify this process, a software utility tool, Automated Software Test Tool (ASTT), will be developed to test these programs and files in a more efficient method.

3 TECHNICAL APPROACH

The Automated Software Test Tool (ASTT) which will be developed using Object-Oriented Design (OOD) and implemented in Object-Oriented Programming (OOP), is divided into two major parts. The first part will read a command database file containing a description of all valid spacecraft commands and create test case scenario files. The second part will run each test case scenario file through the software chain to verify the validity and correctness of every program and initialization file. If at any time, a program in the chain fails to produce a correct output file or fails altogether, the program will be terminated and the chain will be concluded. A report will be generated to indicate the success or failure of each test case.

The ASTT will produce approximately 1600 test cases, one per command. Each file will contain one command with all the permutation of parameter values. For instance, if a command has three parameters with the first parameter having four possible values, the second parameter having three possible values, and the third parameter having two possible values, then the test file would contain $4 \times 3 \times 2 = 24$ instances of that one command. Command parameters range from zero to thirty, with an average of four

parameters per command. Each parameter has about two to twenty values, with an average of five values per parameter. To process a test of this magnitude, it would take approximately seven days of non-stop processing on a 100 Million of Instruction Per Second (MIPS) Hewlett Packard 735 workstation. To shorten the test duration, ASTT will have the capability to direct test execution tasks to multiple workstations based on users request. ASTT will have the ability to access 3 Hewlett Packard 735 workstations and 15 Hewlett Packard 725 workstations (that run at 50 MIPS). The user will interact with the ASTT via a Graphical User Interface (GUI).

This project will be accomplished in three incremental phases. The software for generating the test case scenario files will be developed in the first phase, along with the command database interface, and processing of each command. The second phase will include developing the supporting scripts which will be used to run the test files through the software chain for verification and validation. Also the preliminary part of the GUI will be developed in the second phase. Enhancing the capabilities of ASTT will be done in the third phase, such as distributing the processes over multiple workstations, enhancing the supporting scripts and GUI to support the process distribution. Each delivery phase will include a specified period of time allocated for testing the delivered products and their capabilities.

Work breakdown structure is as follows:

. Phase One

1. Develop command database interface.
2. Process commands.
3. Create "test case scenario" files (which is the initial input file).
4. Verify and validate phase one capabilities.

. Phase Two

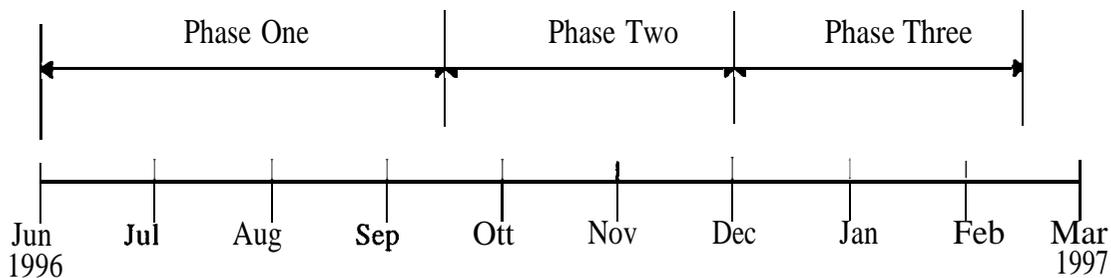
1. Develop scripts in order to run the test files through the software chain and detect any failure.
2. Develop the preliminary part of the GUI.
3. Verify and validate phase two capabilities.

. Phase Three

1. Distribute the processes over multiple workstations.
2. Enhance the supporting scripts and GUI to support the process distribution.
3. Verify and validate phase three capabilities.

4 SCHEDULE

The elapsed time for this project will be June 1996 through February 1997. The chart below depicts the milestones.



Note: The phase and steps are defined in Section 3 of this document.

5 CRITERIA FOR SUCCESS

The minimum success criteria for validating and evaluating the ASTT are as follows:

1. Generation of test case scenario files using the command database interface.
2. Development of a major script file used to run a test case through the chain of software programs on one workstation.
3. Distinguishing between successful or unsuccessful "failed" test cases.
4. Availability of a user-friendly GUI that utilizes and monitors the success criteria 1, 2, and 3.

The desired success criteria for validating and evaluating the ASTT are as follows:

5. Distribution of processes onto multiple workstations.

6. Successful production and implementation of the ASTT to support users with effective and automated testing capabilities.

6 Reference

- [1] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen. "Object-Oriented Modeling and Design." Prentice-Hall, New Jersey, 1991
- [2] D. Norman. "The Design of Everyday Things." Doubleday, New York, 1989.
- [3] B. Shneiderman. "Designing the User Interface." Addison-Wesley, New York, 1992. Second Edition.
- [4] R. Pressman. "Software Engineering: A Practitioner's Approach." McGraw-Hill, New York, 1992. Third Edition.

7 Acknowledgment

Portions of the work described were performed at the Jet Propulsion Laboratory, California Institute of Technology under contract with National Aeronautics and Space Administration.

CALIFORNIA STATE UNIVERSITY,
NORTHRIDGE

Automated Software Test Tool

A graduate project submitted in partial fulfillment of the
requirements for the degree of Masters of Science

Computer Science

by

Eugan Hacopians

May 1997

The Graduate project of **Eugean Hacopians** is approved:

Suzanne R. Dodd

Date

Professor Steven Stepanek

Date

Professor Shari **Barkataki**, Chair

Date

California State University, Northridge

I dedicate this thesis to my loving wife **Roubina**.

TABLE OF CONTENTS

ABSTRACT	VI
1. INTRODUCTION	1
2. WHAT IS SOFTWARE TESTING?	6
2.1. UNIT TESTING	7
2.1.1. <i>WHITE BOX TESTING</i>	7
2.1.2. <i>BLACK BOX TESTING</i>	8
2.2. INTEGRATION TESTING.....	8
2.3. SYSTEM TESTING	9
2.4. ACCEPTANCE TESTING.....	10
3. ASTT - THE PRODUCT	11
3.1. DATABASE READER	13
3.2. ASTT INTERFACE.....	15
3.3. PROCESS DISTRIBUTOR	22
3.4. TEST DATA GENERATOR	23
3.5. RUN SCRIPTS	24
3.6. ERROR ANALYSIS.....	25
4. DEVELOPMENT OF ASTT	27
4.1. PROBLEM DISCOVERY	28
4.2. PROBLEM ANALYSIS.....	29
4.3. DEVELOPMENT PROCESS	30
4.3.1. PHASE ONE: HIGH LEVEL REQUIREMENTS	31
4.3.2. <i>PHASE ONE: DEVELOPMENT</i>	32
4.3.3. <i>PHASE ONE: EVALUATION</i>	34
4.3.4. <i>PHASE TWO: HIGH LEVEL REQUIREMENTS</i>	38
4.3.5. <i>PHASE TWO: DEVELOPMENT</i>	39
4.3.6. <i>PHASE TWO: EVALUATION</i>	44
4.3.7. <i>PHASE THREE: HIGH LEVEL REQUIREMENTS</i>	47
4.3.8. <i>PHASE THREE: DEVELOPMENT</i>	47
4.3.9. <i>PHASE THREE: EVALUATION</i>	48

5. CONCLUSION	49
6. REFERENCES	53
6.1. SOFTWARE ENGINEERING REFERENCES	53
6.2. CASSINI DOCUMENT	53
6.3. USER INTERFACE REFERENCES	53
6.4. PROGRAMMING REFERENCES	54
7. ACKNOWLEDGMENTS	56
<u>GLOSSARY</u>	58
<u>ACRONYM LIST</u>	61

TABLE OF FIGURES

FIGURE 1: THE MSS SOFTWARE CHAIN (IDEA)	2
FIGURE 2: ASTT PROGRAM COMPONENTS AND PROCESS FLOW	14
FIGURE 3: ASTT GRAPHICAL USER INTERFACE	16
FIGURE 4: ASTT PROGRAM SAMPLE RUN	2 1
FIGURE 5: PHASE ONE ASTT DATA FLOW DIAGRAM.....	33
FIGURE 6: ASTT TEST DIRECTORY STRUCTURE	34
FIGURE 7: PHASE TWO ASTT_{DATA} DIRECTORY STRUCTURE	40
FIGURE 8: PHASE TWO ASTT DATA FLOW DIAGRAM(DATABASE READER'S OPERATION)	4 1
FIGURE 9: PHASE TWO ASTT DATA FLOW DIAGRAM(ASTT'S OPERATION)	41
FIGURE 10: PHASE TWO ASTT DATA FLOW DIAGRAM (PROCESS DISTRIBUTOR'S OPERATION)	4 2

ABSTRACT

Automated Software Test Tool

BY

Eugean Hacopians

Masters of Science in Computer Science

Software testing is an important phases in any software development project [4]. The testing typically consumes twenty to twenty five percent of the total effort of software development. Unfortunately, in a typical project the actual time for testing is decreased due to schedule overruns in earlier phases of software development [5]. To generate, execute, and analyze the result from test cases is very crucial but time consuming task. Using automated tools for testing can help meet the testing requirements. Automating any or all parts of the test process will help developers to meet the delivery schedules [4].

This Graduate project describes the analysis, design, and implementation of a software testing tool for use in testing a JPL-

specific system. This system **contains** several large inter-linked¹ programs developed by the Mission Sequence System (**MSS**) group for the **Uplink Operations Element (ULO)** of the **CASSINI** project. The **MSS** is used for commanding the **CASSINI** spacecraft that is scheduled to be launched towards Saturn in October of 1997. This project will continue until 2008 with possibility of extended mission, if funding is approved,

From July 1996 through May 1997, the Automated Software Test Tool (**ASTT**) has been fully operational and is being used to automate up to sixty percent of the testing process of all programs in the **MSS**. The use of **ASTT** reduced the cost of testing from estimated twelve engineering work weeks (best case if performed manually) to one engineering work week (worse case if performed by **ASTT**) and increased the overall efficiency of the testing process. The integrity of the **MSS** was significantly improved by **ASTT** through the automation of test case generation, test case execution in parallel utilizing multiple workstations, and analysis of the result from **the**

¹inter-linked: Input file of one program is generated by another program. In other words, in order to prepare the input files and execute program number two, program number one must be executed first.

test cases. The MSS is planning to use ASTT for testing future programs delivered for the duration of the **CASSINI** project.

The soundness of the MSS software is increased by removing the tedious tasks from the test team and allowing the team to concentrate on in-depth testing and analysis of MSS software. The ideology behind ASTT can be a tremendous cost savings and can increase software quality, a goal for any software project.

Portions of the work described here were performed at the Jet Propulsion Laboratory, California Institute of **Technology** under contract with National Aeronautics and Space Administration.

1. INTRODUCTION

The Multi-mission Ground System Office (MGSO) is a part of the Jet Propulsion Laboratory (JPL) organization. MGSO produces a set of core software programs to assist in the generation of flight sequences that are uplinked to spacecraft through the Deep Space Network (DSN). MGSO develops these core software programs by collecting all of the common requirements from different JPL/NASA projects. Upon delivery of the MGSO core software programs, each project modifies and tailors each program to their specific needs and requirements by manipulating the necessary files.

These programs are inter-linked together, (see Figure 1). For instance, the output of one software program is an input to another. In addition to expecting the output file of the previous program (in case of program-2, program-3, and program-4), initialization files such as the Command Database (CMD_DB)² and Flight Rules³ are also expected by each program. A sequence team member may generate the initial input file by using the program-1 of the software chain.

²The file that contains all the spacecraft commands with all the possible argument values of every command,

³Set of rules to follow in order to NOT harm the spacecraft, These rules are expressed as algorithms read by software to check if the set of commands will generate a conflicting result for the state of the spacecraft

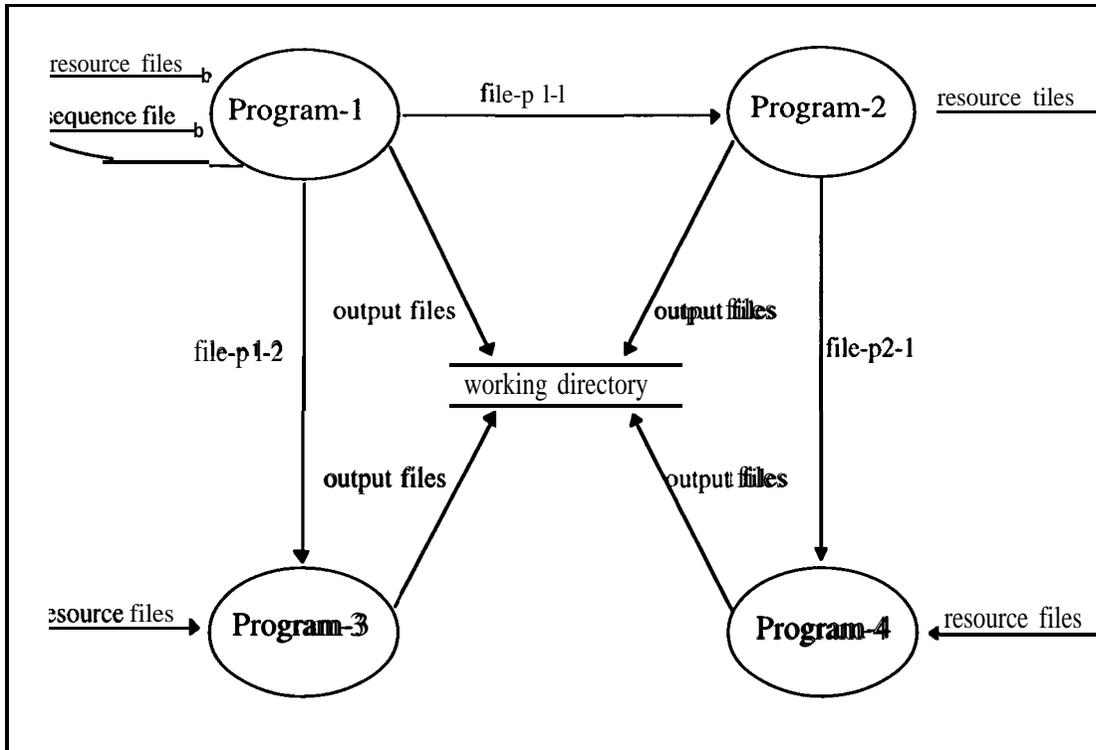


FIGURE 1: THE MSS SOFTWARE CHAIN (IDEA)

One can look at these programs as an “operating system” of the spacecraft, but with some differences. Consider the following: when a UNIX command directive such as “ls” is entered at the command line, the result is a list of the current working directory. However, the steps involved in executing this command directive occur within the operating system and are transparent to the user. The “ls” command, after some translations, is converted into binary, loaded into CPU memory, and then executed. A similar process occurs on the spacecraft which carries the computer system (spacecraft brain)

onboard. Due to the difficult and lengthy process of certifying computer hardware⁴, there is a limited storage (hard disk) and processing power on board the spacecraft computer system. Therefore, only limited capabilities of the operating system are installed onboard, while the remaining portion remains in the ground system. The command translations and binary conversions remain in the ground system, and then command bits are uplinked to the spacecraft for command processing and execution.

The first MSS program includes a special editor to generate the initial file called a sequence file. A sequence file is similar to perl or Bourne shell script files that contain several related commands in sequential order to perform a task, This editor allows the user to combine several related spacecraft commands in a sequence file by selecting the command and its argument values from a Graphical User Interface (GUI). The sequence file is processed through the MSS similar to a Bourne shell script processed through a UNIX system.

⁴Every piece of the spacecraft from the building material to the electronic components must pass a regression testing process against radiation, heat and other environmental elements absent on earth due to the atmosphere.

For example, to install a program in the user's home directory the following commands are executed in sequence:

```
cp /dir 1/dir2/prog.tar.Z -user_ account/.
uncompress prog.tar.Z
tar xvf prog.tar
rm prog. tar
```

If we place the code segment in a file, the Bourne shell script file will look like the following:

```
#!/bin/sh
Cp /dir 1/dir2/prog.tar.Z ~user_account/.
uncompress prog.tar.Z
tar xvf prog.tar
rm prog.tar
```

Similarly, if a spacecraft needs to be rotated in order to send its data to earth, several commands need to be executed one after the other to perform this task.

These programs are large in both size and complexity. Many files can be manipulated during the process of adaptation⁵ and

⁵Tailoring the MGSO software to a project's requirements and specification.

therefore, each software program must be tested at the unit level. Also, other initialization files are created during the adaptation phase either manually, by software, or combination of both. These files must also be tested for completeness and correctness.

The process of testing these initialization files are labor intensive work. Due to the amount of data contained in each file, it is very difficult to test all possibilities. To simplify this process, a software utility tool, Automated Software Test Tool (ASTT), has been developed to test these programs and initialization files more efficiently.

2. WHAT IS SOFTWARE TESTING?

“Software testing is the process of executing a program or system with the intent of finding errors” [2]. The errors found during software testing could be caused by flaws in:

- Design.
- Coding.
- Requirement.

Other factors considered in software testing relate to how well the software has been built or engineered, This includes consideration of documentation, structure, efficiency, ease of understanding, and extendibility [4]. However, testing can not show the absence of software defects, it only can show software defects are present. [3]

There are four distinct levels of testing: Unit Testing, Integration Testing, System Testing, and Acceptance Testing. Individual program modules are Unit Tested. After integrating groups of program modules Integration Testing is performed, and groups of programs are tested together in System Testing.

Completed system is then tested by end users during Acceptance Testing.

2.1. UNIT TESTING

Unit Testing is performed by the programmers. This scheme is an informal testing process used to find logic, structure, typographic and module interface errors. There are two major schemes used to perform Unit Testing, White Box Testing and Black box Testing.

2.1.1. WHITE BOX TESTING

White Box Testing is used to test the internal control structure of a given module by the programmer. The programmer designs and generates test cases to execute each independent path at least once, execute every loop condition at the boundaries and within the operational boundaries, and execute all the logic decisions on their TRUE and FALSE sides. The White Box Testing uncovers errors such as incorrect assumptions, logic errors, or typographical errors [4].

2.1.2. BLACK BOX TESTING

Black Box Testing is used to test the functional requirements of the software without regards to the internal workings of the program. In this testing method the programmer designs and generates test cases using input conditions that will fully exercise all functional requirements. This testing scheme uncovers such errors as incorrect or missing functions, interface errors, data structure errors, performance errors, initialization errors, and termination errors [4].

Black Box Testing is not an alternate to White Box Testing. It is a complimentary approach that is likely to uncover different classes of errors than White Box Testing [4].

Techniques such as Equivalence Partitioning, Boundary Value Analysis, Comparison Testing or Cause-Effect Graphing Techniques are used to design and generate test cases used in Black Box Testing [2].

2.2. INTEGRATION TESTING

Integration Testing begins when several related modules are brought together. This testing scheme is used to test the interfaces

of these modules and ensure that the programs are communicating as expected.

2.3. SYSTEM TESTING

System Testing begins after Integration Testing. This testing scheme is used to exercise all system level functions. Also the software is stressed to uncover its limitations and measure its full capabilities. System Testing relies mostly on the Black Box Testing perspective [4].

System Testing is much more formalized than unit testing. At this level of testing, records are kept to document what is tested and the result of each test is maintained. Automated aids are more useful and more commonly developed and used to reduce the time and cost of System Testing. Test Data Generators that generate test data files based on parameters and report on the differences, are used increasingly [4].

2.4. *ACCEPTANCE TESTING*

Acceptance Testing begins when System Testing is completed. Its purpose is to provide the end user or customer with confidence and assurance that the software is ready to be used.

3. ASTT - THE PRODUCT

The spacecraft command definition (i.e. number of arguments for each command, argument values, type of commands) is specified in a centralized database called the Command Database (**CMD_DB**). In order to define, verify, and translate a command, access to the information of the **CMD_DB** file is needed.

Each of the MSS programs perform one step of the spacecraft command translation process, and therefore, each program needs a subset of the information contained in the **CMD_DB** file. These programs utilize the necessary information in a file of which the format is only readable by that one program.

The **CASSINI** project will have approximately one thousand six hundred spacecraft commands by the end of July 1997. Command arguments range from zero to thirty, with an average of four arguments per command. Each argument has a range of approximately two to two hundred values, with an average of five values per argument. Generating test cases to verify every bit of information in each program's **CMD_DB** file against the centralized database requires a great deal of effort. For example, if command **XYZ** has three arguments with the first argument having four

possible values, the second argument having three possible values, and the third argument having two possible values, $4 \times 3 \times 2 = 24$ instances of that one command must be tested. Therefore, automating the test case generation and execution of each test case through the MSS became necessary.

The Automated Software Test Tool (ASTT) is currently being used to automate the **CMD_DB** file verification of each MSS program against the centralized database. The purpose of this tool is to reduce the amount of time spent on **CMD_DB** file verification. The ASTT reduces the time duration in three different ways:

- Generating the test files.
- Executing test files in parallel on multiple workstations.
- Analyzing the result of the test cases.

This tool also performs some other high level System Testing such as program interface testing, reading in the necessary resource files, and producing the correct output files.

The ASTT is a system comprised of several programs and UNIX Bourne shell scripts. Individual program and script range from

fifty lines to three thousand lines of code. The ASTT components, shown in Figure 2, consists of the following:

- Database Reader - generates partial test cases and command table file.
- ASTT Interface - the user interfaces with in order to create test runs.
- Process Distributor - distributes test cases to multiple workstations and calls the Run Script.
- Test Data Generator - takes the partial test cases (specified in the ASTT Interface by the user) and generates complete test cases.
- Run Scripts - executes MSS programs (specified in the ASTT Interface by the user) with each test case.
- Error Analysis - analyzes the results of the test cases.

3.1. ***DATABASE READER***

The Database Reader is a batch mode C program with approximately two hundred lines of code.

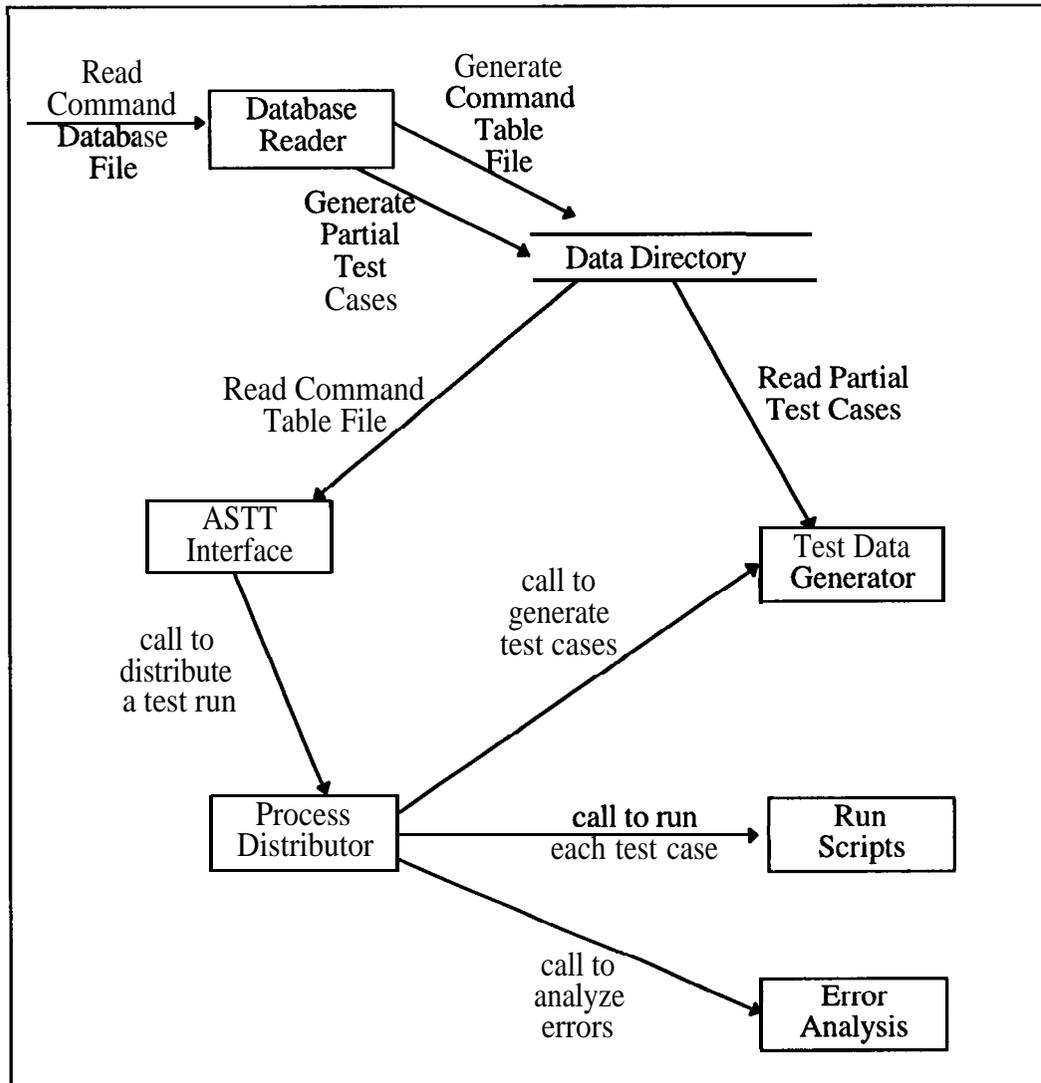


FIGURE 2: ASTT PROGRAM COMPONENTS AND PROCESS FLOW

The System Test Engineer executes Database Reader program once per **CMD_DB** file delivery. The **CMD_DB** file is delivered whenever a set of commands are added, deleted, or modified in a n y way. The Database Reader reads the **CMD_DB** file and generates:

- One partial sequence file per command.
- Command table file.

Each partial sequence file contains several instances of a given command with the combination of its parameter values (i.e. 24 instances for command **XYZ**, as mentioned in the previous example). A sequence file⁶ not only contains the spacecraft commands but also has different packaging schemes. The packaging schemes will be discussed in the ASTT Interface program description. A command table file contains all the command stem S^7 specified in the **CMD_DB** file grouped by subsystem. A subsystem is one of the spacecraft components such as the infrared camera. All commands belonging to each subsystem are distinguished and grouped under that subsystem heading,

3.2. ASTT INTERFACE

The ASTT Interface, shown in figure 3, is the **Graphical User Interface (GUI)** to the ASTT with approximately three thousand lines

⁶ Similar to **bourne** shell script file that contains several related commands in sequential order to perform a task.

⁷Spacecraft command names i.e. “1s” is a command stem.

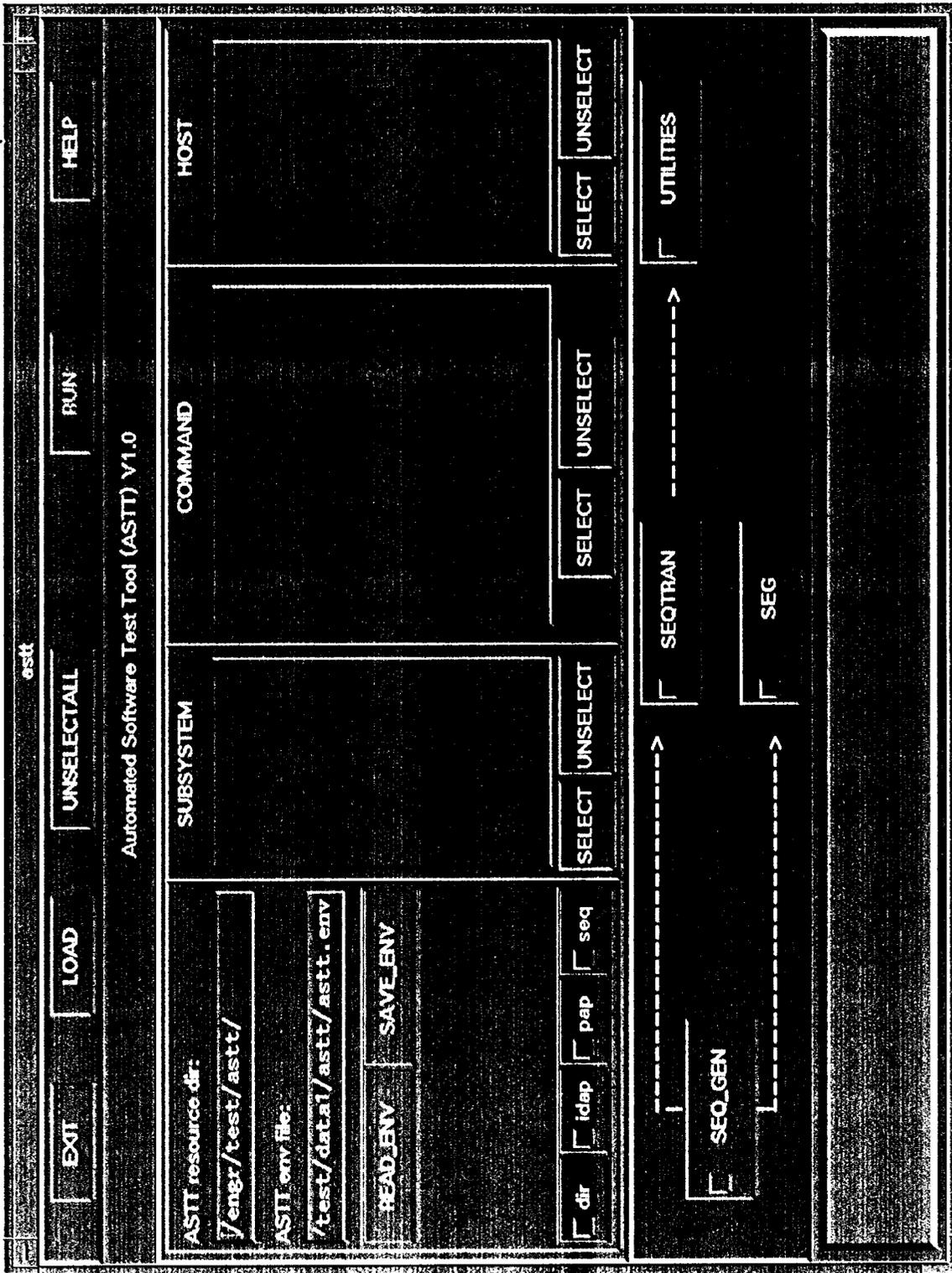


FIGURE 3: ASTT GRAPHICAL USER INTERFACE

of code, This program is an X-motif based GUI with object-oriented C++ classes for data structures. The structure of these classes are discussed in section 4.

The ASTT Interface program reads three input files to populate the corresponding data structures and display the data via the GUI when the LOAD button is pressed. The first input file read is the command table file generated by the Database Reader. The information in this file is stored in the subsystem/command data structure and the subsystems are displayed in the SUBSYSTEM list. By double clicking on a given subsystem, the commands belonging to that subsystem are displayed in the COMMAND list. The next file read is the default packaging schemes of each command. The information read from this file is also stored in the subsystem/command data structure in its corresponding command class.

A sequence file can be packaged in four different ways:

- Direct.
- Sequenced.
- Immediate/Delayed Action Program (IDAP).

- Privileged Action Program (PAP).

Direct packaging means that as soon as the sequence file is received by the spacecraft it will be executed. Sequenced packaging means that the sequence file is stored in the spacecraft memory. One can think of sequenced packaging as time triggered events which the trigger time specified in the sequence file. IDAP packaging is like Sequenced packaging that is limited in data size (118 data words). PAP packaging is like IDAP packaging with an additional restriction. Only limited number of commands that are marked privileged can be placed in a PAP packaging.

After the default packaging scheme file is loaded, the user can see the default packaging scheme of each command by double clicking on that command. This is indicated by the colors of the four toggle buttons that change whenever the user double clicks on a command. For example, if a command can be placed in Direct, Sequenced, and IDAP packaging, the “dir”, “seq”, and “idap” toggle buttons will change color leaving the color of the “pap” toggle button unchanged.

The final input file is the name of the workstations available to the user. The information in this file is stored in the host class data structure and displayed in the HOST list.

The user can select commands with different packaging schemes to build a test run. When a subsystem is selected, all the commands in that subsystem are selected for that test run. For example, the user can select several commands by double clicking on a desired subsystem, highlighting the command names, and pressing the select button under the COMMAND list. In addition, the user can select other commands in other subsystems by following the previously mentioned steps, or selecting an entire subsystem(s) by highlighting subsystem name(s) and pressing the select button under the SUBSYSTEM list. If neither of the packaging scheme toggle buttons are selected, the **default** packaging scheme(s) **is(are)** selected.

The user must select at least one of the MSS programs by pressing the appropriate toggle buttons. Also, the user can select the workstations that will be used to execute this test run. In order to select the workstations the user highlights the host name(s) and presses the select button under the HOST list. If no host is selected

the current workstation will be selected as the default. The user has the choice of deselect any selected options by highlighting the selection and pressing the corresponding **unselect** button.

After the user makes all the selections needed the user must select the “SAVE_ENV” button below the “ASTT env file:” text field. This action will save all the selections made by the user in a file. The name and location that this file will be saved is specified in the “ASTT env file:” text field. After the environment file has been saved the user can press the “RUN” button, at the top of the display, to execute the saved test run. This action will execute a **fork()** function call. The parent process allows the user to return to the ASTT Interface in order to design the next test run. The child process calls the Process Distributor program to perform the test run. The functions of Process Distributor program are discussed in section 3.3. Process Distributor.

For example (see Figure 4), if the user generates two test runs the ASTT will generate two child processes for each test run and calls the Process Distributor program from each child process. Assume test run number one contains sixty test cases with workstations number two through five as it's computing resources.

And also assume test run number two contains fifty test cases with workstations number four through eight as it's computing resources.

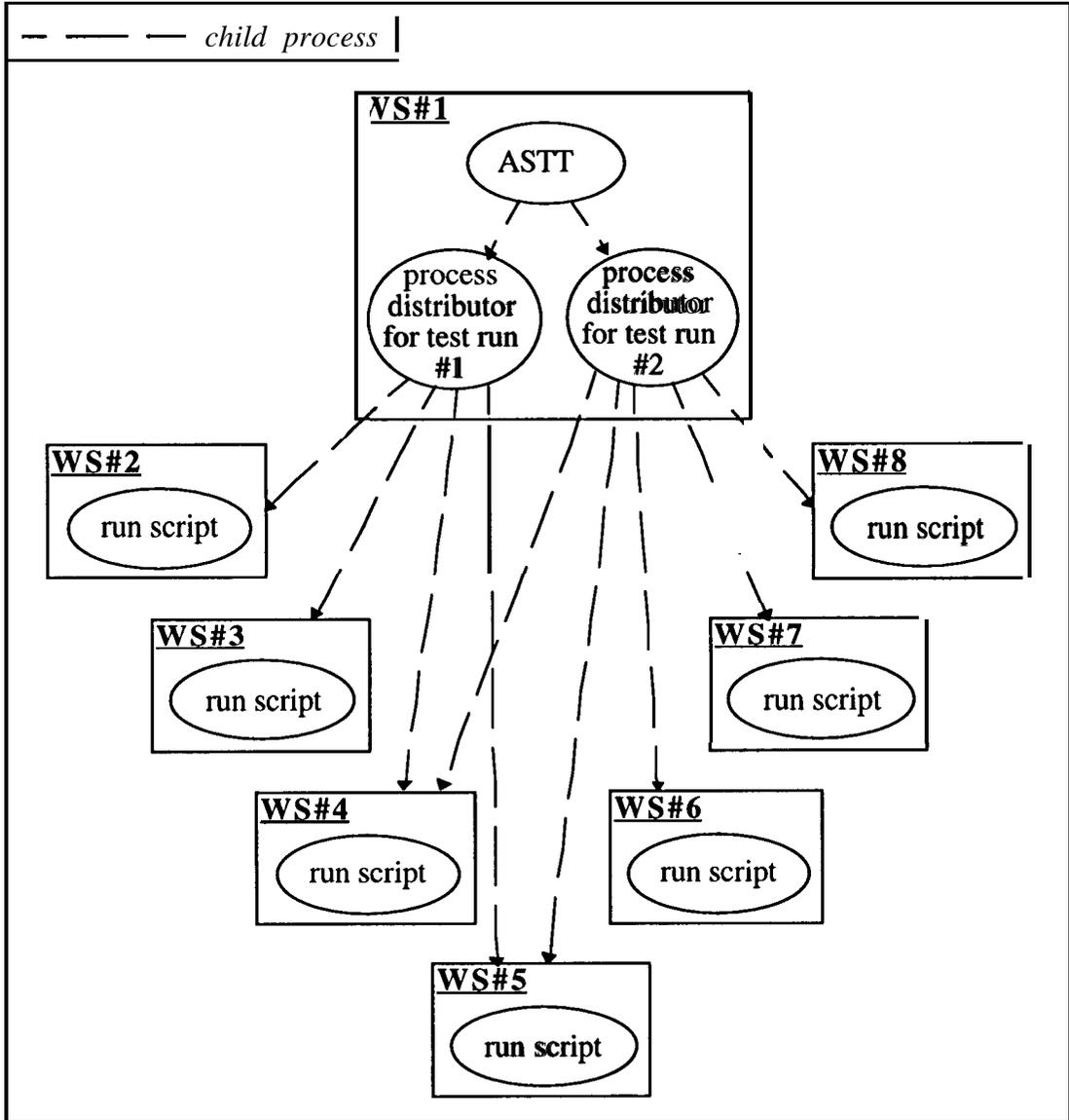


FIGURE 4: ASTT PROGRAM SAMPLE RUN

Each of the process distributors will utilize the computing resources allocated to them by the user to complete the test run allocated to this program.

Any action taken during the test design process is documented in the text output at the bottom of the ASTT Interface. If any error occurs during the test design process, the error is printed in the text output and the error dialog boxes are displayed, informing the user of each error.

3.3. PROCESS DISTRIBUTOR

The Process Distributor is a C++ program with approximately eleven hundred lines of code. This program is called by the child processes of the ASTT Interface in order to execute a test run. The Process Distributor reads the ASTT environment file and executes the test cases specified in this file. The function of this program is to:

- Generate the complete test cases by calling Test Data Generator.
- Copy a test case directory to the temporary directory of the selected workstations.

- Distribute the execution of each test case to the corresponding workstations.
- Move the test result back to the test directory.
- Analyze the generated data by calling Error Analysis program.

The process distribution is done by `fork()` function and remote shell calls. The parent process calls multiple `fork()` functions until every workstation has been allocated a test case. The child process copies the test case to the designated workstation, executes the Run Script, and copies the result back to the test directory. The child process performs these tasks via remote shell calls. After completion of every test case specified in ASTT environment file, the program calls the Error Analysis script.

3.4. TEST DATA GENERATOR

The Test Data Generator is a Bourne shell script with approximately sixty lines of code. This script reads the ASTT environment file that contains all the information to perform a test

run, then generates the complete test files specified in the ASTT environment file. This includes:

- Creating one directory per test file.
- Placing the packaging scheme in each sequence file.
- Copying the necessary MSS program environment files in each test directory.

3.5. RUN SCRIPTS

The Run Script is a **Bourne** shell script with approximately seventy lines of code. This script takes five arguments as input. The first argument is the path to the test directory. The next four specify which programs are needed to be executed. A “1” in any of the four arguments indicates to run that program and a “0” for no execution. For example, if the program-1 and program-2 are executed the arguments to the Run Scripts would be:

```
runScript testpath 1 1 0 0
```

The Run Script executes a test case through the specified program(s) as specified on the script's argument line.

As discussed earlier, the MSS programs are interconnected. Therefore, if any error is detected after the execution of a program in the beginning or middle of the chain, executing the next program is not necessary. The criteria for running the first program in the chain is the existence of the input files. The criteria for running the next programs are to scan the log file of the previous program for errors. If an error is detected, the execution of that test case is terminated for that specific command. Otherwise, the next program is executed. These steps are taken until every MSS program has been executed.

3.6. *ERROR ANALYSIS*

Error Analysis is a **Bourne** shell script with approximately two hundred and fifty lines of code. After all the test cases in a ASTT environment file have been executed, the Process Distributor calls the Error Analysis script to analyze the data generated. This script scans the log file of each MSS program, starting from the beginning of the chain, for expected and unexpected errors. This is determined

according to the error messages detected. These steps are performed for every test case specified in the ASTT environment file.

Test case directories are moved to a separate directory. As mentioned there are two kinds of errors, expected or good and unexpected or bad error cases. Therefore, there are two separate directories to keep the good and the bad error cases. The content of “good errors” directory is the expected error cases (i.e. purposely inserting an incorrect packaging scheme in the sequence file). The content of “bad errors” directory is any unexpected results. The structure of the “good errors” and “bad errors” directories are the same structure as the test directories. All successful test cases remain in the test directory.

After the ASTT run has completed, the user views the test cases in the “bad error” directory to determine the cause of the failure. If the “bad error” directory is empty and the software ran to completion, the user can assume the ASTT run was successful. The test cases in the “good error” directory are also considered successful test cases.

4. DEVELOPMENT OF ASTT

The ASTT evolution, like any other major software product, began with encountering problems. These problems encountered by the System Test Team were the time allocated to perform all System Testing and the concern of not being able to test every MSS program's database files. The next steps in developing the ASTT were problem analysis, generating requirements, design, implementation, testing and maintenance.

Due to the nature of the ASTT and possibility of additional requirements, the Evolutionary Software Development Process was chosen. This process allows the developer to separate the software requirements into related groups. Then one of the groups are selected, developed and delivered to the customer for evaluation. After the customer tries the first build of the software the next group of requirements are selected and development of the next software build starts. The user provides feedback to the developer in regards to the likes and dislikes of the delivered software build. The user suggestions are inserted into appropriate requirement groups and development towards next build of the software continues [2].

4.1. PROBLEM DISCOVERY

The MSS test team received the first MSS delivery for System Testing in July 1995. The MSS Test Team is comprised of William J. Krueger and Eugene Hacopians. The first version of MSS programs could only accept one hundred sixty five of the planned one thousand six hundred commands. During System Testing, various test methods such as Integration Testing, Requirement Testing, and Program Interface Testing were utilized to verify the MSS programs. The test team realized that generating test cases to verify the database files of each program would be a difficult and time consuming task. Therefore, an idea to automate the generation of test cases came about.

From September 1995 to March 1996, efforts were put forth by the test team to investigate the possibility of automating the test case generation process for verifying each program's database file that contains information about the spacecraft commands such as command stems and their possible argument values. During this time, the following issues were studied: spacecraft commanding possibilities, processing of source data (such as command names and argument values), generation of sequence files, and test case storage

directory structure. The study resulted in determining the spacecraft commanding possibilities. As mentioned in Section 3, command **XYZ** with four values for argument one, three values for argument two, and two values for argument three has possible 24 instances. The commanding possibilities for command **XYZ** is 24 which is the combination of the argument values. The issues of processing source data, generating sequence files, and test case storage structure were postponed for further study.

4.2. PROBLEM ANALYSIS

The test team received the second version of MSS programs for System Testing in March 1996. This system now could process about four hundred of the planned one thousand six hundred commands. During System Testing, the test team tried incorporate the idea of commanding possibilities into generating partial test cases. These test cases would resemble the test cases that the automated testing tool was expected to generate. The test team produced good results from these test cases. Good results in testing means that the test cases detected one or more anomalies. Using these simulated test cases, numerous anomalies in the MSS programs were found and

documented. Since the idea and the method of test case generation was successful, the MSS team placed a proposal for funding to develop the ASTT tool. After funding approval, the test team decided to start developing this tool by collecting requirements. Eugene Hacopians was assigned the development tasks of the ASTT.

4.3. DEVELOPMENT PROCESS

Due to the discreet nature of the MSS programs, detailed methods and file formats will not be provided in this report. Since the ASTT utilizes these programs and files, detail design, code, or file formats of ASTT will not be revealed in this report. The information contained in these programs are considered sensitive data/information.

According to **CASSINI** Ground System Security Requirement document: “Non-classified data or information which if compromised could impact the laboratory’s image, pose a threat to personal privacy or pose a threat to human **life**”[5].

4.3.1. PHASE ONE: HIGH LEVEL REQUIREMENTS

The high level requirements for the ASTT consisted of:

- Reading the command information from the **CMD_DB** file.
- Generating one sequence file per command with the combination of the corresponding command argument values.

The following additional requirements were added during ASTT's development:

- Packaging each sequence file in all four packaging schemes of Direct, Sequenced, IDAP and PAP⁸.
- Automating the execution of all test cases through MSS programs.

The capability for automating the execution of test cases were added to the list of requirements to accommodate the execution of grater number of possible test cases in the future. For example, with

⁸ See section 3.2, ASTT Interface for more explanation.

one thousand six hundred commands in each of four possible packaging schemes, there would be at least six thousand four hundred test cases. Therefore, to verify the database files of every MSS program, the test team would be required to execute all four MSS programs approximately six thousand four hundred times. This process would take as long as generating test files. Hence, by automating this process the test team was able to generate and execute test cases automatically to verify database files of all MSS software.

4.3.2. PHASE ONE: DEVELOPMENT

From June 1996 to November 1996, the Planning, Design, and Implementation phases of the ASTT'S first build was completed. The programs developed during this period, shown in Figure 5, consisted of the Database Reader, Test Data Generator, Run Scripts, and a partial Error Analysis. All programs were executed in batch mode.

The Database Reader read the CMD_DB file and generated a command table file⁹. The Test Data Generator read the command table file and

⁹The command table file is a subset of CMD_DB file that contains information about the command stems and the corresponding arguments.

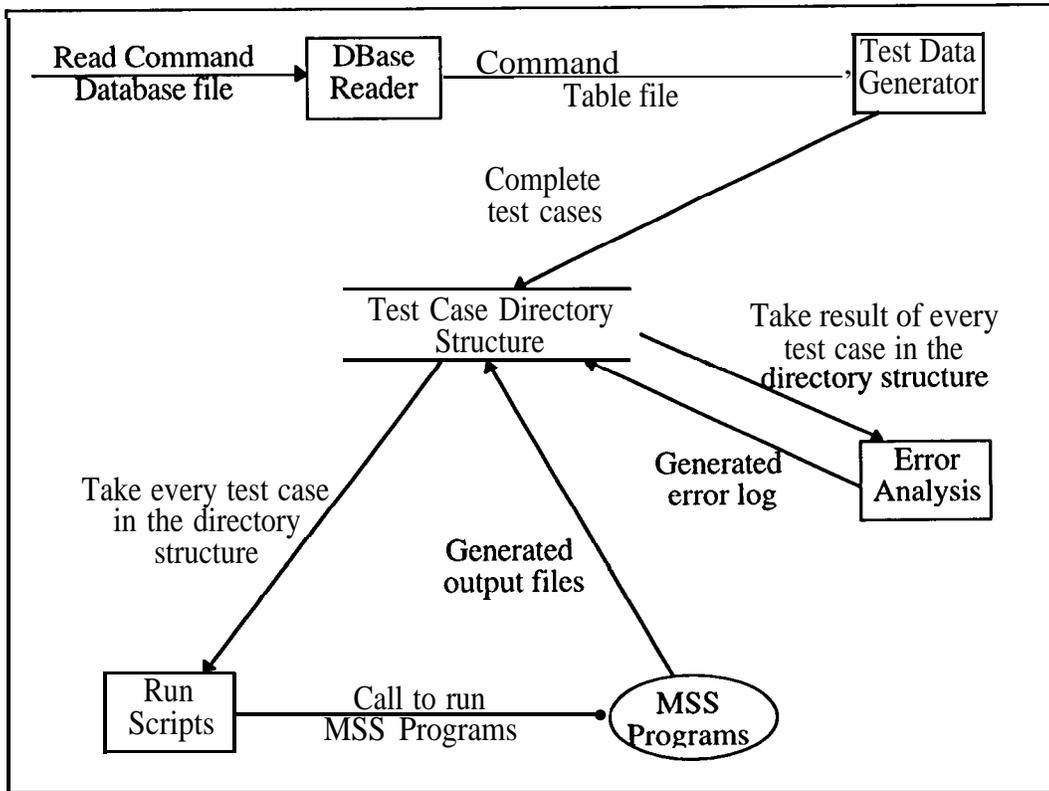


FIGURE 5: PHASE ONE ASTT DATA FLOW DIAGRAM

generated four complete sequence files for each command specified in the command table file. This was accomplished by generating one partial test case per command and placing each partial test case in all four packaging schemes. All of the information was stored in a pre-defined directory structure (see Figure 6). The directory structure was designed to separate each test case by subsystem, command and packaging scheme.

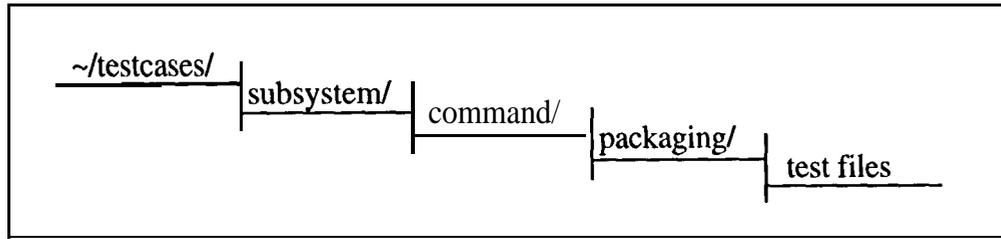


FIGURE 6: ASTT TEST DIRECTORY STRUCTURE

The Run Script iterated through the test directory structure and executed all the test cases in the test directory structure through all four MSS programs. After the execution of all the test cases were completed the Error Analysis script was executed. The Error Analysis script iterated through the test directory structure searched for errors in the log files generated by the MSS programs. Upon detecting an error, the Error Analysis script moved **those** test cases to an “error” directory and generated a log file for documentation purposes.

4.3.3. PHASE ONE: EVALUATION

The test team received the third version of the MSS programs for System Testing in November 1996. This system now could process approximately one thousand of the planned one thousand six hundred commands. During this period the test team used the

ASTT'S first release to test the MSS programs. Evaluation phase of the ASTT's first build was completed during System Testing.

The ASTT generated approximately three thousand eight hundred test cases out of four thousand possible, and ran the test cases through every MSS program. It was discovered that generating test cases for about fifty out of one thousand commands were not possible, due to complicated logic of those commands. The Test Data Generator also was not able to generate those test cases that one command argument depended on the other argument of the same command. However, the rest of the test cases found additional anomalies. These anomalies were due to the discrepancy between the CMD_DB file and one of MSS program's database file. These discrepancies were from:

- Addition of new command(s) to the system.
- Deletion of existing command(s) from the system.
- Change(s) in the command structure (i.e. change in number of arguments).
- Change(s) in the argument values of a command.

The benefits from using ASTT were:

- Generating all test cases.
- Executing all test cases in a much shorter time than manual process.
- Verifying the correction of the previously found anomalies.
- Finding additional anomalies.

Every partial sequence file and its corresponding four test cases (one complete test case per packaging scheme) were stored within the same test directory structure. The Run Scripts executed all the test cases through the MSS programs. Since the workstation that was executing the Run Scripts only had two and one half gigabytes disk space available the system ran out of disk space. The following reasons contributed to the shortage of storage:

- The quantity of test cases generated by ASTT.
- The amount of data generated by every MSS program.

Due to this problem additional data storage was allocated to the test team from other workstations. Accessing these data storage were done via Network File Systems (NFS) mounts.

The amount of data generated was approximately nine gigabytes. To work around the data storage problem the Run scripts were modified to run test cases of one subsystem at a time instead of all the test cases. Another modification to the Run Script was to remove the unnecessary output files from each test case directory. After these modifications the amount of data generated was approximately seven **gigabytes**.

Running three thousand eight hundred test cases took approximately six days of non-stop processing on a Hewlett Packard 735 workstation with a rating of one hundred Millions of Instruction Per Second (MIPS).

Another problem was the manual processing portion of error analysis. As mentioned above, the Error Analysis script detected all the errors and did not distinguish the “good errors” from the “bad errors”¹⁰. Therefore, the user had to examine every test case within

¹⁰ See section 3.6, Error Analysis, for more detailed explanation of GOOD and BAD errors.

the “error” directory instead of examining only the test cases in the “bad errors” directory.

Another encountered disadvantage of ASTT was the ability of executing only one command with a specific packaging at a time. For instance, if a user needed to test command X with only “dir” packaging scheme, the user had to execute the command with all four packaging schemes.

4.3.4. PHASE TWO: HIGH LEVEL REQUIREMENTS

The high level requirements for the next development phase were:

- Distributing the execution of test cases over multiple workstations to reduce the test execution time.
- Modifying the test case generation process to separate the partial sequence files and the test cases.
- Modify the Error Analysis script to distinguish the “good errors” and the “bad errors” and separating them into different directories.

- Generating a table containing the default packaging schemes of every command in the system.
- Introducing a Graphical User Interface (GUI) for ease of selecting commands, packaging schemes, and workstation names for process distribution.
- Allowing the user to select any or all packaging schemes including the default packaging schemes.

4.3.5. **PHASE TWO: DEVELOPMENT**

From December 1996 to March 1997, the Planning, Design, and Implementation phases of the ASTT'S second build was completed. The programs developed during this period, shown in Figures 8-10, consisted of the Database Reader, ASTT Interface, Test Data Generator, Process Distributor, Run Scripts, and Error Analysis.

The separation of the directory structure was one of the major changes from the phase one release of ASTT. The difference was that the phase one release of ASTT kept all the files necessary in the test directory. These files consisted of various packaging schemes, partial sequence files recourse files, and completed test cases. The phase two release in conjunction with the current files required additional

files. These files were necessary to incorporate the additional capabilities specified in the high level requirements. Therefore, the files necessary to configure the ASTT such as resource files and partial sequence files were separated and placed in a directory by themselves labeled “~/astt/”, (see Figure 7). Everyone had read access to this directory structure, however, write access was restricted only to the System Test Engineer.

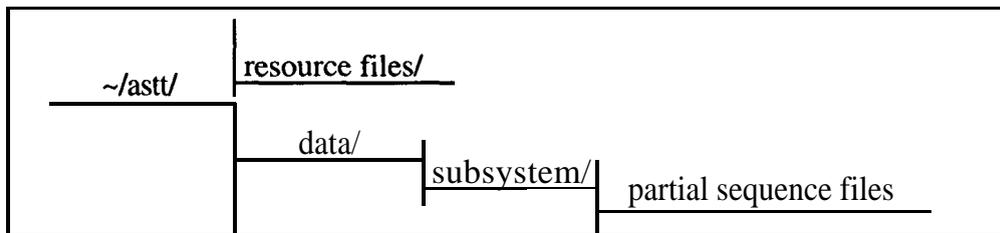


FIGURE 7: PHASE TWO ASTT DATA DIRECTORY STRUCTURE

Similarly, the Database Reader was accessible only by the System Test Engineer and was modified to write the command able and partial sequence files in the “DATA” directory structure (see Figure 8). The partial sequence files were stored in a “data” directory structure instead of the phase one’s test directory structure (see Figure 7),

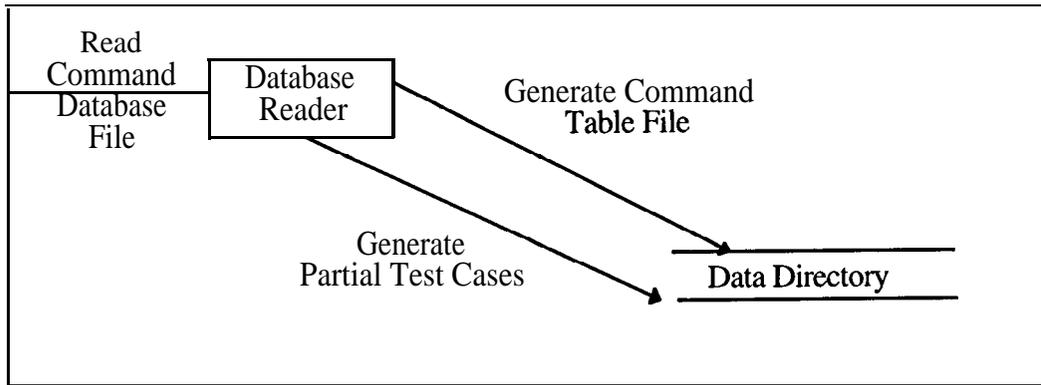


FIGURE 8: PHASE TWO ASTT DATA FLOW DIAGRAM(DATABASE READER' S OPERATION)

The ASTT Interface was developed to read resource files from the pre-specified “DATA” directory structure and displayed the information to the user via the GUI (see Figure 9). The ASTT read the command table generated by the Database Reader, packaging

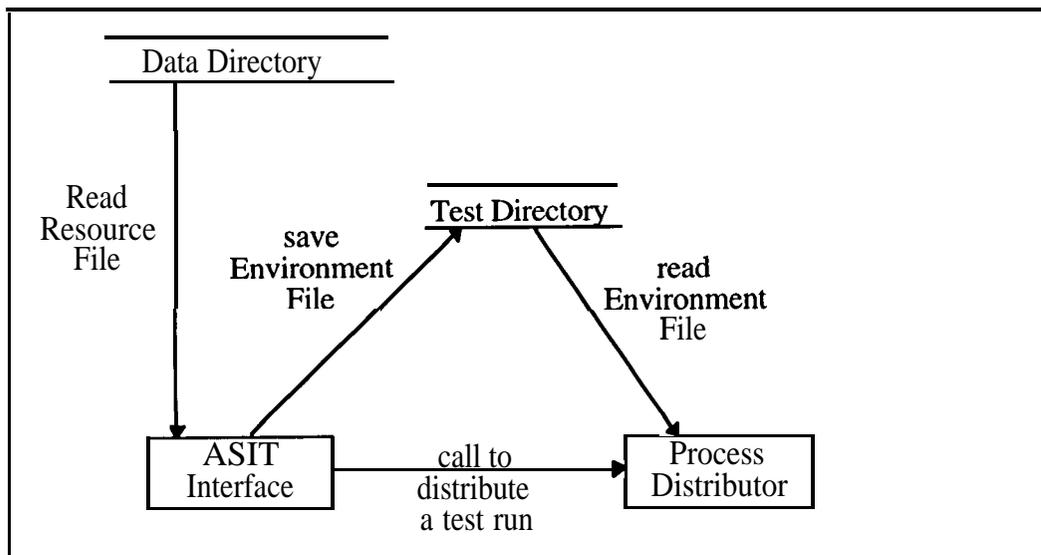


FIGURE 9: PHASE TWO ASTT DATA FLOW DIAGRAM(ASTT'S OPERATION)

scheme table, and workstation host names. After the user made the desired selections, the ASTT generated an environment file when the user pressed the “SAVE_ENV” button and executed the Process Distributor when the “RUN” button was pressed by the user.

The Process Distributor read the environment file generated by the ASTT Interface, and executed the following scripts (see Figure 10):

- Test Data Generator.
- Run Script (multiple times via `fork()` function calls).
- Error Analysis.

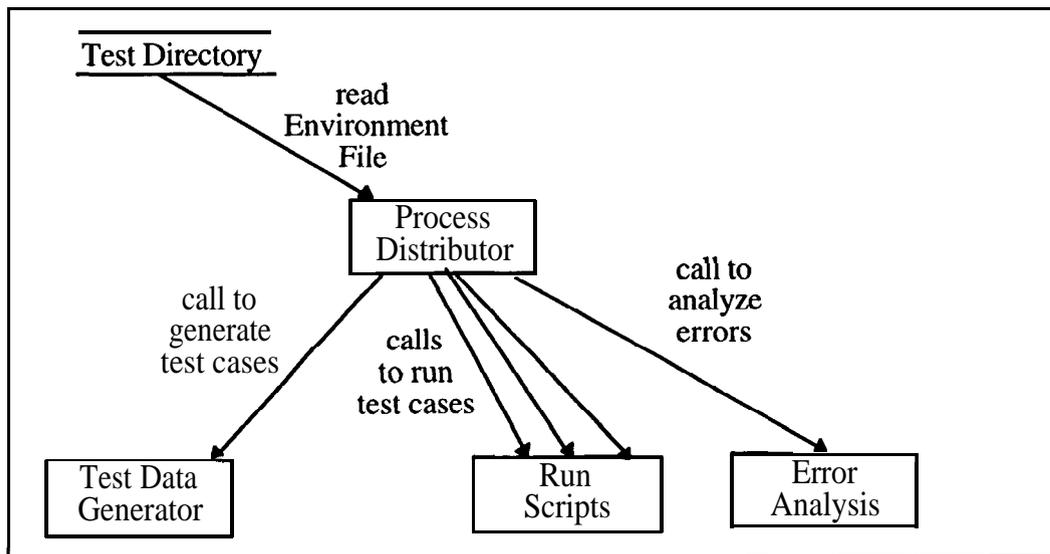


FIGURE 10: PHASE TWO ASTT DATA FLOW DIAGRAM (PROCESS DISTRIBUTOR'S OPERATION)

The call to Data Generator generated a complete test file for each test case specified in the ASTT'S environment file. Then the Process Distributor executed every test case. The execution of each test case was done by copying each test case to one of the selected host's temporary directory, creating new processes, calling the Run Script from each child process to execute that test case through the selected MSS program(s), and copying the result back into the test directory. After all the test cases specified in the ASTT'S environment file have been processed, the Error Analysis was executed.

Test data generator read the environment file generated by the ASTT Interface and created a new test directory or overwrote an existing test directory and created the specified complete test files. The data for generating these complete test files was resident in the "DATA" resources directory. The partial sequence files were picked from the "data" directory and the packaging schemes were picked up from the resource directory of "astt" (see Figure 7).

The Run Script executed a test case through the selected MSS program(s).

The Error Analysis script read the environment file generated by the ASTT Interface and scanned the log files of specified test case for errors. After encountering an error, the script compared the detected error message to the set of error messages that were expected for the “good error” cases. If the detected message matched, the test case directory was moved to “good errors” directory. Otherwise, the test case directory was moved to “bad errors” directory. All the cases without any errors remained in the test directory.

4.3.6. PHASE TWO: EVALUATION

The test team received the fourth version of the MSS programs for System Testing in March 1997. This system could now process about one thousand two hundred of the planned one thousand six hundred commands. During this period, the test team used the ASTT’s second release to test the MSS programs. Evaluation phase of the second build was completed during System Testing. The test team also received a sixteen gigabyte hard drive to resolve the data storage problem encountered in the previous build.

Phase two version of ASTT was much more visual and flexible than the phase one version. This version of ASTT was more visual due to the GUI. This version of ASTT was flexible because it allowed the user to generate as few as one test case and up to four thousand eight hundred test cases. Additionally, the ASTT could now distribute multiple test runs and each test run could be distributed to multiple workstations. By using a test run, data generated is more manageable than the previous version of ASTT. This is due to the ability to select subsets of test cases verses all test cases in phase one.

The most significant improvement for phase two of ASTT was the reduction in amount of time to execute the test cases. Although a test run could be designed with a small number of test cases, for benchmarking purposes the same number of test cases as in phase one was used. This test run utilized ten, one hundred MIPS HP 735 workstations, with all the MSS applications selected. The execution time was about seventeen hours of non-stop processing. Theoretically, using ten of the same workstations should execute three thousand eight hundred test cases in a one-tenth of the time. But, due to copying and moving files from one workstation to another

the execution of these three thousand eight hundred test cases was completed in one-seventh of the time. Currently there are ten, one hundred MIPS HP 735 and twenty, fifty MIPS HP 725 workstations available to the test team.

The Error Analysis script was improved to distinguish the “good errors” from the “bad errors”. After detecting the errors, the script moved the erred test cases in the corresponding “good errors” and “bad errors” directories. In this phase, the user only viewed the “bad errors” directory and skipped the “good errors” directory. If the “bad errors” directory was empty, then the user skipped that subsystem altogether. In the first build of the ASTT, the manual process of examining error cases took approximately eight work days. This time was reduced to four work days in the second build of ASTT.

Including the two capabilities mentioned above to the ASTT, it took a maximum of five work days to verify the database files of each MSS program (one work day for test run and the remaining days for error analysis), down from fifteen work days (six work days for test run and the remaining days for error analysis).

4.3.7. PHASE THREE: HIGH LEVEL REQUIREMENTS

The high level requirements for the next development phase are:

- Updating the Database Reader to extract additional information from the centralized database such as command packaging information.
- Adding a capability to the ASTT Interface to read the previously generated environment file in order to modify the test run, or execute the original test run in multiple occasions.
- Adding a GUI to the Process Distributor to indicate the work load and the capability to suspend or stop a test run.

4.3.8. PHASE THREE: DEVELOPMENT

From April 1997 to May 1997 the Planning, Design, and Implementation phases of the third build will be completed. The programs that will be enhanced during this period will be the

Database Reader, ASTT Interface, Test Data Generator, and Process Distributor.

The development of the third build of ASTT will start in the first week of April 1997.

4.3.9. PHASE THREE: EVALUATION

The test team is scheduled to receive the fifth version of MSS programs for System Testing in May 1997. This system should process most of the planned one thousand six hundred commands. During this period the test team will use the ASTT'S third phase to test the MSS programs.

Evaluation phase of the third build will be completed during System Testing.

5. CONCLUSION

Software testing is a very important issue in software development. This is due to the cost of correcting errors before releasing the software verses post [4]. Unit Testing and system requirements testing are the critical testing levels in most software developments [3].

In the case of the **CASSINI** project, the testing and verification of the content of each MSS program's database files is a time consuming and tedious effort, due to the number of test cases necessary to perform a thorough testing and repetitious steps involved in generating a test case. In order to send an acceptable command to the spacecraft, the MSS needs to verify each program's database files against the **CMD_DB** file. This is necessary to ensure the commands are translated correctly and sent to the spacecraft with the acceptable values. If the command's name or one of its argument values is incorrect, it could jeopardize the entire mission.

Prior to the introduction of **ASTT**, the only method the test team had available was to hand build test cases utilizing all the commands with all the possible argument values thorough every program. In the beginning this was not a problem since the MSS

could only process approximately three hundred commands. In order to generate test cases for the MSS that could process one thousand two hundred commands, it would take an estimated effort of twelve engineering work weeks to complete the task. However, the task of generating such test cases was humanly impossible, due to requiring a test engineer to repeat continuous key strokes and mouse movements to generate a test case. The magnitude of test cases required to verify each MSS program's database files could result in physical injuries such as Carpel Tunnel Syndrome. Even if the test team was required to perform this magnitude of testing, they would not have had enough scheduled time or funds to accomplish this task. The test team usually has a total of six work weeks to complete all System Testing with two test engineers.

The first release of ASTT reduced the effort of CMD_DB verification by a total of approximately ten engineering work weeks and reduced liability of physical injuries by reducing number of repetitious steps involved in generating a test case. After the second release of ASTT, the effort of CMD_DB verification was reduced much further.

The second release of ASTT allowed one test engineer to complete the **CMD_DB** verification in about five engineering work days; one day to run ASTT and the remaining time to evaluate the error cases. This was the worse case scenario after the second release of ASTT. The optimum case was two engineering work days to complete the **CMD_DB** verification compared to two engineering work weeks for the first release of ASTT or estimated twelve work weeks when ASTT was not available. With the second release the liability of physical injury was reduced further by virtually eliminating repetitious steps altogether.

The improvements that allowed the MSS test engineers to accomplish such an extensive testing of the **MSS** software were the following ideas:

- Automating the test case generation process.
- Automating the execution of the test cases through MSS'S programs.
- Automating most of the error analysis.

To reduce the execution time of the test cases the following technologies were used:

- Ability to easily create multiple test runs.
- Distributing the test case processing to multiple workstation for parallel execution.

These improvements allowed MSS to reduce cost, reduce liability of physical injury, insure quality, and allow enough time to perform other testing. The use of ASTT either directly or indirectly ensured the soundness of MSS's programs.

6.4. PROGRAMMING REFERENCES

- [9] D. Heller, P. Ferguson. "The Definitive Guides to the X Windowing System." O'Reilly & Associates, 1994. Volume 6A.
- [10] P. Ferguson. "The Definitive Guides to the X Windowing System." O'Reilly & Associates, 1994. Volume 6B.
- [11] E. Cutler, D. Gilly, T. O'Reilly. "The X Windowing System in a Nutshell." O'Reilly & Associates, 1992, Second Edition.
- [12] J. Peek, T. O'Reilly, M. Loukides. "UNIX Power Tools." O'Reilly & Associates, 1993
- [13] A. Silberschatz, P. Galvin. "Operating System Concepts." Addison-Wesley, New York, 1994. Fourth Edition,
- [14] M. Sobell. "A Practical Guide to the UNIX System." Addison-Wesley, New York, 1995. Third Edition.
- [15] B. Blinn. "Portable Shell Programming." Prentice-Hall, New Jersey, 1996.
- [16] I. Pohl. "Object-Oriented Programming Using C++." The Benjamin/Cummings, California. 1993.
- [17] D. Smith. "Concepts of Object-Oriented Programming." McGraw-Hill, New York, 1991.
- [18] S. Teale. "C++ IOStreams Handbook." Addison-Wesley, New York, 1993.
- [19] B. Stroustrup. "The C++ Programming Language." Addison-Wesley, New York, 1994. Second Edition.
- [20] S. Lioman. "C++ Primer." Addison-Wesley, New York, 1995. Second Edition.
- [21] H. Schildt. "C++ The Complete Reference" McGraw-Hill, New York, 1995. Second Edition.

[22] B. Kernighan, D. Ritchie. "The C Programming Language."
Prentice-Hall, New Jersey, 1988. Second Edition.

[23] J. Kay, B. Kummerfeld. "C Programming in a UNIX Environment"
Addison-Wesley, New York, 1989.

7. ACKNOWLEDGMENTS

Portions of the work described were performed at the Jet Propulsion Laboratory, California Institute of Technology under contract with National Aeronautics and Space Administration.

I would like to thank my loving wife **Roubina** for her support and patience throughout the duration of my project. Without her love and understanding during my schooling, I would have never completed my masters degree. I will be forever in her grace.

To my thesis advisor, Shari Barkataki, I give my thanks and gratitude. His guidance and motivation helped throughout the development of my thesis topic and during my masters course work.

Many thanks and appreciation to Steven Stepanek for his knowledge and assistance in scripting, object-oriented design and programming. Without his support, my thesis would not have been possible.

A special acknowledgment goes to Suzanne Dodd, Uplink Operations Element Manager at **JPL**. Without her approval and the funding of the project, none of this would have been possible. She allowed me to use the development steps and the results of this project for my graduate project.

Acknowledgment to my co-workers for contributing knowledge and ideas for developing this tool. A special thank you to William J. Krueger for his ideas towards the development of ASTT. To **Jaymie Truschel** for her suggestions in writing and organizing the project information. To **Henry Hartounian** for his support as a fellow student, co-worker, and a friend. To my mentor, Annette Larson, for her support and guidelines for this project. To my former technical supervisor, Scott Lever, I give my thanks for believing that my project would be a success, beneficial to **CASSINI** project, and supporting me during development. Thanks to **Kathy Weld** and **Marie Deutsch** for giving me the opportunity to work on the **CASSINI** project, A special thanks to Cheryl Johnson for having the faith and giving me the opportunity to work at JPL. Her faith and support reinforced my motivation to not only continue my education but to realize my true potential academically and professionally.

Lastly, my love and respect to my parents Flora and Eric. Their continuous reiteration of the importance of education and their allowing me to find my true career path has been the key to my success. The support they have provided me has been priceless.

GLOSSARY

Certifying Flight Hardware:

Every piece of the spacecraft from the building material to the electronic components must pass a regression testing process against radiation, heat and other environmental elements absent on earth due to the atmosphere.

Command Database file (CMD_DB):

The file that contains all the spacecraft commands with all the possible argument values of every command.

Command Stem:

Spacecraft command names e.g. “ls” is a command stem.

Command table file:

The command table file is a subset of CMD_DB file that contains information about the command stems and the corresponding arguments.

Flight Rules:

Set of rules to follow in order to NOT harm the spacecraft. These rules are expressed as algorithms read by software to check if the set of commands will generate a conflicting result for the state of the spacecraft

Inter-Linked:

Input file for one program is generated by another program. In other words, in order to prepare the input files and execute program number two, program number one must be executed first.

Partial Sequence File:

It is a sequence file without the file header information.

Software Adaptation Process:

Tailoring the MGSO software to a project's requirements and specification.

Software chain:

Sequential run of Inter-linked programs.

Sequence File:

Similar to **bourne** shell script file that contains several related commands in sequential order to perform a task.

ACRONYMLIST

ASTT	Automated Software Test Tool.
CMD_DB	Command database.
DSN	Deep Space Network.
GUI	Graphical User Interface.
IDAP	Immediate/Delayed Action Program
JPL	Jet Propulsion Laboratory.
MGSO	Multi-mission Ground system Office.
MSS	Mission Sequence System.
PAP	Privileged Action Program
SIE	Sequence Integration Engineer.
ULO	Uplink Element.

CALIFORNIA STATE UNIVERSITY
NORTHRIDGE

Automated Software Test Tool

Zegean Hacopians

Master's Defense

April 16th 1997

Acknowledgment

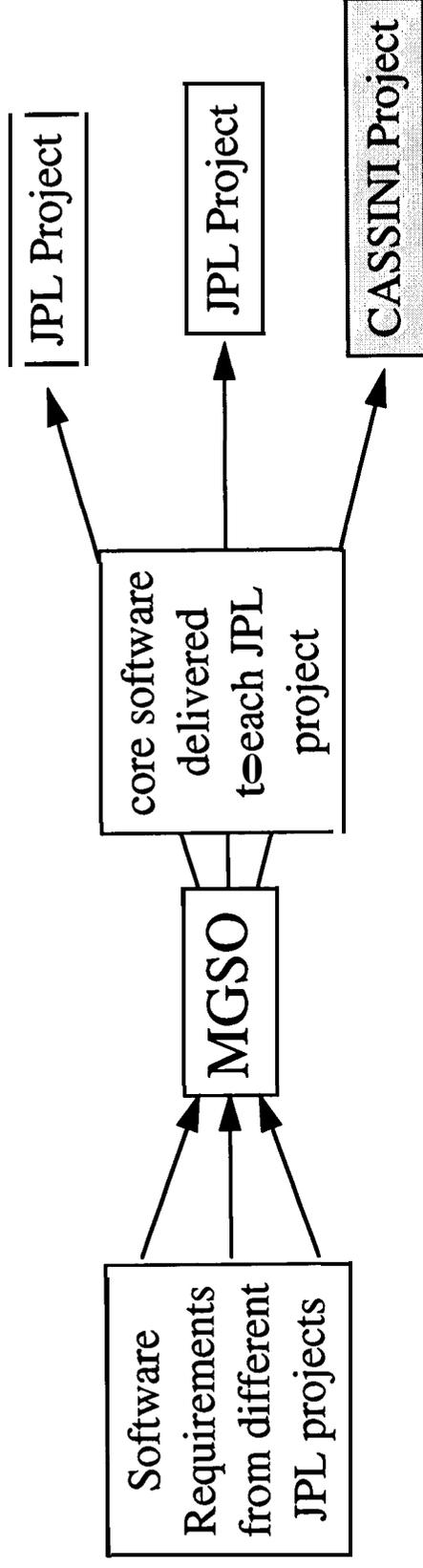
Portions of the work described here were performed at the Jet Propulsion Laboratory (JPL) California Institute of Technology under contract with National Aeronautics and Space Administration (NASA)

Organization Overview

- ❑ National Aeronautics and Space Administration (NASA).
- ❑ Jet Propulsion Laboratory (JPL).
- ❑ Multi-mission Ground System Office (MGSO).
- ❑ Uplink Operations Element (ULO).
- ❑ Mission Sequence System (MSS).
- ❑ Deep Space Network (DSN).
- ❑ Centralized Command Database File (CMD_DB).

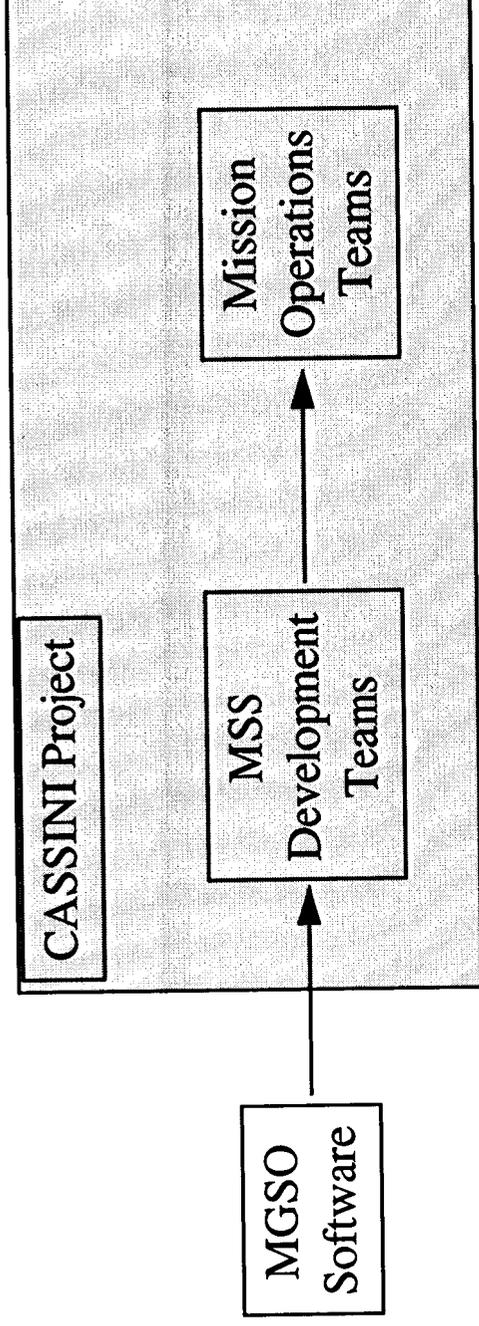
Introduction

- MGSO software development process.



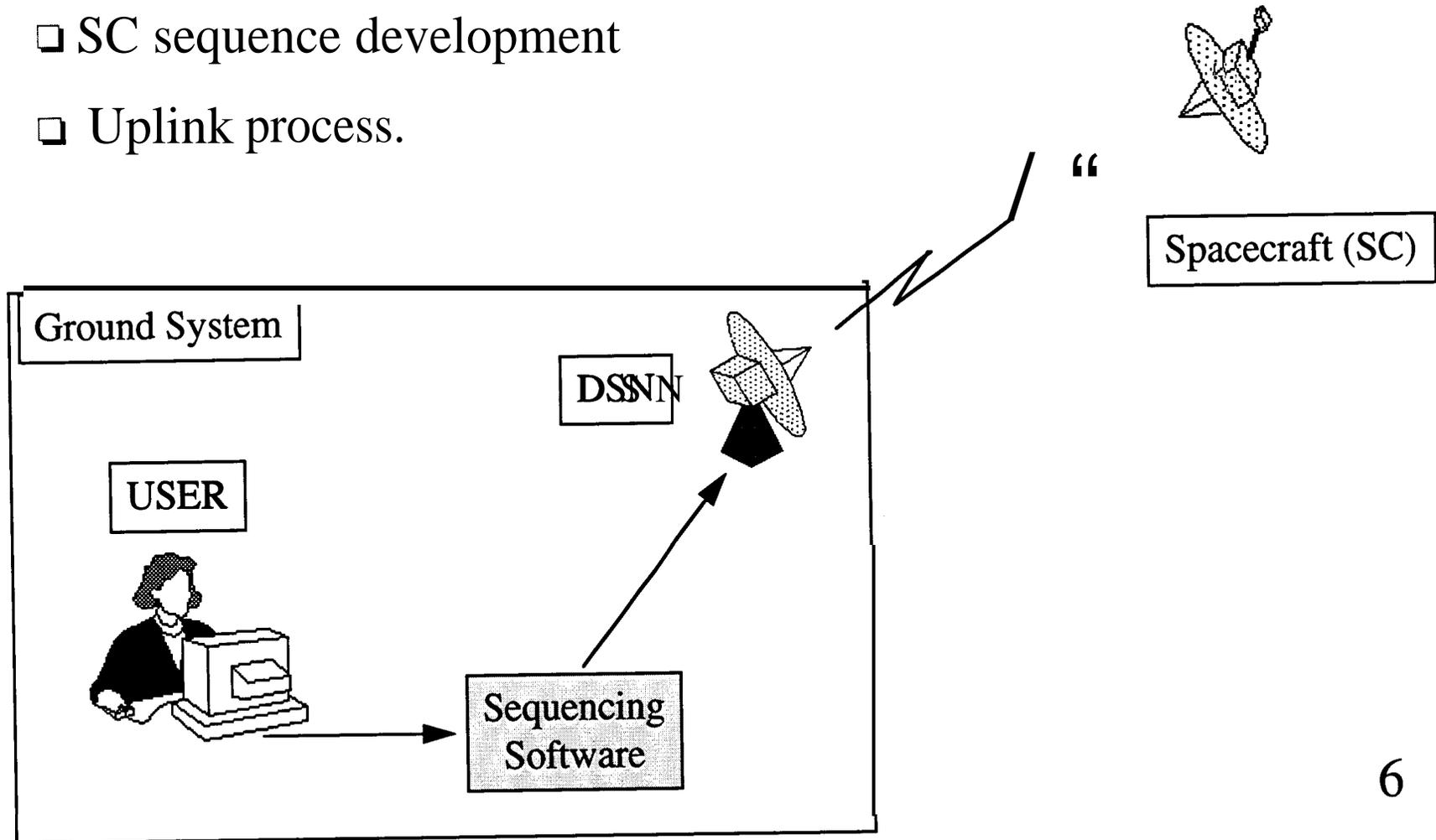
Introduction (cont.)

- CASSINI[®] project software development process.



Introduction (cont.)

- ❑ SC sequence development
- ❑ Uplink process.



Introduction (cont.)

- ❑ SC Operating System is divided into two major parts:
 - » Ground System Software
 - » Flight Software
- ❑ For Example:
 - » UNIX command “ls -al dir”
- ❑ SC components must be “Flight Qualified”
 - » Pass a regression testing process against radiation, heat and other environmental elements absent on earth.

Introduction (cont.)

□ Example of SC sequence file:

» UNIX commands (SC commands)

```
cp / dir1/ dir2/ prog.tar.Z ~user_account/ .  
uncompress prog.tar.Z  
tar xvf prog.tar  
rm prog.tar
```

» UNIX shell script file (SC sequence file)

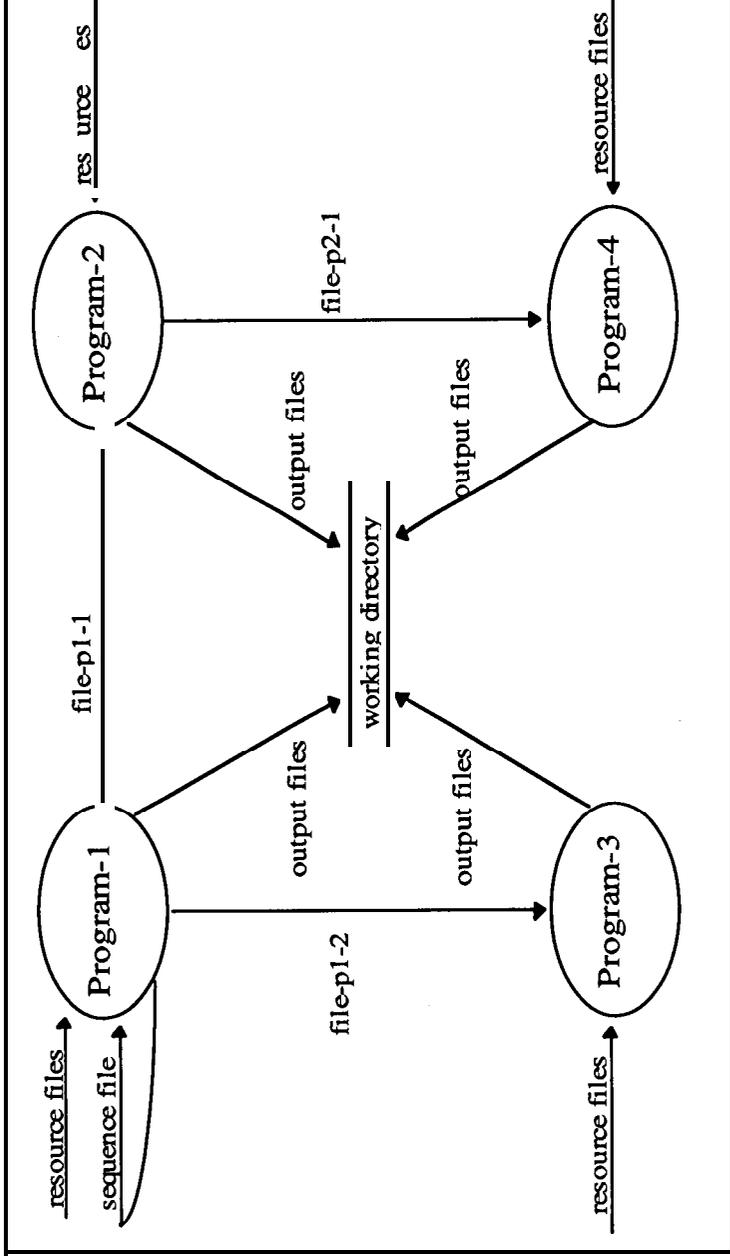
```
#!/ bin/ sh  
cp /dir 1/ dir2/ prog.tar.Z ~user_account/ .  
uncompress prog.tar.Z  
tar xvf prog.tar  
rm prog.tar
```

Introduction (cont.)

- ❑ Each MSS program:
 - » Performs one step of the translation process.
 - » Needs subset of the (CMD_DB).
 - » Accesses many different resource files.
- ❑ Testing these files are labor intensive work.
- ❑ Automated Software Test Tool (ASTT) was developed to simplify this process.

Introduction (cont.)

- MSS software chain.



What is Software Testing?

- “Software testing is a process of executing a program or system with the intent of finding errors.”
- Levels of software testing:
 - » Unit Testing:
 - white BOX Testing: Testing the internal control structure of a given module.
 - Black BOX Testing: Testing the functional requirements w/o regards to internal code.
 - » Integration Testing:
 - Testing the interfaces when several modules are brought together.
 - » System Testing:
 - Testing the entire system when all the modules are integrated.
 - » Acceptance Testing:
 - The end user tests the system (sanity check).

Automated Software Test Tool (ASTT)

□ Why ASTT is needed?

» CASSINI project will have ~1600 commands.

- Each command has 0 to 32 arguments.
- Each argument has 2 to 200 values.

» For example:

- Command XYZ has 3 arguments:

- > **Arg1 has 4 values.**
- > **Arg2 has 3 values.**
- > **Arg3 has 2 values.**

- A test case for this command will have $4 \times 3 \times 2 = 24$ instances command XYZ.

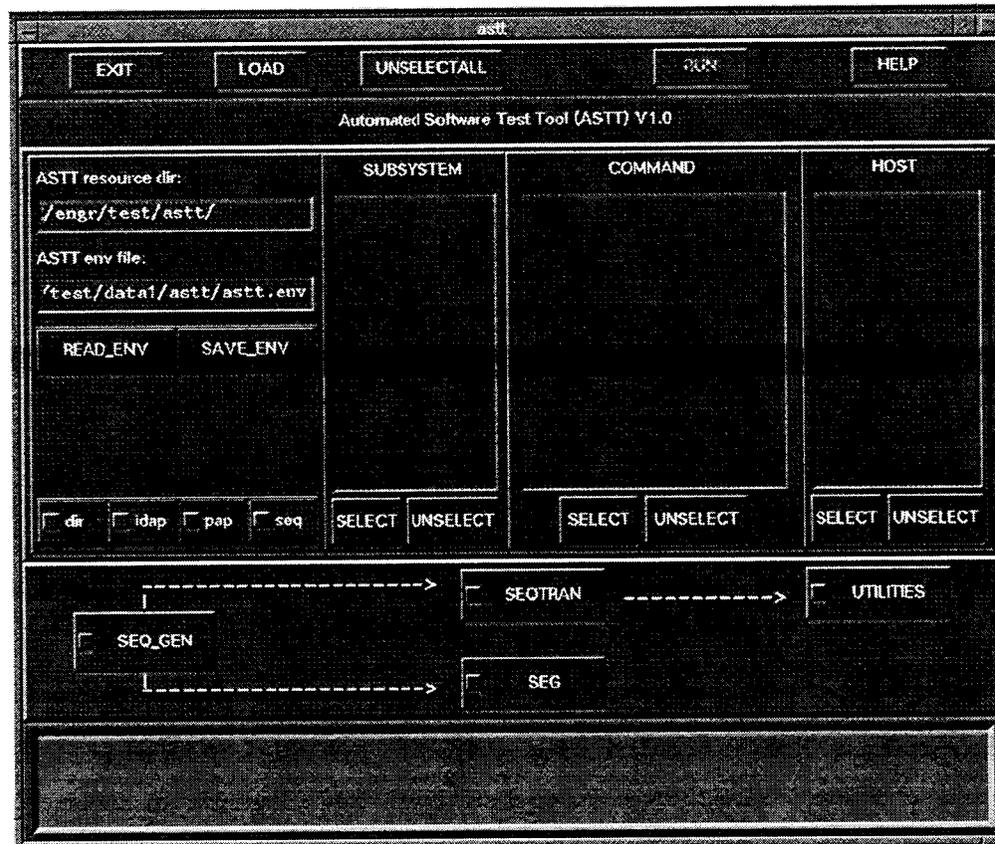
ASTT (cont.)

□ Tasks performed by ASTT:

- » Generate the test files.
- » **Execute test files in parallel** on multiple workstations.
- » Analyze the result of the test cases.

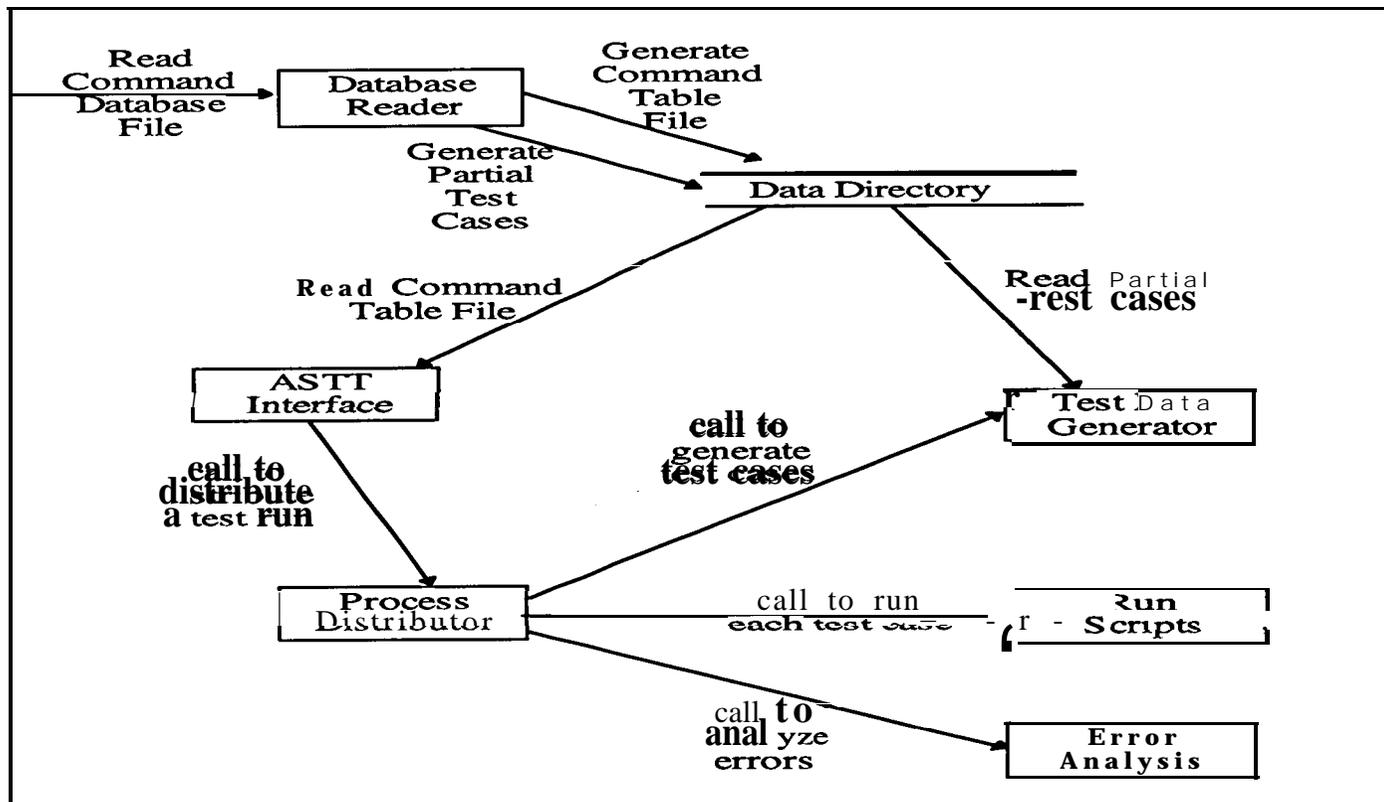
ASTT (cont.)

□ ASTT Interface.



ASTT (cont.)

□ Components of ASTT:



ASTT (cont.)

□ Components of ASTT (cont.) :

» Database Reader.

- Generates partial test files.
- Generates command table file.

» ASTT Interface.

- The user interfaces with this GUI interface in order to create test runs.

» Process Distributor.

- Distributes test cases to multiple workstations and calls the Run Script.

ASTT (cont.)

□ Components of ASTT (cont.):

» Test Data Generator.

- Generates complete test cases from the partial test cases by adding the appropriate file header.

» Run Script.

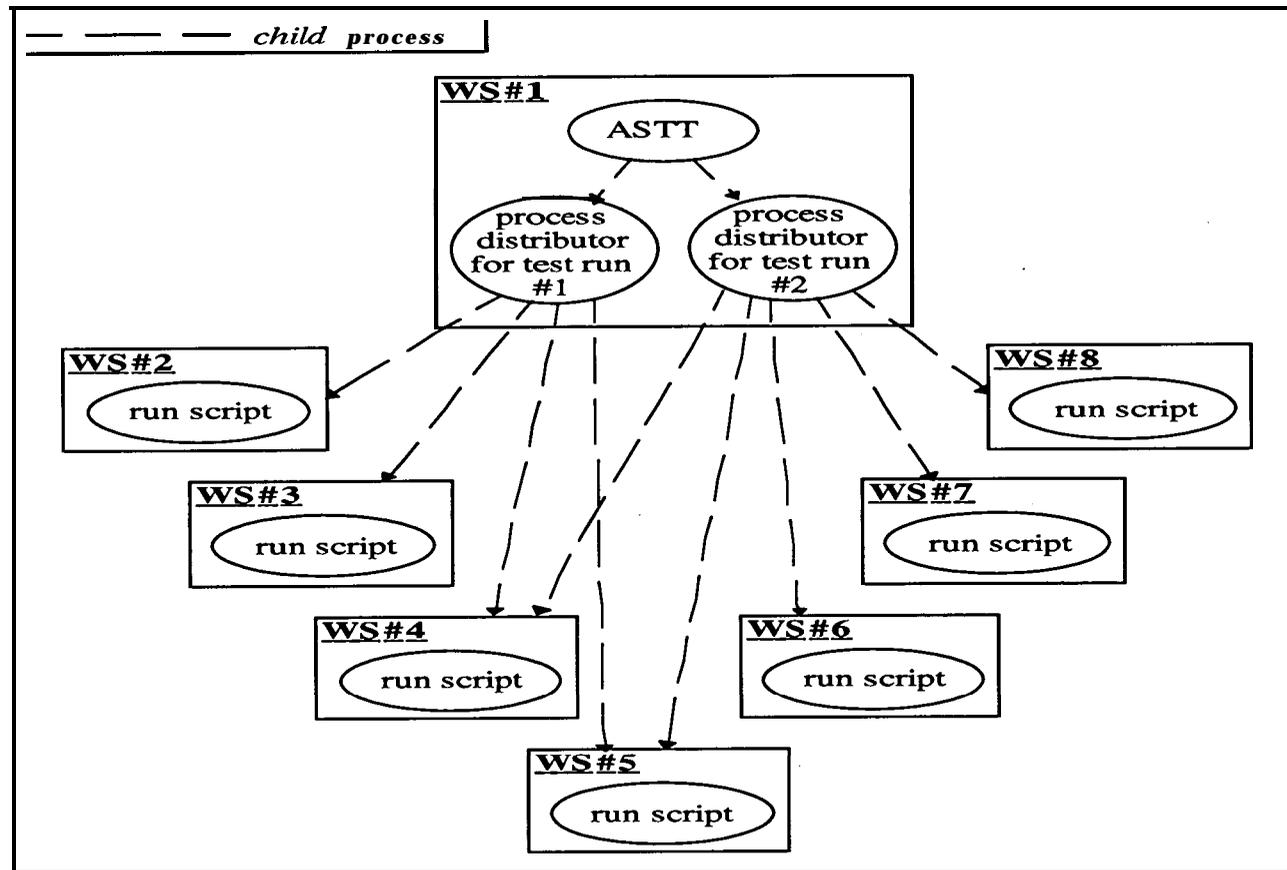
- executes the MSS programs with a given test case

» Error Analysis.

- Analyzes the result of the test cases.

ASTT (cont.)

□ ASTT sample run.



Problem Discovery

- 7/95: 1st version of MSS software was capable of processing 165 out of planned 1600 commands.
 - » The system test team realized verifying the command database file of each MSS's programs would be very time consuming.
- From 8/95 to 3/96 efforts put forth to investigate the possibility of automating the test file generation.
 - » Spacecraft commanding possibilities. (e.g.24 for command XYZ)
 - » Processing of command names and their argument values.
 - » Generation of test files.
 - » Directory structure for storing test cases.

Problem Analysis

❑ 3/96: 2nd version of MSS software was capable of processing 400 out of planned 1600 commands.

» The test team_ tried to incorporate the idea of commanding possibilities into generating test cases.

– These test cases resembled the test cases that ASTT was expected to generate.

❑ Analysis Outcome.

» Good results were produced from these test cases.

– Good result: detected anomalies that previous level of testing did not discover.

» The MSS test team placed a proposal for funding to develop ASTT.

» Eugene Hacopians was assigned the task of developing ASTT.

Development of ASTT (cont.)

□ Prior to ASTT:

- » Test cases were generated manually.
- » This was not a problem at the beginning with 400 commands.
- » Generating test cases for 1200 commands would take an estimated effort of **12** engineering work weeks.
- » This task was not humanly possible due to:
 - Number of key strokes.
 - Number of mouse movements.
 - Eye and wrist strain.

Development Process

- ❑ Due to the nature of ASTT and possibility of additional requirements, the Evolutionary Software Development Process was chosen.
- ❑ This development process allows:
 - » The software requirements to be ^{be} ~~Separated~~ into related groups.
 - » The software is developed one build at a time.
 - » Every build is delivered to the customer for evaluation.
 - » Upon user evaluation, the customer to provide feedback.

Development of ASTT: Phase One

□ High Level Requirements:

- » Read command information from the CMD_DB file.
- » Generate sequence files.
- » Automate the execution of all test cases.

□ Development:

- » Programs / scripts developed:
 - > Database Reader.
 - > Test Data Generator.
 - > Run Scripts.
 - > Error Analysis.

□ Evaluation:

- » 11/96: 3ed version of MSS software was capable of processing 1000 out of planned 1600 commands.

Result of Phase One

□ First Release of ASTT.

- » ASTT generating -3800 test cases.
- » The processing of these test cases through the MSS software took:
 - 6 days of non-stop processing on a 100MIPS HP workstation.
 - **One** eng. workweek to analyze the data generated.
 - Generated 9 gigabyte of data.
- » Reduced effort by approximately **10** eng. work week.
- » Reduced the liability of physical injury.

Development of ASTT: Phase Two

□ High Level Requirements:

- » Distribute the execution of test cases over multiple workstations.
- » Introduce a GUI for ease of selecting commands, workstation names for process distribution.
- » Modify the test case generation process.
- » Modify the Error Analysis script to distinguish between “good errors” and “bad errors”.

□ Development:

- » Programs / scripts developed:
 - > ASTT Interface.
 - > Process Distributor.

Development of ASTT: Phase Two

□ Development (cont.):

» Programs / scripts modified:

> Database Reader.

> Run Scripts.

> Test Data Generator.

> Error Analysis.

□ Evaluation:

» 3/97: 4th version of MSS was capable of processing 1200 out of planned 1600 commands.

» The ASTT was:

> More visual due to the GUI.

> More flexible by allowing the user to generate as few as one test case up to 4800 test cases.

> Able to generate multiple test runs.

> Able to distribute test cases in each test run to multiple workstations.

Result of Phase Two

- Second Release of ASTT.
 - » ASTT generated ~4500 test cases.
 - » The processing of these test cases through MSS software took:
 - ~3800 test cases were selected for benchmarking purposes.
 - 17 hours of non-stop processing on 10 100MIPS HP workstation.
 - 4 eng. work days to analyze the data generated (worst case).
 - » Reduced effort by one eng. work week.

Development of ASTT: Phase Three

□ High Level Requirements:

- » Update the Database Reader to extract additional information from the CMD-DB file.
- » Add the capability to the ASTT interface to read a previously generated test run file in order to modify and re-execute the test run.
- » Add a GUI to the process distributor to indicate the workload and the capability to suspend or stop a test run.

□ Development:

» Programs / scripts modified:

> ASTT Interface..

> Process Distributor..

> Database Reader.

Test Data Generator.

» This phase is currently under development:

Conclusion

- ❑ To send an acceptable command to the SC, the MSS needs to verify each programs database files against the CMD-DB file.
 - » This task is a time consuming and tedious effort.
 - Due to number of test cases necessary to perform a thorough testing.
 - Due to repetitious steps involved in generating a test case.
- ❑ ASTT allowed MSS test engineers to extensively test the MSS software by:
 - Automating the test case generation.
 - Automating the execution of test cases through MSS programs.
 - Distributing the test case processing to multiple workstations for parallel execution.
 - Automating most of the error analysis.
 - Ability to easily create multiple test runs.

Conclusion (cont.)

- Testing Effort.

