Proteum - A Tool for the Assessment of Test Adequacy for C Programs
User's guide

Version 1.1 - C
18/3/96

Márcio Eduardo Delamaro
José Carlos Maldonado

Proteum 1.1

Abstract

This technical report presents the main features of Proteum (**Pro**gram **Te**sting **U**sing **M**utants), a testing tool that supports **Mutation Analysis** criterion. Proteum can be configured for testing programs in many procedural programming languages. This guide reports the version **1.1 - C** that works with the C language on SUN workstations, under OPENWINDOWS environment. Proteum has been developed at University of São Paulo (USP), São Carlos, SP, Brazil [DEL93] and used in teaching and researching activities at USP and at SERC/Purdue University.

Acknowledgment

Table of Contents.

1 - Introduction.

This guide presents the main features of Proteum (**Pro**gram **Te**sting **U**sing **M**utants), a testing tool that supports **Mutation Analysis** criterion [DEM78]. A previous knowledge of mutation testing is assumed. The reader may refer to [DEM78, WON93] for basic principles of mutation testing.

Proteum can be configured for testing programs in many procedural programming languages. This guide reports the version **1.1 - C** that works with the C language on SUN workstations, under OPENWINDOWS environment.

In Proteum a test is guided by test sessions. In each session the tester can create a program test, interrupt it and resume it later. Thus the first step is to create a program test. A program test is identified by the filename of the source program under test, filename of the executable program, names of functions to be tested, a compilation command to create the executable file from the source file and a name that identifies the test and used for saving/retrieving a testing session. In addition, the tester can choose a "type" for the program test. It can be a *normal* or a *research* test. These two types differ in the way mutants are executed. We get back to this point later in this section.

Once a test has been created, the tester can begin working on it. In particular one can save the current session to guarantee that the work done will not be lost, can quit Proteum and can retrieve the same or another test, using the name given at the time of its creation.

The central tasks in a test are to define a test set, to generate a set of mutants and to execute these mutants against the test set. Test cases can be inserted in the test set in one of two ways. When creating test cases interactively, Proteum asks the tester for the initial parameters (command line parameters) and, after the parameters have been provided, Proteum begins executing the program under test. The tester interacts with the program and can provide run time inputs to the program. The tester can see the outputs of the program during execution and decide whether the outputs are as expected or not. If not, an error is found and the tester might want to correct the program under test. Otherwise, input and output data are saved by Proteum and constitute a test case. Currently, only keyboard inputs and tty outputs are supported.

Test cases may also be inserted into a test set from files. Proteum uses data stored in files as run time inputs and executes the program under test on this data, generating the corresponding outputs. Again, inputs and outputs are stored as a test case. Proteum can import three types of files: plain ASCII files, POKE TOOL[1] test files and other Proteum test case files.

Test cases may be deleted or disabled. Deleting a test case physically removes it from a test set. Disabling does not remove a test case but logically excludes it from the test set. When executing mutants, the disabled test cases are not used. Disabled test cases can be re-enabled allowing the tester to try different test sets without re-entering test cases.

---

[1]POKE-TOOL is a data-flow and control-flow based structural testing tool.

The generation and execution of mutants is done in a simple way. First, the tester selects the mutant operators to be used. Proteum provides a set of 71 mutation operators divided into 4 classes: Statement, Operator, Variable, and Constant mutation operators (see [AGA89]). For each operator, the tester can specify the percentage of mutants to be generated. Proteum randomly generates only the specified percent of mutants. Mutants can be generated incrementally thereby allowing the test to be divided into different steps. In each step a small set of mutants may be generated for faster execution and easy analysis. The decision as to which mutant operators to select is a tradeoff between the cost of executing the mutants and quality of test set. This decision is based on the test requirements.

Once the tester requests Proteum to evaluate a test set, Proteum builds, compiles, executes each mutant, and compares its behavior with that of the program under test. The comparison of outputs is based on: outputs, return code and the execution time. Only the outputs written to "stdio" and "stderr" are considered for comparison. The return code is used to differentiate mutants that have the same output but abnormal termination. The execution time is used to distinguish mutants that enter into an infinite loop and are aborted to terminate their execution.

There are two ways Proteum executes mutants. One way, used when the test type is "Normal," is to execute each mutant from the first test case until it is killed or, if no test case kills it, execute it until the test set is exhausted. In case the test type is "Research" each mutant is executed against all test cases, regardless of whether the mutant has been killed or not. This approach enables extra data gathering. For example it is possible to analyze which operators are less efficient – those that have their mutants more easily killed – or counting the number of test cases that killed their mutants. This method of executing mutants certainly is not a practical way to evaluate test sets but may be useful for research aimed at establishing guidelines and cost effective strategies for industrial applications.

After executing the mutants, the tester has an evaluation of the test set. Proteum displays summary statistics about the current test session. For instance, the total number of mutants, the number of live mutants, the number of anomalous mutants and the mutation score are a few of the several useful statistics displayed.

Mutation score is a measure of test set adequacy. A large number of live mutants, implying a low mutation score, is generally attributed to a low quality test set. In addition, some of the live mutants may be equivalent to the program under test, and hence cannot be killed. Such mutants do not contribute to the adequacy test set. However, the current test set may be improved by adding test cases aimed at killing live mutants. In Proteum, it is a tester's task to decide whether a mutant is equivalent or not. Proteum displays the original and the mutated program, the mutant, and allows the tester to mark it as equivalent if that be the case. The tester can also interactively execute the mutant and try different data in order to kill it; this data is not inserted in the test set unless explicitly requested for insertion by the tester.

Proteum provides a report about the effectiveness of each test case. The tester can select what information is desired in the report. For instance, the number of mutants

executed using the test case; number of killed mutants, classified by the cause; number of live mutants; the input and output.

The remainder of this report explains details of the operation of Proteum.

2 - Environment

The directory where Proteum is installed, e.g. */usr/bin/proteum*, must contain the following files:

Executables:
>   proteu
>   tcase_ex

Tables:
>   chave.c (C keywords)
>   tabsin.c (C syntactic table)

Proteum must be invoked from a command window (console or cmdtool) in the Openwindows environment. Before invoking Proteum you need to set the directory where Proteum is installed.by setting the environment variable *PROTEUMHOME*.The following UNIX command may be executed to set the environment variable (assuming that the directory is */usr/bin/proteum*).

> *setenv PROTEUMHOME /usr/bin/proteum*

Having set the location of Proteum, type the command */usr/bin/proteum/proteu* to invoke Proteum. On startup Proteum shows the screen as in Figure 2.1.

3 - Testing Sessions

As mentioned earlier, Proteum's operation is based on **testing sessions**. A program test may be started, interrupted and resumed from the point stopped. To begin a session Proteum needs to be informed of parameters that identify a test session. It can be a new or an interrupted session. The *Program Test* button in the main menu can be used to inform Proteum about the name of the session. A new program test is started by choosing the option *New* or an old one is resumed by choosing the option *Load* (see Figure 3.1).
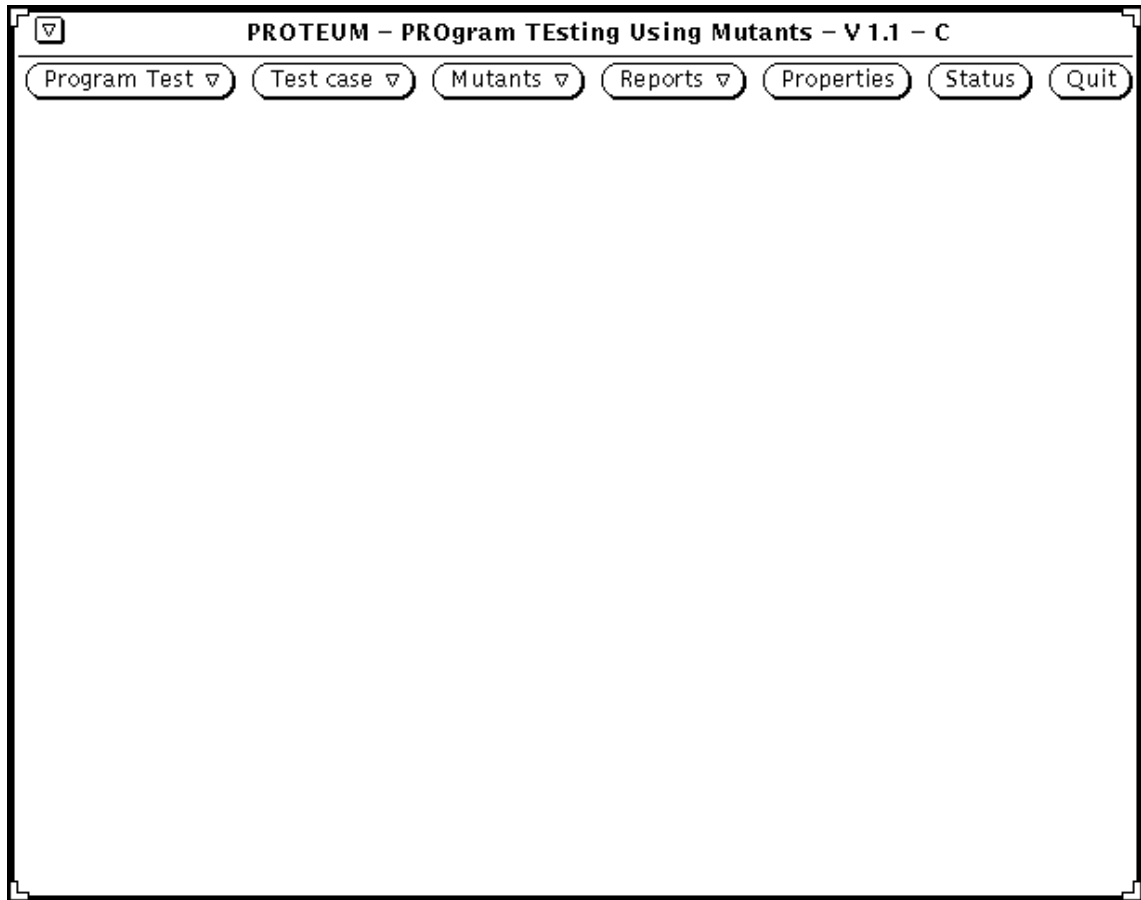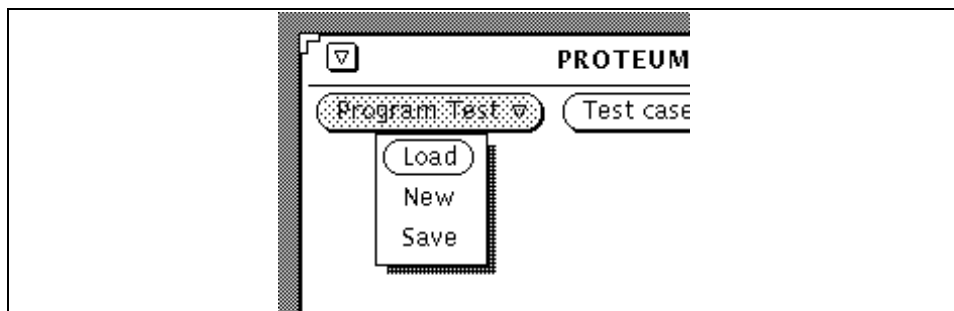
Figure 2.1 - Main Menu

Figure 3.1 - Starting a test session.

3. 1 - Creating a New Test

Selection of the *New* option pops up the panel which provides parameters of a new test (see Figure 3.2). These features determine, among others, the program to be tested.

Figure 3.2 - Creating a New Test

Directory:

In this field enter the name of the directory in which the program test is to be built. This directory must already contain the source file and the corresponding executable program file. In this directory, referred to as "work directory", Proteum creates the work files described below. If this directory does not exist, Proteum gives a warning and does not accept the name.

Program Test Name:

This field contains the name that identifies a test session. Often one may not care to select the source file name or executable file name to identify a test session. When testing a program in different ways, for instance when using different sets of mutants, one needs to select different names for a test session.

For example, suppose that in a test session program *myprog* is to be tested and is to be built from the file *myprog.c* (Figure 3.3). One may name this session as *myprog1*. In this case, Proteum creates the following files in the work directory:

myprog1.PTM

This file has general information about the test provided by the tester at the time of creation of a test session.

myprog1.TCS
myprog1.IOL
>These files together contain the test set used in test session *myprog*.

myprog1.IND
myprog1.MUT
>These files together contain the mutants used in test session *myprog*.

myprog1.LI
>This file has a simplified representation of the source file - called *Intermediate Language* - that is used by Proteum in the generation of mutants.

myprog1.c
>This file is created by *cpp*, the UNIX C preprocessor. Proteum uses this file in a test session and not the original source file (*myprog.c*).

```
#include        <stdio.h>

main()
{
int     a, b;

  scanf("%d %d", &a, &b);
  if (a > b)
       printf("\n%d is greater than %d", a, b);
  else
  if ( a < b)
       printf("\n%d is less than %d", a, b);
  else
       printf("\n%d is equal to %d", a, b);
  return 0;
}
```

Figure 3.3 - Program *myprogram.c*

Source Program:

>This is the name of the source file. It must have the suffix *.c*. and it must be in the working directory. If this file does not exist, Proteum gives a warning and does not accept the name. To test program *myprog* above, one must supply *myprog.c* as the name of the source program.

Executable Program:

This is the name of the file containing the executable (e.g., *myprog*). It must be in the same directory where the source file is located. If it does not exist, Proteum issues a warning.

Compilation Command:

This field contains the system command to create the executable program from the source program. This field allows the use of any compiler and any compiling and linking option. There is just one restriction: the source file name and executable file name must be used at least once in the compilation command.

Proteum uses this field to create executable programs from the mutants it generates. This is done by replacing the source file name by the mutant source file name; the executable file name is replaced by mutant executable file name. Proteum interprets the command *cc myprog.c -o myprog* as *cc <source file> -o <exec. file>*.

Link options and other file names may also be used to build the executable program. For example: *cc myprog.c yourprog.o -lmath -o myprog* is a valid command; it creates the executable file *myprog* by compiling *myprog.c* and linking it with the object file *yourprog.o* and with the library *libmath.a*.

Type:

Two types of test sessions may be selected: *Test* or *Research*. Using *Test* causes Proteum to behave in a conventional manner while executing and killing the mutants. Thus, when a mutant is killed by a test case, it is not executed on any remaining test cases. Using *Research* causes Proteum to execute each mutant on all test cases regardless of whether a mutant is dead or alive. Using *Research* also allows cross reference data gathering about test sets, aimed at the minimization of test sets mutation operator effectiveness analysis.

Functions:

This allows the selection of functions or subprograms to be tested. Thus, parts of a source file may be selected for mutation. The *All* option allows mutating the entire program. The *Select* option allows a tester to select one or more functions for mutation. In the case of selective mutation Proteum examines the source file and presents a list of functions found. One may then select functions from this list.

Once all the fields have been filled one clicks *Confirm* to complete the test session creation process. The *Cancel* button may be clicked to cancel the creation of a test session.

If a test session with the same name is already in the working directory, Proteum issues a warning. The existing test session can be overwritten. This situation is shown in Figure 3.4.
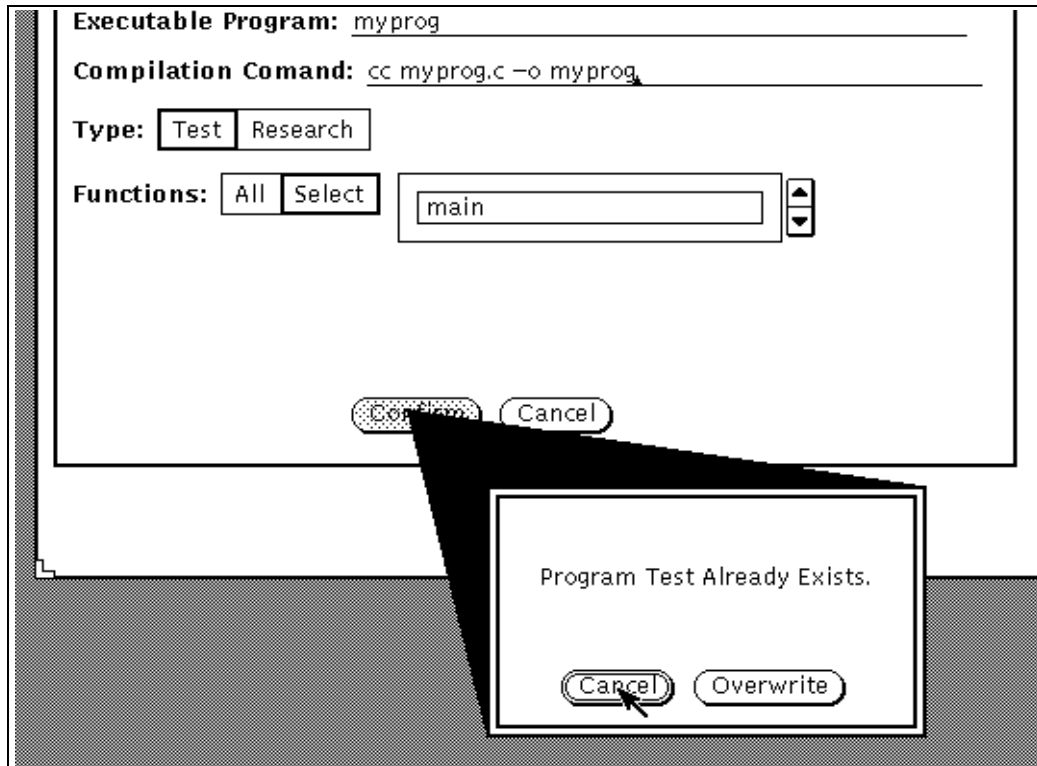
Figure 3.4 - Confirming a New Program Test

3.2 - Retrieving a Program Test

The Load option is used to retrieve an already created test. In this case (illustrated in Figure 3.5), Proteum requests for the directory where the test session was created and the name of the session.
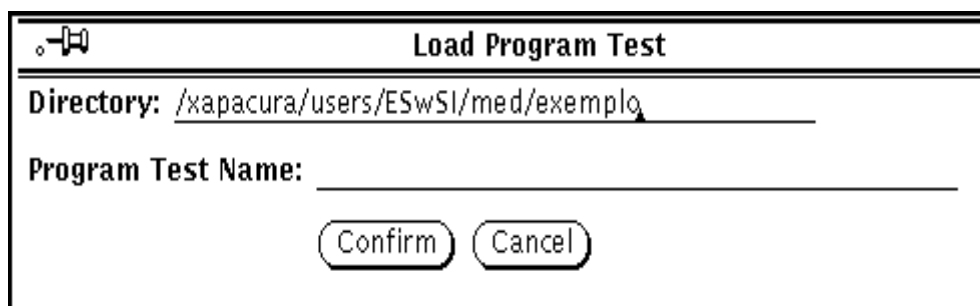
Figure 3.5 - Retrieving a Program Test

When a test session is resumed Proteum makes a copy of all work files of that session. This allows one to abandon a working session without saving the operations

performed during the session, thereby retaining the test state that prevailed at the beginning of the current session.

During a test session, if a request is made to retrieve an old test session or create a new one, Proteum completes the current session and allows one to save or abort the current session (see Figure 3.6). If option *Abort* is selected, Proteum restores the backup files created at the beginning of the session. The *Save* option confirms all operations thus updating the test state and terminating the session. The *Cancel* button aborts the load/create operations and the current test session is retained.

Backup files are created in the work directory and share their prefix with the names of the original files; the last character is replaced by the % symbol as follows:

myprog.PT%
myprog.TC%
myprog.IO%
myprog.IN%
myprog.MU%

3.3 - Saving a Session.

Operations done during a test session are saved using the *save* option from the menu in Figure 3.1. This operation forces Proteum to copy the updated work files to the backup files. Thus, even if one selects the *Abort* option when exiting a test session, the state of the test session until the last *Save* is retained.
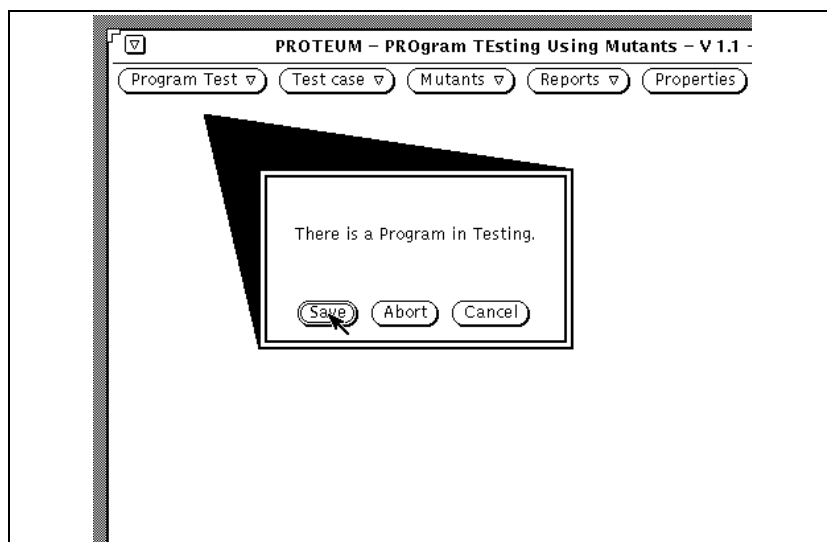
Figure 3.6- Choosing to Abort or to Save a Test Session

4 - Exiting Proteum

To exit Proteum one may select the Quit button in the main menu or in the frame menu (see Figure 4.1). In both cases Proteum verifies if there is an ongoing test session. If yes then Proteum prompts with the *Save/Abort/Cancel* menu.
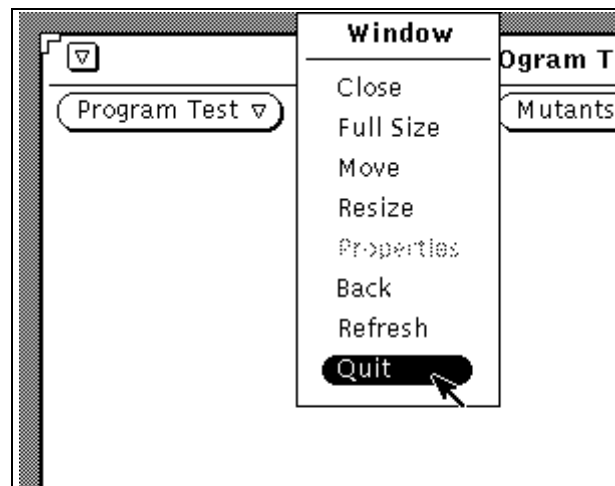
Figure 4.1 - Exiting Proteum

Some operations cannot be done at the same time. For example, one can not quit Proteum during mutant execution. The correct procedure in this case is to stop mutant execution or wait until mutant execution terminates and then quit Proteum.

5 - The Test Set

In addition to the program under test, the test set and mutants created are items that characterize a test session in Proteum.

Proteum provides four operations to manipulate test cases. These operations are showed in Figure 5.1: *Add* to insert new test cases; *View* to view existing test cases; *Delete* to delete test cases; and *Import* to get test cases from different sources.
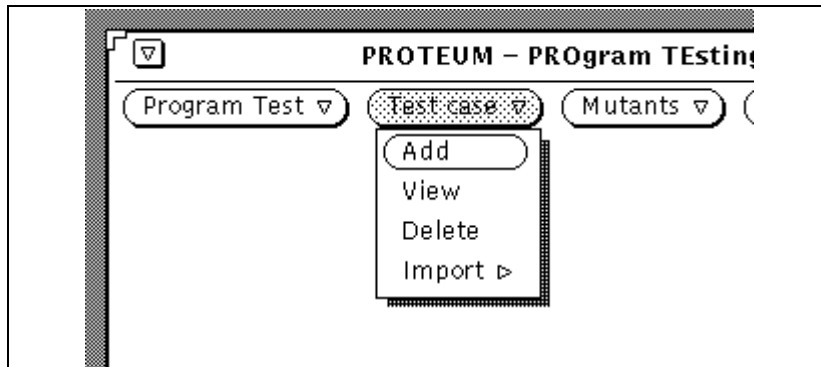
Figure 5.1 - Working with Test Cases

5.1 - Inserting Test Cases

To insert test cases Proteum needs program parameters. First the data that the program gets from system command line is provided. For example, in the command below, program *cd* is called with parameter */usr/bin*.

> *cd /usr/bin*

After obtaining the initial parameters Proteum begins executing the program under test. The tester may now interact with the program and supply the data it needs during execution. The output generated by the executing program is also shown on the screen. Proteum captures and stores the input supplied during program execution and the output that the program has produced. Together, the input and output from one program execution constitute a test case. Only text input/output are allowed; mouse input and graphic output are not supported by Proteum. It is the tester's responsibility to evaluate the correctness of the output. If the output is incorrect then the program contains at least one fault and needs to be corrected.

To implement the steps described above, Proteum first presents a panel (Figure 5.2) where the initial parameters are provided. If the test case does not need initial parameters, the panel may be exited, keeping it empty, by simply striking the *<return>* key.
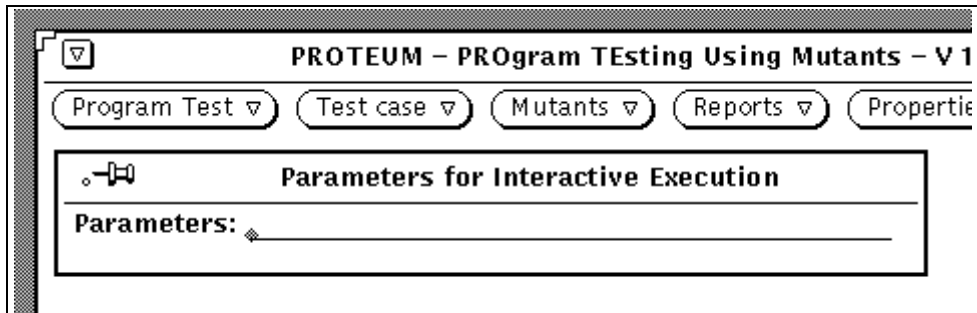
Figure 5.2 - Supplying the Initial Parameters

Next, a TTY subwindow is started where the tester interacts with the program. In this subwindow, the name of the program under test and an advisory that one may abort the test case insertion using *<CTRL><C>* are shown.

Figure 5.3 shows an example. Notice that at the top of the TTY subwindow no initial parameters were given. The remaining part of the window shows program *myprog* waiting for input from the tester. At the end of execution, the tester can determine if the results presented are correct or not. Based on this correctness assessment the tester may choose to confirm or to cancel the insertion of the test case as illustrated in Figure 5.4.

Figure 5.3 - Inserting a New Test Case

Figure 5.4 - Confirm/Cancel Test Case Insertion

5.2 - Querying a Test Set.

Test cases inserted may be viewed using the *View* option in the test case menu. Proteum presents a panel (Figure 5.5) showing the items that feature each test case. First field, on the top left corner is the test case number. Using this field, the tester may access any test case, by entering its number. If the number typed is greater than the last test case number, the last one is shown.

The tester may also walk through the test set using the up and down arrows. Proteum will show the next or the last test case.

Other fields in the panel present information about a test case. The field labeled *Parameters* presents the initial parameters of a test case. Field *Exec. Time* shows the CPU time (in hundreds of seconds) consumed by the current test case. This value is used to kill a mutant by timeout. Some mutants, when executing, can enter an infinite loop. Proteum monitors the amount of CPU time spent by each mutant. If this time exceeds a multiple of the CPU time spent by the original program executing the same test case, Proteum kills this mutant considering it to be in an infinite loop.

```
┌─────────────────────────────────────────────────────────────────┐
│ o─┤┼┤                        View Test Case                      │
├─────────────────────────────────────────────────────────────────┤
│ Test Case:  1̲_____  [▲|▼]        [▽] Enabled          │
│                                                                   │
│ Parameters: _____             │
│                                                                   │
│ Exec. Time (1/100 Seconds): 1̲3̲__       Return Code: 0̲__          │
│                                                                   │
│ The windows below show the Input and Output Logs.                │
│                                                                   │
│ ┌──────────────────────────────────────────────────────────┬──┐ │
│ │ 1 2                                                      ║▲║ │ │
│ │                                                          ║▼║ │ │
│ │                                                          ║  ║ │ │
│ │                                                          ║  ║ │ │
│ │                                                          ║  ║ │ │
│ ├──────────────────────────────────────────────────────────┼──┤ │
│ │ 1 eh menor que 2                                         ║▲║ │ │
│ │                                                          ║▼║ │ │
│ │                                                          ║  ║ │ │
│ │                                                          ║  ║ │ │
│ └──────────────────────────────────────────────────────────┴──┘ │
└─────────────────────────────────────────────────────────────────┘
```
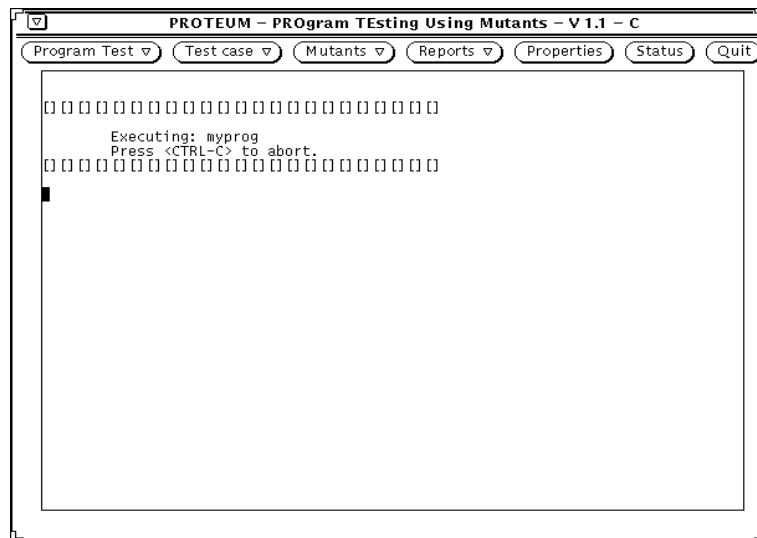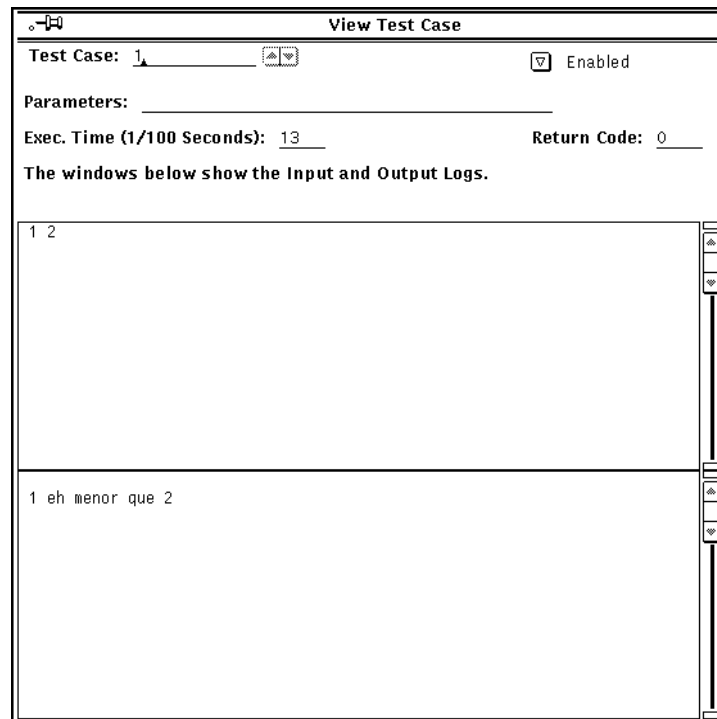
Figure 5.5 - Seeing a Test Case

The *Return Code*, as seen in the example, is the value that indicates the cause of program termination. A tester may consider it an indicator of execution success or failure. This is yet another item used to analyze the behavior of mutants. The mutant return code need not be the same as the return code generated when the test was executed on the program under test. If the test case and the mutant have a normal termination, Proteum considers their return codes as matching.

Subwindows below the panel show the inputs and outputs associated with a test case. The first subwindow shows the data typed when the tester inserted the test case and the second shows the data generated (output) by the program. Proteum compares this output with the output of the mutant to decide whether the mutant is dead or alive.

At the top right corner there is a mutually exclusive list with two options: *Enable* and *Disable*. Choosing the second disables the current test case. A disabled test case is not used in the execution of mutants. Any mutant already killed only by this disabled test case will become live. Disabling a test case does not delete it from the test set. In fact a test case may be enabled after it has been disabled using the *Enable* option as shown in Figure 5.6.
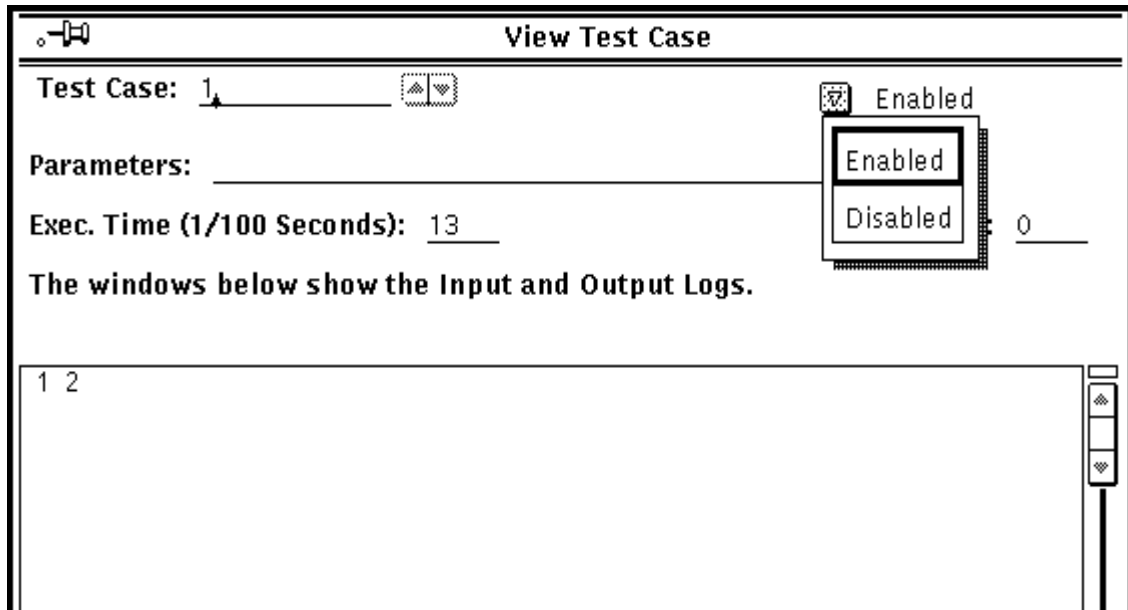
14

Figure 5.6 - Disabling a test case

5.3 - Deleting Test Cases

Test cases may be deleted using the *Delete* option (see Figure 5.1) from the panel in Figure 5.7. A range of test cases may be marked for deletion.
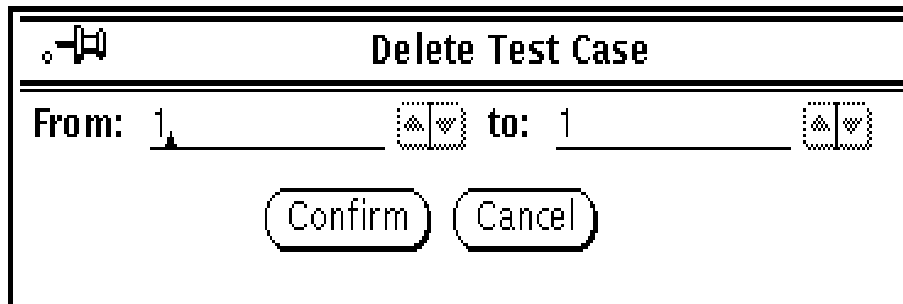

Figure 5.7 - Deleting Test Cases

After one or more test cases have been deleted, the remaining cases are renumbered. For example, a test set with 10 test cases is numbered from 1 to 10; deleting test cases numbered from 4 to 7, you get a set with test cases renumbered from 1 to 6. Thus, test case 8 prior to deletion becomes test case 4 after deletion, test case 9 becomes 5, and so on.

Once a test case has been deleted, it cannot be retrieved. To be able to use a deleted test case one must insert it again into the test set.

5.4 - Importing Test Cases

Proteum is able to import test cases from three sources: from other Proteum sessions; from a POKE-TOOL test; and from an ASCII file. Figure 5.8 shows these options. Importing test cases allows a tester to evaluate a test set that may have been created in other testing activities, perhaps using other testing tools.
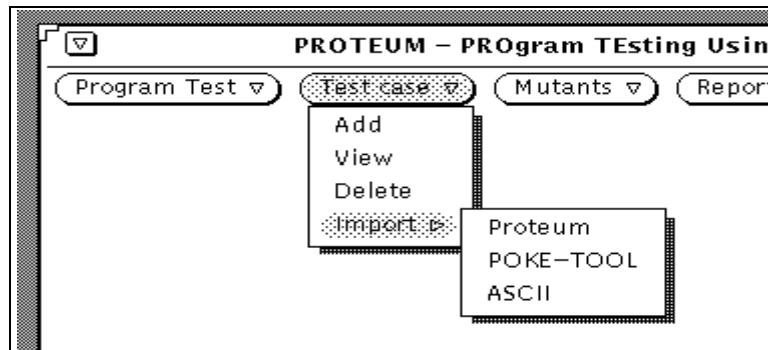

Figure 5.8 - Importing Test Cases

5.4.1 - Importing from a Proteum Session

One may want to test the same program with the same test set but with different features; for example, using different mutation operators. In this situation there is no need to insert the same test cases for each session. Instead an existing test set, created using another session, may be imported.

Proteum requests the directory and the session name from where tests cases are to be imported. Test cases are added to the existing test set in the current session. An example is shown in Figure 5.9.
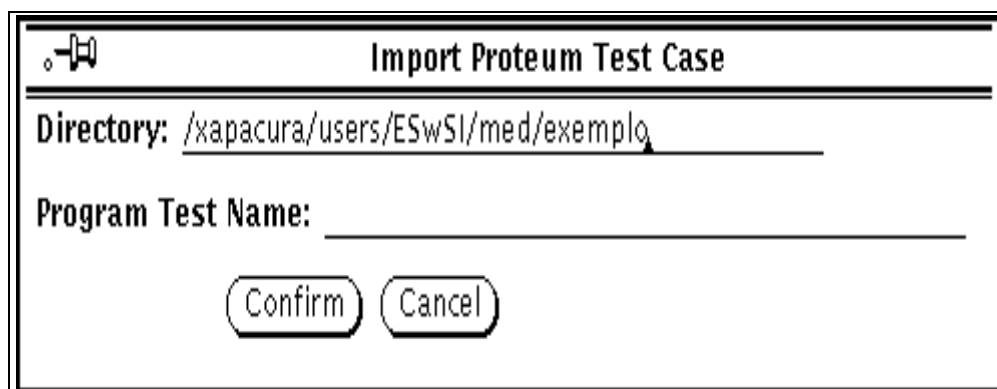

Figure 5.9- Importing Test Cases from Proteum

5.**4.2 - Importing from POKE-TOOL**

POKE-TOOL supports structural testing using the following control and data flow criteria: all-nodes, all-edges and **Potential Uses Criteria** [MAL91, CHA91]. The process to import test cases from POKE-TOOL process is almost the same as that for importing from Proteum. The difference is that Proteum needs to know the directory name where the test set is stored. Figure 5.10 shows the panel that allows to import test cases from POKE-TOOL.
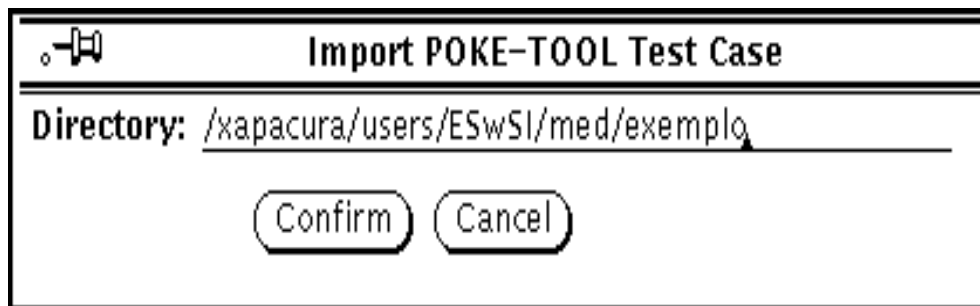

Figure 5.10 - Importing Test Cases from POKE-TOOL

5.4.3 - Importing from ASCII Files

Test cases may be imported from ordinary ASCII files. The contents of the ASCII file are used as a test. The output is obtained by executing the program under test.

Each ASCII file represents one test case. As in Figure 5.11, the file name and a range of numbers is provided by the tester. For example, if the file name is *test case* and the numbers in the panel are from 1 to 10, Proteum uses files named *testcase1*, *testcase2*,..... and *testcase10* to import test cases.
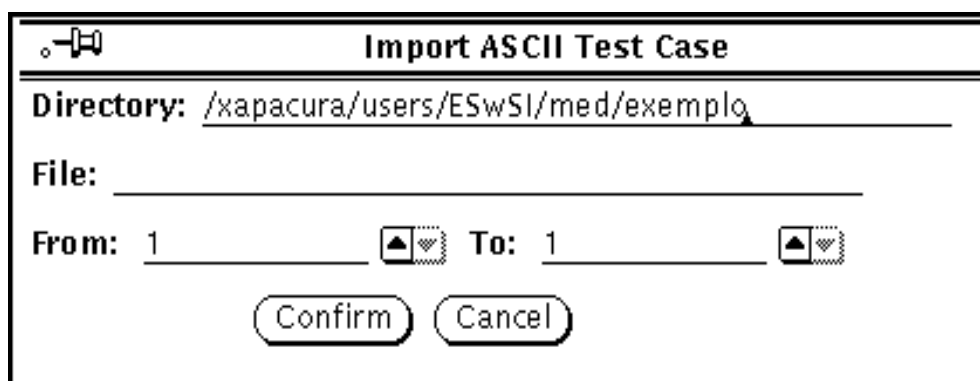

Figure 5.11 - Importing Test Cases from ASCII Files

6 - Working with Mutants

Figure 6.1 shows the operations provided by Proteum on files representing mutants. It is through these operations that test sets are evaluated.
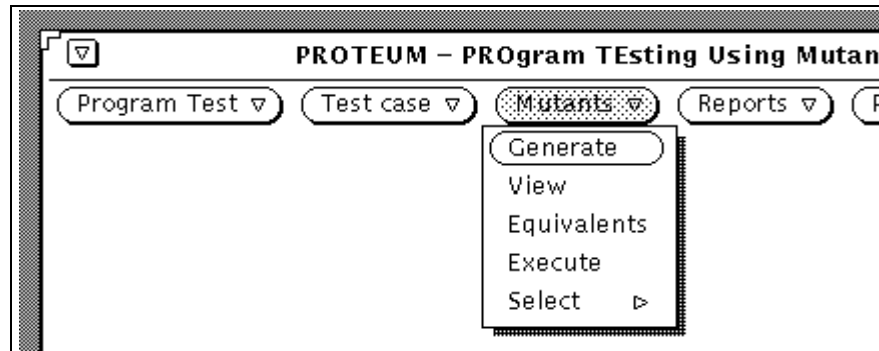

Figure 6.1 - Working with Mutants

6.1 - Generating Mutants

  By selecting option *Generate*   a tester selects the mutants to be generated.   More precisely, one may select the mutation operators to be applied on a program under test.

Figure 6.2 shows a panel with the choice of one of four mutation operator classes. Once a class is selected, Proteum presents the list (Figure 6.3) of operators in that class. For each operator two values are shown: the number of mutants already generated, and their percentage.   If, for some operator, the mutants are already generated, i.e. the percentage is different than zero, these values are shown in gray and cannot be modified. In the case the percentages are zero, the generating percentage for each operator may be specified.
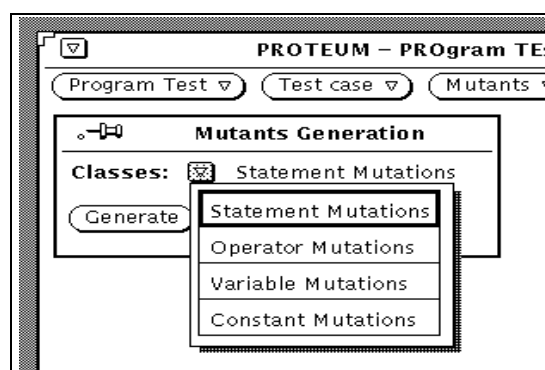

Figure 6.2 - Mutation Operators Classes
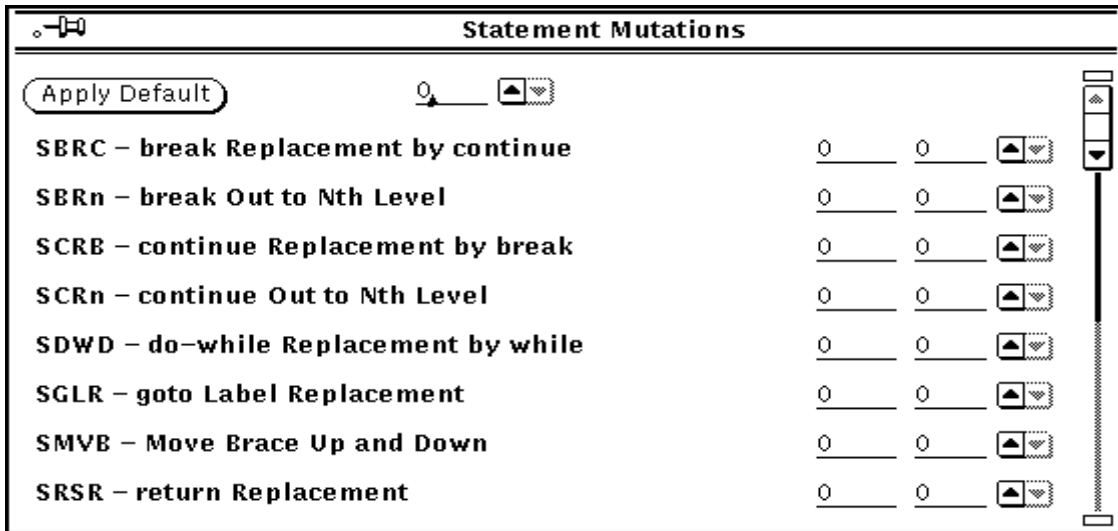
Figure 6.3- Choosing Generating Percentages

On top of the operator's panel there is a place where default value for all operators in a class can be provided. Changing this value and clicking the button *Apply Default*, the generating percentage of all operators (not in gray) in the class is changed.

After the generating percentages are selected, *Confirm* needs to be selected. This allows Proteum to begin generating mutants. While generating mutants Proteum shows the numbers of mutation operators being applied.

Option *Generate* can be used several times. After the first time, mutants can be generated using only the operators not used before, i.e operators for which the generating percentage was 0%. This situation is shown in Figure 6.4. Notice that operators already in use are disabled and their generating percentages cannot be altered.

Figure 6.4 - Generating More Mutants

19

6.2 - Executing Mutants

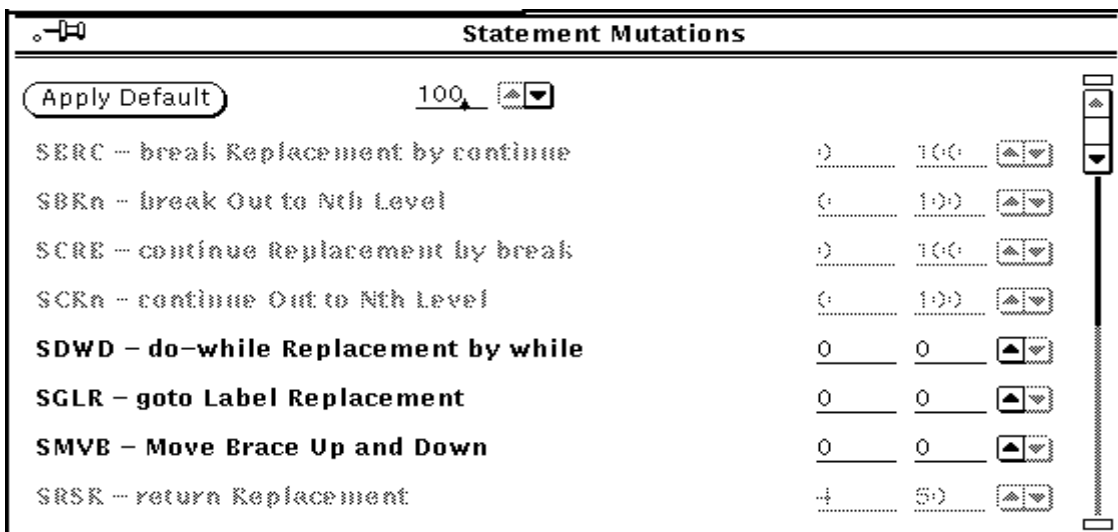The third option from the mutants menu allows the execution of the generated mutants. Through this option mutants are executed against enabled test cases within a test set. If a mutant gives different results from the program under test it is marked as **dead** by Proteum; else it remains **live**.

If the test session type is *Research* (see Section 3.1), each mutant, including dead mutants, is executed against all the enabled test cases. Thus one may obtain data relating all mutants to all enabled test cases. If the mutants have been executed before inserting a new test case, the tester needs to re-execute them thereby forcing all mutants to be executed with the new test case.

If the type of session is *Test*, only the live mutants are executed. For example: if a test set has 5 test cases and a mutant was killed by test case number 1, then it is not executed against test cases 2 to 5. Similarly, when a new test case is inserted only the live mutants are executed against the new test case.

It is also possible to select a subset of mutants to be executed. Through the *Select* option (Section 6.4), determine the mutants to be executed, i.e. the **active** and **inactive** mutants. The mutation score does not take into consideration the inactive mutants (more details in Section 7).

Disabling a test case can affect the state of a mutant. Taking the example above, if test case 1 is disabled, all the mutants killed by this test case are considered "live" once again. If the test session is of type *Test*, each live mutant is executed against the remaining test cases (numbers 2 to 5) until it is killed or until the test set is exhausted. In *Research* mode mutants are executed with all test cases not used in previous executions.

It is important to note that maintaining integrity of the relationship between a test set and the mutant status demands that mutants be re-executed each time the test set is altered. Thus, insertion, deletion, enabling, or disabling of one or more test cases must be followed by a re-execution of the mutants.

As shown in Figure 6.5, mutant execution can be interrupted by clicking the *Cancel* button. In this case, only the executed mutants at the time of interruption are taken into consideration while computing the mutation score.
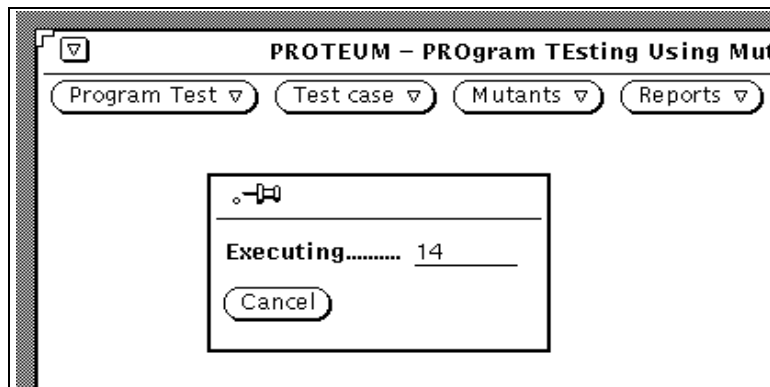
Figure 6.5 - Executing Mutants

6.3 - Analyzing Mutants

Once the mutants have been executed it is possible to analyze them. For example, it is possible to check why a mutant was killed. Similarly one may view mutants by the mutant number or by browsing sequentially through the set of mutants (see Figure 6.6).

Figure 6.6 shows a panel in which a mutant is presented. On the top right corner is a list to determine the type of mutants for viewing. For example, to view only the live mutants, first turn off buttons *Dead, Equivalent* and *Anomalous*. Then, while browsing through the mutant set, only the live mutants are presented. Inactive mutants may also be viewed by selecting the *Inactive button* in the list.

On the top left corner, the first field shows the mutant number. This field determines which mutant is to be presented. If the selected mutant number does not exist or its status does not agree with the types enabled to search (in the list described above), then Proteum shows the "closest" mutant with the correct status. For example, suppose that the viewing panel exhibits mutant number 8 and buttons *Live* and *Inactive* are selected. Now one types number 15 in the mutant number field. If mutant 15 is not live the next live mutant is presented, if there is one. If there is none, mutant number 8 continues to be shown in the panel.

The second field shows the mutant status which can be *Live*, *Dead* or *Anomalous*. For a dead mutant the reason it was killed is also indicated. Various reasons can be: mutant output is different than that of the original program (sent on *stdout)*; the return code from the mutant is different from that of the original program (*return code*), or mutant execution time exceeds, by some constant, that of the original program (*timeout*). If the type of the test session is *Research,* more than one test case can kill a mutant; in this case, the reason shown is the one associated with the first test case that killed the mutant.

The fourth field is a check box to indicate if the mutant is equivalent or not. Hence, the viewing operation is also used to mark equivalent mutants. Remember that equivalent mutants are not executed and therefore Proteum does not attempt to kill them. For dead and anomalous mutants, this field is shown disabled and cannot be used to mark a mutant as equivalent.

In the current version of Proteum, a mutant is anomalous if it has a invalid construction. In principle, a mutation operator should not generate a syntactically erroneous mutant but for some reasons, e.g. compiler features, one may get a mutant that can not be successfully compiled. This rarely occurs but in some cases anomalous mutants are generated.
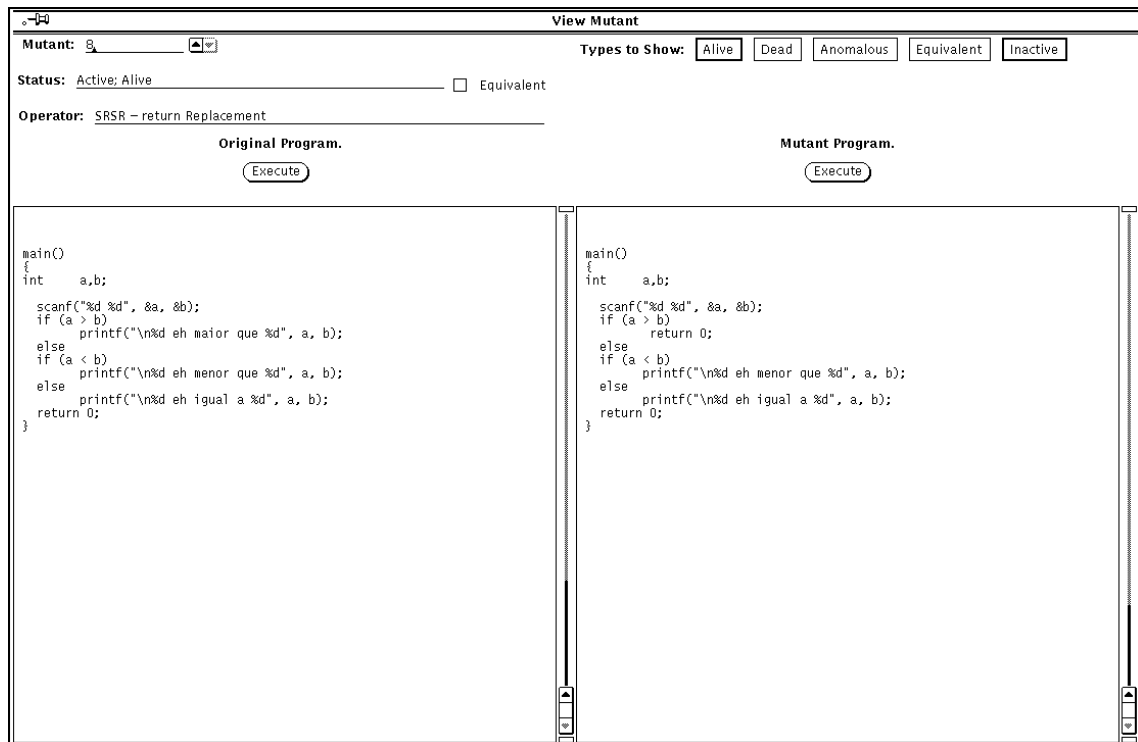
Figure 6.6 - Analyzing Mutants

The next field has the name of the mutation operator that created the current mutant. At the bottom there are two subwindows showing the original source code and the mutant source code. One may compare these and see how the source program was mutated. The point in source program (or first point if there are more than one) where the change occurs is (are) always shown in the subwindows. Above these subwindows there are two buttons both marked *Execute*. Clicking these buttons starts the execution of the original/mutant program. This way one may attempt to kill a mutant by trying out different test inputs.

6.4 - Selecting Mutants

Option *Select* enables the selection of subsets of mutants from already generated mutants. It means that one may determine that some mutants are deactivated and are not to be considered during the test session. Inactive mutants are not executed and are not considered when computing the mutation score. There are two criteria to select mutants: randomly selecting mutants from mutation operators (option *By Operator*) and selecting the number of mutants by program block (option *By Block*).

6.4.1 - Selecting Mutants by Mutation Operators

Using the option *By Operator*, generates a panel similar to the one used when generating mutants. In this panel one may select the percentage of mutants to be generated for each mutation operator or for each class of mutation operators. Proteum randomly selects mutants as per the percentage specified. For example, if one has specified that 50% of mutants be generated from the statement mutation mutants and 80% of the mutants from this same class, then one obtains 40% of active mutants from this class (0.5 x 0.8 = 0.4).

Note that for a mutation operator it is not the same to generate X% mutants and then to select Y% or to generate Y% and then to select X%; the resulting active set in these two cases are not the same. This is true only if X (or Y) is 100%; if one has selected 100% and then selected 30% one gets the same active set that one gets when generating 30% of mutants from a mutation operator or from the mutation operator class.

6.4.2 - Selecting Mutants by Block

This feature has not been implemented in Version 1.1-C.

7 - Test Status

Option *Status* from the main menu provides some information about the current test session (see Figure 7.1). The most important of this information is the *Mutation Score*, which indicates the adequacy of the test set used for testing the program under test. This score is relative to the active mutants.

In the status panel there is a field that contains the number of executed mutants. It is not necessarily the same as the number of generated mutants. This occurs because one may abort mutant execution. Hence only the executed mutants are considered while computing the mutation score. Thus, mutation score can be defined as:

$$ms = \frac{\#executed - \#inactive - \#alive - \#anomalous - \#equivalent}{\#executed - \#anomalous - \#equivalents}$$

Note that the inactive mutants are subtracted from the total executed mutants. This means that the number of inactive mutants shown in the status panel is obtained only from the executed mutants and not from the total generated mutants.

The mutation score can vary from 0 to 1. The goal of a mutation tester is to construct a test set that has a mutation score as near to 1 as possible. This also implies that the number of alive non-equivalent mutants must be close to 0.
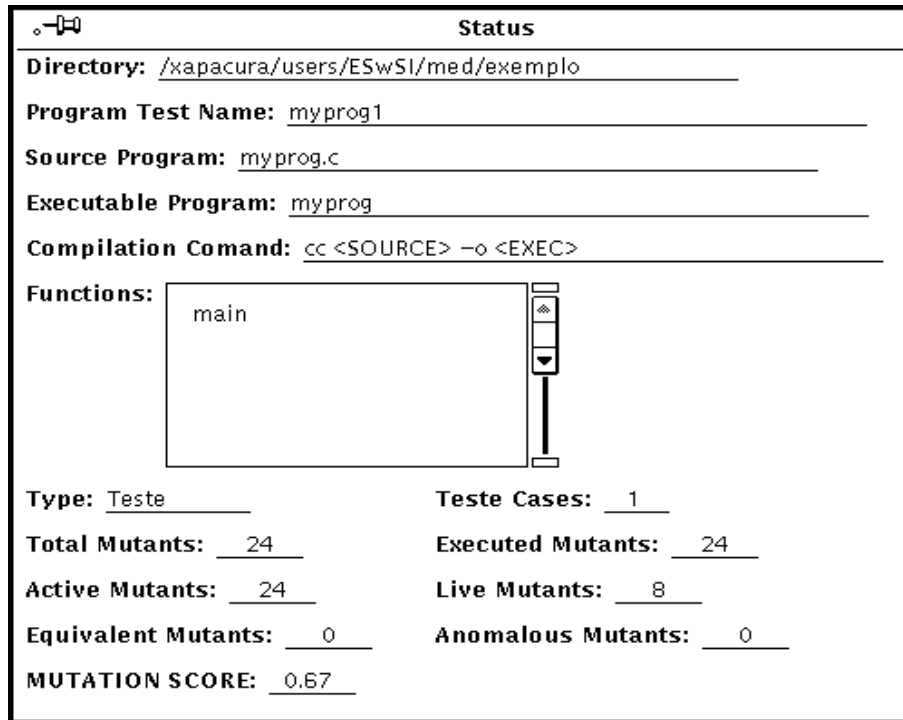
Figure 7.1 - Verifying Test Status

8 - Creating Reports

Another way to verify test status is through Proteum's reports. The current version of Proteum provides only the test case report described below.

8.1 - Test Case Report

Clicking the button *Report* and option *Test Case* gets a panel where to select what data is needed in the test set report. For example, the selection in Figure 8.1 produces a simple report shown in Figure 8.2. A complete report (Figure 8.3) is obtained by selecting all report options.
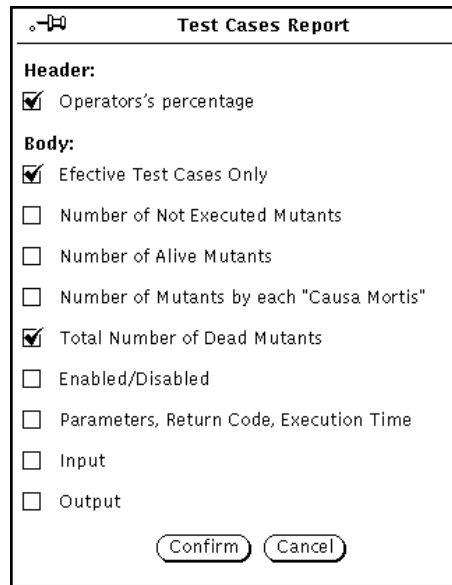
Figure 8.1 - Selecting Report Options
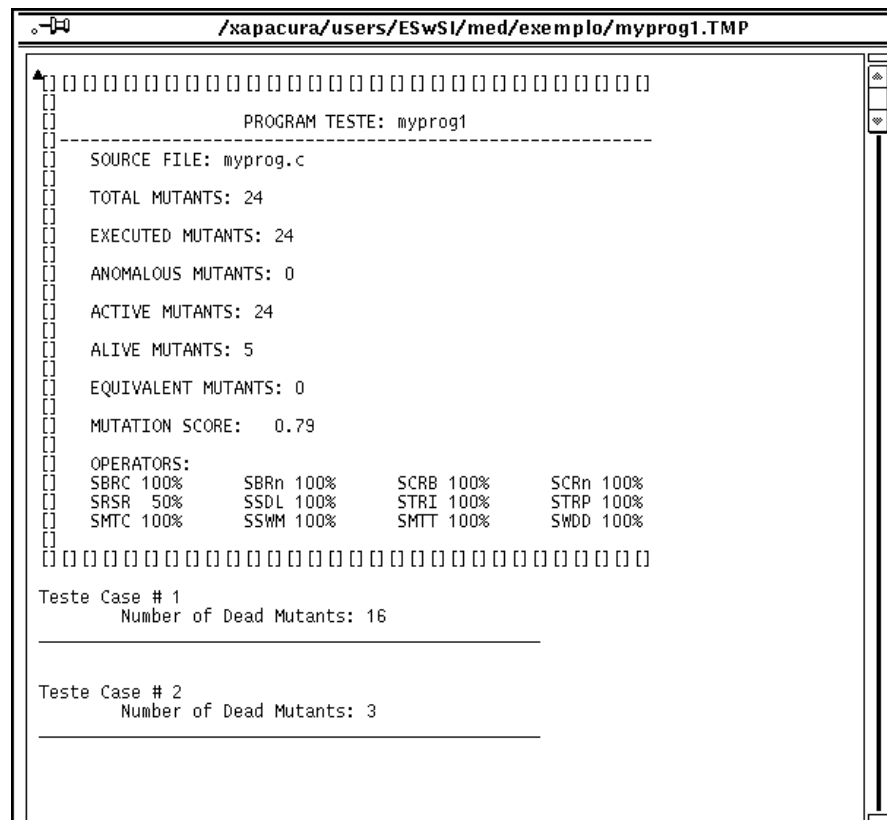
Figure 8.2 - A Simple Report

```
 ┌─────────────────────────────────────────────────────────────┐
 │ o▭⊞        /xapacura/users/ESwSI/med/exemplo/myprog1.TMP     │▲
 ├─────────────────────────────────────────────────────────────┤│
 │ []                                                          ││
 │ []    EQUIVALENT MUTANTS: 0                                 ││
 │ []                                                          ││
 │ []    MUTATION SCORE:   0.79                                ││
 │ []                                                          ││
 │ []    OPERATORS:                                            ││
 │ []    SBRC 100%       SBRn 100%      SCRB 100%     SCRn 100%││
 │ []    SRSR  50%       SSDL 100%      STRI 100%     STRP 100%││
 │ []    SMTC 100%       SSWM 100%      SMTT 100%     SWDD 100%││
 │ []                                                          ││
 │ [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] []││
 │                                                             ││
 │ Teste Case # 1                                             ││
 │        Number of Not Executed Mutants: 0                   ││
 │        Number of Alive Mutants: 8                          ││
 │        Number of Mutants Dead by Stdout: 8                 ││
 │        Number of Mutants Dead by Retcode : 0               ││
 │        Number of Mutants Dead by Timeout: 0                ││
 │        Number of Mutants Dead by Trap: 8                   ││
 │        Number of Dead Mutants: 16                          ││
 │        Enabled                                             ││
 │        Execution Time (1/100 sec.): 13                     ││
 │        Retorn Code: 0                                      ││
 │        Parameters:                                    ┌─┐  ││
 │        Input:                                         │▲│  ││
 │ 1 2                                                   ├─┤  ││
 │                                                       │ │  ││
 │        Output:                                        │ │  ││
 │                                                       └─┘  ││
 │ 1 eh menor que 2                                           ││
 │                                                           ││
 │   ─────────────────────────────────────────               ││
 │   ──────────────────────────────────────────              ││
 │                                                           ││
 │ Teste Case # 2                                            ││
 │        Number of Not Executed Mutants: 16                 ││
 │        Number of Alive Mutants: 5                         ││
 │        Number of Mutants Dead by Stdout: 1                ││
 │        Number of Mutants Dead by Retcode : 0              ││
 │        Number of Mutants Dead by Timeout: 0               ││
 │        Number of Mutants Dead by Trap: 2                  │▼
 └─────────────────────────────────────────────────────────────┘
```

Figure 8.3 - A Complete Report

9 - Configuring Proteum

Option *Properties* from the main menu enables one to configure some environmental variables used by Proteum. The panel in Figure 9.1 shows these variables.

The first field, *Default Directory*, determines the directory in which the test will be conducted. Every panel where a directory name is requested is initialized to this name. For example, the *Program Test/Load* panel (Figure 3.5) needs a directory name from where a test are going to be loaded. When the panel is first shown, the *Directory* field is filled with the default directory name. This name can be altered. At the beginning, the default directory is the current directory from where Proteum is invoked.

The second environmental variable is a time-out value for mutant execution. When generating a mutant, a change to the original program may produce a program (mutant) that, on some test cases, enters an infinite loop. The execution time is a parameter used to distinguish such mutants from the original program. If a mutant execution time is greater than the execution time of the original program execution time on the same test case by a fixed constant, then that mutant is considered dead. The user supplied time-out value determines this fixed constant. A very low time-out value can make non-infinite looping mutants being considered dead. On the other hand, a high time-out value will increase the total mutant execution time. The default time-out value is 5.
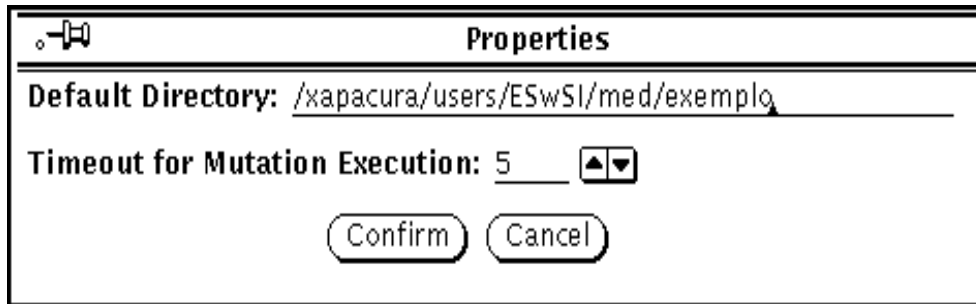
Figure 9.1 - Configuring Proteum

References

[AGA89] Agrawal, H., et al.- Design of Mutant Operators for the C Programming Language, Technical Report SERC-TR-120-P, Software Engineering Research Center, Department of Computer Sciences, Purdue University, W. Lafayette, IN 47907.

[CHA91] Chaim, M.L., POKETOOL - A Tool to Support Data-flow Based Structural Test of Programs, MSc Thesis, DCA/FEE/UNICAMP, April 1991.

[DEL93] Delamaro, M.E., Proteum - A Mutation Analysis Based Testing Environment, MSc Thesis, ICMSC-USP, October 1993.

[DEM78] DeMillo, R.A., Lipton, R.J., Sayward, F.G., "Hints on Test Data Selection: Help for the Practicing Programmer", Computer, vol. 11(4) , April 1978.

[MAL91] Maldonado, J. C., Potential Uses Criteria: A Contribution to Structural Test of Software, Ph.D. Dissertation, FEE/UNICAMP, Campinas - S. P., 1991.

[WON93] Wong, W.E., On Mutation and Data Flow, Ph.D. Thesis, C.S. Department, Purdue University, Dec 1993.