U S E R ' S G U I D E

SMARTS

Version 6.5

Software Maintenance and Regression Testing System

SOFTWARE RESEARCH, INC.

This document property of:

Name:_____

Company:_____

Address:_____

_____

Phone_____

SOFTWARE RESEARCH, INC.

1663 Mission Street, Suite 400

San Francisco, CA 94103

Tel: (415) 861-2800

Toll Free: (800) 942-SOFT

Fax: (415) 861-9801

E-mail: support@soft.com

http://www.soft.com

Table of Contents

List of Figures

Preface

This preface explains how this user's guide is organized.

Congratulations!

By choosing the TestWorks integrated suite of testing tools, you have taken the first step in bringing your application to the highest possible level of quality.

Software testing and quality assurance, while increasingly important in today's competitive marketplace, can dominate your resources and delay your product release. By automating the testing process, you can assure the quality of your product without needlessly depleting your resources.

Software Research, Inc. believes strongly in automated software testing. It is our goal to bring your product as close to flawlessness as possible. Our leading-edge testing techniques and coverage assurance methods are designed to give you the greatest insight into your source code.

TestWorks is the most complete solution available, with full featured regression testing, coverage analyzers, and metric tools.

Audience

This manual is intended for software testers who are using SMARTS to automate the work of organizing and managing tests. This test manager works seamlessly with its companion tools of the STW/Regression Suite: Capbak/X, an automated capture and playback tool, and Exdiff, an extended file and differencing system that compares corresponding bit-mapped images, ASCII or keysave files.

You should be familiar with the X Window System and your workstation.

Description of Contents

This manual is organized to aid you after installation has been completed. (See the *Installation Instructions* if you are trying to install.) It is divided into the following sections:

Chapter 1	*Introduction to SMARTS* discusses test planning and how SMARTS is used.
Chapter 2	*Quickstart* explains how to set up SMARTS and use its various functions.
Chapter 3	*Understanding the GUI* explains the user interface.
Chapter 4	*Creating an ATS* explains how to write and use an automated test script.
Chapter 5	*Invoking SMARTS* describes starting SMARTS from both the GUI and the command line interface.
Chapter 6	*Executing Tests* gives the step-by-step procedures for executing tests.
Chapter 7	*Viewing Execution Tests* describes using SMARTS to view tests.
Chapter 8	*Using the ASCII Menu Interface* describes the menu structure and commands.
Appendix A	*Customizing the GUI Environment* describes modifying the default settings to your own requirements.
Appendix B	*Recommended Usage* offers suggestions for using SMARTS most effectively.

Typefaces

The following typographical conventions are used in this manual:

boldface	Introduces or emphasizes a term that refers to TestWorks' window, its sub-menus and its options.
italics	Indicates the names of files, directories, pathnames, variables, and attributes. Italics are also used for chapter, manual and book titles.
"Double Quotation Marks"	Indicates chapter titles and sections. Words with special meanings may also be set apart with double quotation marks the first time they are used.

`courier`	Indicates system output such as error messages, system hints, file output, and *CAPBAK/X*'s keysave file language.
`Boldface Courier`	Indicates any command or data input that you are directed to type, such as prompts and invocation commands. (For instance, **stw** invokes TestWorks.)

CHAPTER 1

Introduction to SMARTS

This chapter describes the basic functions of *SMARTS*, how it can help you, and its role in Quality Assurance.

1.1 Automated Testing — An Overview

In the past, application and operating environments were relatively simple. Manual testing or a few written test scripts stored in batch files were usually sufficient to fully exercise the product. Today's applications, however, are much more complex, as are the environments in which they run.

The stages of software production involve multiple versions. Over a single production cycle, software may have to be tested several times. Performing these tests manually usually involves a large investment of time and money.

If an application is automated, each test can be performed automatically, accurately, and often un-manned each time developers create a new version of the software. Although it does take some time to develop the test operation, this time is more than compensated during the middle-to-later stages of testing. Considering the resources involved, automating test operation can drastically reduce the overall time needed to test a software product.

1.1.1 Test Planning and Script Writing

The effectiveness and reliability of any tool, manual or automated, depends greatly on the manner in which it is employed. As applications become more complex, planning assumes a more important role in automated testing.

In the past, ad hoc testing was an accepted and adequate method for uncovering most program errors. A few testers manually tested the product's functionality and then reported any errors to the programmer(s). When a number of bugs had been fixed, the software was shipped.

Today, this kind of ad hoc testing spells disaster. Many of today's applications contain dozens of user-selectable functions, each of which can have several major and minor options.

Just as a software application must be developed with an eye towards both reliability and revisions, a testing procedure must mimic this capacity in its ability to verify discrepancies and maintain relevancy throughout the various incarnations of the application under test.

Therefore, the analytical process used to develop an application should also be employed to develop a testing procedure. Before attempting to write test scripts for any type of application, a test plan should be created which addresses the following basic elements:

- Scope of the application to be tested.
- Extent of testing that will be performed.
- Tools that will be required during testing as well as the tasks that each tool will execute.
- Automated methods that will be used.
- Verification methods that will be used.
- Criteria that will determine the program's quality and fitness for distribution.
- Time needed to complete testing.
- Description of the test suites.
- Test data that will be used.

Having created a comprehensive test plan, the parameters and specific goals of the test scripts to be written can now be more concisely defined.

Although developing incisive test scripts can often be the most time-consuming phase of the testing process, the effort will be more than compensated by a thorough and accountable testing procedure.

While no application yet exists which can automatically produce scripts based on a testing plan (just as no application can automatically produce code based on software specifications), certain tools can facilitate and expedite the testing process by automating the procedure, providing an effective means of determining discrepancies and monitoring the effects of regression.

1.2 The STW/Regression Solution

Software Research, Inc. offers a solution, *STW/Regression*™, that can automate testing following the test planning and script writing process. *STW/Regression* is designed to overcome the tedious and error-prone process of manual testing.

Test outcomes are recorded and compared automatically with baselines. Any discrepancies are recorded and stored for further analysis. Extraneous or irrelevant discrepancies, however, can be discarded in the comparison process. Test execution, reports and statistics are available for viewing. *STW/Regression* improves the overall quality of testing by providing technically sophisticated support for full automation of regression testing, test capture / replay, and results comparison.

Although functionality varies slightly, *STW/Regression* products can be invoked either via an X Window System graphical user interface (GUI) or from ASCII menus. *STW/Regression* includes the following products:

- *CAPBAK/X*™ is an automated capture and playback tool for the X Window System that creates automated tests. It incorporates captured keystrokes and mouse movements into test scripts, and can save screen or screen fragment bitmap information as files.
- *EXDIFF*™ compares screen fragments and disregards irrelevant discrepancies with its masking capabilities.
- *SMARTS* automates the work of controlling, executing, re-executing, and analyzing the results of complex sets of tests.

SMARTS is the focus of this manual. For complete information on use of the other *STW/Regression* products, please consult the proper manuals.

An *STW/Regression* flow chart is indicated below. Boxes with darkened backgrounds represent the main components of *STW/Regression*.

SMARTS is central to the *STW/Regression* process, controlling test executions and results. *SMARTS* can run a variety of tests, including playing back hundreds of test scripts created from *CAPBAK*. *SMARTS* can also determine if *CAPBAK/X*'s tests PASSED or FAILED by making a simple call to *EXDIFF* to compare saved bitmap images from *CAPBAK/X*'s tests.

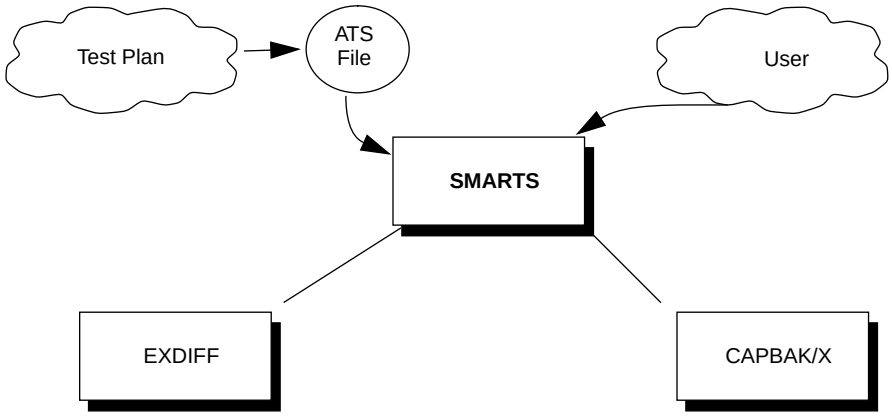

FIGURE 1 STW/Regression Dependency Chart

1.3 SMARTS' Role

SMARTS automates the testing process by reading a user-designed test description file, referred to as an Automated Test Script (ATS). The ATS is written in *SMARTS* code, which is similar in syntax to the C programming language.

Test cases are organized within the ATS and can be supplemented with activation commands, comparison arguments and evaluation methods. From the ATS, *SMARTS* is able to create a "test tree hierarchy" of the groups and tests, which is similar in structure to an outline. The test tree provides a means of interactively controlling and monitoring the testing process.

SMARTS programming capability allows the use of `if, else` and `while` control structures within its test script. Test execution can therefore be tailored to the system environment, allowing the testing process to be repeated with greater reliability than manual testing.

Although tests and test commands must be organized within the ATS description file, *SMARTS* provides a utility, **makeats**, designed to expedite the creation of the scripts.

When generated, *SMARTS* executes the ATS, compares the test output against the expected results (test baseline), and accumulates a detailed record of the test results into a log file. Based on the log file, *SMARTS* also generates reports indicating the status and execution time of any test or group of tests, the percentage of PASS/FAIL results and, as necessary, regressive test output. It should also be noted that in addition to built-in automatic differencing, *SMARTS* also provides the option of "manual" or visual evaluation where the test outcome is determined by the user.

Further information on the ATS and the **makeats** utility is available in a later chapter. (See CHAPTER 4 - "Creating an ATS" on page 87.).

By organizing all tests that apply to a given application, *SMARTS* can improve the quality of that software throughout its life cycle. Through developing and re-running a library of test suites, efforts can be focused on constructing new tests and evaluating test results — and detecting defects — rather than on the largely mechanical task of running tests and checking outputs.

The following data flow diagram depicts *SMARTS*'s processing components and procedures.

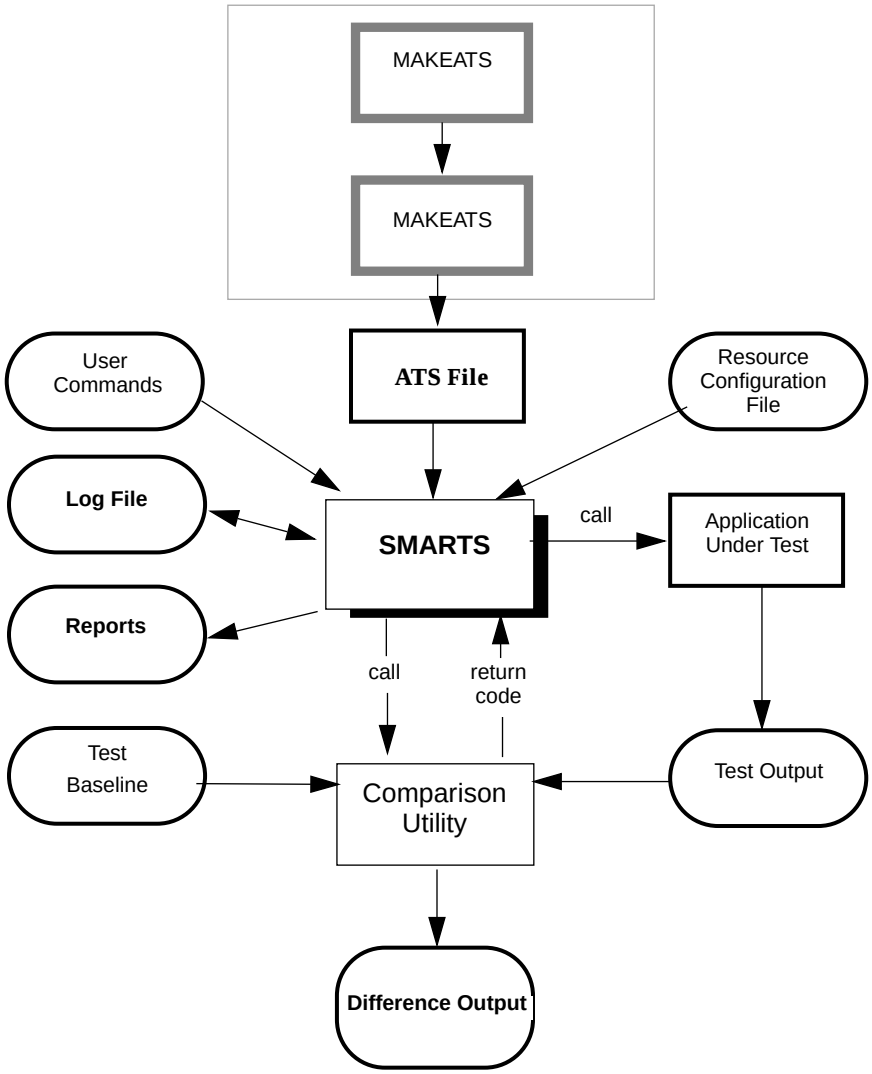

FIGURE 2 SMARTS System Chart

1.4 How SMARTS Is Used

To automate regression testing with *SMARTS*, perform the following five steps:

1. Set up test baseline results.
2. Create an ATS.
3. Execute test actions specified in the ATS.
4. Evaluate test outputs (PASS / FAIL).
5. Generate test reports.

1.4.1 Establishing Test Baselines

The initial procedure to automating the testing process is to create a series of baseline files containing correct, expected (baseline) program outputs. Later, test outputs will be compared with these baseline results.

To establish a baseline, execute a test and save the correct output in a reproducible form, e.g., a text file or an image file. **SR**'s *CAPBAK/X* also allows the results of user sessions to be used as test baselines. These user sessions can be recorded, with the results used as test baselines, and then replayed, with the results used as test outputs.

1.4.2 Creating an Automated Test Script

Once the baselines are established, the ATS can be created using any ASCII editor — for example, **vi**. The test control file must be written using *SMARTS*' C language code. *SMARTS*' **makeats** expedites ATS creation.

Within the ATS thousands of tests can be organized by test **group**, test **sub-group** and test **case**. A test group is the *root* or main group, under which all the sub-groups and test cases reside. Often, but not always, sub-groups contain specific kinds of test cases. If the X-client calculator were being tested, a group may refer to tests in general, whereas sub-groups may consist of specific kinds of tests, such as addition, subtraction, multiplication, and division.

SMARTS executes these test tree elements in sequence according to the "tree-like" group structure.

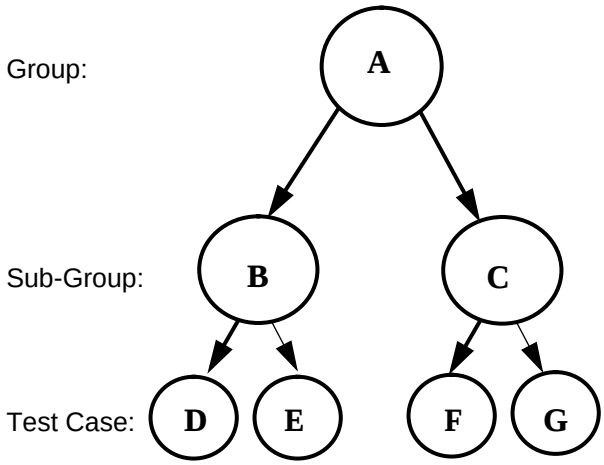

FIGURE 3 Hierarchical Test Tree Structure

This "tree" resembles the directory structure of UNIX. Either part or all of the test set can be executed according to the user's needs.

For each test group or sub-group, the ATS identifies and names the group or test-group by title and comments.

For each test case defined, the ATS includes the following three clauses:

- A `source` clause describes the nature of the test with comments.
- An `activation` clause consists of a command or series of commands to be executed by the operating system. These commands may contain executables (including *CAPBAK/X*) and shell scripts.
- An `evaluation` clause indicates how *SMARTS* will evaluate the test's output (i.e. PASS/FAIL). Four evaluation modes are available: `no evaluation`, `evaluation with user`, `evaluation with baseline` and `evaluation with function`.

NOTE: *SMARTS* also has `include`, `environment` and `termination` clauses. Further details are located in a later chapter (See CHAPTER 4 - "Creating an ATS" on page 87.)

1.4.3 Executing the ATS

Once the ATS has been constructed, testing can proceed under *SMARTS* control, with little or no further manual administration in either interactive or batch modes.

Test execution consists of two basic steps:

1. Test case selection.
2. Test case activation.

Test Case Selection

SMARTS executes only the group, sub-group or individual test case selected. This is an important feature which can be exploited to minimize overhead when re-testing a modified software product. The purpose of selection is to execute only the test groups or cases of interest, without invoking the remaining tests.

The key to test case selection is the *current position* in the test tree structure (analogous to the current working directory of UNIX). Initially, when invoking the *SMARTS* system, the *root* of the test hierarchy (the first test group or test case in the ATS) is the *current position*. The selected tests are all the test cases that fall *under* the current position.

You can change the current position by clicking the mouse on the desired test group or case (for the GUI version) or using specific commands (for both the GUI and ASCII menu versions) which allow the test hierarchy to be traversed and a specific test node to be located. For example:

1. Change the current position to an immediate descendant node (group or test case) whose name is specified.
2. Change the current position to the node just above it.
3. Change current position to a node whose name ends with a specified pattern.
4. Move the *n*th node below the current position.

GUI and ASCII menu commands are also available to display information regarding the content and hierarchy of an ATS' test tree such as:

1. Display the hierarchical test structure starting from the current position.
2. List all sub-tests in hierarchy relative to the current position.
3. List only test cases containing a specified string in their case name.
4. List test cases containing a specified string as part of their source field.

1.4.3.1 Test Case Activation

Tests are activated from *SMARTS* based on commands specified in the ATS by the `activation` keyword. *SMARTS* executes the commands as synchronous sub-processes, waiting for overall completion before regaining control.

Any number of *SMARTS* commands may be used to activate the test case, up to the limit imposed by available memory, and the length of each activation command is unlimited. Furthermore, any command that can be executed via the standard operating system command line can also be executed within the `activation` command clause.

Prior to the testing process, *SMARTS* provides various execution options. The most commonly used are:

1. Execute selected tests until there are no more test cases.
2. Execute selected tests and stop execution on the first test that fails.
3. Execute a user specified group when a test case fails.
4. Repeat the selected test as many times as specified by you.

During test execution, *SMARTS* displays processing information on the screen, including the current group, sub-group, test case names, activation command(s), and test outcomes. If a test fails, the GUI and ASCII menu interfaces display the following message:

```
**********************************************
TEST FAILS (/group/subgroup/testcase)
**********************************************
```

1.4.3.2 Test Log File

After each test execution, *SMARTS* accumulates a detailed record of test results into a log file. This information can be accessed at any time to display statistics about test execution time, such as:

- Cumulated elapsed execution time for a test case or group.
- Total execution time for a given node.

SMARTS also generates reports based on log file information, as described in another section of this book (See Section 7.2 - “Invoking the Report Window” on page 134.).

1.4.4 Evaluating Test Outputs

Following each test execution, *SMARTS* evaluates the test output according to evaluation instructions in the ATS.

There are three principal methods of evaluation:

1. No test result evaluation.
2. Manual test result evaluation.
3. Automatic test result evaluation.

No Evaluation

If a test is defined in the ATS by the entry `noevaluation`, the test will automatically pass.

Manual Evaluation

User evaluation is necessary for tests which require the test output to be manually inspected to determine outcome. Tests in this category are defined within the ATS by the entry `evaluation with user`. After a manual test is activated, the user indicates the appropriate test outcome by:

- Clicking on **PASS** or **FAIL** (for the GUI version).

or

- Typing **p** to indicate that a test passes or **f** to indicate failure (for the ASCII menu interface).

Automatic Evaluation

Tests in this category are defined in the ATS by two possible keyword phrases:

- evaluation with baseline.
- evaluation with function.

Automatic evaluation of tests for which `evaluation with baseline` is required results in comparison of the baseline and response files listed in the ATS' `evaluation` clause. Text files are compared using a file comparison utility such as **diff** and bitmap image files are compared using *EXDIFF*'s **Xexdiff** utility. *All* pairs of files must be identical if the test is to pass. If any pairs are not identical, the test fails.

The default comparison utility for `evaluation with baseline` is **diff**. However, a different comparison utility can be specified in the resource configuration file (default *smarts.rc*) for the GUI and the ASCII menu interface. The user can also set a different command using the **Set Difference Utility** option under the **Option** menu in the **Main** window of

the GUI. All the difference utilities must return a zero if the pair is identical and non-zero otherwise.

In cases where baseline files are empty or do not exist, the ATS uses three special keyword phrases that will allow these tests to pass. This is important in situations where the user does not predict a response file will be created, or if the file will be empty based on the baseline's output. This often occurs, i.e, when an application does not create an error file or the file is empty. In such cases, the user may add in these `evaluation with baseline` phrases.

- `empty` passes a test if a specified file does not exist or is empty.
- `not_exist` passes a test if a specified file does not exist.
- `not_empty` passes a test if a specified file exists and is not empty.

Of course, automatic `evaluation with baseline` will be performed only if a set of such baseline files (containing correct, expected program outputs) has been created prior to test case execution.

The `evaluation with function` feature allows any other form of evaluation to be used if a special program exists which can be used in place of the standard **diff** utility as the evaluation procedure. A wide range of function calls, or sequences of function calls, can be used. The test will pass only if all of the functions called return a 0 (successful execution).

1.4.5 Generating Test Reports

After each test execution, *SMARTS* accumulates a detailed record of the test results into a log file. Based on the log file, *SMARTS* produces a variety of test reports providing a means of determining which tests to examine for possible program errors.

The system produces four types of reports:

- **Status** report.
- **History** report.
- **Regression** report.
- **Certification** report.

The **Status** report contains the following information regarding the most recently executed tests: test name(s), activation date, outcome (PASS / FAIL), execution time (seconds), and an application's return code.

The **History** report lists the test name(s), activation date and outcome (PASS / FAIL) of all log file test entries (as opposed to the **Status** report, which lists only the most current tests) for a given node.

The **Regression** report indicates only those tests whose outcome has changed, thereby identifying bugs which have been fixed or introduced since the last time the tests were activated.The report lists test name, outcome, and activation date.

The **Certification** report provides a brief overview of testing status, indicating the number and percentage of tests that have passed and tests that have failed and the total number of tests executed.

1.5 Main Features

The following list indicates the more significant features of the *SMARTS* application:

- Starts up with only slightly varying functionality, either through the X Window System GUI or from the command line.
- Controls and monitors tests, organized either by the user or the *SMARTS* **makeats** utility, in a (tree) structure. The hierarchy of an ATS description file facilitates code traversing and reporting, and is similar to a directory structure.
- Executes a user-supplied ATS description file which specifies the tests, commands and evaluation methods.
- Runs any command executable from the command line, including commands to play back a user session captured via *CAPBAK/X*, from within the ATS description file.
- Supports conditional test execution.
- Generates reports indicating the status and execution time of any test or group of tests, the percentage of PASS / FAIL results and, as necessary, regressive test output.
- Compares captured windows and sub-windows with calls to *EXDIFF*.

CHAPTER 2

Quick Start

This chapter presents a step-by-step run-through of a basic *SMARTS* test session executed using the graphical user interface (GUI).

2.1 Instructions

We recommended that you complete the instructions in this chapter *before* going on to other chapters.

The *SMARTS* application program provides a directory containing demonstration files and pre-written programs. The tutorial test session performed in this chapter is invoked from this directory.

On completion of this chapter, you should be familiar with the following activities involved in executing a *EXDIFF* test session: invoking the *SMARTS* application, compiling a program from the testing process, setting up a *SMARTS* Automated Test Script (ATS), running a suite of tests, analyzing the test outcome via *SMARTS* reports, examining any test regression, purging any log files and exiting the *SMARTS* application.

For an overview of the *SMARTS* Graphical User Interface (GUI), please refer to Chapter 3, “UNDERSTANDING THE GRAPHICAL USER INTERFACE”.

2.1.1 STEP 1: Setting Up SMARTS

1. Initialize an xterm-type window. How initialization is executed is dependent on the window manager being used. The xterm window will serve as the *SMARTS* invocation window. Please contact your system administrator if you are unsure about your windowing environment.
2. Move the window to the upper left corner of the screen. Windows are moved by clicking on the window's title bar and dragging the window to the desired location.
3. Change to the *$SR/demos/regression/smarts.demo* directory and display its contents. The *smarts.demo* directory is provided with the *SMARTS* product and consists of the following files:
   - A resource configuration (RC) file named *smarts.rc*. The RC file consists of pre-established runtime parameters, such as an ATS specification parameter.
   - Two versions of the *search* demonstration program (*searchp.c* and *searchi.c*).
   - A sample ATS named *search.ats*. When executed, this pre-written script instructs *SMARTS* to run a set of test suites for the demo program.
   - Several sample baseline and test input files.

The screen display should now look like this:

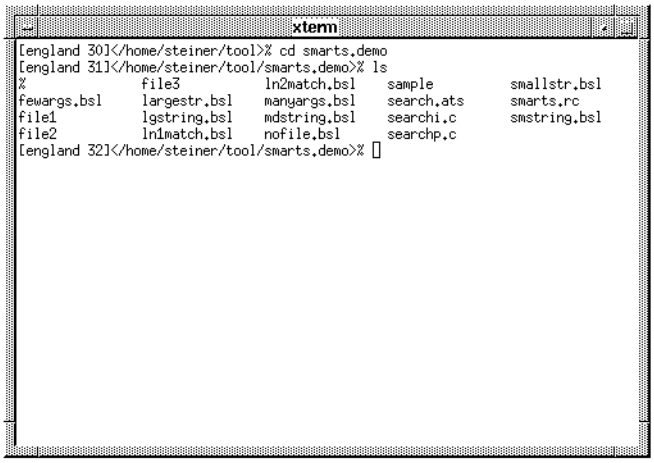

FIGURE 4 Setting Up the Display

2.1.1.1 Analyzing the Test Setup

SMARTS automates the testing process by reading a user-designed test description file, referred to as an Automated Test Script (ATS). Tests are organized within the ATS by group, sub-group, and cases, and can be supplemented with activation commands, comparison arguments, and evaluation methods. From the ATS, *SMARTS* creates a "test tree hierarchy". When the ATS is run, an outline of the test tree is displayed in the **ATS Test Tree** display area of the **Main** window. This outline provides a means of interactively controlling and monitoring the testing process.

For the purposes of this demonstration, two source files have been suppled: *searchp.c* and *searchi.c*. The *searchp.c* file is the *perfect* version of the demonstration program. The *searchp.c* program has been tested and is known to work correctly. The other file, *searchi.c*, modifies the original program and is known to have an error; thus, the *imperfect* version of the demonstration program.

Ten accompanying baseline files (indicated by the file extension *.bsl*) are used for comparisons to the output of the current program run. Also supplied are three sample test files for the *search* demonstration programs: *file1*, *file2* and *file3*. The previously described "test tree hierarchy", as well as relevant activation commands, comparison arguments and evaluation methods, are listed within the supplied ATS *search.ats*.

2.1.2 STEP 2: Compiling the Perfect Test Program

As indicated in the previous step, the *smarts.demo* directory contains two versions of the search program: *searchp.c* and *searchi.c*. The supplied ATS named *search.ats* controls and monitors the testing process for the current program run of an executable file named *search*. Therefore, before the ATS can be run by *SMARTS*, the demonstration source program to be tested must be compiled and given the executable file name *search*.

1. Because *searchp.c* is known to work correctly, compile and run it first. As the ***.c*** file name extension signifies that the *searchp.c* program is written in the C language, enter the compilation command for C programs:

```
cc searchp.c -o search
```

NOTE: When using *SMARTS*, programs and test scripts may be written in any programming language. The executable file name, however, must be the same as the executable file specified within the ATS.

After compiling the *searchp.c perfect* program, the screen display should look like this:

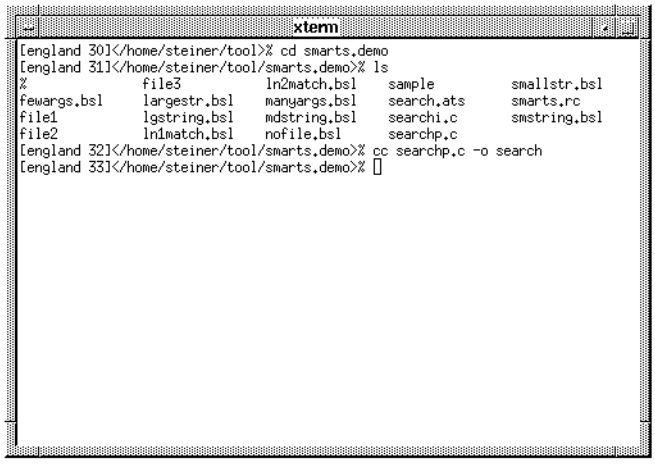

FIGURE 5 Compiling the Perfect Program

2.1.3 STEP 3: Invoking SMARTS

To invoke the GUI-version of *SMARTS:*

1. Position the mouse so that it is located in the invocation window.
2. Activate the invocation window by clicking the right mouse button on it.

   This window now becomes the main control window. During the test session, all status messages and warnings are displayed in this window.
3. Invoke *SMARTS* from the working directory by typing:

   ```
   Xsmarts
   ```

   The **Main** window pops up.

   SMARTS automatically loads the *search.ats* ATS file (which is specified in the default resource configuration file smarts.rc) and builds an outline of the file's hierarchy. This outline is referred to as a test tree. The *search.ats* test tree is displayed in the **ATS Test Tree** display, providing an immediate overview of the test groups and test cases to be executed by the *searchp.c* sample program. That is, each test group or test case indicated in the *search.ats* test tree may represent a series of commands, test files or baseline files to be invoked by the *searchp.c* demonstration program. In the current example, the *search.ats* test tree depicts three test groups and ten test cases for the *searchp.c* program.
4. Move the **Main** window to the lower left corner of the screen by clicking on the window's title bar and dragging the window to the desired location.

2.1.3.1 Re-Starting SMARTS

Re-start a test session as follows:

1. Terminate the current test session from the **Main** window by clicking on the **File** menu.
2. Select **Exit**.
3. Perform the previously described steps for invoking the *SMARTS* application.

After invoking *SMARTS* and relocating the **Main** window, the screen display should look like this:

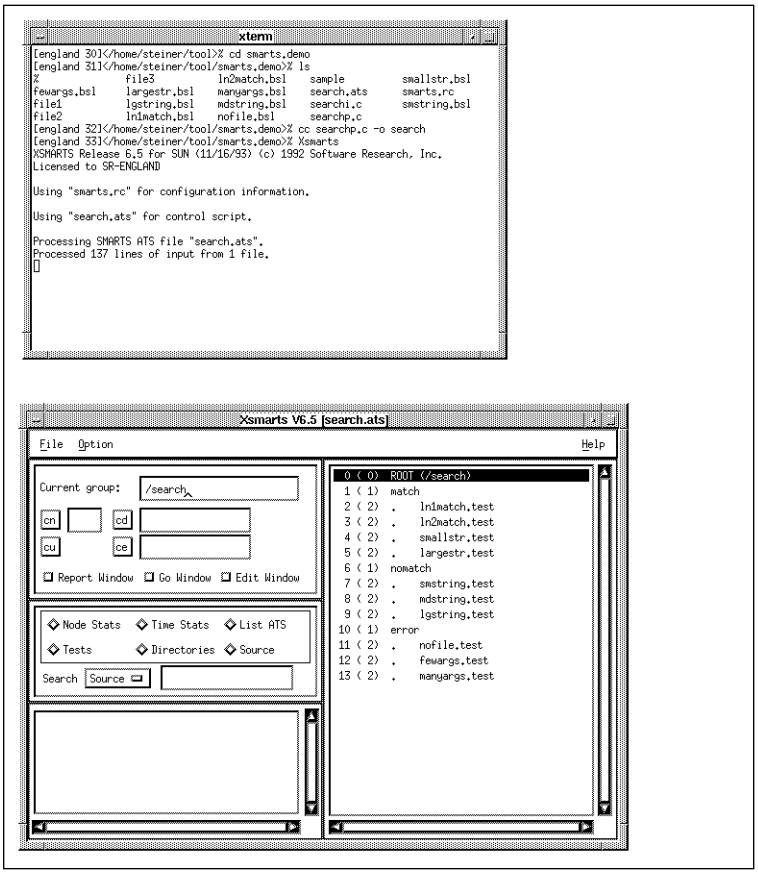

FIGURE 6 Invoking SMARTS

2.1.4 STEP 4: Opening the Go Window

In the GUI-version of *SMARTS*, tests are executed from the **Go** window. To invoke the **Go** window, perform the following:

1. In the **Main** window, click on the **Go Window** button.

   The **Go** window pops up.

2. Drag the **Go** window to the upper right corner of the screen display.

After initializing and relocating the **Go** window, the screen display should resemble the following:

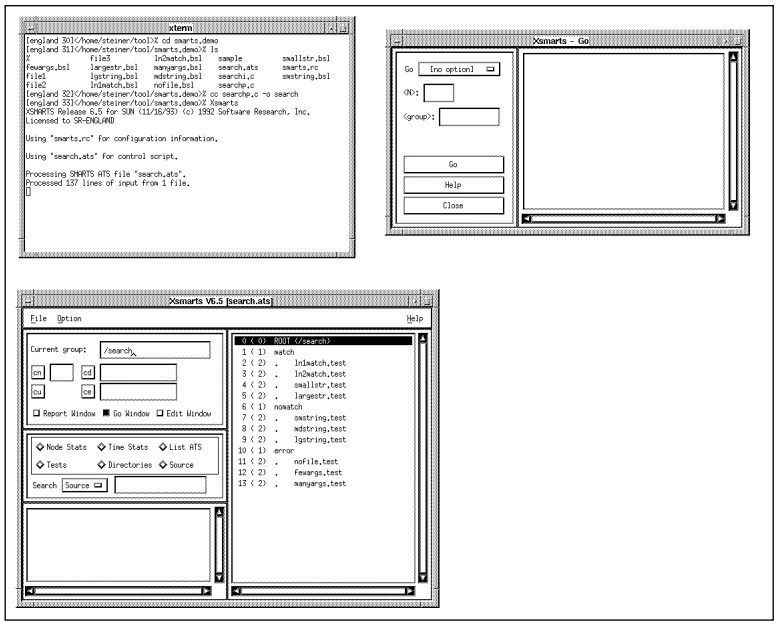

FIGURE 7 Initializing the Go Window

2.1.5 STEP 5: Opening the Report Window

In the GUI-version of *SMARTS*, the various reports generated following test execution can be viewed from the **Report** window. To invoke the **Report** window, perform the following:

1. In the **Main** window, click on the **Report Window** button.
2. The **Report** window pops up.
3. Drag the window to the lower right corner of the screen display.

Having initialized and relocated the **Report** window, the screen display resembles the following:

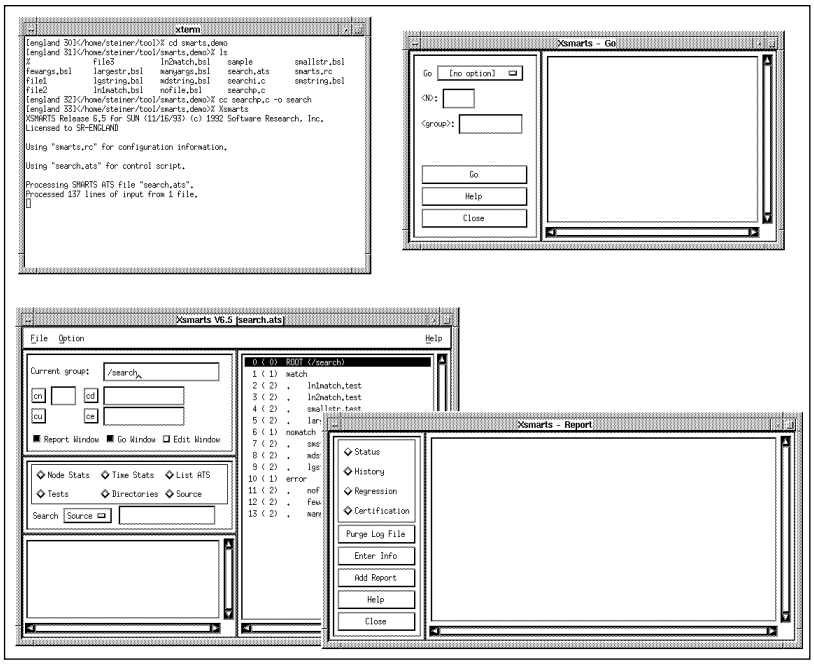

FIGURE 8 Initializing the Report Window

2.1.6 STEP 6: Testing the Perfect Version of the Demo Program

As indicated in **STEP 1,** the *search.ats* test tree displayed in the **ATS Test Tree** display of the **Main** window provides an immediate overview of the test groups and test cases to be executed by the *searchp.c* sample program.

A group's test cases or an individual test case is selected for execution by clicking the mouse on desired group or test case in the **ATS Test Tree** display. This is called the *current position*. The current position is identified by a highlighted line in the **Current group** text field of the **Main** window.

When an ATS file is initially read into the **ATS Test Tree** display, the default current position is the topmost group or test case, or the root of the test tree hierarchy. In this tutorial demonstration, all the test groups and test cases presented in the ATS will be executed.

To execute tests from the default current position of `/ROOT (/search)` onwards, perform the following:

1. Click on the **Go** window's **Go** button.

   The **Go** button toggles to **Stop** and its background / foreground color is reversed. As the `ROOT (/search)` is the top-most (or main) test group, *SMARTS* will execute all test groups and test cases indicated in the ATS, running all the tests one at a time as specified in the ATS.

   As each test executes, the current test outcome is compared to the baseline files specified within the test. The entire process, as well as test outcomes, are scrolled in the display area of the **Go** window.

   Since this is the *perfect* version of the demonstration program, all ten test cases should pass, indicated in the display area of the **Go** window by the statement: `TEST PASSES`

   When the test tree is finished executing, this message box pops up:

2. Confirm the message by clicking on the **OK** button.
3. If you wish, use the scroll bars to maneuver throughout the **Go** window's execution messages.

Note: In addition to built-in automatic differencing, *SMARTS* also lets you determine the outcome by using "manual" or visual evaluation. This is further explained in other sections (See Section 4.2.2 - "Test Case Clauses" on page 91.), (See Section 4.3.4 - "Conditional Expressions" on page 97.).

Following test execution, the screen display should look like this:

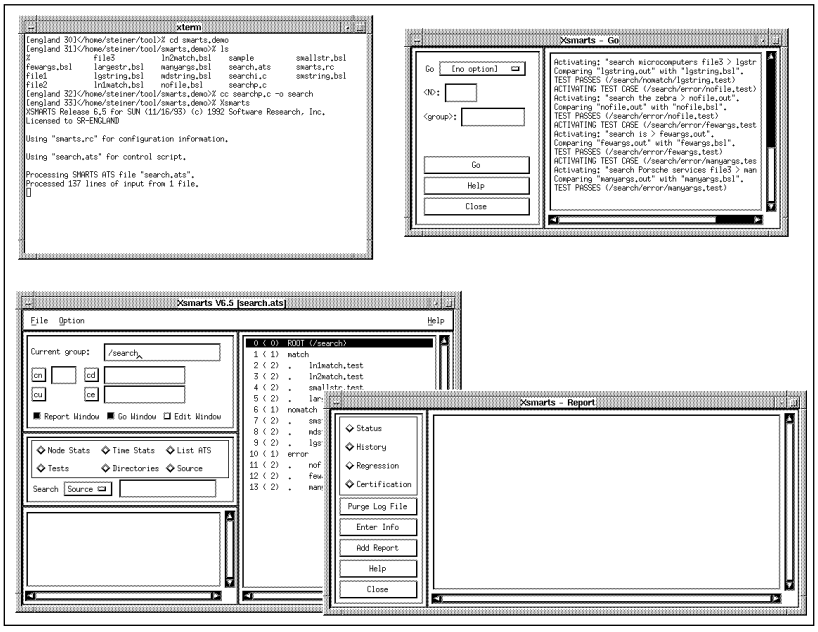

FIGURE 9 Executing a Test

2.1.7 STEP 7: Editing the ATS

After a test is executed for the first time, users may have ideas on improving the ATS file. This step shows you *SMARTS*' editing utility that lets you edit the actual ATS.

This step assumes that the current system contains xterm-like windows and a **vi** text editing utility. The edit utility allows the ATS to be edited at the location of its current position (in this case, the root directory).

Because modifications will effect the ATS, **do not** actually edit the sample ATS.

1. In the **Main** window, click on **Edit Window** button.

   The **Edit** Window pops up, displaying the *search.ats*.

2. Drag the **Edit** window so that it resides over the **Go** window and the **Report** window.

   Notice the cursor position in the ATS corresponds to the current position specified in the **ATS Test Tree** display of the **Main** window.

3. **Exit without editing or saving the file**, by entering the following command:

   :q

   The ATS is exited.

When using the **Edit** window, the screen display should look like this:

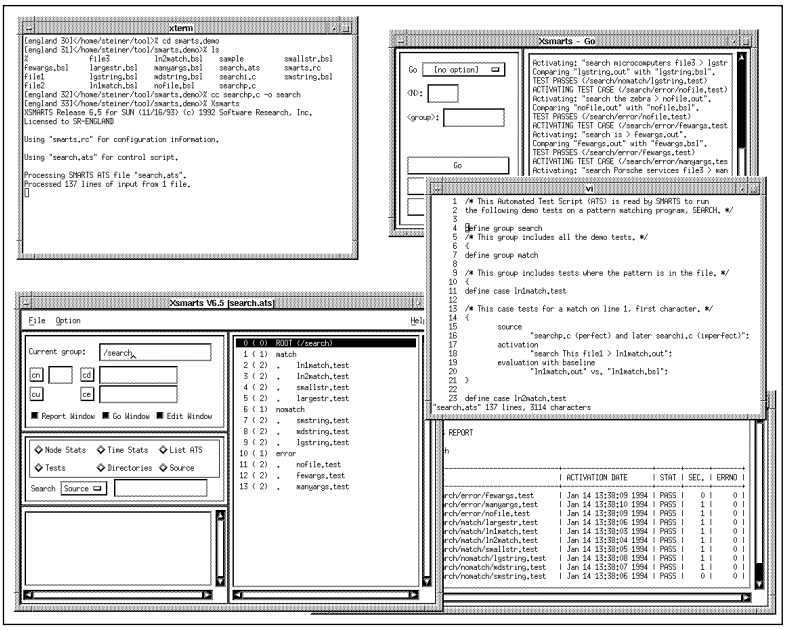

FIGURE 10 Editing an ATS

2.1.8 STEP 8: Analyzing Test Status

When tests are executed, *SMARTS* runs a difference check on the test output against the test baseline files, and accumulates a detailed record of the test outcomes into a log file. Based on the log file, *SMARTS* also generates reports.

The **Status** report depicts the current status of the selected test nodes executed, listing the test name, activation date, execution time of any test or group of tests, PASS/FAIL test outcome, and the application's (used for the evaluation method) return code.

Display the **Status** report by performing the following:

1. In the **Report** window, click on the **Status** button.

   The **Status** report appears in the **ATS Test Tree** display of the **Report** window. In the current example, all executed tests pass.
2. Use the scroll bars to maneuver through the report.

Other reports generated by *SMARTS* are:

History report — Reflects the PASS/FAIL test outcome history for all the tests from the log file. This report is useful when a number of tests have been run and a history of modification effects is needed.

Regression report — Shows only those test cases whose PASS/FAIL outcomes have changed from a previous execution, identifying bugs that been introduced or fixed. Because the log file only contains the *perfect* version's test results, this report is currently empty but will be further explored in **STEP 12**.

Certification report — Lists the total number and percentage of tests that passed or failed in the current execution. This report is ideal when a summary form of PASS/FAIL results is sufficient.

The general procedure for viewing *SMARTS* reports is similar to viewing the **Status** report:

1. From the **Report** window, click on the appropriate button. The selected report is shown in the display area.
2. Report text can be further viewed using the scroll bars.

When viewing the **Status** report, the screen display should look like this:

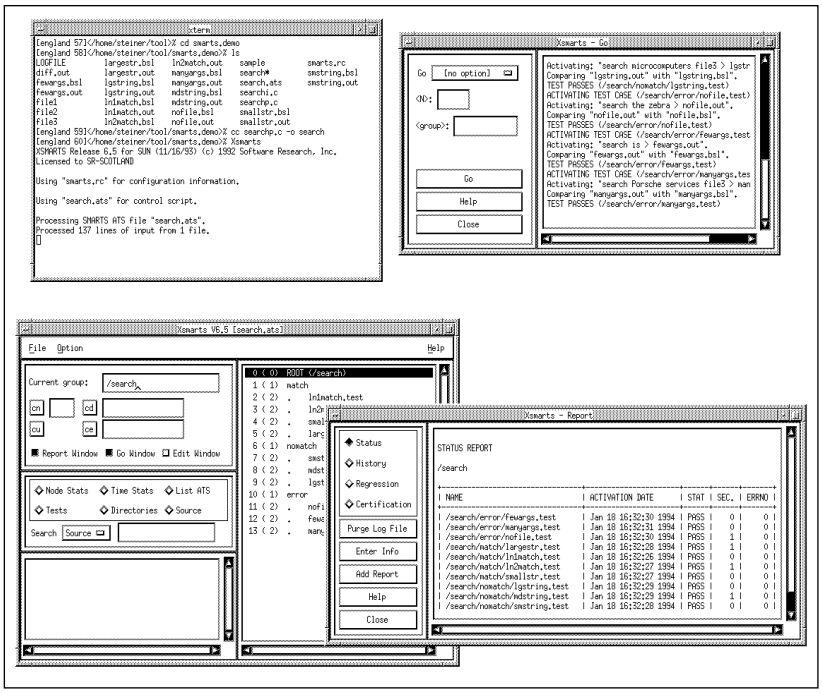

FIGURE 11 Looking at a Status Report

2.1.9 STEP 9: Exiting the Testing Session

After running the *perfect* program and viewing the **Status** report, you can terminate the testing session as follows:

1. Close the **Report** window by clicking on the window's **Close** button.
2. Close the **Go** window by clicking on the window's **Close** button.

Once you having closed any open *SMARTS* windows, exit the application itself:

1. In the **Main** window, click on the **File** menu.
2. Select the **Exit** option.

NOTE: The *SMARTS* application need not be terminated following each test session. The application is exited throughout this tutorial for you to practice.

After you have closed all open *SMARTS* windows and exited the *SMARTS* application, the screen display should look like this:

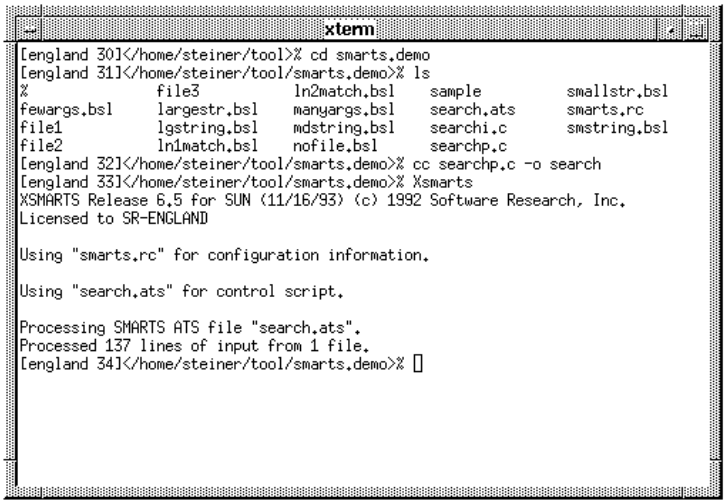

FIGURE 12 Completing the Perfect Version's Session

2.1.10 STEP 10: Compiling the Imperfect Version of the Sample Program

As previously described in this chapter, the *SMARTS* demonstration directory contains two programs: *searchp.c* and *searchi.c*. At this point in the tutorial, you should have already compiled and run the supplied *perfect* version of the demonstration program using *SMARTS*.

In the following three **STEPS**, the imperfect version of the demonstration program (*searchi.c*) is run and the program's output is examined. As previously indicated, the *SMARTS* product controls and monitors the testing process and output via a user-defined ATS. The supplied demonstration ATS *search.ats* refers to an executable file named *search*. Therefore, before the ATS can be run by *SMARTS*, the *searchi.c* source program is compiled and given the executable file name *search*.

- As the *.c* file name extension signifies that the *searchi.c* program is written in the C language, enter the compilation command for C programs:

```
cc searchi.c -o search
```

After compiling the *searchi.c imperfect* program, the screen display should look like this:

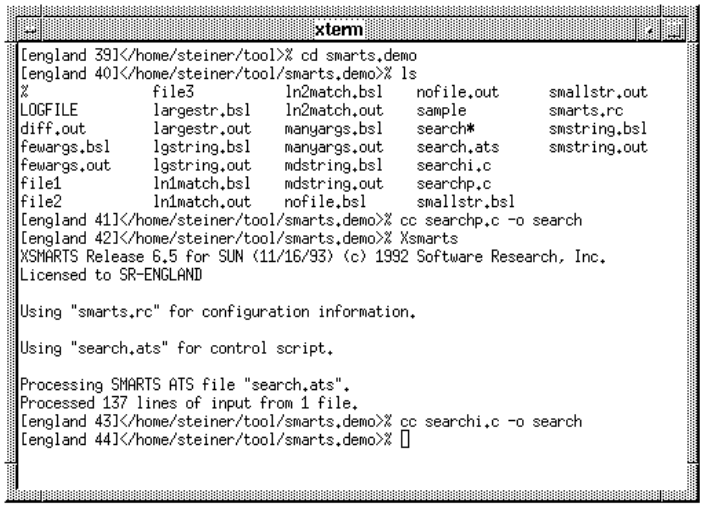

FIGURE 13 Compiling the Imperfect Program

2.1.11 STEP 11: Testing the Imperfect Version of the Demo Program

Prior to executing the test cases from *smarts.ats* for the *imperfect* version of the sample program:

- Invoke *SMARTS*.
- Initialize the **Go** and **Report** windows from the **Main** window.

To review these procedures, refer to **STEP 3** through **STEP 5**.

Previously, all test cases indicated within the *search.ats* ATS were executed when the compiled *searchp.c perfect* version of the sample program was run. When *SMARTS* is invoked, the */search* group should be selected.

To execute test cases, from the default position `/search` onward:

1. Click on the **Go** window's **Go** button.

   The **Go** button toggles to **Stop** and the background/foreground is reversed. As the `ROOT (/search)` is the top most (or main) test group, *SMARTS* will execute all test groups and test cases indicated in the ATS. *SMARTS* will execute each test in the order specified in the **Main** window's **ATS Test Tree** display.

   As each test executes, the current test outcome files are compared to the baseline files specified within the test. The entire process, as well as test outcomes, are scrolled in the display area of the **Go** window. In this *imperfect* program, all but one of the ten test cases should pass as indicated by the following statement:

   ```
   TEST PASSES
   ```

   One test case, named */search/match/smallstr.test*, fails as indicated by this statement:

   ```
   TEST FAILS
   ```

   When the test tree is finished executing, this message box pops up:

2. Confirm the message by clicking on the **OK** button.
3. If you wish, use the scroll bars to maneuver throughout the **Go** window's execution messages.

After executing a test, the screen display should look like this:

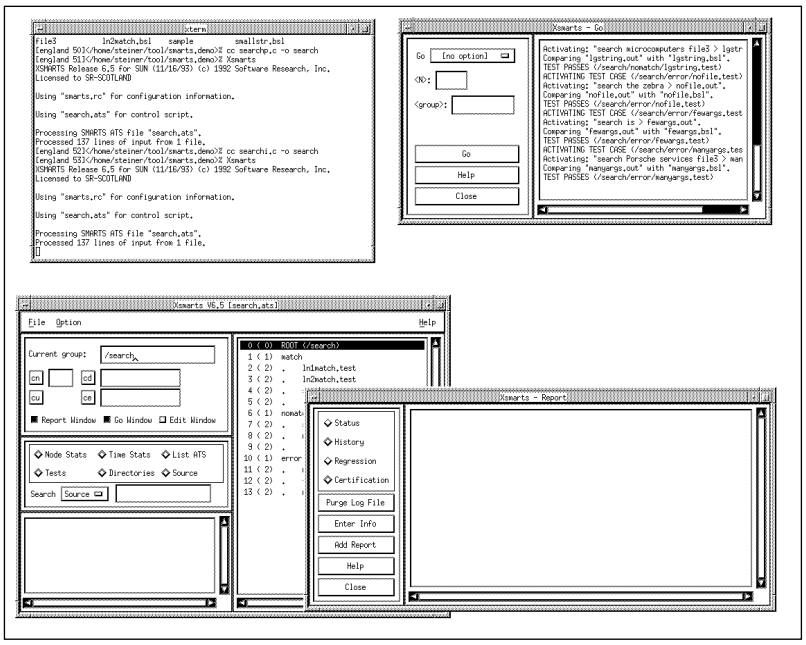

FIGURE 14 Executing a Test

2.1.12 STEP 12: Analyzing the Imperfect Demo Program — Determining Test Regression

Since all the test cases in the *perfect* program passed and one test case failed in the *imperfect* program, the test regressed. *SMARTS* offers a **Regression** report which lists only those tests whose outcome has changed, thereby identifying bugs which have been fixed or introduced since the last time the tests were activated.

2.1.12.1 Viewing Test Regression

The **Regression** report is viewed following the same procedure used to display any of the reports:

1. In the **Report** window, click on the **Regression** button.
2. The **Regression** report appears in the display area of the **Report** window.

This report displays only those test cases whose PASS/FAIL outcomes have changed from a previous execution.

3. Analyze the other reports with the **Status**, **History** and **Certification** buttons. Please refer to **STEP 8** regarding information displayed for each report type.

When viewing the **Regression** report, the screen display should look like this:

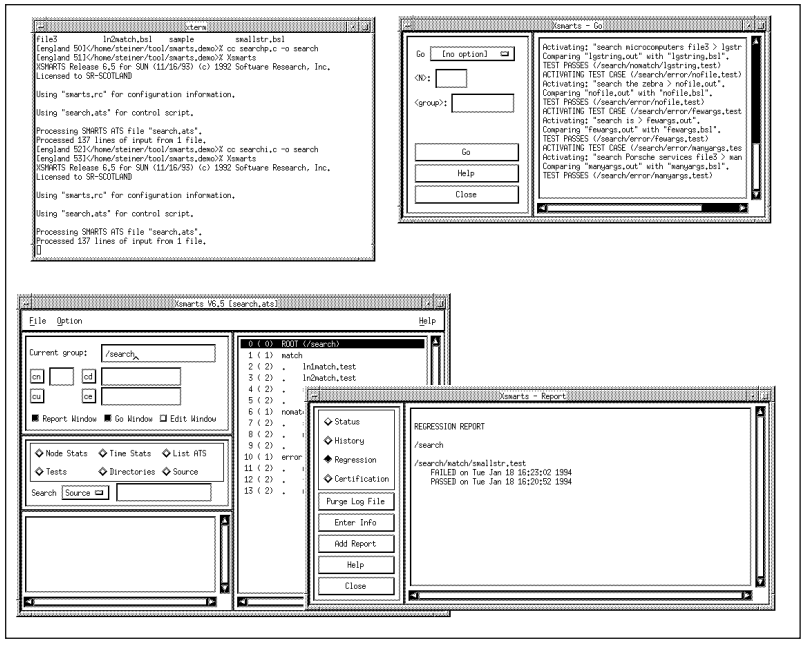

FIGURE 15 Analyzing a Regression Report

2.1.13 STEP 13: Purging the Log File

All testing data (such as test case names, start and end times, test case status, and test return values) is contained in a log file, default name *LOGFILE*. Over time, this test information accumulates, creating an immense log file and, consequently, acquiring large amounts of disk space. Therefore, it is advised to periodically purge the log file.

Purge the log file as follows:

1. In the **Report** window, click on the **Purge Log File** button. This message box pops up:

2. Click on **OK** to confirm the request. To cancel the operation, click on **Cancel**.

   If you selected **OK**, another message box pops up:

3. Confirm the message by clicking on the **OK** button.

   Only the testing data from the last run of the test cases will remain in the log file. For example, if twenty two programs have been run, following a **Purge Log File** execution, only the testing data for the twenty-second program execution would remain in the log file.

After purging the log file, the screen display should look like this:

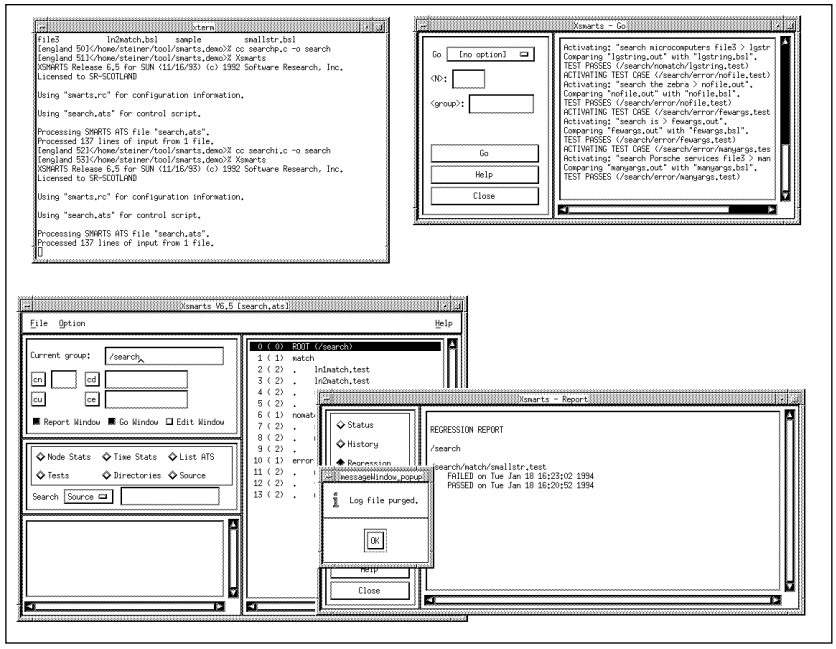

FIGURE 16 Purging the Log File

2.1.14 STEP 14: Exiting the SMARTS Product

To exit the *SMARTS* application:

1. Exit any open windows by clicking on the window's **Close** button.

   Once all *SMARTS* windows are closed, you can now exit the application itself.
2. From the **Main** window, select the **File** pull-down menu.
3. Select the **Exit** option to terminate the current test session.

After exiting the *SMARTS* application, the screen display should look like this:

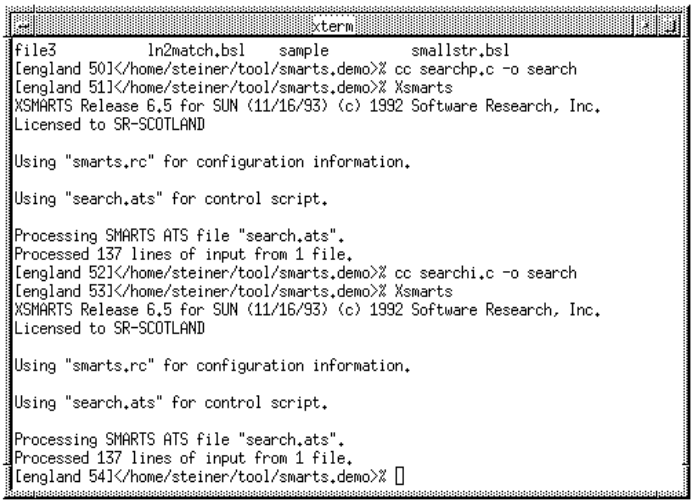

FIGURE 17 Completing a SMARTS Session

2.2 Summary

Once you have successfully completed the preceding fourteen steps, you've seen and practiced the basic skills you need to use *SMARTS* productively. In this chapter, you have examined how to invoke *SMARTS*, how to run a suite of tests, how to analyze test outcome, how to look for test regressions, and how to purge a log file.

For further practice we suggest:

- Repeat **STEPS 1 - 14** without the manual.
- Re-examine the product-supplied *smarts.ats* to review the ATS' "test tree hierarchy", supplemental commands, arguments and evaluation methods.
- Refer to Chapters 4-7 for complete information on creating an ATS file and using the *SMARTS*' GUI to manage your tests.

CHAPTER 3

Understanding the GUI

This chapter summarizes *SMARTS*' windows, menus and commands. Individual application of commands is described in detail in the relevant chapters of this guide.

3.1 Basic OSF/Motif User Interface

This section demonstrates using file selection dialog boxes, help menus, message dialog boxes, option menus, and pull-down menus. If you are familiar with the basic OSF / Motif graphical user interface (GUI) style, you can go on to another section (See Section 3.2 - "The Main Window" on page 51.).

3.1.1 File Selection Windows

SMARTS' file selection windows (Figure 18) allow you to select an existing test file name or specify a new test file name.

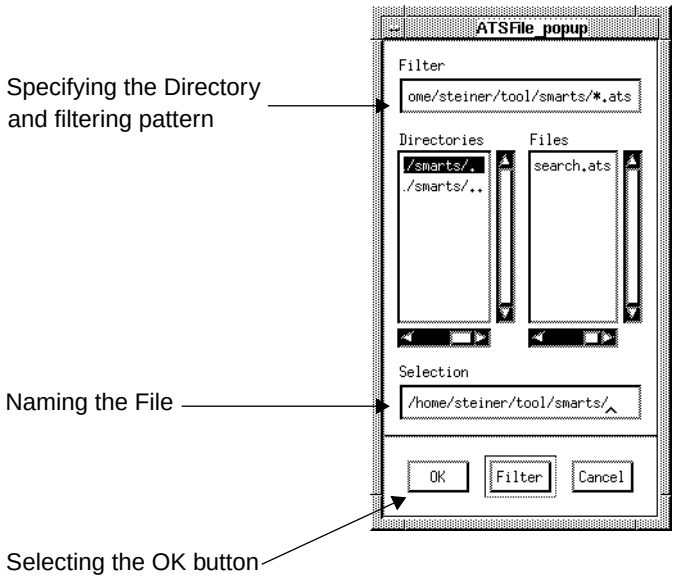

FIGURE 18 File Selection Window

The components of the file selection window are as follows:

Filter entry box	Specifies a directory mask. When you click the **Filter** button, the directory mask filters files or directories that match the indicated mask (or pattern).
Directories list box	Lists directories in path defined in the **Filter** entry box.
Files list box	Lists files for the path defined in the **Filter** entry box.
Scroll bars	Allows vertical and horizontal movement within the **Directories** and **Files** list boxes.
Selection entry box	Accepts a file name for selection.

Use the three buttons at the bottom of the window to issue commands:

OK	Accepts the file in the **Selection** entry box as the new file or the file to be opened and then exits the window.
Filter	Applies the pattern specified in the **Filter** entry box and lists the directories and files that match that pattern.
Cancel	Cancels any entered selections and exits the window.
Scroll bars	Move up / down and side / side in the **Directories** and **Files** list boxes.

3.1.1.1 Using a File Selection Window

The file selection operation can be restricted to a named region (directory path) by either:

- Typing a directory path name in the **Filter** entry box, or
- By clicking on a path name in the **Directories** list box.

  The **Filter** push button is then clicked on.

To select a file name, perform one of the following activities:

- Double click on the file in the **Files** list box.
- Highlight the file in the **Files** list box or type in the file name in the **Selection** entry box and click on **OK**.
- Highlight or type in the file name and press the **Enter** key.

When a procedure requires a file to be created, create a new file by either:

- Double clicking on an existing file in the **Files** list box to be overwritten, or
- Entering a new file name in the **Selection** entry box and either clicking on **OK** or pressing the **Enter** key.

3.1.2 Help Windows

SMARTS provides on-line information for each of its windows. When a window's **Help** option is activated, an information window containing pertinent text is displayed. In other words, if invoked from the **Record/Play** window, the **Help** window will automatically display information relevant to the **Record/Play** window.

To access on-line help:

1. Click on the window's **Help** option.

   The **Help** window pops up, listing informational text corresponding to the window from which the **Help** option was activated.

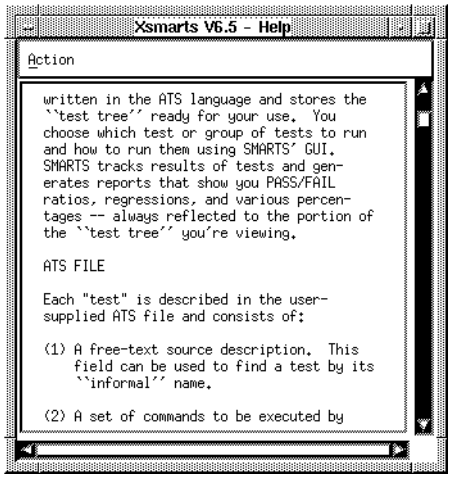

FIGURE 19 Help Window

The displayed text can be traversed by either clicking on the text and dragging the mouse, or using the vertical and horizontal scroll bars.

2. Click on the **Action** menu and select the **Search** option to search for specific text. The following dialog box pops up:

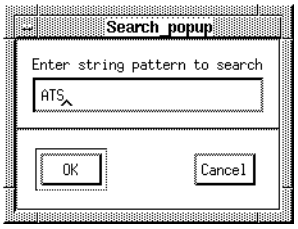

FIGURE 20 Search Dialog Box

3. Click the cursor in the **Enter string pattern to search** region and type in the search pattern string.
4. Either click on **OK** or press the **Enter** key.

   If the pattern is found, then the window will automatically scroll to the location of the specified pattern.

   If the pattern is *not* found, the following message box is displayed:

FIGURE 21 Help Message Box

NOTE: If the **Help** window is currently displayed and a **Help** option from another window is activated, the **Help** window automatically scrolls to the relevant text for the new window.

5. To close the **Help** window, click on the **Action** menu and select the **Exit** option.

3.1.3 Pull-Down Menus

Pull-down menus are located within the menu bar of *SMARTS*' windows. They often contain several options.

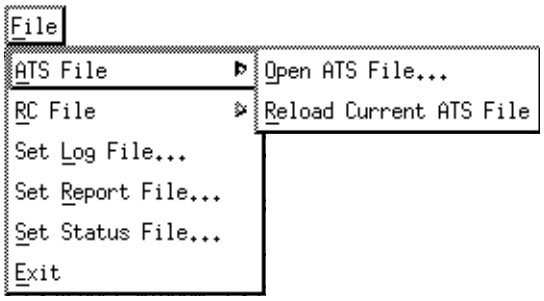

FIGURE 22 Pull-Down Menu

To use pull-down menus and their options, perform the following steps:

1. In the title bar, place the mouse pointer over the menu name.
2. Display the menu's options by holding the left mouse button down.
3. While holding down the left mouse button, slide the mouse pointer to the desired menu option. The menu option is highlighted in reverse shadow.

NOTE: Three dots to the right of a menu item indicates that selecting the item will display a pop-up window, such as a file selection window. An arrow to the right of the menu indicates that item has a submenu.

---

4. To activate a command, release the mouse button while the desired item is highlighted. To exit without selecting an item, simply drag the mouse pointer off the menu before releasing the mouse button.
5. To display the submenu, slide the mouse pointer over the arrow. You can then select an item on the submenu.
6. Release the mouse button while the desired item is highlighted to activate the command. To exit without selecting anything, simply drag the mouse pointer off the menu before releasing the mouse button to not activate anything.

3.1.4 Option Menus

Activation buttons containing smaller, raised buttons display option menus. The option indicated on the activation button for an option menu is the current default. To use an option menu, perform the following:

1. Using the left mouse button, click on the activation button for the option menu. A list of available options is displayed.

NOTE: To continue displaying the option menu, the left mouse button must be held down.

---

2. Drag the mouse to the desired option menu item.
3. Release the mouse. The activation button for the option menu now displays the selected item, indicating that the option is now activated.

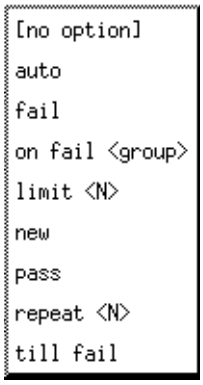

FIGURE 23 Option Menu

3.1.5 Message Boxes

Pop-up message boxes serve three purposes:

- Display warnings and error information.
- Prompt for a command.
- Request verification for command execution.

To either remove a message box or execute the current command, click on the **OK** button.

To cancel a command, click on the **Cancel** button.

FIGURE 24 Message Box

3.2 The Main Window

When the *SMARTS* graphical user interface (GUI) is invoked from the OSF / Motif X Window System, the **Main** window is first displayed.

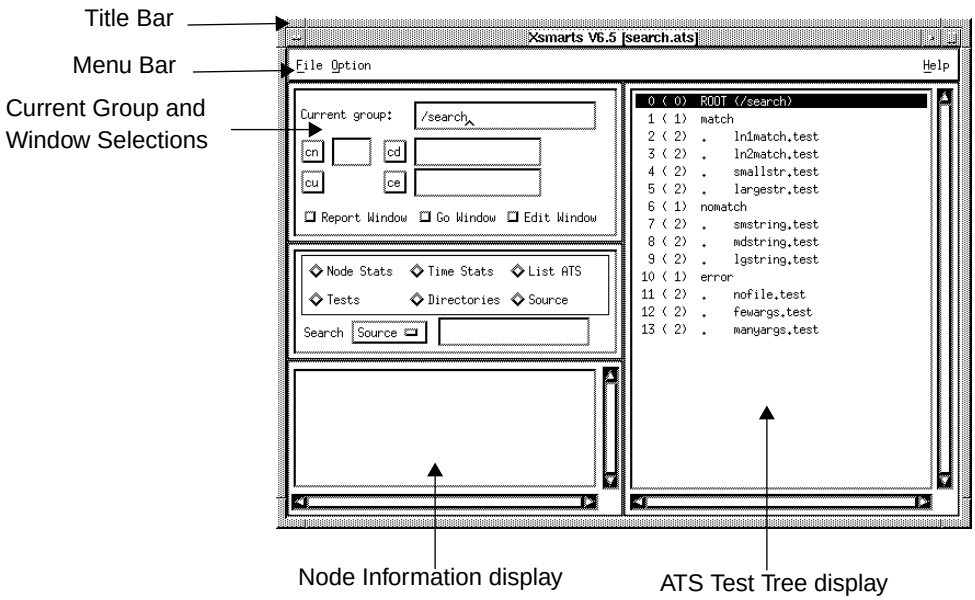

FIGURE 25 Main Window

The **Main** window is divided into the following four sections:

1. Menu bar.
2. **ATS Test Tree** display.
3. **Current group** and window selections.
4. **Node Information** display.

Note: Potential discrepancies between the **Main** window invoked and the **Main** window depicted above will be addressed throughout the chapter.

3.2.1 Menu Bar

The Menu bar section spans the length of the top of the **Main** window and is comprised of the following items:

1. **File** menu.
2. **Options** menu.
3. **Help** option.

3.2.1.1 File Menu

The **File** menu allows you to set files that can be used instead of the files specified in the configuration file or with one of the options during invocation.

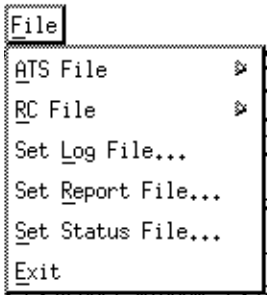

FIGURE 26 File Menu

ATS File As indicated by the arrow to the right of the option, the **ATS File** cascading menu accesses a submenu (or cascading menu).

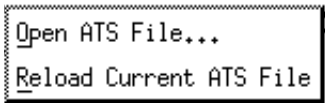

FIGURE 27 ATS File Submenu

The ATS File submenu consists of the following two options:

Open ATS File... Invokes a file selection window through which an existing Automated Test Script (ATS) file can be selected. Once selected, the ATS is automatically loaded into the **ATS Test Tree** display.

The file selection window uses a ***.ats** file mask.

Reload Current ATS File

Reloads the current ATS file into the **ATS Test Tree** display. This option is useful if the ATS file is modified with the **Edit** window and you want the modifications to be reflected in the **ATS Test Tree** display. Furthermore, this option eliminates the need to invoke the **Open ATS File** dialog box when selecting the same ATS file.

RC File As indicated by the arrow to the right of the option, the **RC File** option accesses a submenu (or cascading menu). The **RC File** cascading menu refers to the resource configuration file which consists of a series of run-time parameters (file names to be accessed, processing time limits, etc.). Run-time parameters can be set or modified in the RC file using any of the following methods:

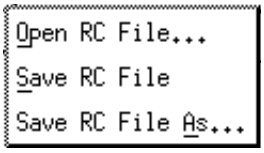

FIGURE 28 RC File Submenu

Open RC File... Invokes a file selection window where an existing resource configuration file can be opened.

The file selection window uses a ***.rc** file mask.

Save RC File... Saves the current resource configuration file under its existing file name.

Save RC File As... Invokes a file selection window to save the current resource configuration file under a different name.

The file selection window uses a ***.rc** file mask.

Set Log File...	Invokes a file selection window to rename the current name of the log file. The log file is where current and past test information is stored. If you do not set a specific log file name to store test information, all information is automatically stored to the file specified in the current resource configuration file or *SMARTS*' default file named *LOGFILE*. The file selection window uses a ***** file mask.
Set Report File...	Invokes a file selection window to specify a file where report information will be stored. After viewing a report from the **Report** window, its information can then be stored to the previously specified file using the **Report** window's **Add Report** button. If the **Add Report** button is not executed, the previously specified file will remain empty. The file selection window uses a ***** directory mask.
Set Status File...	Invokes a dialog box to specify a file to store all runtime messages displayed in the **Go** window during test activation. The file selection window uses a ***** directory mask.
Exit	Terminates the *SMARTS* application.

NOTE: The file selection window's directory masks can be changed by editing the *SR* file. Please see Appendix A, "CUSTOMIZING THE GUI ENVIRONMENT", for further information. Please refer to other sections for resource configuration information (See Section 5.4 - "Resource Configuration File Processing" on page 125.) and for command line information (See Section 5.3 - "Command Line Invocation" on page 118.).

3.2.1.2 Option Menu

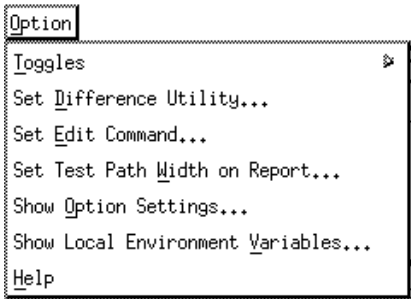

FIGURE 29 Option Menu

Toggles As indicated by the arrow to the right of the option, the **Toggles** option accesses a submenu.

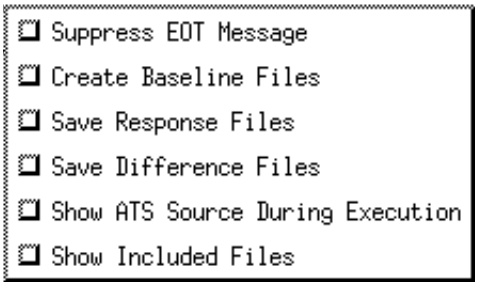

FIGURE 30 Toggles' Cascading Menu

The **Toggles** submenu check button items are as follows:

Suppress EOT Message

Suppresses the **Go** window's pop-up dialog box indicating test case execution completion with the comment: `Execution completed.`

Create Baseline Files

For each response-to-baseline comparison indicated in a failed test's `evaluation with baseline` clause, the response file is copied to the corresponding baseline file.

Note: Within the syntax for the `evaluation with baseline` clause, the response file is indicated first:

`evaluation with baseline`
response_file **vs.** *baseline_file*

This option serves two purposes: (1) When non-existent, creates baseline files. (2) Overwrites defective baseline files. This option is only relevant for tests containing the `evaluation with baseline` clause.

Save Response Files Saves the current response file. If this option is left off, the current response files are removed after test execution. It is recommended that this option be turned off if response files accumulate too much disk space.

Save Difference Files Saves the difference output for each test case that fails to a file named *basename*.`diff`. This option is particularly helpful in identifying and locating bugs. This option works only with the following evaluation modes: `evaluation with baseline` and `evaluation with function`.

Show ATS Source During Execution

During test execution, displays the ATS `source` clause(s) relevant to the current position in the **Go** window.

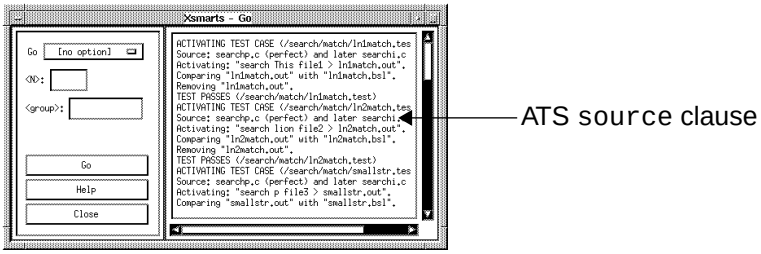

FIGURE 31 Showing the Source Clause During Execution

Show Included Files

If files are included in the ATS, this option displays the included files in the **ATS Test Tree** display's hierarchy.

Note: The **Toggles**' options may permanently defaulted on or off by modifying the GUI's *SR* file. Please see Appendix A, "CUSTOMIZING THE GUI ENVIRONMENT", for further information. The current resource configuration file can also be modified. See another chapter for further information (See CHAPTER 5 - "Invoking SMARTS" on page 115.).

Set Difference Utility...

Invokes a message box where the command to difference baseline and response files can be set for the `evaluation with baseline` evaluation mode.

SMARTS defaults to the command set in the current resource configuration file upon start-up.

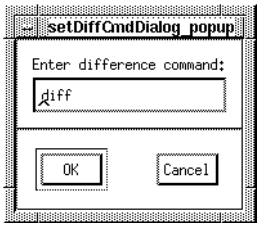

FIGURE 32 Setting the Difference Command

One of four utilities can be specified:

- **diff** to difference ASCII files. This is *SMARTS*' default.
- **exdiff** to extended differences of ASCII and binary files.
- **Xexdiff** for differences of saved X Window bitmap files.
- Other differencing utilities.

exdiff and **Xexdiff** are *EXDIFF*'s utilities. Please refer to the *EXDIFF User Manual* for further information on these utilities.

Set Edit Command...

Invokes a message box where a command to initialize an xterm-type window and to establish a text editor (in this case, the vi editor) for the **Edit** window can be specified.

FIGURE 33 Setting the Edit Command

The **Edit** window allows you to edit the ATS file. It displays the ATS according to the current position selected.

SMARTS defaults to the command set in the current resource configuration file upon start-up.

Set Test Path Width on Report...

Invokes a message box where the width for the **Status** and **History** reports can be set.

FIGURE 34 Setting the Report Width

Width is specified in terms of characters. The width is defaulted to 30 characters. The width must be greater than 3 characters but less than 513 characters.

SMARTS assumes the width set in the current resource configuration file.

Show Option Settings...

Invokes a message box that displays the current configuration file's settings. Unless a resource configuration file is set during invocation or with the **RC File** option (See Section 3.1.1.1 - "Using a File Selection Window" on page 45.), *SMARTS* assumes the default file, *smarts.rc*.

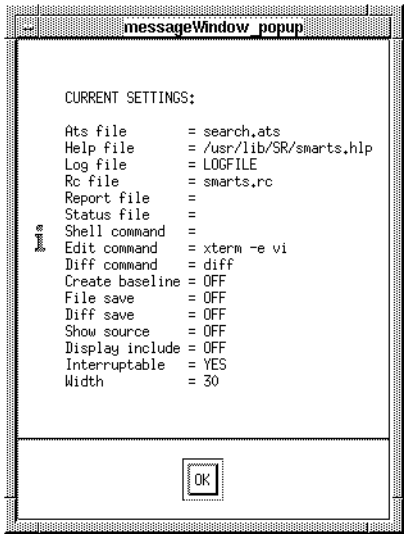

FIGURE 35 Displaying the Configuration File's Settings

Show Local Environment Variables

Invokes a window that displays the aggregate environment variables defined in the ATS' `environment` clauses for the executed test cases in the most recent invocation of *SMARTS*. Please refer to other sections for further information on the `environment` clause (See Section 4.2.2 - "Test Case Clauses" on page 91.), (See Section 4.3.6.2 - "The Environment Clause" on page 101.).

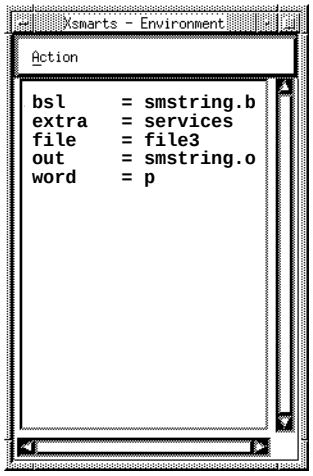

FIGURE 36 Showing the Local Environment Variables

Help Initiates a **Help** window which contains information for the **Option** menu.

3.2.1.3 Help Option

The **Help** option invokes a **Help** window which describes the basic features of the **Main** window.

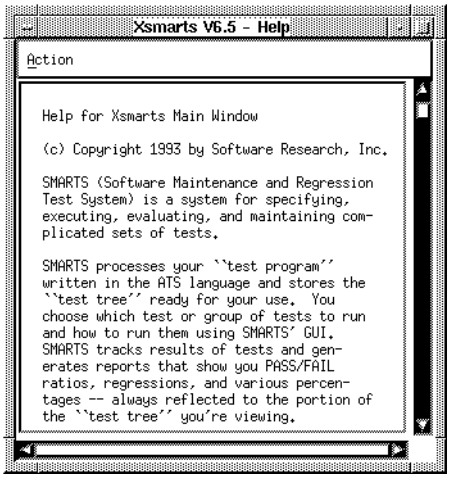

FIGURE 37 Help Window for the Main Window

3.2.2 ATS Test Tree Display

The **ATS Test Tree** display is located at the right side of the **Main** window. When an Automated Test Script (ATS) is selected, *SMARTS* automatically generates a directory type listing of the ATS in the **ATS Test Tree** display.

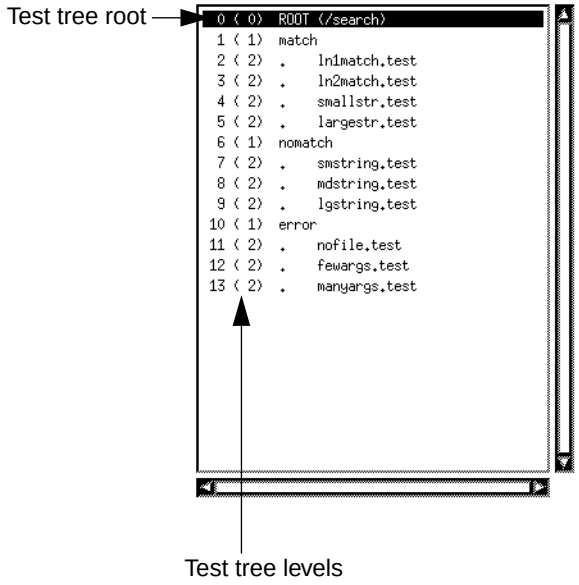

FIGURE 38 ATS Test Tree Display

Tests are organized in a tree structure that represents the group, subgroup and test cases of the ATS. The left-most number is the test tree identification number (0-13 in the example above). The parenthesized number is the level that the group or test case is at in the test tree. In the example above there are three levels: `(0)` represents the *root*, or the main group of the test tree, `(1)` represents the sub-groups (`match`, `nomatch`, and `error`), and `(2)` indicates test cases.

This "test tree hierarchy" provides a means of selectively executing only those tests of interest. When *SMARTS* initially loads the chosen ATS file into the **ATS Test Tree** display, the root of the test hierarchy (the first test case or group in the ATS file) is the *current position*. Selected tests are all test cases that fall *under* the current position.

The current position can be altered by either using the mouse to click on a test or via the **Current group** text field of the **Main** window. For further information on the **Current group** text field of the **Main** window, refer to

the section that details it (See Section 3.2.3 - “Current Group” on page 67.). Or the current position can be changed by directional commands **cn**, **cd**, **cu**, or **ce**, described in another section (See Section 3.2.3 - “Current Group” on page 67.). The **ATS Test Tree** display can be traversed using either a mouse or the vertical and horizontal scroll bars.

3.2.3 Current Group

The **Current group** section is located beneath the **Title Bar** in the upper left corner of the **Main** window.

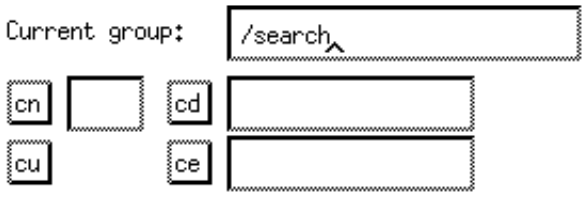

FIGURE 39 Current group Section

The test group or case to be used as the current position is determined via the **Current group** section. The current position is either selected with the mouse in the **ATS Test Tree** display, entered in the **Current group** text field or established using the following options:

cn (**c**urrent *n*th) text field

Changes the current position to the *n*th descendant listed beneath the current position. In the **cn** text field *n* is entered, where *n* indicates the *n*th descendant of the current position.

cd (**c**urrent **d**escendent) text field

Changes the current position to the *name* specified in the text field; the text entered must be an immediate descendant of the current position. For example, in the test tree hierarchy shown in the figure (See Figure 40 "Hierarchical Test Tree Structure" on page 68.), consider A as the current position. If A is the current position, C becomes the current position after typing **C** in the text field. G, however, is not accessible directly from A. The inverse operation of **cd** is entering **cu** in the text field.

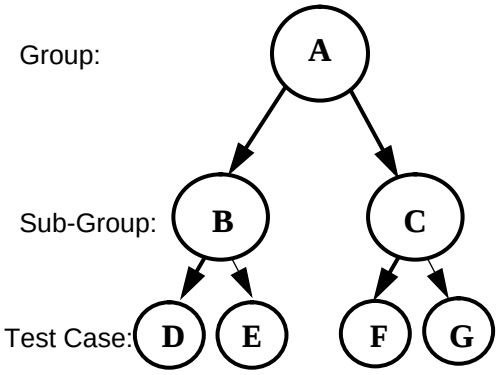

FIGURE 40 Hierarchical Test Tree Structure

cu (**c**urrent **u**p) Option

Changes the current position to the preceding test group or case. Referring to the figure above, if test case G were the current position, C would be the current position after **cu**. The inverse operation of **cu** is entering **cd** in the text field.

ce (**c**urrent **e**nd) string text field

Changes the current position to a descendant with a substring ending in some *pattern*. For example:

```
test.33
test.34
test.35
test.36
test.37
```

In the above listing, if **35** is entered in the **ce** text field, the current position would change to `test.35`.

3.2.4 Window Selections

Within the **Current group** section of the **Main** window, the following three windows can be invoked:

1. The **Report** window.
2. The **Go** window.
3. The **Edit** window.

The functionality of these windows is examined next.

3.2.4.1 Report Window

When invoked from the **Main** window, the **Report** window is displayed as follows:

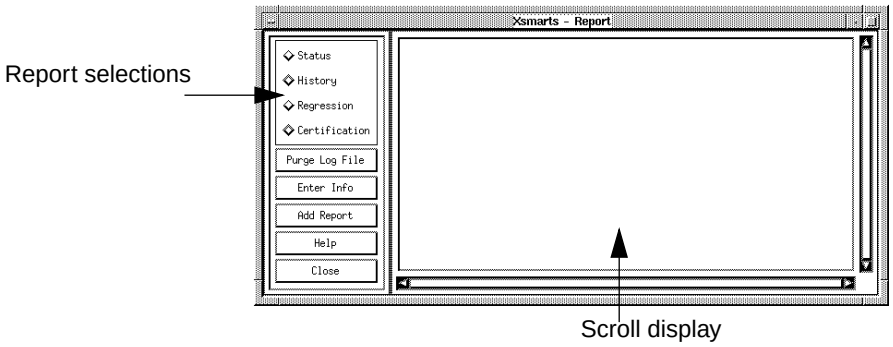

FIGURE 41 Report Window

Proceeding test execution with the **Go** window, four different types of reports can be accessed using the following buttons:

- **Status** button.
- **History** button.
- **Regression** button.
- **Certification** button.

The **Report** window also contains the following features:

- Scroll display.
- **Purge Log File** button.
- **Enter Info** button.
- **Add Report** button.
- **Help** button.
- **Close** button.

Each of these features is described next.

Status button

Based on the log file, produces a report which reflects the test outcomes of the most recent test execution, in respect to the current position selected. Therefore, more information will be generated if the test was executed from a root or a group than if only a single test case is executed.

The **Status** report contains the name of each test case within the current position, the PASS/FAIL status of each test case, the date of test activation,execution time in seconds, and includes the error value returned by each test case. Provided information can help locate test cases that failed during execution.

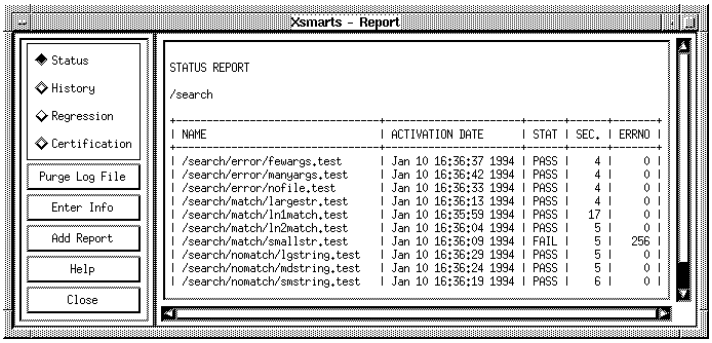

FIGURE 42 Status Report

History button — Produces a summary report of all test outcomes maintained in the log file, providing an overview of test regression throughout the testing process. If log file has not been purged in a while, then the **History** report can be quite extensive, covering all old and new test executions.

Test information is organized within the **History** Report by test case name. For each test case, the date(s) and PASS/FAIL status of all recorded test executions are listed. Below is a sample **History** report.

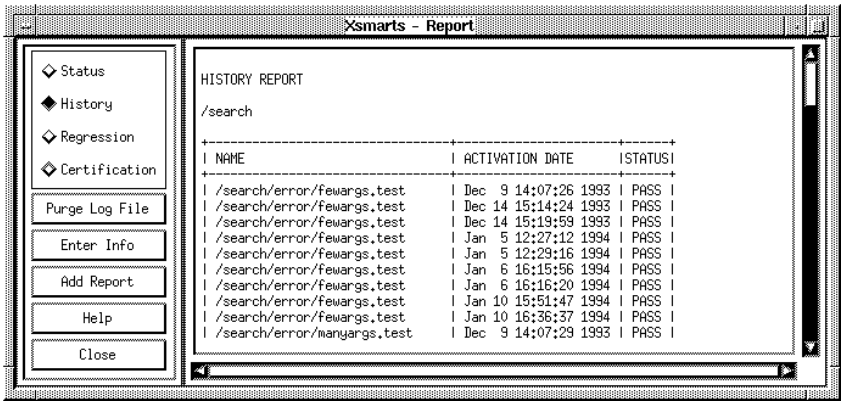

FIGURE 43 History Report

Regression button

Based on the log file, produces a report which indicates only the most recently executed test cases whose outcomes have changed since the previous execution.

The **Regression** report helps to identify bugs that have been either fixed or introduced since the last time the tests were activated.

The **Regression** report contains the name of the test group (if applicable) or the test case name whose outcome has changed, the PASS/FAIL status and date of the most recent test execution, and the PASS/FAIL status and date of the previous test execution.Using this information, the source code to locate a bug can be quickly inspected. A sample **Regression** report is shown below.

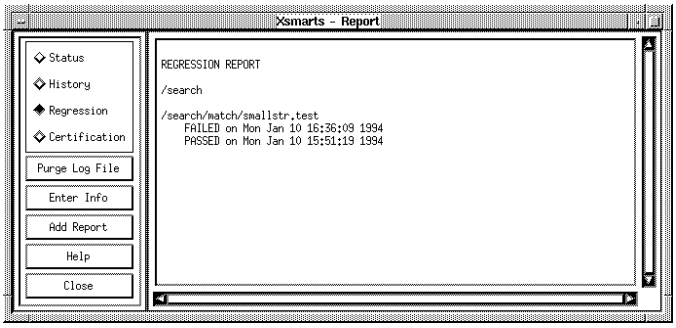

FIGURE 44 Regression Report

Certification button

Based on the log file contents, summarizes the total number and percentage of PASS/FAIL outcomes for the current position's selected test(s).

The **Certification** report contains the name of the current position, the PASS/FAIL status for each test case and the overall PASS/FAIL status for the current position. Below is a sample **Certification** report.

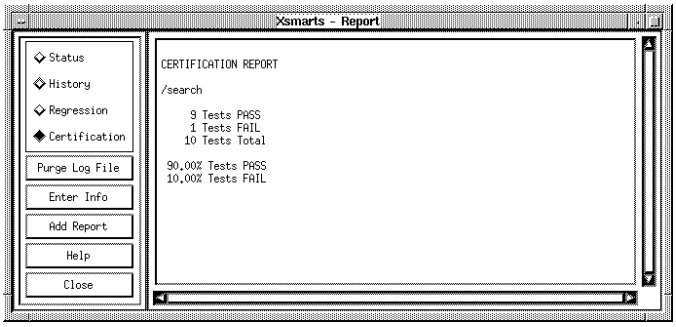

FIGURE 45 Certification Report

Purge Log File button

All testing information, such as test case name, timing information and return values, is stored in the log file. (The default name for this file is *LOGFILE*.) Over a period of time, the log file can become quite large and accumulate extensive amounts of disk space.

The **Purge Log File** button deletes old log file records and maintains only the most current data on each test case. If a test, for instance, were executed twenty times, only the twentieth activation would remain in the log file after purging.

Purging will not affect the **Status** and **Certification** reports; however, the **History** report will only have the most recent data and the **Regression** report will be empty until tests are executed again.

Enter Info button

The **Enter Info** button brings up a dialog box for entering text for the header of any of the four reports:

- **Tester's Name:**
- **Version Number:**
- **Test Info:**

Following is a sample of the information window.

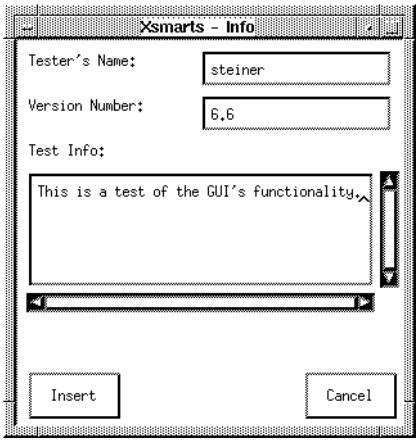

FIGURE 46 Enter Info Window

Add Report button — Automatically enters the currently displayed report into a previously specified file. The specified file can be set either using the **Main** window's **Set Report File...** option (See Section 3.2.1.1 - "File Menu" on page 52.) or indicating the file name in a resource configuration file. The file can be displayed later using any text editor.

Help button — Invokes a **Help** window which provides on-line explanation of the **Report** window.

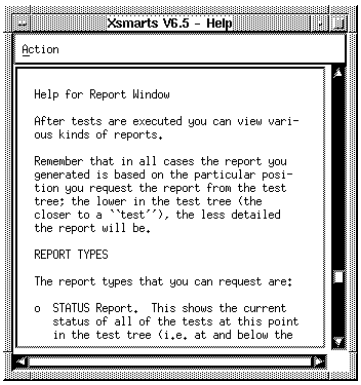

FIGURE 47 Help Window for the Report Window

Close button — Closes the **Report** window.

3.2.4.2 Go Window

When invoked from the **Main** window, the **Go** window is displayed:

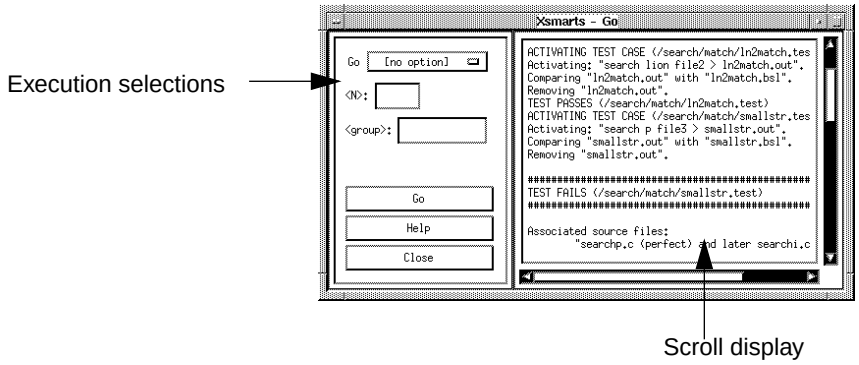

FIGURE 48 Go Window

Regardless of the current position selected in **Main** window's test tree, all functions necessary to execute all or part of the ATS file are accessible from this window.

The **Go** window contains the following features:

- **Go** option menu.
- **<N>** text field.
- **<group>** text field.
- **Go** button.
- **Help** button.
- **Close** button.
- Scroll display.

These features are described on the following pages.

Go option menu — The ATS file is executed from the **Go** window according to one of the following optional commands:

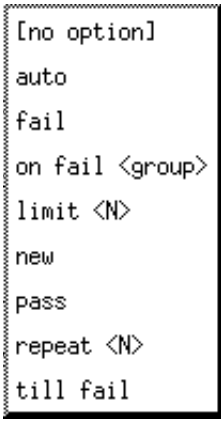

FIGURE 49 Go Option Menu

[no_option] — Commands *SMARTS* to execute from the current position onward until there are no more tests. In the figure below, consider A as the current test. When the **[no_option]** is specified, tests D, E, F, and G will all be executed. Note, however, if C is the current test only F and G will be run.

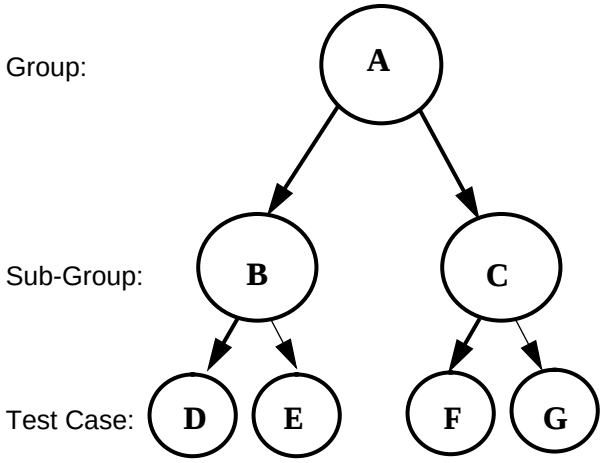

FIGURE 50 Hierarchical Test Tree Structure

auto If an ATS file contains `evaluation with user` clauses, this option is recommended to run the ATS file unattended. If the **auto** option is not selected for such an ATS file, then a confirmation message box will pop up every time an `evaluation with the user` clause is encountered. The confirmation message box requires the **PASS** or **FAIL** button to be clicked before continuing the test execution process. Otherwise, cases with the "`evaluation with user`" clause is ignored.

fail Based on the existing contents of the log file, commands *SMARTS* to execute (from the current position onward) only those test cases that failed during the previous execution.

on fail <*group*> Commands *SMARTS* to execute from the current position onward until a test fails. Should a test fail, *SMARTS* then executes the *group* specified. When the indicated test *group* has completed execution, control returns to the **Go** window.

NOTE: Prior to activating this option, the **<group>** text field *must* be set. Refer to **<group>** text field explanation.

---

limit <*N*> Commands *SMARTS* to execute the current group within a time limit of *N* seconds per test. If any test is longer than *N* seconds, the test fails. The testing process, however, continues.

Note: Prior to activating this option, the <*N*> text field *must* be set. Refer to <*N*> text field explanation.

---

new Commands *SMARTS* to execute from the current position onward only those test cases that have not been previously recorded in the current log file.

pass Commands *SMARTS* to execute (from the current position onward) only those test cases that passed during the previous execution.

repeat <*N*> Executes the ATS file *N* times from the current position.

NOTE: Prior to activating this option, the **<N>** text field *must* be set. Refer to **<N>** text field explanation.

`till fail` Commands *SMARTS* to execute from the current position onward until the first fail, which terminates test execution.

`<N>` text field Used in tandem with the **limit <N>** and **repeat <N>** optional **Go** commands. The *N*umber of seconds limiting individual test execution is entered in the **<N>** text field prior to selecting the **limit <N>** option. The *N*umber of times the ATS file will be executed from the current position onward is entered in the **<N>** text field before selecting the **repeat <N>** option.

`<group>` text fieldUsed in tandem with the **on fail <group>** optional **Go** command. The *group* name that is to be executed should a test fail is entered in the **<group>** text field prior to selecting the **on fail <group>** option. When the indicated test group has completed execution, control returns to the **Go** window.

`Go` button Selected after establishing the current position and type of test execution with the **Go** option menu. When activated, *SMARTS* will execute the selected test(s) and the **Go** button will toggle to the **Stop** button. When the selected tests have completed execution, all information is stored in a log file. To terminate execution at any point, click the **Stop** button and the test execution will stop, following the completion of the currently executing test case.

Help button Displays a **Help** window which explains the **Go** window.

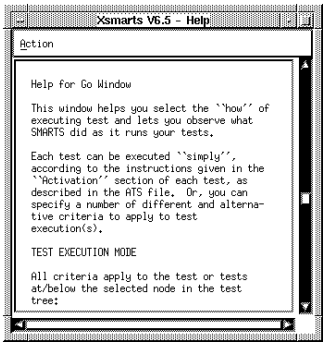

FIGURE 51 Help Window for the Go Window

Close button Closes the **Go** window.

3.2.4.3 Edit Window

The **Edit** window is an xterm-like window. It displays the ATS and indicates the current position with a cursor. When a current position is changed with the **ATS Test Tree** display or with the **Current group** sections of the **Main** window, the **Edit** window's cursor moves to the new current position's location in the ATS.

Editing is done in respect to the kind of text editor specified in the current resource configuration file or with **Option** menu's **Set Edit Command** option in the **Main** window.

If you make changes to the ATS that you want reflected in the **Main** window's **ATS Test Tree** display, use the **File** menu's **Reload Current ATS File** option in the **Main** window. Please see the section that details this information (See Section 3.2.1.1 - "File Menu" on page 52.).

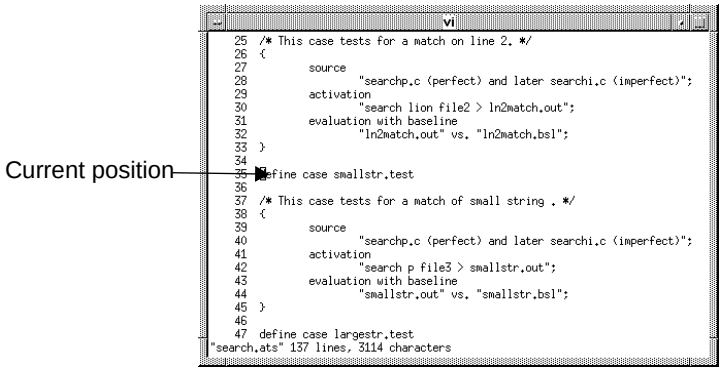

FIGURE 52 Edit Window

3.2.5 Node Information Display

The **Node Information** display is located in the lower left corner of the **Main** window. The **Node Information** buttons list various types of statistical file and test information in the scroll display. The scroll display can be traversed using either a mouse or the vertical and horizontal scroll bars. Selection of any button activates the display.

Each of the **Node Information** buttons are examined next.

Node Stats button — Lists the hierarchical statistics for the current position, displaying the number of levels in the test tree and the number of test groups or cases in each level.

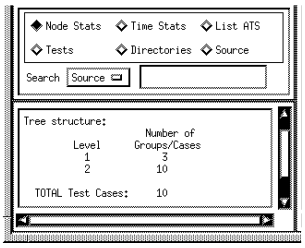

FIGURE 53 Node Stats Button Example

Time Stats button — Lists the individual processing time of each test executed from the current position, as well as the total execution time.

As the **Time Stats** data is derived from the log file, the **Time Stats** display will be empty prior to test execution.

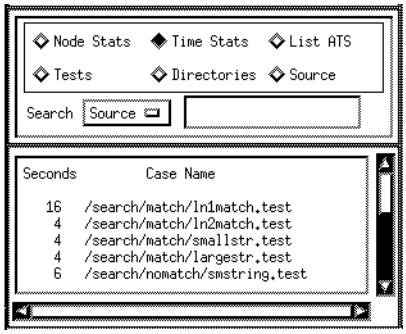

FIGURE 54 Time Stats Button Example

List ATS button — Lists all test names and sections (`source`, `activation`, and `evaluation` clauses) in the ATS file for each test relative to the current position.

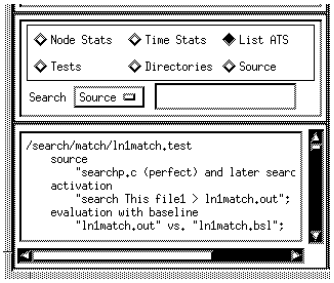

FIGURE 55 List ATS Button Example

Tests button — Lists all test case names relative to the current group's position. If the current position is an individual test case, no data will be displayed.

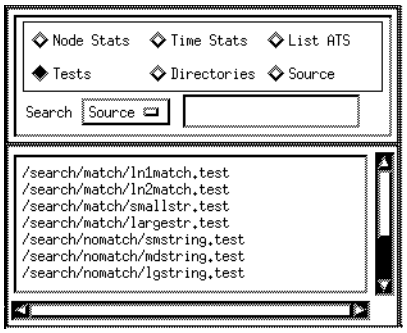

FIGURE 56 Tests Button Example

Directories button Lists the sub-group directories (`match`, `nomatch`, and `error`) relative to the current position. If the current position is a sub-group directory, then only the test cases relative to the sub-group directory are displayed. If the current position is an individual test case, no data will be displayed.

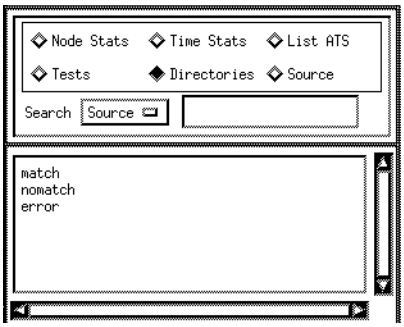

FIGURE 57 Directories Button Example

Search option menu Displays the search options (**Source** / **Case**).

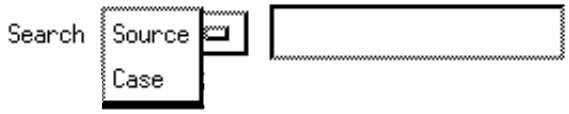

FIGURE 58 Search Option Menu

Search text field Accepts a user entered **Source** or **Case** *string* to be searched for from the current test tree position onward.

Source/Case button

In the **Node Information** scroll window, lists instances of the *string* entered by the user in the **Search** field for the current test tree position.

That is, when the default **Source** button is activated, the entered *string* is searched for in the ATS `source` clause*(s)* from the current test tree position. Conversely, when the **Case** button is activated, the entered *string* is searched for in the ATS for the current test tree position.

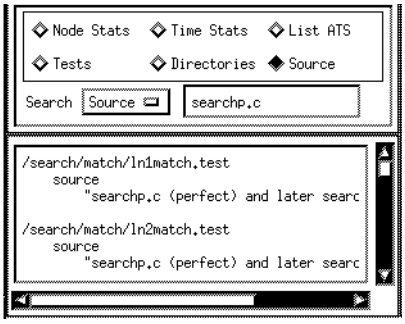

FIGURE 59 Source Button Example

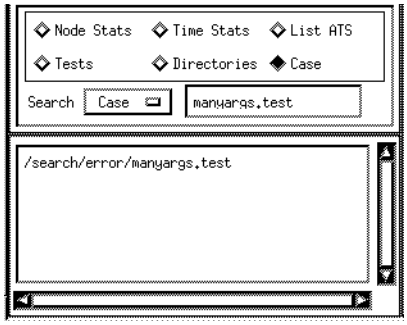

FIGURE 60 Case Button Example

CHAPTER 4

Creating an ATS

This chapter explains the Automated Test Script (ATS) language, how to establish an ATS and how to expedite the process using the **makeats** utility.

4.1 Automated Test Script

After developing test scripts based on a comprehensive test plan, an ATS is created. The ATS file is a structured description file which references a test suite. Using the *SMARTS* Description Language, the hierarchically organized tests can be supplemented with activation commands, comparison arguments and PASS/FAIL evaluation methods. Creation of the ATS can be expedited using the *SMARTS*-provided **makeats** utility. When executed, *SMARTS* performs the pre-stated actions, runs a difference check on the outputs against the baseline, and accumulates a detailed record of the test results.

4.2 ATS Structure

A test suite may be organized within the ATS as either a simple list of test cases (a flat hierarchy), or distinct test groups wherein different categories of tests are isolated (a tree-like hierarchy). Employing either hierarchical approach, thousands of tests can be referenced within the ATS file.

If the test suite is organized as a tree-hierarchy, the result resembles a UNIX directory structure:

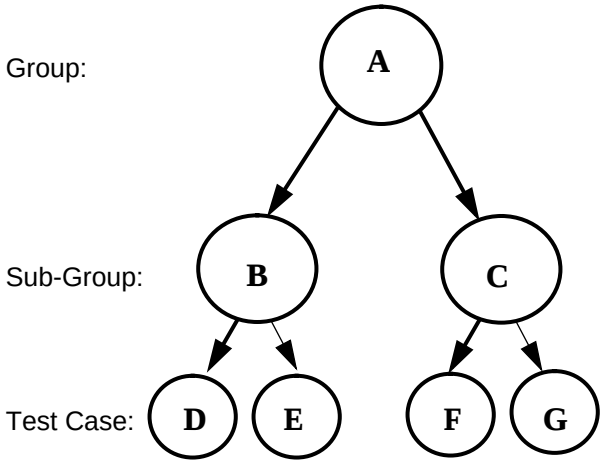

FIGURE 61 Hierarchical Test Tree Structure

As depicted in the diagram above, the ATS test tree is organized by hierarchical levels, groups (or directories) and cases.

The ATS file structure has two advantages. First, either part or all of the test set can be executed according to the user's needs. Second, as the functions of most programs are organized hierarchically, the ATS file structure emulates the functionality of the program to be tested.

4.2.1 ATS Structure Description

Using the *SMARTS* Description Language, the ATS file can reference a test group. The depicted test group might contain nested test group references, which may in turn contain test case references. Conversely, the ATS file may just reference one test case. Regardless of the number of referenced tests, however, there must be a unique root test group in the test suite hierarchy.

The *SMARTS* Description Language which defines each test group or case is referred to as a section. Following the initial test group or case declarative statement, each section is enclosed by braces.

When defining groups and test cases use the following syntax:

`define group` *name*

The declarative statement precedes the section definition for each test group. A test group section can contain nested test groups or cases, where each nested test group or case section definition is enclosed by braces. You must enclose sub-groups or case definitions with braces ({ }). A ({) brace tells SMARTS to include any sub-group and test cases under that group's hierarchy. A (**}**) brace tells *SMARTS* not to include anything else in that group or sub-group.

For example: **define group** *name*

```
{
.
.
.
}
```

`define case` *name*

The declarative statement which precedes the section definition for each test case.

```
define group name
{
.
.
.
}
```

NOTE: Nested groups and cases are not permitted within a case test section.

The ATS also allows for optional comments. Comments are enclosed by the slash and asterisk characters, beginning with **/*** and ending with ***/**.

As previously indicated, tests are organized hierarchically within the ATS file. The *SMARTS* product includes a *smarts.demo* directory containing a sample ATS file named *search.ats*. If you followed the tutorial (See CHAPTER 2 - Quick Start" on page 15.), you may already be familiar with it.

When creating an ATS, it is recommended that you make reference to this example.

The *search.ats* file organizes tests as follows:

- Level 0: A root group called `/search`.
- Level 1: Three sub-groups (directories) named for expected outcome: `/match`, `/nomatch`, and `/error`.
- Level 2: Ten test cases designed to thoroughly exercise the sample *search* program. (Normally, the limit of manageability is between 50-100 test cases per directory.)

After the ATS file is created, *SMARTS* automatically generates a test structure resembling the one shown below for the *smarts.ats* file:

```
Root (/search)
      match
            ln1match.test
            ln2match.test
            smallstr.test
            largestr.test
      nomatch
            smstring.test
            mdstring.test
            lgstring.test
      error
            nofile.test
            fewargs.test
            manyargs.test
```

4.2.2 Test Case Clauses

The most explicit component of the ATS file hierarchy is the test case section. It is within the test case section that activation commands, comparison arguments and PASS/FAIL evaluation methods are specified for the invoked test(s). A test is any executable file, such as an executable or script file.

After a test tree hierarchy is established, use the following clauses to define a test case.

The **`source, activation`** and **`evaluation`** clauses are required. **`environment`** and **`termination`** are optional.

source

The `source` clause contains comments which may specify the intent and origin of the test(s) invoked by the test case. *SMARTS* displays the `source` clause comments when an activated test case is evaluated as FAIL, thereby indicating which files should be inspected.

```
source
"searchp.c (perfect) and later searchi.c
(imperfect)";
```

activation

The `activation` clause indicates any system commands to be performed during the test case execution. The sequence of system commands may be as numerous as needed and enable redirection of input and output. For example, in the test case `ln1match` (see the sample test case section below), *SMARTS* issues the operating system command `search This file1` with output redirected to the file `ln1match.out`.

```
activation
      "search This file1 > ln1match.out";
```

evaluation

The `evaluation` clause specifies the evaluation method by which test output is determined to PASS or FAIL. There are three possible modes of evaluation you can use:

- **`noevaluation`** — All tests are assumed to PASS.
- **`evaluation with user`** — Interactive evaluation by the user for manual tests.
- **`evaluation with baseline`** — Automatic comparison of pairs of files.

- **`evaluation with function`** — Automatic testing using a user-designed function.

environment

When included in a test case section, the `environment` clause defines local environment variables that can be used in the `activation` and `evaluation` clauses. Once set, these environment variables remain active until modified. The `environment` variables are defined in the same manner as shell variables (i.e., *var=val*) and are referred to as `$var` in the `activation` and `evaluation` clauses.

Below is a sample of an `evaluation with baseline` clause:

```
evaluation with baseline
      "ln1match.out" vs. "ln1match.bsl;
```

termination

Commands defined in a `termination` clause are executed when a test is aborted due to an `activation` clause terminating process.

The sample test section definition for *search.ats*' `ln1match` test case is as follows:

```
define case ln1match.test
/* This case tests for a match on line 1,
first character. */
{
      source
            "searchp.c (perfect) and later
searchi.c (imperfect)";
      activation
            "search This file1 >
ln1match.out";
      evaluation with baseline
            "ln1match.out" vs.
"ln1match.bsl";
}
```

In the sample case test section above, the instruction `evaluation with baseline` informs *SMARTS* to compare the indicated files: if the *ln1match.out* output file is identical to the *ln1match.bsl* baseline file, the test passes; otherwise, the test fails.

For further information on writing test section clauses, please refer to the section that details it (See Section 4.3.6 - "Syntax for Test Cases" on page 100.).

4.3 ATS Description Language

The *SMARTS* Description Language is free format, meaning that lines, columns, and blanks are not significant. There are exceptions to this rule; for example, blanks are significant within strings, and keywords and identifiers must be separated with white space. Generally, however, programs may be formatted in any way the user desires.

4.3.1 Character Set

The character set of the *SMARTS* Description Language contains all possible characters, including ASCII alphabetic, numeric, control, null, etc. Unusual characters usually appear in strings or comments. On some systems, *SMARTS* may exclude null characters (ASCII value 0).

Character Set All ASCII and non-ASCII characters.

4.3.2 White Space Characters

The following "white space" characters are used to separate token types in the *SMARTS* Description Language.

blank	*tab*	*newline*
carriage	*return*	*form feed*

See the following description of token types for further details.

Note: When indenting, use only tabs rather than blank / space.

4.3.3 Token Types

Programs in the *SMARTS* Description Language are broken up into the following tokens types:

keyword	*string*	*number*
identifier	*delimiter*	*comment*

NOTE: Keywords and identifiers *must* be separated by a "white space" character. It is not necessary, however, to separate keywords and identifiers from other token types. For example:

```
define case interrupt { ...
```

may be written as

```
define case interrupt{ ...
```

as the left brace ({) is a delimiter. However, the following

```
definecaseinterrupt{ ...
```

is unacceptable because keywords and identifiers must be separated.

4.3.3.1 Keyword

Token keywords are reserved (may not be used as an identifier) and must appear in lower case. Within comments and strings, keywords have no special meaning.

The keywords for the *SMARTS* Description Language are as follows:

```
activation      evaluation      not_exist
baseline        group           source
case            if              termination
define          include         user
empty           noevalution     vs.
function        not_empty       while
environment     with
```

The keyword `vs.` must include a period.

4.3.3.2 Identifiers

Identifiers are used to indicate test group and test case names. Identifiers must begin with an alphabetic character, and can be followed by zero or more characters which are neither delimiters nor white space. Identifiers may be from 1 to 255 characters in length.

Sample Identifiers:

IDENTIFIER	***VALIDITY***
A	*valid*
a2	*valid*
user.interface	*valid*
communication_protocol	*valid*
X!@#$%^&()-_=+''~\/?.><[]:*	*valid*
2	*invalid*
2xyz	*invalid*
characters{};,	*invalid*

4.3.3.3 Strings

Strings are character sequences enclosed in quotes (“ ”) in the *SMARTS* Description Language. To place a quote character within a string, use two quotes in sequence, e.g.:

```
“Plato’s “”Laws”” “
```

Any character in the *SMARTS* character class may appear in a string.

There is no limit imposed on the length of strings. However, if a string is very long, it may be broken up across lines for formatting purposes in the ATS file. This is accomplished by placing a backslash (\) at the very end of the line within the string. See the examples below.

```
“This is a string”
“”
“Four score and \
seven years ago \
our fathers brought \
forth...”

“He said “”Four score and seven years\
ago...”” to them.”
```

4.3.3.4 Delimiters

Delimiters are as follows:

```
()  {}  ;  ,  #
```

4.3.3.5 Numbers

Numbers are sequences of one or more digits. Only integer numbers are permitted; floating point formats are excluded.

4.3.3.6 Comments

Comments are surrounded by the slash and asterisk characters, as in the C programming language. A comment is enclosed by a combination of backslash and asterisk characters: that is, a comment is preceded by `/*` and followed by `*/`. A comment may contain any character in the *SMARTS* character set and there is no limit on the length of comments.

Comments may not appear in strings, and may not be nested (a comment may not appear in a comment).

Sample Comments:

```
/* This is a comment */
/* Comments may appear in any column */

/*
** Comments may also be
** formatted in this or any
** similar manner.
*/
```

4.3.4 Conditional Expressions

The use of conditional expressions in evaluating groups and cases allows you to specify under what environmental conditions a set of groups or cases should be executed. The conditional statements supported by *SMARTS* are **`if`**, **`else`** and **`while`**. These are defined below.

4.3.4.1 if ( ) { } Clauses

Following the **`if`** keyword is the system command to be executed enclosed in parentheses and quoted - e.g., **`if ("cat file"`**). Next is a left brace ({) followed by definitions of sub-groups, conditional expressions and cases. Eventually, a closing right brace (}) ends the **`if`** block. Braces must be balanced within the entire ATS file. If the system command or program exits with (0), meaning the command executed without error, the **`if`** block is executed.

NOTE: Unlike C, after which the structure of *SMARTS* conditionals are patterned, most UNIX commands (such as **`cat`**) exit with (0) if termination is proper, else they return a non-zero error code.

4.3.4.2 else {} Clauses

Following the **`else`** keyword is a left brace ({) followed by definitions of sub-groups, conditional expressions and test cases. Eventually, a closing right brace (}) ends the else block.

Just as in C, if the proceeding un-`elsed` **`if`** returned false (meaning the **`if`** block was not executed), the **`else`** block in the `if...else` section is executed. If no **`else`** block exists, the next group, sub-group, test case, or program that follows will be executed.

The form of `if` and `if ... else` constructions follows that used in C language, as the following syntax explanation shows.

```
if and else Syntax:
if (“syscall”) {
     [define] group <identifier> {
         if (“syscall”) {
             [define] case <identifier> {
               ...
           }
       }
         else {
           if (“syscall”) {
               [define] case <identifier>
{
                 ...
              }
           }
       }
     }
     [define] case <identifier> {
          ...
     }
     ...
}
```

Here a `syscall` is a string that is given to the underlying operating system for execution (a.k.a. system call). The return code for the `syscall` is used to determine the future flow of control with *SMARTS* execution.

4.3.4.3 while ( ) { } Clauses

Following the **while** keyword is the system command to be executed enclosed in parenthesis and quoted - e.g. , w**hile** ("cat file"). Next is a left brace ({) followed by definitions of sub-groups, conditional expressions and cases. Eventually, a closing right brace (}) ends the **while** block. If necessary, you can click on the **Go** window's inverse **Stop** button to exit an infinite **while** loop if you are using the GUI-version of *SMARTS*. So long as the system command or program exits with (0) - meaning the command was executed with no errors (i.e., true) - the **while** block is re-executed. Execution of the block stops when the returned value is non-zero (i.e. false).

```
while Syntax:
while ("syscall") {

     [define] group <identifier> {

          [define] case <identifier> {
              ...
          }
     }

     while ("syscall") {

               [define] case <identifier> {
                ...
          }
     }
     ...
}
```

Just as test groups may be nested, conditional expressions and iteration clauses may also be nested. The depth of nesting is limited by memory capacity only. However, an automated test script that involves more than 10 levels of nesting of **if** and **while** statements is likely to be too complex to be managed reliably.

4.3.5 #include Statements

Different ATS files may be included together using the **`#include`** command. It helps to organize the tests into separate groups, which may, if required, be included into a single main ATS file. Often these groups would be logically distinct, and would be physically organized into separate directories. The only physical limit to the size of an ATS file is the amount of RAM available. This is because *SMARTS* creates an in-memory image of the ATS file.

The syntax for the keyword `include` is similar to that for `#include` in the C language. Any text file can be included in any position, provided it obeys the normal *SMARTS* syntax rules:

```
#include <test group/case file>
```

4.3.6 Syntax for Test Cases

4.3.6.1 The Source Clause

The `source` clause contains comments generally used to briefly describe the premise of each case. The multiple clauses are specified as strings separated by commas; the last string must be followed by a semicolon. There is no limit on the number of `source` clause an ATS can contain.

Source clauses are displayed as header / flagging lines when test case activation is evaluated as a FAIL. This helps mark which files need to be inspected.

Syntax for Source Clauses:

```
[define] case identifier {
        source
            string,
              ...
            string;
        ...
}
```

4.3.6.2 The Environment Clause

Environment variables can be defined with the keyword **`environment`**. It is a sequence of variables and their values. ATS environment variables are defined in the same way shell variables are defined, i.e. `variable=value`, where `variable` is the environment variable name and `value` is the corresponding value in that test case.

When defined in the `environment` clauses, environment variables can be used as variables in the `activation` and `evaluation` clauses. Once set, these environment variables stay in place until they are changed.

They are referred to as **`$variable`** in the **`activation`** and **`evaluation`** clauses. The `environment` clauses are specified as strings of the form `var=val` separated by commas. The last string must be followed by a semicolon.

Syntax for Environment Clauses:

```
[define] case identifier {
      ...
      environment
            string,
      ...
            "var=val";
      ...
}
```

4.3.6.3 The Activation Clause

Activation is triggered by the keyword **`activation`** and is the sequence of system commands which perform the test case. The `activation` clauses are specified as strings separated by commas. The last string must be followed by a semicolon. There is no limit on the number of activation commands.

Syntax for Test Activation:

```
[define] case identifier {
        ...
        activation
            string,
            string,
             ...
            string;
        ...
}
```

4.3.6.4 The Evaluation Clause

Evaluation is the method by which the test case is determined to PASS/FAIL. Four methods are available: no evaluation, with user evaluation, automatic evaluation with baselines and evaluation with defined functions.

In some testing situations, it is desirable to perform actions that serve merely to set up or manage the testing environment. Bypassing evaluation is appropriate in these circumstances. Using the keyword `noevaluation` will let you activate commands without performing or recording evaluation.

A manually evaluated test is specified with the keywords `evaluation with user`. User evaluation is needed for tests which require you to inspect the test output manually to determine the outcome. After activation of a manual test, the system will ask you for the test outcome; you should either click on **PASS** for PASS or **FAIL** for FAIL (for the GUI-version) or enter either **p** for PASS or **f** for FAIL for the ASCII menu version.

Syntax for No Evaluation

```
[define] case identifier {
        ...
        noevaluation;
}
```

Syntax for User Evaluation:

```
[define] case identifier {
        ...
        evaluation with user;
        ...
}
```

An automatic test can be specified with the keywords `evaluation with baseline`, followed by a sequence of string pairs separated by commas. The last evaluation must be followed by a semicolon.

The sequence of string pairs denote pairs of files that are to be compared. Between string pairs the literal keyword is `vs.` (The period is important.). The contents of the strings is interpreted as file names.

The `empty`, `not empty`, and `not exist` features are used to assess the status of a file produced by a test case. These three parameters effect evaluation in the following manner:

file `empty`	The test will pass if *file* does not exist or is empty.
file `not exist`	The test will pass if *file* does not exist.
file `not empty`	The test will pass if *file* exists and is not empty.

Syntax for Baseline Evaluation:

```
[define] case identifier {
        ...
        evaluation with baseline
            file vs. file,
            file vs. file,
            ...
            file not_exist,
            ...
            file not_empty,
            ...
            file empty,
}
```

Note: The default comparison utility is **diff**. You also set the **exdiff** command to difference ASCII and binary files or **Xexdiff** to difference bitmap images. The chosen command can be set in the resource configuration file (See Section 5.4 - "Resource Configuration File Processing" on page 125.), the **Set Difference Utility** option in the GUI's **Main** window(See Section 3.2.1.2 - "Option Menu" on page 56.), or with **newdiff** option in the ASCII menu's **OPTIONS** menu.

An automatic test incorporating a user-defined function can be specified with the keywords `evaluation with function` followed by one or more strings which may be any system commands of user programs.

Syntax for Evaluation With Function

```
[define] case identifier {
        ...
        evaluation with function
            "command string",
            ...
            "command string";
}
```

4.3.6.5 The Termination Clause

SMARTS also provides for the execution of concurrent processes in order to test the timing of specific test cases and terminate them if necessary. Terminating processes are specified by preceding the system command with "%" within the string — e.g. `%sleep 10`. An activation command may be any valid system command.

Example of Test Activation with Termination:

```
[define] case <identifier> {
        source
            "example of termination command.";
        activation
            "%sleep 10",
            "test1",
            "%sleep 100",
            "%sleep 20",
            "test2";
        evaluation with baseline
            "ln1match.out" vs. "ln1match.bsl";
        termination
            "cat error.file";
        }
```

In the previous example, `sleep 10` would be executed concurrently with `test1`.

Note that the terminating process `sleep 10` should be listed before the corresponding `test1` command in the `activation` clause.

If `sleep 10` finishes before `test1` the activation phase will be terminated, the test case will FAIL, the evaluation phase will be skipped and the termination phase will be executed. If, however, `test1` finishes first, the terminated process will be aborted and processing will continue with the next activation line, which in this case contains `%sleep 100`.

Because terminating commands are executed concurrently, `sleep 100`, `sleep 20` and `test2` will be executed concurrently. As with `test1`, if `sleep 20` or `sleep 100` finish before `test2`, the `activation` clause will be terminated.

Note: Only re-directions are allowed in system commands.

4.4 BNF Description of ATS Language

The BNF description of the *SMARTS* ATS language is shown below as a reference.

```
sourceFile:==    def.s
def.s   :==    def.s def
        |      null

def     :==    DEFINE GROUP ID '{' def.s '}'
        |      DEFINE CASE ID '{' item.s '}'
        |      CASE ID '{' item.s '}'
        |      IF '(' string.s ')' '{' def.s '}'
        |      WHILE '(' string.s ')' '{' def.s '}'
        |      ELSE  '{' def.s '}'
        |      DEFINE CASE ID '{' item.s '}'
        |      CASE ID '{' item.s '}'

item.s                :==    sourceItem environment-
Item  activateItem
       evaluateItem terminateItem
        |      null

sourceItem:==    SOURCE string.s ';'
        |      null

environmentItem:==    ENVIRONMENT string.s ';'
        |      null

activateItem:==    ACTIVATION string.s ';'
        |      null

string.s:==    STRING ',' string.s
        |      STRING

evaluateItem:==    EVALUATION WITH USER ';'
        |      EVALUATION WITH BASELINE compare.s ';'
        |      EVALUATION WITH FUNCTION string.s ';'
        |      NOEVALUATION ';'
        |      null

terminateItem:==    TERMINATION string.s ';'
        |      null

compare.s:==    compare ',' compare.s
        |      compare
compare                       :==    STRING VS STRING

null    :==
```

4.5 The makeats Utility Overview

To produce the automated test suite, considerable effort is initially required to create ATS files. The **makeats** utility substantially reduces this effort by producing ATS files from minimal information. It is our experience that **makeats** can be used to create most ATS files from very few details, and much can be assumed. In the lifetime of an ATS file, detailed changes can easily be edited directly into the ATS file.

4.5.1 Invocation and Use of makeats

makeats is invoked by the command **makeats**. Optional run-time parameters may be specified on the command line with a dash, followed by an option code letter. If no parameters are specified, default values are assumed. Invalid parameters are ignored.

Below is the required syntax you should use for **makeats**:

makeats *infile outfile* [**options**]

infile outfile — These are the names of the input file and the output file. Pipeline rules are assumed. To read from standard input, just replace *infile* with the standard UNIX "-". For standard output, replace *outfile* with "-". For example, the command:

```
makeats - -
```

will read from standard input (the keyboard) and write to standard output (the screen).

Following is a description of the command line options.

```
-F
-help
-K basename number
-s N
-t
```

-F — Fast Generation Switch. The input file interprets quickly into a set of **#include** structures that you can later fill in. Please see another section for an example (See Section 4.5.1.3 - "Fast Operation Example" on page 112.).

-help	Help Switch. Generates a full description of the correct calling parameters for **makeats**. This information is also generated whenever an incorrect calling sequence is used.
-K *basename number*	Special Keysave File Standard Script Generator. In this case a standard script is generated that plays back the file *basename.**ksv*** and compares *number* files named *basename.**bxx*** against *basename.**rxx***, where *basename.ksv* is a keysave file from *CAPBAK/X*, *basename.bxx* is a captured baseline file and *basename.rxx* is the corresponding response file. Please see the section that details an example (See Section 4.5.1.4 - "Keysave File Operation Example" on page 113.). No input file is needed for this option.
-S *N*	Space Insertion Switch. Inserts *N* line spaces between major ATS elements.The default value is 1.
-t	Test Output Switch. Bypasses normal mode ATS production and displays the hierarchical *SMARTS* test structure for the input file that would be produced if the **-t** switch were not present.

4.5.1.1 makeats Regular Input Description

The **makeats** input file structure is based on the UNIX **make** utility. It generates a hierarchical *SMARTS* test structure consisting of groups, sub-groups, and test cases. The number of sub-groups and test cases is unlimited.

Each input file line should take the form:

group_name`:` *subgroup_name test_cases* `...`

Each group definition requires a separate line; to continue a line to the next, end the first line with a backslash (\). A sub-group must appear on the right side of a group definition; thereafter the sub-group can be defined anywhere in the input file. Legal punctuation includes the colon (`:`), equal sign (`=`),and quotation mark (`"`) .

File names for test input and output, and for baseline comparison, are generated from the test case names. To modify these file names or the `source` clause description of a group or test case, or to include a file in the ATS, use one of the following options:

$source = *string*

Places **source** *string* in the test group on the input file line following the command. To modify the source clause, place another command in the proper input file location. To turn source off, insert a command with an empty string.

$evaluation = *method*

Selects the test evaluation method for evaluation clause. Placement works like a **source** command. *method* is one of the following three values:

automatic = `evaluate with baseline`
manual = `evaluation with user`
noevaluation = `noevaluation`

The default *method* is **automatic**.

$activation = *string*

Changes the test activation command for the activation clause to the specified *string*. Placement works like a **source** command. Default activation is the test case name. To turn activation off, insert a command with an empty string.

$command = *string*

Changes only the activation command in the `activation` clause. The default activation command is the test name (i.e. `"test_name"` **<test_name.in>** `test_name.out`). An empty *string* returns default.

$include = *file_name*

Inserts an `#include` statement before the test group on the input file line following the command.

Note: Any non-ambiguous substring of the above options or evaluation methods will be accepted by **makeats**. For example:

$eval = *auto*
$s = *source string*

4.5.1.2 Regular Operation Example

Following is a sample input file and sample resulting output file for **makeats**.

Sample Input File:

```
$source = "This is part 1"
$include = "file1"
one: two three four five
three: three1 three2 three3 \
three4

$source = "This is part 2"
$eval = "man"
$activ = "testact"

three2: three2a three2b three2c
two: top bottom
four: intro begin mid end conclusion
three2b: center
```

makeats then produces an ATS which is ready for input into *SMARTS*:

```
#include "file1"
define group one {
    define group two {
        define case top {
            source
                "This is part2";
            activation
                "testact";
            evaluation with user;
        }
        define case bottom {
            source
                "This is part2";
            activation
                "testact";
            evaluation with user;
        }
    }
    define group three {
        define case three1 {
            source
                "This is part 1";
            activation
                "three1 < three1.in > three1.out";
            evaluation with baseline
                "three1.out" vs. "three1.bsl";
        }
        define group three2 {
            define case three2a {
                source
                    "This is part2";
                activation
                    "testact";
                evaluation with user;
            }
            define group three2b {
                define case center {
                    source
                        "This is part2";
                    activation
                        "testact";
                    evaluation with user;
                }           }   define case end {
            source
                "This is part2";
            activation
                "testact";
            evaluation with user;
        }
        define case conclusion {
            source
                "This is part2";
            activation
                "testact";
            evaluation with user;
        }
    }
    define case five {
        source
            "This is part 1";
        activation
            "five < five.in > five.out";
        evaluation with baseline
            "five.out" vs. "five.bsl";
    }
```

4.5.1.3 Fast Operation Example

The alternative ‘‘fast’’ version of **makeats** works only using a tab-delimited file and produces only a ‘‘top’’ ATS file. The inputs and output are shown below.

Below is a sample input file and sample resulting output file for **makeats**.

Sample Input File:

```
outline
        some
        with
        contents
                and
                others
with
        structure
        descriptors
```

makeats then produces the following ATS file:

```
group outline {
        case some {
                #include "some.ats"

        }
        case with {
                #include "with.ats"
        }
        group contents {
                case and {
                        #include "and.ats"
                }
                case others {
                        #include "others.ats"
                }
        }
}
group with {
        case structure {
                #include "structure.ats"
        }
        case descriptors {
                #include "descriptors.ats"
        }
}
```

4.5.1.4 Keysave File Operation Example

The "keysave file" version of **makeats** uses the **-K** switch. It allows you to produce a basic ATS file that consists of a playback of a keysave file and comparison of zero or more baseline files with corresponding response files. It requires no input file.

The next two examples demonstrate the "keysave file" feature.

Example 1:

Command: **`makeats -K NONE 5`**

Output:

```
define group EXAMPLES {
   define case FIVE {
      source
         "Play a keysave file that saves 5 images.";
      activation
         "Xplabak -k FIVE.ksv";
      evaluation with baseline
         "FIVE.r01" vs. "FIVE.b01",
         "FIVE.r02" vs. "FIVE.b02",
         "FIVE.r03" vs. "FIVE.b03",
         "FIVE.r04" vs. "FIVE.b04",
         "FIVE.r05" vs. "FIVE.b05";
   }
}
```

Example 2:

Command: **`makeats -K NONE 0`**

Output:

```
define group EXAMPLES {
   define case NONE {
      source
         "Play a keysave file that saves 1 images.";
      activation
         "Xplabak -k NONE.ksv";
      noevaluation;
   }
}
```

CHAPTER 5

Invoking SMARTS

This chapter explains how to invoke both the GUI and ASCII versions of *SMARTS*. It also explains how to establish a resource configuration file.

5.1 Invoking SMARTS' GUI

To invoke the graphical user interface (GUI) version of *SMARTS* from the working directory, enter the following command:

```
Xsmarts
```

The **Main** window pops up:

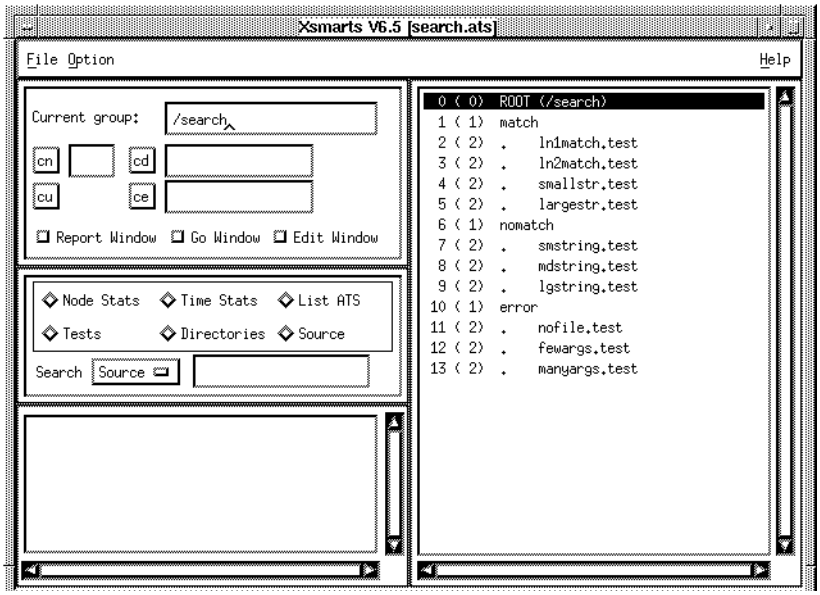

FIGURE 62 Invoking the GUI's Main Window

If the *STW* product bundle is available on the current system, *SMARTS* can be invoked. Here's how:

1. In your working directory, type **`stw`**.
2. The **TestWorks** window pops up.
3. Select the **Regression** button.
4. The **STW/Regression** window pops up.
5. Select **SMARTS**.
6. The SMARTS **Main** window pops up.

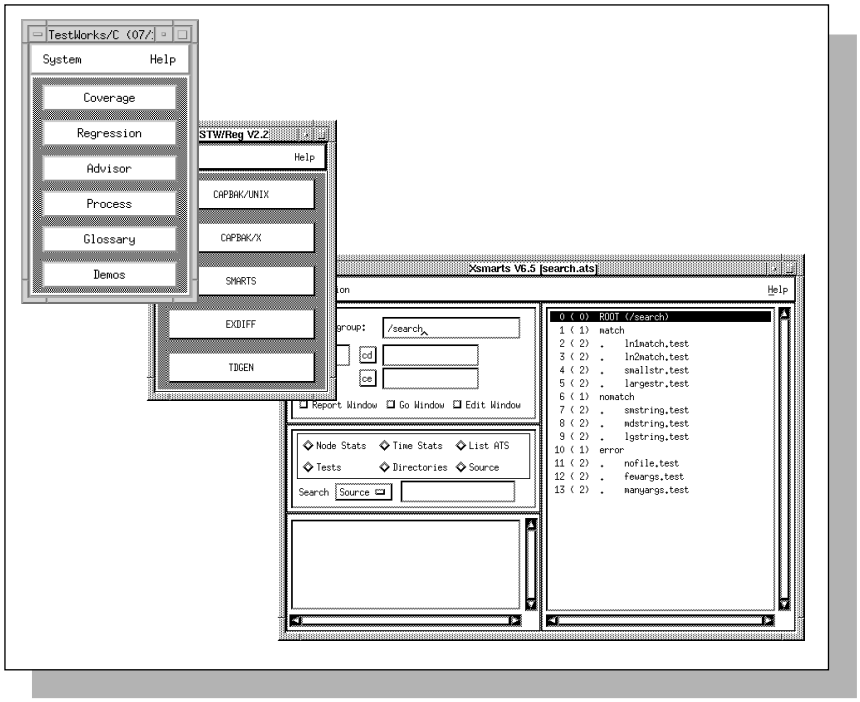

FIGURE 63 Invoking SMARTS with STW

5.2 Invoking SMARTS' ASCII Menu Interface

To invoke the graphical user interface (GUI) version of *SMARTS* from the working directory, enter the following command:

```
smarts
```

smarts will provide header information and initialize the **MAIN** menu as shown below:

```
SMARTS Release 6.1 for SYSV386 (03/17/93) (c) 1991
Software Research, Inc.
Using "smarts.rc" for configuration information.
Using "search.ats" for control script.

Processing 137 line of input from 1 file.
SMARTS:MAIN:
```

FIGURE 64 Invoking the ASCII Menu's MAIN Menu

5.3 Command Line Invocation

The *SMARTS* system is activated by either of the following single commands:

Xsmarts Invokes the Graphical User Interface (GUI) version for the X Window System version.

smarts Invokes the ASCII menu version.

Either command is a single executable file which may reside in any file system directory, as long as it is included on the execution search path. However, in the event that the *SMARTS* description file (ATS file) contains `activation` and `evaluation` clauses which refer to the current system directory or subdirectory, it may be necessary to invoke *SMARTS* from a particular directory in the file system.

A configuration file which includes run-time parameters is also invoked when either the `Xsmarts` or `smarts` command is executed. In *SMARTS*, this file is referred to as the resource configuration (RC) file and *smarts.rc* is the default RC file. If *SMARTS* is invoked without the RC file, then the embedded default values for the *SMARTS* application are used.

Run-time parameters can be set or modified with any of the following methods:

1. Edited in the RC file prior to invoking *SMARTS*, as explained in another section (See Section 5.4 - “Resource Configuration File Processing” on page 125.).
2. Indicated during the command line invocation of *SMARTS* as described in another section (See Section 5.3.2 - “smarts’ Runtime Options” on page 122.).
3. Specified from the GUI, as described in another chapter (See CHAPTER 3 - Understanding the GUI” on page 43.).
4. Specified from the ASCII menus, as described in another chapter (See CHAPTER 8 - Using the ASCII Menu Interface” on page 139.).

5.3.1 Xsmarts’ Runtime Options

SMARTS run-time options may be specified on the command line with the following syntax:

`Xsmarts -option` *parameter*

If no parameters are specified, default values are assumed. Invalid parameters are ignored. The GUI-version of *SMARTS* can be invoked using the following run-time options.

```
Xsmarts
        -f ats_file
        -G go_mode
        -l log_file
        -N line
        -P path_name
        -R report_type
        -rpt report_file
        -S
        -T
        -X
```

-f ats_file ATS File Name Specification Switch. The *ats_file* is used as the ATS file instead of the file specified in the current RC file.

-G go_mode Batch Execution Mode Specification Switch. The *go_mode* is used to specify the type of test case execution. *go_mode* can any one of the following:

- **no_option | null** executes until there are no more tests. This is the default.
- **auto** runs the ATS file unattended if an ATS contains evaluation with user clauses. This option forces all such tests to pass without user intervention.
- **fail** runs onlys those test cases that failed during the previous execution.
- **on_fail** <*group*> runs tests until a test fails. Should a test fail, *SMARTS* then executes the test <*group*> specified.
- **limit** <*N*> executes the current goup within a limit of *N* seconds per test. If a test is longer than *N* seconds, the test fails.
- **new** executes only those tests that have not been previously recorded in the current log file.
- **pass** executes only those tests that passed during the previous execution.
- **repeat** <*N*> executes the ATS file *N* times.
- **till_fail** executes tests until the first test fails.

Note: The *go_mode* can be modified with any of the **Go** option menu's options in the **Go** window. Please see another section for further information (See Section 3.2.4.2 - "Go Window" on page 77.).

-l *log_file*	Log File Name Specification Switch. The *log_file* is used as the log file instead of the default file *LOGFILE*. This option allows unique log files for different groups of tests to be maintained in the same directory.
-N *line*	Test Line Number Specification Switch. *line* is the test tree hierarchy line number in the **ATS Test Tree** display of the **Main** window. This option changes to the current position of the test tree from the topmost group or test cases to the line number specified.
-P *path_name*	Test Path Execution Specification Switch. *path* is the path of the group or test case to selected as the current position in the **ATS Test Tree** display of the **Main** window. This option changes to the current position of the test tree from the topmost group or test cases to the group or test case specified.

-N and **-P** are mutually exclusive. When used with the **-G** option, **Xsmarts** will run the tests from the current position without popping up the GUI. In other words, tests are run in the background without user interaction on the GUI. At the end of the test, the message box below pops up:

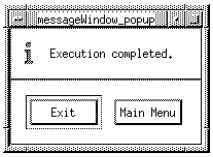

FIGURE 65 Ending Background Test Execution

You can then exit **Xsmarts** or invoke the **Main** window. When **-N** and **-P** are used without the **-G** option, **Xsmarts** will bring up the **Main** window and select the test specified from the command line.

-R *report_type* — Report Type Specification Switch. *report_type* can either be

- status
- history
- regression
- certification

to represent the four different kinds of reports available. This option generates reports from the current log file. The report is appended to the report file (if one exists) or to standard output if no report file is specified. A report file can be specified with the **-rpt** command line option (see below), in the configuration file (See Section 5.4 - "Resource Configuration File Processing" on page 125.) or with the GUI's **Set Report File** option (See Section 3.2.1.1 - "File Menu" on page 52.).

-rpt *report_file* — Report File Specification Switch. *report_file* is the file to which report information is appended. *report_file* can also be specified in the configuration file (See Section 5.4 - "Resource Configuration File Processing" on page 125.) or with the GUI's **Set Report File** option (See Section 3.2.1.1 - "File Menu" on page 52.).

-S Silent Batch Mode Operation Switch. Stops the end-of-execution message box from popping up when the **-N** or the **-P** switch is used with the **-G** switch. See the previous page for an explanation on this message box.

-T Test Tree Output Switch. Produces the current ATS's test tree hierarchy in standard output.

-X Purge Log File Switch. This switch purges the current log file.

5.3.2 smarts' Runtime Options

SMARTS run-time options may be specified on the command line with the following syntax:

smarts -option *parameter*

If no parameters are specified, default values are assumed. Invalid parameters are ignored. The ASCII version of *SMARTS* can be invoked using the following run-time options.

```
smarts
        -c
        -e
        -f ats_file
        -i
        -l log_file
        -r rc_file
        -rpt report_file
        -s status_file
```

-c Create/Copy Baselines Switch. Whenever a test fails which contains an **evaluation** with **baseline** clause, baseline files are **c**reated by **c**opying the response file. That is, for each response-to-baseline comparison indicated in the failed test's e**valuation** with **baseline** clause, the response file is copied to the corresponding baseline file.

Copying the response file to the baseline file guarantees that the next execution of the test will PASS, as the files will now be identical.

Within the syntax for the **evaluation** with **baseline** clause the response file is indicated first:

evaluation with **baseline**

response_file **vs.** *baseline_file*

This switch is only applicable for tests which include the `evaluation with baseline` clause.

Baseline files can also be created with the ASCII menu's create option (See Section 8.5 - "OPTIONS Menu" on page 147.).

-e Echo Enablement Switch. Input commands are **e**choed to standard output. This is useful for batch file execution when command input is from a file redirected with ">".

-f *ats_file* ATS File Name Specification Switch. The *ats_file* is used as the description file instead of the file specified in the current RC file.

-i Ignore Resource Configuration File Switch. The resource configuration file *smarts.rc* or the RC file specified with the **-r** switch. is **i**gnored. When *SMARTS* is invoked without an RC file, the embedded default values for the *SMARTS* application are used and an ATS file must be manually specified.

-l *log_file* Log File Name Specification Switch. The *log_file* is used as the log file instead of the default file *LOGFILE*. This option allows unique log files for different groups of tests to be maintained in the same directory. The log file can also be specified in the current configuration file (See Section 5.4 - "Resource Configuration File Processing" on page 125.) or with the ASCII menu's *logfile* option (See Section 8.5 - "OPTIONS Menu" on page 147.).

-r *rc_file* Resource Configuration File Specification Switch. Parameters are read from *rc_file*. When this switch is not present, the default RC file used is *smarts.rc*.

-rpt *report_file* Report File Specification Switch. *report_file* is the file to which report information is appended. *report_file* can also be specified in the configuration file (See Section 5.4 - "Resource Configuration File Processing" on page 125.) or with the

ASCII menu's **`rptfile`** option (See Section 8.6 - "REPORT Menu" on page 149.).

`-S` *status_file* — Report File Specification Switch. *status_file* is the file where test execution statements are stored. *status_file* can also be specified in the configuration file (See Section 5.4 - "Resource Configuration File Processing" on page 125.).

A batch or script file may also be used to run **smarts** using standard input and output redirection. This works because smarts is an ASCII menu-based system.

A batch file can be useful for running controlled test suites from several different system directories each with separate log files and test scripts. A sample invocation for such use could be:

```
smarts -e -f test.ats < test.in > test.out
```

The command required to run all tests can be entered into a file using a text editor and should mimic the normal iteration use. Such a file could look like:

```
browse
go
report
status
exit
exit
exit
exit
```

The **`-e`** option will echo these commands to standard output.

5.4 Resource Configuration File Processing

Prior to command line invocation, run-time parameters for **Xsmarts** or **smarts** may be set from a resource configuration (RC) file. The default RC file is *smarts.rc*.

A RC file consists of a series of parameters which are set one per line and listed in any order. These parameters may be altered and saved from either the **File** and **Option** menus (in the GUI version) or the **OPTIONS** menu (in the ASCII version). A new default RC file can also be specified from either of the previously indicated menus. If no parameters are specified, then the embedded defaults values for the *SMARTS* application are used.

When a *file* is indicated in the upcoming list of available parameters, the *file* is presumed to be located in the directory from which *SMARTS* is invoked. If the *file* is not located in the working directory, the entire path name should be specified. A sample RC file is listed in another section (See Section 5.4.1 - "Sample Resource Configuration File" on page 127.).

The following parameters can be set in the RC file:

`atsfile =` *"file"* — Reads the ATS *file*. This parameter can be overridden by the command line **`-f`** run-time option. The default is *smarts.ats* if the **`-f`** option is not used.

`diff =` *"command"* — Uses *command* as the comparison utility. The default comparison utility is **`diff`**.

`diffsave` — Saves the difference output for each test case that fails. The default parameter is **`nodiffsave`**.

`edit =` *"command"* — Use *command* as the editor when editing the ATS file from within the interactive menus. The default editor is **`xterm -e vi`** (in the GUI interface) and **`vi`** (in the ASCII interface).

`filesave` — After test comparisons are made, maintain all response files indicated within a test case's evaluation with baseline clause. The default parameter is **`filesave`**. See the **`nofilesave`** parameter.

`help =` *"helpfile"* — Use the *file* when accessing the help file. The default help file locations are */usr/lib/SR/smarts.hlp*. This parameter only applies to the ASCII menu version of *SMARTS*.

logfile = *"file"*	Use **file** as the log file. The default is *LOGFILE*. This parameter can be overridden by the command line **-l** command line option.
nodiffsave	Does not save the difference output when a test case fails. This is the default parameter.
nofilesave	After test comparisons are made, delete all response files indicated within a test case's **evaluation with baseline** clause. This parameter is used primarily to save disk space. The default parameter is **filesave**.
noshowinclude	Suppress showing the names of included files during start-up. The default parameter is **showinclude**.
noshowsource	Do not display source lines during test execution. This is the default parameter.
reportfile = *"file"*	Saves reports to **file**. This can be overridden by the **-S** command line option for the ASCII menu version.
statusfile = *"file"*	Saves execution status messages to **file**.
showinclude	Displays the included files within the test tree hierarchy. This is the default parameter.
showsource	Displays the source clause from the ATS file for each test case during test execution. The default parameter is **noshowsource**.
width = *n*	Sets the width for the **Status** and **History** reports to *n* characters. The default width is 30 characters.

5.4.1 Sample Resource Configuration File

Below is the default RC file *smarts.rc*:

```
atsfile = "search.ats"
help = "/usr/lib/SR/smarts.hlp"
logfile = "LOGFILE"
reportfile = ""
statusfile = ""
edit = "xterm -e vi"
diff = "diff"
nofilesave
nodiffsave
noshowsource
showinclude
width = 30
```

CHAPTER 6

Executing Tests

This chapter explains how to select the current position, which test execution is based upon, and how to execute tests from that point in the Xsmarts application.

6.1 Selecting the Current Position

Prior to executing tests, you must select the current position. The selected test(s) to be executed are all the test cases that fall ***under*** the current position. If the current position is an individual test case, then only the selected test case is executed. The current group is identified in the **Current group** text field.

To select the current position:

1. Use the mouse to click on the current position in the **ATS Test Tree** display of the **Main** window.
2. Use the options from the **Current group** section of the **Main** window. (See Section 3.2.3 - “Current Group” on page 67.).

6.2 Invoking the Go Window

At this point, the hard work is finished. All that remains is to execute the tests and view the *SMARTS*-generated reports. This chapter describes performing the test execution. Viewing reports is examined in the following chapter.

Tests are executed from the **Go** window. To invoke the **Go** window, perform the following:

1. From the **Current group** section of the **Main** window, click on the **Go Window** button.
2. The **Go** window indicated below pops up.

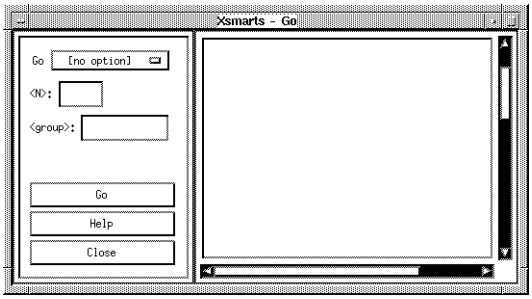

FIGURE 66 Invoking the Go Window

6.3 Selecting Execution Options

A log file name must be set. The log file contains all the test execution information. It can be set in the following ways:

- Using the resource configuration file. Please see another section for information on creating and editing a resource configuration file (See Section 5.4 - "Resource Configuration File Processing" on page 125.).
- Using the command line's **`-l log_file`** option. Please refer to another section for further information (See Section 5.3.1 - "Xsmarts' Runtime Options" on page 118.).
- Using the **Main** window's **File** menu to select the **Set Log File** option. Please see another section for further information (See Section 3.2.1.1 - "File Menu" on page 52.).
- Using the default *LOGFILE*. Set no option and all test execution PASS/FAIL information will be written to this file.

The following kinds of options can be set for prior to execution:

- A file where all the execution messages are stored. This file can be set with the **File** menu's **Set Status File** option in the **Main** window or in the resource configuration file. Please refer to other sections for further information on the **Set Status File** option (See Section 3.2.1.1 - "File Menu" on page 52.), and the resource configuration file (See Section 5.4 - "Resource Configuration File Processing" on page 125.).
- Various test execution runtime options can be set with the **Option** menu's **Toggles**' options and the **Set Difference Utility** option and in the resource configuration file. Please refer to other sections for further information on these GUI options (See Section 3.2.1.2 - "Option Menu" on page 56.), the resource configuration file (See Section 5.4 - "Resource Configuration File Processing" on page 125.).
- The type of execution to be performed can be set with **Go** window's **Go** menu and the **<N>** and **<group>** text field options, or the command line option's **-G** option. Please refer to other sections for further information on the **-G** option (See Section 5.3.1 - "Xsmarts' Runtime Options" on page 118.), and the **Go** menu's options (See Section 3.2.4.2 - "Go Window" on page 77.).

6.4 Executing the Test Cases

After determining the type of test execution processing, execute the selected ATS test(s) by performing the following:

1. Click on the **Go** activation button. The **Go** activation button toggles to **Stop** and the activation button's background / foreground colors are reversed.

   As each test executes, the current test outcome is compared to the baseline files specified within the test. Clicking on the inverse **Stop** button will stop the execution following completion of the currently executing test case.

   The entire process, as well as test outcomes, are scrolled in the display area of the **Go** window. A test case which passes is indicated by the statement:

   `TEST PASSES` (*group identifier)*

   A failed test case is indicated by the statement:

   `TEST FAILS` (*group identifier)*

   As the *group identifier* is listed in the display area of the **Go** window, the location of a failed test is easily determined.

2. When the test session has completed executing, the end-of-test-execution message box pops up:

3. Click on **OK**.

Note: This message box can be suppressed by selecting the **Suppress EOT Message** option from the **Main** window's **Toggles** cascading menu.

6.5 Exiting the Go Window

The **Go** window need not be exited following each test session. The window is exited throughout the user manual for the purpose of practice.

Exit the **Go** window by clicking on the window's **Close** button.

CHAPTER 7

Viewing Execution Tests

This chapter explains how to select and view execution reports relative to the current position in the Xsmarts application.

7.1 Selecting the Current Position

Prior to executing tests, you must select the current position. The selected reports display the execution results from the current position. If the current position is an individual test case, then only the selected test case is executed. The current group is identified in the **Current group** text field.

To select the current position:

1. Use the mouse to click on the current position in the **ATS Test Tree** display of the **Main** window.
2. Use the options from the **Current group** section of the **Main** window (See Section 3.2.3 - “Current Group” on page 67.).

7.2 Invoking the Report Window

When the selected test(s) are executed, *SMARTS* automatically stores all the test information in the log file established or the default *LOGFILE* and generates various reports based on the log file data.

Reports are displayed in the Report window. To invoke the **Report** window, perform the following:

1. From the **Current group** section of the **Main** window, click on the **Report Window** button.
2. The **Report** window indicated in the next figure pops up.

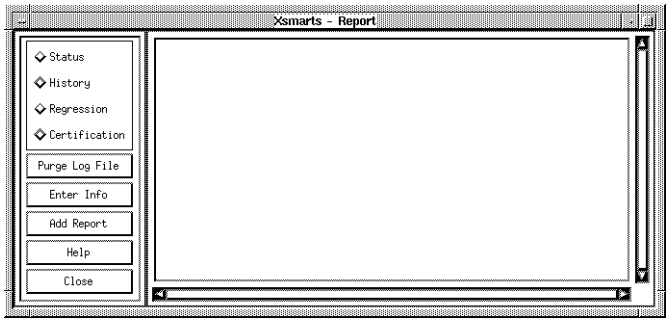

FIGURE 67 Invoking the Report Window

7.3 Selecting Report Options

Prior to selecting reports, the following options can be set:

- A file where reports can be saved after viewing. The file can be set in the resource configuration file, with the `-rpt` command line option, or with the **File** menu's **Set Report File** option in the **Main** window. Please see other sections for setting the file in the resource configuration file (See Section 5.4 - "Resource Configuration File Processing" on page 125.), and the `-rpt` option (See Section 5.3.1 - "Xsmarts' Runtime Options" on page 118.), or setting the file with the **Set Report File** option (See Section 3.2.1.1 - "File Menu" on page 52.).
- The report width of the **Status** and **History** reports can be set with the **Option** menu's **Set Test Path Width on Report** option or set in the resource configuration file. For further information see the sections (See Section 3.2.1.2 - "Option Menu" on page 56.), (See Section 5.4 - "Resource Configuration File Processing" on page 125.).

7.4 Selecting Reports

When selecting a report, keep in mind that the information will only reflect the most recent log file data for each test case. That is, if an entire test suite has not been executed in a while and an individual test case is selected for execution, the generated reports will indicate the recently-executed data for the individual test case. The data provided for the remaining cases in the test suite, however, will reflect information from previous test executions.

The **Report** Window offers the following reports:

Status report	Reflects the test outcomes of the *most recent test execution*, respective to the selected current position. Therefore, more information will be generated if the test was executed from a root or a group than if only a single test case is executed.
History report	Produces a summary report of *all* test outcomes maintained in the log file, providing an overview of test regression throughout the testing process.
Regression report	Includes only the *most recently executed test cases* whose outcomes have changed from the previous executions. The **Regression** report helps to identify bugs that have been fixed or introduced since the last time the tests were activated. This report is recommended for most test executions (except first runs, of course).
Certification report	Lists the total number and percentage of PASS/FAIL outcomes for the most recently executed test suite. This report is ideal for an immediate assessment of test outcomes.

You can view samples of these reports in another section (See Section 3.2.4.1 - “Report Window” on page 70.).

To view one of the previously generated reports from the **Report** window, click on the radio button for the desired report.

The selected report is automatically loaded into the **Report** window’s scroll display. The displayed report can be traversed using either a mouse or the vertical and horizontal scroll bars.

While a report is being viewed, header information can be added to the report, and the report saved to a file.

To add header information to a report:

1. Click on the **Enter Info** button. This window appears:

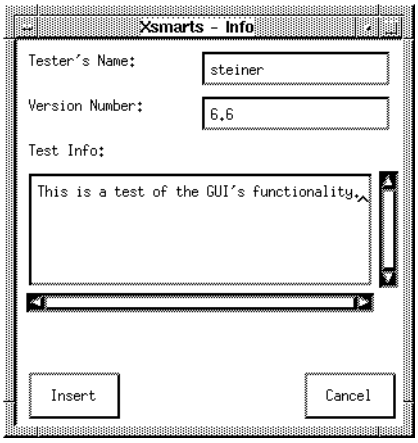

FIGURE 68 Enter Info Window

2. Click on the appropriate text field and begin entering information when the cursor appears.
3. Click on **Insert**. If **Cancel** is clicked on, the information will not be entered in the report.

To save a report's output:

The **Add Report** option is used to append a selected report to a previously specified file. The file can be set either by using the **File** menu's **Set Report File** option, by indicating the file name in the resource configuration file, or using the **`-rpt`** command line option.

1. Click on the **Add Report** button. A message box is displayed:

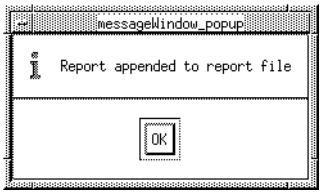

FIGURE 69 Add Report Dialog Box

2. Click on **OK**. If **Cancel** is clicked on, the displayed report will not be appended to the report file.

7.5 Purging the Log File

Over a period of time, the log file can become very large from the accumulating test information and may require extensive disk space. After executing a large number of test executions, it is recommended to purge the log file. Purging will leave only the most recent test execution information for each test case.

To purge the log file:

1. From the **Report** Window, click on the **Purge Log File** button. A message box pops up:

FIGURE 70 Purge Log File Message Box

2. Click on **OK**. If **Cancel** is clicked on, the log file will not be purged.

If the **OK** button was clicked, another message box is displayed:

FIGURE 71 Purge Log File Confirmation Message Box

3. Click **OK** to confirm the process.

7.6 Exiting the Report Window

The **Report** window need not be exited following each test session. The window is exited throughout the user manual for the purpose of practice. Exit the **Report** window by clicking on the window's **Close** button.

CHAPTER 8

Using the ASCII Menu Interface

This chapter explains how to use *SMARTS* from the ASCII menu interface. For how to invoke the menus, (See Section 5.3.2 - “smarts’ Runtime Options” on page 122.).

8.1 Menu Structure

After you have created baseline files and an ATS file (See CHAPTER 4 - Creating an ATS” on page 87.), you can use the *SMARTS* interactive menu interface to run tests. It is based on the following menus:

- **MAIN** is the root menu which appears when the **`smarts`** command is used.
- **BROWSE** is the menu from which tests are activated.
- **OPTIONS** is the menu from which the resource configuration is selected.
- **REPORT** is the menu from which reports are selected.

The menu’s options are typed at the keyboard, followed by a carriage return. Some commands have parameters which must be separated from the command with blanks or tabs.

It is not necessary to type the complete command name. For example, the command **`browse`** may be entered as **`br`**. However, shortened forms of commands must not be ambiguous; if they can be confused with more than one command, an error message is issued.

It is not necessary to use only lower case letters for a command. For example, **`HELP`** and **`HelP`** work just as well as **`help`**.

NOTE: To invoke the menus, please follow the instructions in another section (See Section 5.3.2 - “smarts’ Runtime Options” on page 122.).

8.2 Global Menu Commands

The following options may be used from all four menus:

!cmd — Passes the command *cmd* to the operating system command interpreter for execution. **smarts** waits for execution of the command to complete before regaining control. This feature allows you to run a system command without terminating **smarts**.

!! — Executes the previously issued system command (if one exists). This command allows repetition of system commands without retyping. **Caution**: The text of the repeated command is not shown.

DEL key — Allows you to abort the test run or system command.

exit — Exits the current level, or in the case of the **MAIN** menu, exits from **smarts**.

help — Displays a list of commands for which help is available.

help [*opt***]** — Displays help for the any of the menu's options. *opt* is the name of menu option. The help utility searches the default path for the help file */usr/lib/SR/smarts.hlp/smarts.hlp*.

If this file needs to be moved, the **help** parameter can be set for the current resource configuration file. Please see another section for further information on **help** (See Section 5.4 - "Resource Configuration File Processing" on page 125.).

Otherwise, you will need to specify the help directory with the **OPTION** menu's **chnghelp** option whenever **smarts** is run.

release — Commands *SMARTS* to report the version number and other implementation details.

settings — Displays all the current run-time settings on the screen.

8.3 MAIN Menu

The following menu commands are typed from the **MAIN** menu, which appears when **smarts** is invoked:

```
SMARTS:MAIN:

Options:

       browse   --Browse through the test hierarchy.
       report  -- Report on the test results.
       options  --Change current runtime options.
       settings -- Display current runtime settings.
       release  -- Show the current release of SMARTS.
       help [opt] -- Display HELP text for a command.
       exit    -- Exit to the system.
```

browse Calls the **BROWSE** menu from which you "walk through" your hierarchy of tests and activate tests individually or in groups.

exit Stops **smarts** and returns to the caller, usually the operating system command processor. All *SMARTS* files are automatically closed.

options Calls the **OPTIONS** menu from which you can view and the resource configuration's run-time parameters.

report Calls the **REPORT** menu from which you select summary reports which are based on log file data from tests you have run.

You can issue commands by typing the first few letters of each command. The only requirement is that the letter sequence be unique. *SMARTS* will inform you when a command you issue is ambiguous or matches two or more possible commands.

You can move from the **MAIN** menu to any other menu at will. *SMARTS* remembers the sequence of events in its own internal stack. This means that when switching from one menu to another, you can return immediately to the prior menu with the exit command.

8.4 BROWSE Menu

The following menu commands are typed from the **BROWSE** menu:

```
SMARTS:BROWSE:
Options:
tree            -- Display the test directory structure.
edit            -- Edit ATS file.
stats           -- Display number of nodes at each level.
cd <name>       -- Change down to a lower sub-group.
ce <pattern>    -- Same as cd <name>, but to subgroup w/ ending
                pattern.
cn <N>          -- Same as cd <name>, but to Nth subgroup.
cu              -- Change up to the previous group.
dir             -- Display the sub-groups.
tests           -- List the test structure and test names.
list_ats        -- List the Automated Test Script.
source <string> -- List only test cases with string as the source
source case <string> -- List only those test cases with the name
                string
go [opt]        -- Execute the tests in the sub-group.
time            -- Display the execution time for tests.
totime          -- Display only the total execution time for
                tests
report          -- Report on the test results in logfile.
options         -- Change current runtime options.
help [opt]      -- Display HELP text for command.
settings        -- Display current runtime settings.
exit            -- Exit current level.
```

The **BROWSE** menu can be selected from any menu and lets you select and activate tests in the hierarchical test suite, view the hierarchical tree structure, generate reports, or find out the amount of time a test or group of tests takes to execute.

There is always a specific test group or test case associated with the **BROWSE** menu prompt. This is generally referred to as the current position.

You can change the current position with either the **cd** or **cu** commands. The name of the current position is displayed at the **BROWSE** prompt, prefixed with **group** or **case** depending on whether it has been defined as a group or a test case. Initially, the current position is the first test case or test group defined in the ATS file, and is the **root** of the test tree. All the commands operate relative to the current position, i.e., **go** will execute the current position if it is a test case, or all sub-tests if it is a test group.

Consider the figure below in the following explanations of **BROWSE** menu commands (they are presented in alphabetic order). Note that in this illustration, Group A is the root of the test tree.

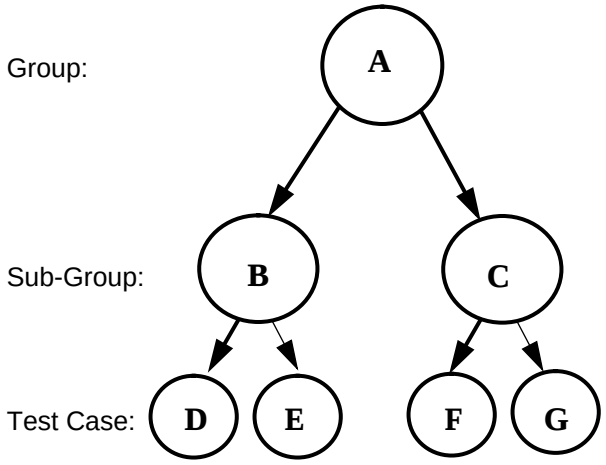

FIGURE 72 Hierarchical Test Tree Structure

`case` *string* — Lists only those test cases that have *string* in their case name. The list is detailed in the manner of **list_ats** commands.

`cd` *name* — Changes the current position to the *name* specified; *name* must be an immediate descendant as given by the **dir** command. If A is the current position, C will become the current position after typing **cd C**; however, G is not accessible directly from A. The inverse operation of **cd** is **cd..** .

`ce` *pattern* — The same as **cd** but changes the current position to a descendant with a substring ending in *pattern*. Consider the following list generated by **`dir`**:

```
test.33
test.34
test.35
test.36
test.37
```

If a **`dir`** command listed the above, then **`ce 35`** would change the current position to **`test.35`**.

cn *N*	The same as **cd** but changes the current position to the *N*th descendant beneath the current position listed in a **dir** command. For example, in the **dir** listing on the following page, **cn 4** would change to **test.36**.
cu; cd ..	Changes the current position to the one just above it. For example, if test case G were the current position, C would be the current position after **cu**. The inverse of **cu** is **cd**.
dir	Lists the sub-tests and groups under the current position. Only the immediate descendants will be listed. If A is the current position, **dir** would display:
go	Executes the current test if the current position is a test case, or all sub-tests if it is a group. A **go** command with no option will execute until there are no more test cases. If A is the current position, tests D, E, F, and G will be run; if C is current, only F and G will be run.
go auto	Executes the current test and/or sub-tests, and ignores all `evaluation with user` tests; they are assumed to PASS.
go fail	Begins execution from the current group but will only execute cases that FAILed according to its prior status as recorded in the current log file.
go on_fail *file*	Executes tests from the current position until a test fails. When a test fails, executes the group *file*.

go limit *N* — Executes the current test, and / or sub-tests, with a time limit of *N* seconds per test. If any test takes longer than *N* seconds, the test FAILs and an error message is printed. However, the testing process will go on to completion.

go new — Begins execution from the current group but will only execute cases that have not been logged in the current log file.

go pass — Begins execution from the current group but will only execute cases that PASSed according to the existing contents of the log file.

go repeat *N* — Executes from the current position *N* times.

go till_fail — Begins execution from the current position but will stop and return to the **BROWSE** menu on the first test that fails.

list_ats — Displays the actual ATS from the current position.

stats — Lists the hierarchical statistics for the current position, displaying the number of levels in the test tree and number of test groups or cases in each level.

```
Tree structure:
                    Number
        Level        Tests
            0            5
            1           10
            2            9
            3          199
        TOTAL    223
```

source *string*
Lists only those test cases that have *string* as part of their `source` clause.

tests
Lists, line by line, all sub-tests in the hierarchy relative to the current position.

Test case and group names are separated with the slash (`/`) separator. If the current position is A, **tests** will display the following paths:

```
/A
/A/B
/A/B/D
/A/B/E
/A/C
/A/C/F
/A/C/G
```

time
Reports the elapsed execution time for the test case or group based on information in the log file relating to the current position. If the tests were activated more than once, the most recent execution time is used, otherwise a question mark is displayed. If the current position is a group, then all sub-tests are reported and the result is accumulated.

totime
Reports the total elapsed execution time, without printing out the intermediate execution times or each test.

tree
Displays the group, sub-group and test hierarchy starting from the current position.

If the current position is A, **tree** will display the following:

```
( 1) A
( 2) .    B
( 3) ..        D
( 3) ..        E
( 2) .    C
( 3) ..        F
( 3) ..        G
```

8.5 OPTIONS Menu

The following menu commands are typed from the **OPTIONS** menu:

```
SMARTS:OPTIONS:
Options:

browse          -- Browse through the test hierarchy.
report          -- Report on the test results.
atsread <file> -- Read in a new ats file <file>.
logfile <file> -- Specify a new logfile <file>.
rcread <file>  -- Read in a new rc file <file>.
rcsave <file>  -- Save current settings to a rcfile.
rprtfile <file>-- Specify a file to send the reporting
                  information.

create          -- Toggle creation of baseline files on or off.
filesave        -- Toggle the savefile mode on or off.
diffsave        -- Toggle the saving of the diff output on or
                   off.
showsource      -- Toggle printing of ATS source during
                   execution.

displayinclude -- Toggle the display of including files  on or
                   off.
newdiff <path> -- Specify a new diff utility specified  in path.
newedit <path> -- Specify a new edit utility specified  in path.
chnghelp <path>-- Change the directory of the smarts  help
                   files.
width <size>   -- Change default status report width.
settings        -- Display current runtime settings.
help [opt]      -- Display HELP text for a command.
exit            -- Exit to the system.
```

The **OPTIONS** menu can be called from any menu. From this menu you can view and change *SMARTS'* run-time parameters, which have been set from either the command line or the configuration file.

Use the **OPTIONS** menu commands to perform the following functions:

`atsread` *atsfile* — **smarts** uses *atsfile* instead of the file specified in the current resource configuration file or the file specified with the **`-f`** command line switch.

`chnghelp` *path* — Changes the path specified in the current resource configuration file to be searched for the help file *smarts.hlp*. The default is */usr/lib/SR*.

`create` — Creates baseline files automatically whenever a test fails which contains an **`evaluation with`**

	baseline clause. The **-c** command line option can be also used during start-up.
diffsave	Toggles to the **diffsave** and **nodiffsave** modes set in the current resource configuration file.
displayinclude	Toggles between **displayinclude** and **nodisplayinclude** modes set in the resource configuration file.
filesave	Toggles between **filesave** and **nofilesave** modes set in the current resource configuration file.
logfile *file*	Uses **file** as the log file instead of the file specified in the resource configuration file or with the **-l** command line option. The default is *LOGFILE*.
newdiff *path*	Uses the comparison utility found at *path* instead of the utility specified in the current resource configuration file. The default comparison utility is **diff**.
newedit *path*	Uses the ATS edit command found at *path* instead of the command in the current resource configuration file. The default name used is **vi**.
rcread *rcfile*	Uses *rcfile* as the resource configuration (RC) file to be read in instead of the file specified with the **-r** command line option.
rcsave *rcfile*	Saves the current run-time parameters set with the **OPTIONS** menu to the resource configuration file *rcfile*.
rprtfile *file*	Uses *file* instead of the file specified in the resource configuration file or with the **-rpt** command line option.
showsource	Toggles resource configuration file's **noshowsource** to **showsource**. **showsource** displays the source clause for each test executed.
width *N*	Uses the *N* width for the **Status** and **History** reports instead of the width specified in the current resource configuration file.

Note: Please refer to other sections for information on the resource configuration file's settings (See Section 5.4 - "Resource Configuration File Processing" on page 125.), and the command line options(See Section 5.3.2 - "smarts' Runtime Options" on page 122.).

8.6 REPORT Menu

The following menu commands are typed from the **REPORT** menu:

```
SMARTS:REPORT:Options
browse          -- Browse through the test hierarchy
options         -- Change current runtime options.

status [width] -- Report on the status of the test results.

history [width]-- Report on all entries for each node (test).

regression     ---Report on the test result changes.

certification  -- Report on the statistics of test results.

information    -- Enter testing information for reporting.

purge          -- Delete all test results in the logfile.

help [opt]     -- Display HELP text for command.

settings       -- Display current runtime settings.

exit           -- Exit current level.
```

The **REPORT** menu allows you to select and generate reports on tests previously executed. The reports are generated from the log file that resides in the current directory.

The **REPORT** menu can be called from any menu, from the **BROWSE** menu, or from the **OPTIONS** menu. This menu contains commands to generate four different reports: the **Status** report, the **Regression** report, the **Certification** report, and the **History** report. Sample reports are shown at the end of this chapter. In addition, the **purge** command can be used to maintain the log file by deleting old execution records.

REPORT menu commands perform the following functions:

`certification` Generates a **Certification** report that summarizes the total number and percentage of PASS/FAIL outcomes for current position executed.

`history [`*width***`]`** Generates a **History** report that summarizes all test outcomes maintained in the log file. The *width* variable, if present, gives the width of the report. This is useful if the path names for tests are particularly long. The default is set to 30 characters.

`information` This allows you to enter information to be used as a header to any report. The following information can be entered:

- Tester's Name
- Test Version
- Test Description

To enter the tester's name and the test version, just type in the name followed by the carriage return. *SMARTS* then prompts for the test version. For test description, just enter text followed by a return for each line. When finished, type control-D (**^D**).

`purge` Deletes the old log file records and keeps the most recent information. The only information remaining after purging is the very latest. When run on a defective or damaged log file, **purge** indicates that it has found an error and cleans up the file.

`regression` Generates a **Regression** report that shows only the most recently executed test cases whose outcomes have changes since the previous execution.

`status` *[width]* Generates a **Status** report that reflects the test outcomes of the most recent test execution.The argument *width* may be used to specify the number of columns to be used for the test name in the **NAME** field of the report. The *width* parameter must be greater than 3 and less than 513.

Note: Examples of *SMARTS* reports can be found in another section (See Section 3.2.4.1 - "Report Window" on page 70.).

APPENDIX A

Customizing the GUI Environment

This appendix explains where the graphical user interface (GUI) setup information is stored and gives you instructions on how to change it.

A.1 SMARTS Setup Information

SMARTS' GUI can be customized by modifying the X Window System resource or setup files.

Resource, or setup files are text files which can be edited with any standard UNIX text editor. All the graphical user interface defaults are set in the *SR* resource file supplied with the product. The *SR* file is copied to the */usr/lib/X11/app-defaults* directory during the installation procedure. For further information on the *SR* file, please refer to the *Installation Instructions*.

To avoid constantly resetting the GUI parameters, the set defaults can be modified by manually changing the *SR* file. Here is a list of the GUI defaults:

```
smarts*optionSuppressEOTMessage.set: False
smarts*optionCreateBaselineFiles.set: False
smarts*optionSaveResponseFiles.set: True
smarts*optionSaveDifferences.set: False
smarts*optionShowATSSourceDuringExecution.set: False
smarts*optionDisplayIncludedFiles.set: False
smarts*ATSFile.dirMask: *.ats
smarts*RCFile.dirMask: *.rc
smarts*saveAsRCFile.dirMask: *.rc
smarts*logFile.dirMask: *
smarts*reportFile.dirMask: *
smarts*statusFile.dirMask: *
```

These defaults effect the default settings for options in the **Main** window of *SMARTS*. Each defaults is described on the following pages.

A.2 Default Settings

When opening the GUI-version of *SMARTS*, the on / off status of options and default values of options are determined by the default switch setting in the resource file. Within the resource file, switch settings are indicated by `set`, with `False` interpreted as off .

These defaults determine the on / off status of the **Toggles** submenu's options. `False` is interpreted as off; `True` is interpreted as on.

`smarts*optionSuppressEOTMessage.set: False` –

The **Suppress EOT Message** option is defaulted to off. Change False to True to suppresses the **Go** window's end-of-execution message.

`smarts*optionCreateBaselineFiles.set: False` –

The **Create Baseline Files** option is defaulted to off. Change False to True to overwrite baseline files with response files when test execution fails.

`smarts*optionSaveResponseFiles.set: True` –

The **Save Response Files** option is defaulted to on, thereby saving all response files. Change True to False to remove the response files after each test execution.

`smarts*optionSaveDifferences.set: False` –

The **Save Difference Files** option is defaulted to off. Change False to True to save the difference output for each test case that fails to *basename.diff*.

`smarts*optionShowATSSourceDuringExecution.set: False` –

The **Show ATS Source During Execution** option is defaulted to off. Change False to True to display the Automated Test Script's (ATS) source clause is displayed in the Go window during test execution.

`smarts*optionDisplayIncludedFiles.set: False` –

The **Show Included Files** option is defaulted to off. Change False to True to display any included files in the ATS file in the ATS Test Tree display.

For further information on the options affected by the GUI defaults (See Section 3.2.1.2 - "Option Menu" on page 56.).

A.3 Default Directory and Directory Mask Settings

Most users prefer to name certain types of files with unique extensions, i.e. naming all ATS files **`filname.ats`** and all configuration files **`filename.rc`**. When opening files, *SMARTS* has default masks to filter files that match the masks, allowing for easy file location of particular types of files.

These defaults determine the masks for **File** menu's options:

`smarts*ATSFile.dirMask: *.ats` –

Sets the ATS file directory mask to ***.ats** for the **Open ATS File** option.

`smarts*RCFile.dirMask: *.rc` –

Sets the configuration file directory mask to ***.rc** for the **Open RC File** option.

`smarts*saveAsRCFile.dirMask: *.rc` –

Sets the configuration file directory mask to ***.rc** for the **Save RC File As** option.

`smarts*logFile.dirMask: *` –

Sets the test execution log file mask to ***** for the **Set Log File** option. ***** will lists all the files in the current directory listed.

`smarts*reportFile.dirMask: *` –

Sets the report file mask to ***** for the **Set Report File** option, listing all the files in the current directory.

`smarts*statusFile.dirMask: *` –

Sets the test execution status file mask to ***** for the **Set Status File** option, listing all the files in the current directory.

Please refer to other sections for instructions on using directory masks (See Section 3.1.1 - "File Selection Windows" on page 44.), **File** menu's options (See Section 3.2.1.1 - "File Menu" on page 52.).

APPENDIX B

Recommended Usage

This appendix explains where the setup information is stored and gives you instructions on how to change it.

B.1 Automated Regression Testing

SMARTS is a powerful and effective tool for use in automated software testing. It is important to appreciate, however, that even though the *SMARTS* application is a comprehensive organization tool, the "engine" of an effectively automated testing system is test planning and script writing.

In practice, using *SMARTS* is only *part* of automating the regression testing process. Other parts of the process include creating test baseline (which may involve *CAPBAK/X*) and employing a variety of comparison methods.

B.2 Organizing Tests

Several common sense guidelines to remember when organizing a regression test suite:

- Organize tests into a hierarchy to facilitate test selection.
- When possible, divide tests into smaller, function-driven test suites. Should test re-structuring be desired, modular tests will expedite the re-structuring process.
- Like tests — tests for similar features of a system — should be grouped together.
- Always try to detemine the most effective way to execute comparisons between baseline and response files. If the *STW/ Regression* product bundle is available on the current system, the *EXDIFF* utility can be employed. (The *EXDIFF* utility has both ASCII and image file comparison capabilities.)

B.3 ATS Creation

After developing comprehensive test scripts, the test plan is transformed into an ATS (Automated Test Script). The ATS file is read by *SMARTS* to automate the testing process. Creating an ATS file is the most important and time-consuming part of using *SMARTS*. Follow the suggestions below to lessen the burden.

- In many ways, the ATS file structure should emulate the recommended organization of test suites. Within the ATS file, a minimal number of function-driven test groupings should be hierarchically arranged. This structure facilitates both the debugging and ATS restructuring process. For a first-time user, a large, detailed ATS file can be overwhelming.
- Use the **`environment`** clause to establish internal variables. Variables can define long path names as abbreviated paths, thus minimizing the amount of typing to be done in the ATS. Tests can also become portable from directory to directory by simply defining path as a variable. Please refer to other sections for further information (See Section 4.2.2 - "Test Case Clauses" on page 91.), (See Section 4.3.6.2 - "The Environment Clause" on page 101.).
- Take advantage of the product-supplied **makeats** utility. This utility expedites the process of creating an ATS file, especially when executed with the **`-F`** (Fast Generation switch) and the **`-K`** (Keysave File Standard Generation) switch options. Please refer to other sections for further information on this utility (See Section 4.5 - "The makeats Utility Overview" on page 106.), (See Section 4.5.1 - "Invocation and Use of makeats" on page 106.).
- Indent each group and test case within the ATS to improve readability. Use tabs only.

For complete information on organizing tests within the ATS file, please (See CHAPTER 4 - Creating an ATS" on page 87.).

B.4 Executing the Test Suite and Generating Reports

Having created the ATS file, the majority of user-required procedures are completed. The remaining procedures are to execute the test suite and view generated reports.

- Execute tests in the background with the **`-G`** *go_mode* option.
- Take advantage of the GUI's **Create Baseline Files** option in the **Main** window or the ASCII menu's **create** option in the **OPTIONS** menu. When an evaluation fails, this option overwrites a baseline file with its corresponding response files. Use this option the first time tests are executed, bypassing the manual creation of baseline files.
- To avoid clicking **OK** every time on the GUI's end-of-test execution message,activate the **Toggles** cascading menu's **Suppress EOT Message** option.
- When a test is executed, all of the resulting information is stored in a log file. Over time this log file can become very large, accumulating extensive amounts of disk space. Therefore, use the **Report** window's **Purge Log File** option in the GUI or the **REPORT** menu's **purge** option after several executions.
- To immediately determine whether a test has regressed since its previous execution, display the **Regression** report.

B.5 SMARTS and STW/Regression

SMARTS is designed as the central part of the *STW/Regression* product bundle. Using *CAPBAK/X*, a user session can be captured, and baseline files created. The user session and baseline files can then be referenced within the ATS file. The ATS file can command the test session to play back and compare the captured images.

The following is an example of how to set up a standard *SMARTS* play-back script. The ATS file can be manually created or the **makeats** utility can be used.

1. To run this script, we modified the default **diff** command to read **Xexdiff** in the *smarts.rc* file. This modification allows the **`evaluation with baseline`** clause to compare bitmap images rather than ASCII files. This script runs a standard *CAPBAK/X* keysave file that generates five saved bitmap images. Here is the same script with zero saved images:

```
define case FIVE
{
        source
"Play a keysave file that saves 5 images.";
activation
"Xplabak -k FIVE.ksv";
evaluation with baseline
                "FIVE.r01" vs. "FIVE.b01",
                "FIVE.r02" vs. "FIVE.b02",
                "FIVE.r03" vs. "FIVE.b03",
                "FIVE.r04" vs. "FIVE.b04",
                "FIVE.r05" vs. "FIVE.b05";
        }
```

`define case FIVE` includes a comparison of five images; it is assumed that the indicated images were saved using *CAPBAK/X*.

2. This script runs a standard *CAPBAK/X* keysave file that generates NO saved images.

```
define case NONE
{
        source
"Play a keysave file that does not saves 0 images.";
activation
"Xplabak -k NONE.ksv";
noevaluation
        }
```

`define case NONE` shows how to playback a keysave file in which no images were saved. (The results of playing back such a keysave file will always produce a PASS evaluation.)

B.5.1 Generating Standard Replay Scripts

To automatically generate the previous script passages with the **makeats** utility, use the command:

`makeats -K` *basename number*

basename	The basename of the keysave file.
number	The *number* of images to be compared. For the script to work, there must be at least as many windows as are specified here.

The two ATS passages indicated on the previous page could have been generated using the following command:

1. **`makeats -K FIVE 5`**
2. **`makeats -K NONE 0`**

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

W

X