U S E R ' S G U I D E

TCAT-PATH

Version 8.2

Path Test Coverage Analyzer

SOFTWARE RESEARCH, INC.

This document property of:

Name:_____

Company:_____

Address:_____

_____

Phone_____

SOFTWARE RESEARCH, INC.

625 Third Street
San Francisco, CA 94107-1997
Tel: (415) 957-1441
Toll Free: (800) 942-SOFT
Fax: (415) 957-0730
E-mail: support@soft.com
http://www.soft.com

Documentation: Adam Heilbrun

(Last Update March 4, 1997)
user-manuals/coverage/97covbook/coverage.book/tcat-path/feb97/97t_pathunix.b

Table of Contents

List of Figures

Preface

This preface describes how the User Manual is organized.

Congratulations!

By choosing the TestWorks 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, and the peace of mind it provides our customers is our most valued feature.

Thank you for choosing TestWorks.

Audience

This User Manual is for both training and reference for software quality assurance testers or development staff who will use **TCAT-PATH** to test newly created or modified software.

TCAT-PATH is intended for any of the following Software Engineering professionals:

1. The Software Quality Analyst who intends to develop a complete set of tests for a system released by Research and Development.
2. The Software Metrics or Independent Evaluation Group that will measure and evaluate the testing of either sub or entire systems.**TCAT-PATH** enables this group to ‘‘test the testers." The coverage data might be combined with bug reports, complexity metrics or other data to guide software quality management.

Purpose

TCAT-PATH can be used for either:

1. *Unit testing,* where the focus of attention is one or more interconnected modules that will later contribute to a larger system.
2. *Measurement of the completeness of a test suite* for an entire system consisting of a large number of modules. This is informally known as the ‘‘big bang" testing approach.

Contents of Chapters

Chapter1	*SYSTEM OPERATION* gives a brief overview of **TCAT-PATH** features and operating modes.
Chapters 2, 3, 4, 5, and 6	These chapters explain how to use **TCAT-PATH**.
Chapter 7	*TCAT-PATH COMMAND SUMMARY - MS-DOS* explains source viewing.
Chapters 8 and 9	*TCAT-PATH COMMAND SUMMARY-UNIX* and *FULL TCAT-PATH EXAMPLE* explain the appropriate commands for each platform.
Chapter 10	*GRAPHICAL USER INTERFACE (GUI) TUTORIAL* displays a step-by-step full **TCAT-PATH** for “C” example.
Chapter 11	*SYSTEM RESTRICTIONS AND DEPENDENCIES* displays a step-by-step graphical user interface tutorial.
Chapter 12	*ON-LINE HELP FRAMES* lists restrictions and limitations.
Chapter 13	*COVERAGE MEASURE EXPLAINED* presents all of **TCAT-PATH’s** on-line help frames.

Typefaces

The following typographical conventions are used in this manual.

boldface Introduces or emphasizes a term that refers to **TestWorks'** window, its submenus and its options.

italics Indicates the names of files, directories, pathnames, variables, and attributes. Italics is also used for manual, chapter, and book titles.

"Double Quotation Marks"

Indicates chapter titles and sections. Words with special meanings can 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/MSW*'s keysave file language.

`Boldface Courier`

Indicates any command or data input that you are directed to type. For example, prompts and invocation commands are in this text. (**`stw`**, for instance, invokes *TestWorks*.)

CHAPTER 1

System Operation

This chapter describes how *TCAT-PATH* operates and explains the major operating modes for the package.

1.1 System Features

TCAT-PATH performs detailed path analysis of programs using a series of processing steps. Features of the *TCAT-PATH* system include:

- Automatic generation of structural digraphs from submitted programs.
- Automatic generation of complete path sets based on a unique SR-proprietary path analysis and equivalence class generation algorithm.
- Display of structure of structural digraphs using a special digraph visualization utility.
- Calculation of the cyclomatic complexity of programs.
- Automatic analysis of full trace files for instrumented programs (the instrumentation is generated automatically by the built-in *TCAT-PATH* instrumentor).

TCAT-PATH includes both command-line invocable processes and a fully interactive system.

1.2 System Information Flow

Figure 1 shows an overall data flow diagram of the *TCAT-PATH* system for “C" language.

The parts of the *TCAT-PATH* can all be command-line driven, and are designed to be usable with the standard UNIX pipeline and redirection facility.

In addition, a simple and friendly menu system (the user interface for the interactive version of *TCAT-PATH*) assists novice users in creating special “configuration" files which record the selection of run-time parameters, and executing the programs with on-line help.

1.3 Operating Modes

As Figure 1 on the following page suggests, there are several main modes for *TCAT-PATH* operation:

- **Analyzing** a file to extract digraph information about the included function(s) or procedures.
- **Viewing** the digraph for a particular program, relative to a specified basis path.
- **Generating** the set of paths that correspond to each program's structure.
- **Running** tests on the instrumented program to get aCt-compatible trace file of test coverage data.
- **Computing** the Ct coverage of a module or set of modules and producing reports.

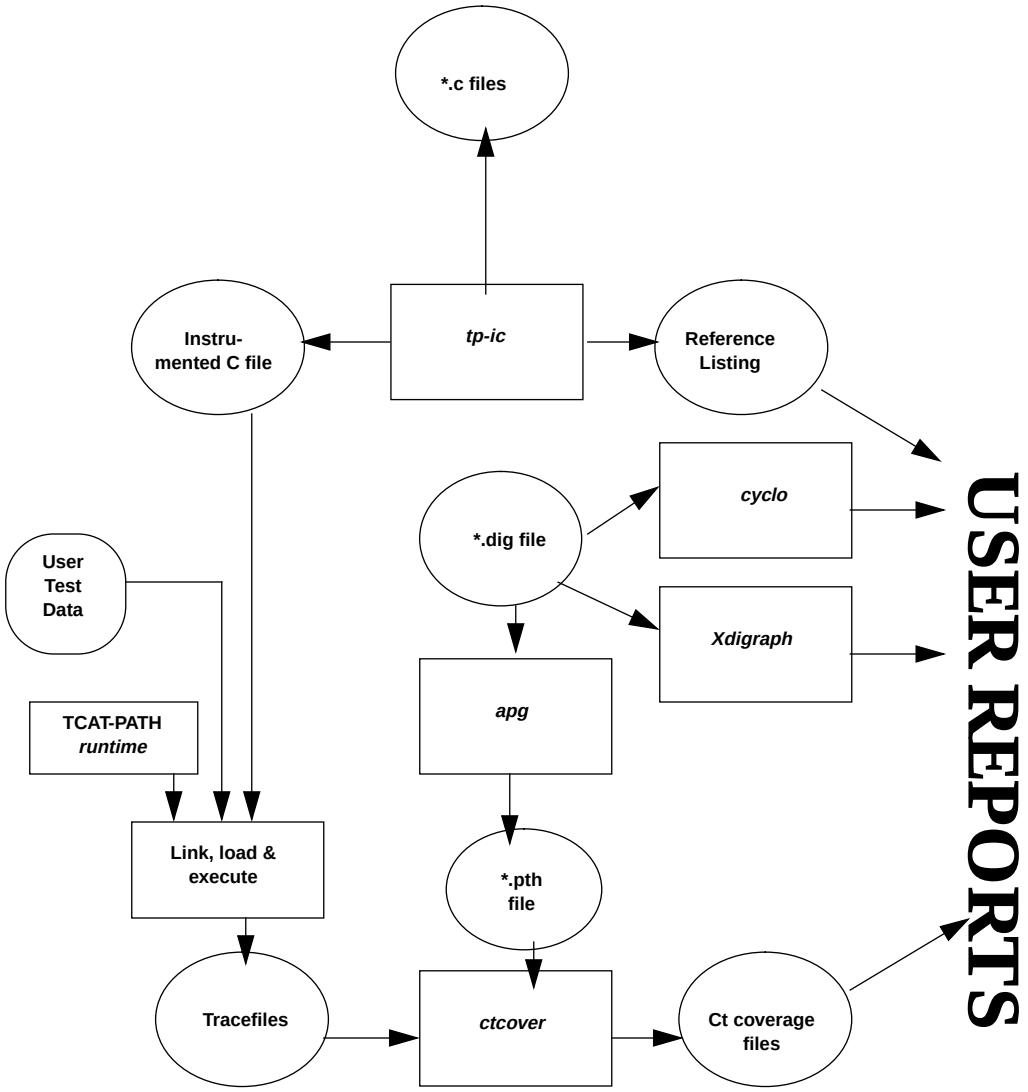


FIGURE 1 TCAT-PATH System Diagram

1.4 TCAT-PATH Functional Methodology

The *TCAT-PATH* package consists of three main systems: the tp-i<lang> instrumentor processor (see Note below), apg, and ctcover which can be used individually as command-line invocable units, or with the *TCAT-PATH* interactive menu system. In addition, there are several other sub-functions and support scripts that can be used independently.

NOTE: The only language-dependent component of *TCAT-PATH* is the instrumentor itself.

For simplicity, and because *TCAT-PATH* is available for a variety of languages, we refer to this element of the system in general terms as tp-i<lang>. Typical forms for this command, which can be modified by the user through the *TCAT-PATH* configuration file, are:

- tp-ic for "C" programs
- tp-ic++ for "C++" programs
- tp-iada for Ada programs
- tp-if77 for FORTRAN (f77) programs

Chapter 8 describes special characteristics of the instrumentor with which *TCAT-PATH* could be supplied.

Here is an informal description of how you can use the *TCAT-PATH* components to measure path coverage.

The methodology for using *TCAT-PATH* is based on the following typical scenario: you want to measure the Ct coverage values for a group of functions that are coded a few at a time into several files.

- STEP 1: Create a Working Directory.Set up a directory in which to keep all of your intermediate files. *TCAT-PATH* uses filename extensions on basenames.

  Your working directory should have copies of the source files, plus any supporting files you need to run tests on these files after they have been instrumented.

- STEP 2: Instrument and Generate Digraphs. You instrument and generate digraphs by processing all of the files with the supplied *TCAT-PATH* instrumentor (the specific digraph processor and instrumentor depends on the language you are processing). If some processed files contain more than one module (function), then the **`tp-i<lang>`** command will split up the digraph data and create separate digraph files each named after the corresponding module.

- STEP 3: Generate Paths.You use the **apg** command to generate the path sets for each module. Some modules may have “too many" paths. You have to make this determination; *TCAT-PATH* does not impose internal size limits, but your situation and other practicabilities may!

  The script **DoPTH** can be used to generate all of the *.pthfiles for all *.dig files in the working directory.
- STEP 4: Study Structure and Properties. Use **cyclo** and **digpic** to study the properties and structure of each ***.dig** file.

  These two commands can help identify “too complex" modules, and gain intuition about the internal structure of the software you are analyzing. You may wish to avoid trying to analyze *Ct* coverage for modules with more than 300 paths (for example).

NOTE: The scripts **DoCYC** and **DoPIC** help you run the cyclo and digpic commands on all ***.dig** files in the working directory.

- STEP 5: Generate Trace Files. You have to re-compile the instrumented programs (generated automatically by *TCAT-PATH* 's **tp-i<lang> command**), and link them with the supplied runtime object module.
- Then you execute the program as you normally would on an uninstrumented program. The result of this will be one trace file per test. If you have multiple tests you can append each test to the end of each trace file (note that the trace files cannot be reduced, because such files do not have essential segment sequence information).
- STEP 6: Evaluate Ct Coverage. For each module, and for the set of all trace files you think are appropriate, you call ctcover to produce the standard Ct coverage report.

  This report contains an image of the ***.pth** file for reference purposes. The script **DoRPT** can be used to handle generating the ***.rpt** files for all basenames (for which there are ***.pth** files) in the working directory.

CHAPTER 2

Instrumentation

This and the next four chapters tell how to use *TCAT-PATH* to increase test coverage and detect more software errors. There are two ways to access *TCAT-PATH*: with command line commands and with menus.

The following command line invocations are the focus of these chapters.

1. Instrumentation (marking segments)
2. Compiling and Linking with Runtime (recording and counting markers) and Executing
3. Path generation (generating complete path sets)
4. Coverage analysis (reporting path hit)

A description of how to use the menus appears in Chapter 6.

2.1 Overview

In brief, *TCAT-PATH* instruments the source code of the system to be tested, that is it inserts function calls at each logical branch. The instrumentation will not affect the functionality of the program. When it is compiled, linked and executed, the instrumented program will behave as it normally does, except that it will write coverage data to a trace file. There is some performance overhead due to the data collection process.

The trace file is processed by a report generator described later. The file resulting from instrumentation is then used for path generation. These generated paths are also processed by the report generator.

Finally, the user looks at the coverage reports to assess testing progress and to plan new test cases. New test cases are added in subsequent passes until a threshold percentage of *Ct* logical path coverage has been reached.The coverage reports guide the addition, or possibly the deletion, of tests.

2.2 Instrumentation

As already mentioned, an instrumented program is one that has been specially modified so that, when executed, it transmits information about Ct coverage at every stage of testing, while in all other respects, the logic functions just as in the original program.

In its operation, *TCAT-PATH*'s instrumentor parses your candidate source code, looking for logical branches. When one is discovered, the instrumentor inserts a function call in the instrumented version of the source code. It is important to note that the resulting source code file is still a legal program, as was the original program. The only difference is the added function calls.

When executed, the inserted function calls write to a trace file. Remember, the instrumented version will otherwise function as the uninstrumented version.

2.2.1 The Instrumentor

This command reads a ***.<lang>** file and produces a ***.dig file** for each module in the ***.<lang>** file. It also instruments the ***.<lang>** file and produces an instrumented version of the file and other reference and statistical files. For a complete listing on the files produced by the instrumentor, please refer to Section 2.4, File Summary.

The generic syntax for command line calls to the instrumentor follows.

```
tp-i<lang>  [options]  file.ext [file.ext]
```

where,

file.ext File(s) to be instrumented. **ext** is language specific (e.g.”c” or “i” for a C file). If there are multiple files, then each is processed in the order presented.

options Instrumentation options are also language specific. Options for the “C" language are presented in the next section. Options for other languages are listed in Chapter 8.

2.2.2 The "C" Instrumentor

The complete syntax for command line calls to ic is listed below.

```
tp-ic file.ext [file.ext]
    [-ce]
    [-cw]
    [-DI deinst-file]
    [-fl value]
    [-fn value]
    [-help]
    [-I]
    [-lj]
    [-m]
    [-m6]
    [-n]
    [-t]
    [-u]
    [-w]
    [-x]
    [-z]
```

This command instruments submitted "C" language file(s). It takes ***.i** source file(s) and produces the instrumented file(s): ***.i.c** (for UNIX) or ***.ic** (for MS-DOS or OS/2). ***.c** is the "C" source file, while ***.i** is the preprocessed file.

It is required that the user preprocess the source file through a "C" preprocessor before passing it to tp-ic. Normally, the preprocessing command is:

```
cc -P file.c (for UNIX)
```

or

```
cl -P file.c (for DOS running Microsoft C)
```

These commands read **file.c** and produce **file.i**. The following options may be used to vary the processing and reports generated by the instrumentor.The options are listed in alphabetical order.

file.ext File(s) to be instrumented.ext can be "c" or "i". If there are multiple files, then each is processed in the order presented.

-ce Preprocesses conditional expressions of the form ? a : b.

-cw Suppresses the "Conditional Expressions Not Processed" warning message.

-DI deinst-file De-instrument Switch. Allows the user to specify a list of modules that are to be excluded from instrumentation. Only the list of module names found in the specified deinst-file is to be excluded from instrumentation. The module names can be specified in any format. White space (such as tabs, spaces) is ignored. This switch effects the instrumented (***.i.c**) file and the reference listing (***.i.A**) file.

-fl *value* Allows the user to specify the maximum length of filename characters that are allowable on the system. If the length of a generated filename exceeds the value, then the instrumentor output will be redirected to files named **Temp.i.?**. These files can be used in subsequent processing.

-fn *value* The flexname switch.Allows the user to specify the maximum characters of function names the instrumentor recognizes. If the function name exceeds the value, then the instrumentor will recognize as distinct only the first value characters of the function name. For instance, a **-fn 5** will recognize the first five characters as distinct. Characters beyond that point, however, will not be recognized for function name purposes.

-help Help Switch. Forces output to show a summary of available switches.

NOTE: This is also the output produced by any illegal command to tp-ic.

-I Ignore Errors Switch. In certain rare cases, when the underlying “C" compiler supports non-standard options and constructs, it may be desirable to “force" instrumentation to occur regardless of errors found.

This is done with the **-I** switch.

CAUTION: When instrumentation is forced using this switch, there is a chance that the instrumented software will not compile.

For example, if you use the **-I** switch to “instrument" a file of text material, you would not expect the output to be compilable (and it probably won't be), even though it may have been “instrumented".

-lj	Processes setjmp and longjmp. This option only works for UNIX.
-m	Recognize Microsoft C 5.1 keywords during the instrumentation process. **Note**: This switch applies only to MS-DOS and OS/2 versions. This switch may produce unusual results if used in UNIX systems.
-m6	Recognize Microsoft C 6.0 keywords during the instrumentation process. **NOTE:** applies only to MS-DOS and OS/2 versions. This switch may produce unusual results if used in UNIX systems.
-n	Will not instrument empty edges (for example: if without else or switch without default.)
-t	Recognize Turbo C keywords during the instrumentation process. Note: This switch applies only to MS-DOS and OS/2 versions.
-u	Forces the instrumentor to recognize _exit as exit.Note: This switch applies only to MS-DOS and OS/2 versions.
-w	Recognize Whitesmith C keywords during the instrumentation process. Note: This switch applies only to MS-DOS and OS/2 versions.
-x	Will not recognize exit as keyword.
-z	Recognize MANX/AZTEC "C" keywords during the instrumentation process. **Note**: This applies only to MS-DOS. This switch may produce unusual results if used on UNIX systems. If there is an error, **tp-ic** gives a response line, or usage line, indicating the set of possible switches and options, which is the same as the **-h** output.

2.3 Instrumenting With ‘make' Files

Most often, *TCAT-PATH* will be used to develop test suites for systems that are created with ‘make' files.Make files cut the time of constructing systems, by automating the various steps necessary to build the system, including compiling and linking.

Fortunately, it is possible to add a few statements to most ‘make' files to enable them to make an instrumented version of the system. The modifications fall into two general categories, based on whether or not the make file explicitly names the compiler.

The rest of this section will assume the use of the “C" compiler. For any other language, the user can substitute the corresponding command in the language.

If the ‘make' file explicitly mentions the “C" compiler with a cc command (for example), it is possible to add the tp-ic command and an extra cc command for preprocessing, instrumenting and compiling causing the make script to instrument and compile the “C" files in question.

Make file lines such as:

```
UNIX:

    sample.o:sample.c
        cc -c sample.c

MS-DOS and OS/2:

    sample.obj:sample.c
        cl c sample.c
```

would be changed to:

```
UNIX:

    sample.o: sample.c
        cc -P $(CFLAGS) sample.c
        tp-ic sample.i
        cc -c $(CFLAGS) sample.i.c
        mv sample.i.o sample.o

MS-DOS and OS/2:

    sample.obj:sample.c
        cl /P $(CFLAGS) sample.c
        tp-ic -m6 sample.i
        rename sample.ic temp.c
        cl /c $(CFLAGS) temp.c
        rename temp.obj sample.obj
```

The other situation is where the compiler is not explicitly mentioned, but given as a “built-in" rule. The user can add the following “built-in" rule:

```
UNIX:

    .c.o:
        cc -P $(CFLAGS) $*.c
        tp-ic $*.i
        cc -c $(CFLAGS) $*.i.c
        mv $*.i.o $*.o

MS-DOS and OS/2:

    .c.obj:
        cl /P $(CFLAGS) $*.c
        tp-ic -m6 $*.i
        rename $*.ic temp.c
```

```
cl /c $(CFLAGS) temp.c
rename temp.obj $*.obj
```

The other change necessary is to add SR runtime modules to the link statement. (More on this in the next chapter.)

2.3.1 Example 'make' Files

This section gives on the following pages several examples of how to create 'make' files that work under MS-DOS and UNIX environments.

The first example 'make' file is an illustrative MS-DOS type 'make' file that is unmodified.

```
######################################################################
##
##   S A M P L E    M A K E    F I L E
##   ----W I T H O U T I N S T R U M E N T A T I O N----------
##
##
##  DOS version make script for SAMPLE
##
######################################################################
#
OBJS = sample.obj sampley.obj samplel.obj tree.obj init.obj \
error.obj dotest.obj help.obj log.obj ui.obj premain.obj license.obj \
pretree.obj preprocl.obj preprocy.obj

CFLAGS = /c /FPi /AL /DMSDOS /DLIMITED
LFLAGS = /STACK:20000
sample.obj: sample.c
sampley.obj: sampley.c
samplel.obj: samplel.c
tree.obj: tree.c
license.obj: license.c
init.obj: init.c
error.obj: error.c
dotest.obj: dotest.c
help.obj: help.c
log.obj: log.c
ui.obj: ui.c
premain.obj: premain.c
pretree.obj: pretree.c
preprocl.obj: preprocl.c
preprocy.obj: preprocy.c
sample.exe: $(OBJS)
        sample.obj license.obj help.obj \
    sampley.obj samplel.obj tree.obj init.obj \
    error.obj dotest.obj log.obj ui.obj premain.obj \
    pretree.obj  preprocy.obj preprocl.obj\
    link @sample.lnk;
```

The file below shows the modifications to the ‘make' file needed to provide for automatic instrumentation. The modifications are in bold face.

```
######################################################################
##
## S A M P L E    M A K E    F I L E
##
## -----------W I T H I N S T R U M E N T A T I O N------------
##
##
## DOS version make script for SAMPLE file
##
######################################################################

OBJS = sample.obj sampley.obj samplel.obj tree.obj init.obj \
error.obj dotest.obj help.obj log.obj ui.obj premain.obj license.obj\
pretree.obj preprocl.obj preprocy.obj

CFLAGS = /c /FPi /AL /DMSDOS /DLIMITED
LFLAGS = /STACK:20000

.c.obj:
    cl $(CFLAGS) /P $*.c
    tp-ic -m6 $*.i
    rename $*.ic temp.c
    cl $(CFLAGS) /c temp.c
    rename temp.obj $*.obj
sample.obj: sample.c
sampley.obj: sampley.c
samplel.obj: samplel.c
tree.obj: tree.c
license.obj: license.c
init.obj: init.c
error.obj: error.c
dotest.obj: dotest.c
help.obj: help.c
log.obj: log.c
ui.obj: ui.c
premain.obj: premain.c
pretree.obj: pretree.c
preprocl.obj: preprocl.c
preprocy.obj: preprocy.c
sample.exe: $(OBJS)
    sample.obj license.obj help.obj \
    sampley.obj samplel.obj tree.obj init.obj \
    error.obj dotest.obj log.obj ui.obj premain.obj \
    pretree.obj preprocy.obj preprocl.obj \fBctrun1l.obj\
    link @sample.lnk;
```

The ‘make' file below shows a typical UNIX/XENIX ‘make' file before modification.

```
########################################################################
##
##  S A M P L E    M A K E    F I L E
##
##  Make file example, no instrumentation.
##
##  UNIX, XENIX
##
########################################################################
# Uses make's knowledge of lex, yacc, cc.
########################################################################
CCextras =
CFLAGS = -s ${CCextras} -DXENIX
YFLAGS = -d
LDFLAGS = -i -ly -ll
LFLAGS = -v
Lextras =
Objects = sample.o sampley.o samplel.o tree.o init.o error.o dotest.o
log.o \
    ui.o premain.o preprocy.o preprocl.o pretree.o help.o license.o
Sources = sample.c sampley.c samplel.c tree.c init.c error.c dotest.c
log.c \
    ui.c premain.c preprocy.c preprocl.c pretree.c sample.h \
    typedef.h error.h y.tab.h preproc.h help.c license.c license.h
# UNIX version.  Compiles and links.
sample:  $(Objects)
  rm -f sample
  cc $(Objects) $(LDFLAGS) $(Lextras) -o sample
#
sampley.c: sampley.y
  yacc $(YFLAGS) sampley.y
  mv y.tab.c  sampley.c
  cp y.tab.h  ytab.h
#
samplel.c: samplel.l
  lex    $(LFLAGS) samplel.l
  mv lex.yy.c samplel.c
#
preprocy.c: preprocy.y
  yacc $(YFLAGS) preprocy.y
  cat y.tab.c | sed -e 's/yy/xx/g' >  preprocy.c
  cat y.tab.h | sed -e 's/yy/xx/g' >  pretab.h
  rm y.tab.c
#
preprocl.c: preprocl.l
  lex    $(LFLAGS) preprocl.l
  cat lex.yy.c | sed -e 's/yy/xx/g' >  preprocl.c
  rm lex.yy.c
lpr:
  pr $(Sources) | lpr

license.o: license.c license.h
```

The changes needed have been made in the modified ‘make' file shown below. The modifications are shown in bold face.

```
######################################################################
##
##  S A M P L E    M A K E    F I L E
##
##  Make file sample, with TCAT-PATH instrumentation
##
##  UNIX, XENIX
######################################################################
# Uses make's knowledge of lex, yacc, cc.
######################################################################
CCextras =
CFLAGS = -s ${CCextras} -DXENIX
YFLAGS = -d
LDFLAGS = -i -ly -ll
LFLAGS = -v
Lextras =
Objects = sample.o sampley.o samplel.o tree.o init.o error.o dotest.o
log.o \
    ui.o premain.o preprocy.o preprocl.o pretree.o help.o license.o
Sources = sample.c sampley.c samplel.c tree.c init.c error.c dotest.c
log.c \
    ui.c premain.c preprocy.c preprocl.c pretree.c sample.h typedef.h
error.h \
    y.tab.h preproc.h help.c license.c license.h
# UNIX version.  Compiles and links.
\fB .c.o:
    cc -P $ (CFLAGS) $*.c
    tp-ic $*.i
    cc -c $(CFLAGS) $*.i.c.
    mv $*.i.o $*.o
#
sample: $(Objects) ctrun1.o
    rm -f sample
    cc $(Objects) \fBctrun1.o\fP $(LDFLAGS) $(Lextras) -o sample
#
sampley.c: sampley.y
    yacc $(YFLAGS) sampley.y
    mv y.tab.c  sampley.c
    cp y.tab.h  ytab.h
#
samplel.c: samplel.l
  lex    $(LFLAGS) samplel.l
  mv lex.yy.c samplel.c
#
preprocy.c: preprocy.y
  yacc $(YFLAGS) preprocy.y
  cat y.tab.c | sed -e 's/yy/xx/g' >  preprocy.c
  cat y.tab.h | sed -e 's/yy/xx/g' >  pretab.h
  rm y.tab.c
#
preprocl.c: preprocl.l
  lex    $(LFLAGS) preprocl.l
  cat lex.yy.c | sed -e 's/yy/xx/g' >  preprocl.c
  rm lex.yy.c
lpr:
  pr $(Sources) | lpr
license.o: license.c license.h
```

2.4 File Summary

This section describes *TCAT-PATH* file naming conventions for the instrumentor (tp-ic).

MS-DOS or OS/2:

```
tp-i<lang> [optional switches] filename.i
```

Input:

<filename>.i Preprocessed source file

Produces:

<filename>.i<lang> Instrumented source

<filename>.iA Segment and node reference listing

<filename>.iE Error listing

<filename>.iL Segment count for each module

<filename>.iS Instrumentation Statistics

<module name>.digFile(s) containing digraph of the named module(s)

NOTE: Digraph filenames of module names that are more than 8 characters long are truncated to 8 characters.

UNIX:

```
tp-i<lang> [optional switches] filename.i
```

Input:

<filename>.i Preprocessed source file

Produces:

<filename>.i<lang> Instrumented source

<filename>.i.A Preprocessed source file

<filename>.i.E Error listing

<filename>.i.L Segment count for each module

<filename>.i.S Instrumentation Statistics

<module name>.dig

File(s) containing digraph of the named module(s)

2.5 Embedded Systems

An added benefit resulting from *TCAT-PATH*'s software instrumentation strategy is that the tool may be used with embedded systems. Because *TCAT-PATH*'s output is a syntactically correct program, the tool can be used on programs that are cross-compiled for target systems. The sequence of steps are: the instrumented code is cross-compiled, linked, then moved to the embedded system.

After execution, coverage data collection occurs on the embedded system, and the trace files are uploaded to the host. The specifics of transferring trace files from the embedded system to the host is dependent on the system in question.

CHAPTER 3

Compiling, Linking and Executing

This chapter explains how to compile, link and execute the instrumented program.

Once instrumentation has been completed, the instrumented version of your program must be compiled and linked with the runtime object modules, sometimes called runtime routines.

The runtime routines are supplied by SR and will write to the trace file. These modules are called from the instrumented code; the added function calls, or “probes", call sub-functions inside the runtime modules.

There are several runtime objects for each computer as described in the next section.

NOTE: Some unreachable code may occasionally be inserted by the instrumentor.

This may cause warning messages when compiling, but they are not fatal and the compiler should proceed in spite of them.

3.1 Runtime Descriptions

As mentioned above, the test engineer using *TCAT-PATH* has a choice of many runtime routines to change the behavior and performance of the instrumented system under test. Different runtimes may be selected by linking in the appropriate module.

Finally, the user can write his own runtime package if he needs to modify *TCAT-PATH* to a particular situation, since the program that is needed is small. For an embedded system where the target system has particular characteristics, rewriting the runtime is a practical way to adapt *TCAT-PATH*.

TCAT-PATH runtime system is compatible with TCAT runtime system but the TCAT runtime system is not compatible with *TCAT-PATH*. That is, you can use the *TCAT-PATH* system with C1-instrumented programs, but you cannot use TCAT's runtime system for *TCAT-PATH*. There are a variety of runtime modules for each language.

The function of each runtime package is specified by the format of its name as defined following:

```
<language>trun<level>.o   (for UNIX)
```

or

```
<language>trun<level><model>.obj (for DOS)
```

Examples:

ctrun0.o -- C, level 0, UNIX

ftrun1.o -- Fortran 77, level 1, UNIX

ctrun0m.o -- C, level 0, DOS, medium memory model.

Several versions of runtime are available depending on your needs.

3.1.1 <lang>trun0 - Raw Trace File

There is no internal processing or buffering. The trace file is the full, unedited trace of program execution. There is no prompting for trace file name, so the user must indicate the trace file identification at the invocation of the program under test.

3.1.2 <lang>trun1 - Standard Trace File

This is the same as <lang>trun0, but with prompts that ask the user for Test Descriptor and the name of trace file.There is no internal processing or buffering. The trace file is the full, unedited trace of program execution. This is the basic version.

3.1.3 MS-DOS Runtimes

MS-DOS has several runtimes available. You must first determine the memory model you are using for memory management on your system. You will then be able to easily choose from the following list of runtimes for “C" language. The standard runtimes are ctrun1, while the “quiet" runtimes are ctrun0.Microsoft C has five memory models: S for small; M for medium; C for compact; L for large; and H for huge.

Turbo C has six memory models: T for tiny; S for small; M for medium; C for compact; L for large; and H for huge.

The following is a partial list of runtimes for “C" language on MS-DOS, as they appear on the distribution diskette:

```
\RUNTIME\TURBO\STD\CTRUN1C.OBJ
\RUNTIME\TURBO\STD\CTRUN1H.OBJ
\RUNTIME\TURBO\STD\CTRUN1L.OBJ
\RUNTIME\TURBO\STD\CTRUN1M.OBJ
\RUNTIME\TURBO\STD\CTRUN1S.OBJ
\RUNTIME\TURBO\STD\CTRUN1T.OBJ
\RUNTIME\TURBO\QUIET\CTRUN0C.OBJ
\RUNTIME\TURBO\QUIET\CTRUN0H.OBJ
\RUNTIME\TURBO\QUIET\CTRUN0L.OBJ
\RUNTIME\TURBO\QUIET\CTRUN0M.OBJ
\RUNTIME\TURBO\QUIET\CTRUN0S.OBJ
\RUNTIME\TURBO\QUIET\CTRUN0T.OBJ
\RUNTIME\MSC51\STD\CTRUN1C.OBJ
\RUNTIME\MSC51\STD\CTRUN1H.OBJ
\RUNTIME\\MSC51\STD\CTRUN1L.OBJ
\RUNTIME\MSC51\STD\CTRUN1M.OBJ
\RUNTIME\MSC51\STD\CTRUN1S.OBJ
\RUNTIME\MSC51\QUIET\CTRUN0C.OBJ
\RUNTIME\MSC51\QUIET\CTRUN0H.OBJ
\RUNTIME\MSC51\QUIET\CTRUN0L.OBJ
\RUNTIME\MSC51\QUIET\CTRUN0M.OBJ
\RUNTIME\MSC51\QUIET\CTRUN0S.OBJ
```

NOTE: Microsoft C 5.1 runtimes should be compatible with 6.0 updates.

3.1.4 Executing the Instrumented Program

The next step is to run your instrumented program and track which logical paths have been exercised by the test data you supply. In essence, this is a matter of noticing the not-hit paths mentioned in the coverage report (refer to Chapter 6), and looking up the corresponding code in the Reference Listing.

TCAT-PATH senses when paths are hit by monitoring the markers inserted during instrumentation and by accumulating the results in a trace file and matching them with the paths in the path file.

To produce the trace file, first run your instrumented and compiled "C" program and follow the prompts.

If you use the standard runtime routines, the system will respond with:

```
Trace Descriptor:
```

Type in a description of the test run. Be as descriptive as needed for your own information in referring to this test run. You can enter up to 80 characters of text in your message. This message will be recorded in the trace file and used in coverage reports.

If you choose to enter no descriptive text, just press the return key. The system next will prompt you for an output filename:

```
Name of tracefile [default is Trace.trc]:
```

Type in any name. The system will create a trace file with the name you enter. To use the default name Trace.trc, just press the return key. The trace file description and name are useful in keeping track of different test runs. Consistent, clear naming conventions are useful in organizing different groups of results.

A common practice is to identify trace files with the filename extension**.trc**.

3.1.5 Performance Considerations

Sometimes, an instrumented program will produce very large trace files. One solution to this is to compile a mixture of instrumented and un-instrumented files so that the program is tested in pieces.

CHAPTER 4

Utilities

This chapter covers the automatic path generation, cyclomatic number calculation, digraph picture generation utilities, “essential" path extractor, and path logical condition extractor.

The first utility generates a complete set of paths for a module, which is used later for coverage reporting along with an execution trace file. The last two utilities are intended for the user to study the structure and properties of the module in question. A cyclomatic complexity number can be computed to identify “too complex" modules. A digraph picture can also be drawn to give the user intuition of the structure of the module.User can also get the “essential" paths, i.e. a minimal subset of paths that will guarantee 100 percent branch level coverage (C1).

Additionally, user can display the logical conditions (or predicates) that need to be satisfied for a given path to be traversed. All utilities use the digraph file produced from the instrumentation as input.

4.1 apg (Automatic Path Generator)

Automatic Path Generator (**apg**) processes a digraph file (***.dig** file) into a path file (***.pth** file). This path information is the input to the coverage analyzer (**ctcover**) which will be discussed in the next chapter.

4.1.1 apg

This program uses a SR-proprietary algorithm that generates sets of equivalence classes of paths. The path classes are either non-iterative or iterative. The output describes iteration in terms of “loop" or “cycles" that can be entered, and then exited (see Chapter 14).

apg uses the notation <..> for 0 and [..] for 1 or more repetitions; **apg** also uses the {..} notation for groups of edges.

apg issues error messages if it is asked to generate paths beyond a maximum path count (the user can modify these values); see below.

```
apg name [-b] [-c] [-g] [-m loop] [-n] [p 2limit][-q]
[-S] [-w width]
```

where,

name — This is the file basename that represents the module to be processed. **<name>.dig** is assumed to exist and to contain a digraph in proper format.

If the named file has a **suffix .dig** then this suffix is stripped off and the resulting name is used as the basename (see the **-g** switch).

[**-b**] — Basis paths only switch. If this switch is present, then apg computes all paths but outputs only those paths which have no iteration.

NOTE: A program must have at least one basis path; otherwise, there is something wrong with the digraph.

[**-c**] — Count paths only switch. If present, apg computes the total number of paths (regardless of the value set by the **-p** switch) found. If the path count is large, apg prints out intermediate messages so that you don't think it is failing. (The intermediate messages happen every 1000 paths). CAUTION: If the path counts is over 100,000 you should be prepared for a long wait.

[**-g**] — Output redirection switch. The output of **apg** normally goes to standard output; if the **-g** flag is present, then the output is written to name.pth.

[**-m** loop]	This is the maximum loop count for the internal path generation process. The default value is loop = 1. This switch overrides values assigned in the configuration file. Note: currently only loop = 1 is supported. This means that all iterations are grouped at exactly one level above the base-path level.
[**-n**]	Each path is preceded by a path number. For example, @2 : 1 3 4 <{4}> 5 6. The number between the @ and: is the path number.
[**-p** limit]	This is the integer maximum number of paths to generate. If the total number of paths to be emitted is limit, then the total number of paths calculated is 16 * limit. The default value is limit = 300.
[**-q**]	The quiet switch will suppress version number and other extraneous outputs.
[**-S**]	If present, after the path computation, **apg** prints a series of statistics that characterize the set of paths. No paths are generated. Path statistics are output to standard output. If the **-g** switch is on, statistics are returned to the **name.stt** file, where name is the module name. A sample is shown below:

[**-w** *width*] The output of **apg** is "folded" - with \\'s protecting the new-line characters - so that it is never wider than width characters. The default value for width (i.e. without the **-w** switch) is 72.

```
apg, Release 3

Path Analysis Statistics.

File name:                testfile.dig

Number of nodes:   16
Number of edges:   20
Cyclomatic number (E - N + 2):12

Number of paths:   236
Average path length (segments):22.45
Minimum length path (segments):12   (Path 23)
Maximum length path (segments):45   (Path 464)
Most iteration groups:4    (Path 14)

Path count by iteration groups:
    0 iteration group(s):4
    1 iteration group(s):66
    2 iteration group(s):14
    3 iteration group(s):0
    4 iteration group(s):0
    5 iteration group(s):16
```

4.1.1.1 Other notes:

There is a supplied script, **DoPTH** that reads the basename of the module, calls **apg**, and writes the ***.pth** file for every ***.dig** file in the current directory.

Sample Output:

Here is a sample input ***.dig** file and the corresponding output ***.pth** file:

Example 1:

```
.Digraph file...

0 1 1
1 4 2
1 2 3
2 2 4
2 3 5
3 4 6
3 4 7

Paths generated...

1 2
1 3 4 <{4}> 5 6
1 3 4 <{4}> 5 7
1 3 5 6
1 3 5 7
```

Example 2:

```
Digraph file...

1 2 1
2 3 2
3 7 3
3 4 4
4 7 5
4 7 6
2 5 7
5 6 8
6 7 9
6 7 10
5 7 11


Paths generated...

1 2 3
1 2 4 5
```

```
1 2 4 6
1 7 8 9
1 7 8 10
1 7 11
```

4.1.2 Processing Program Subgraphs with ‘apg'

In complex cases the *TCAT-PATH* user may wish to declare a subgraph of the original program as one which is to be treated as a “unit” in relation to processing of the larger graph. Doing this will, in many cases, decrease the number of paths generated to a more manageable number. (This is often called “path factoring".)

The following figure shows how **apg** handles one or more sub-digraphs within the specified graph:

```
apg -s filename
```

or

```
apg -s ‘hereis.filename‘
```

or

```
apg -s filename -s filename -s filename
```

(maximum of 16 such filenames)

where in each case the “filename" is another digraph (in the *TCAT-PATH* standard format) where the first appearing node is the assumed entry, and which can have any number of exits.

When **apg** encounters that entry node then it treats **ALL** of the nodes in the named subgraph files as a **SINGLE SEGMENT**, labeled by the name of the filename.This means that the GROUP of edges named in the *-s* file acts like just one edge in regard to path generation. When this option is used, an **apg** output path might look like the following:

```
. . .
. . .

2 5 14 <{ 16 "filename1" 29 30 33 }> \\
        44 49 50 51
. . .
. . .
```

More about **apg** processing of subgraphs is found in Chapter 14.

NOTE: apg processing of subgraphs may not be available in early versions of *TCAT-PATH*.

4.1.3 Blocked Pairs Processing with 'apg'

When the number of paths that apg generates grows very large, it may be desirable to prevent generation of some paths. Note that the user always has the option of editing the ***.pth** file to remove paths. There is, however, one feature of apg which can simplify what would otherwise be complicated editing sessions.

The special flag **-b** can be used as follows to inform **apg** to not include pairs of segments in any path. Here is a typical invocation of apg in this case:

```
apg  -b  filename
```

where filename is the name of the file in the working directory that contains a list of pairs of segments that should not be included (i.e., blocked pairs).

The format for the blocked pair file is as follows:

```
# This is a sample 'blocked pair' file
# for use with TCAT-PATH...

segment-1  segment-2
segment-a  segment-b
segment-x  segment-y
...
```

which means that the indicated pairs are to be used to "block" generation of a path.

The user should be cautious with this capability, however. If critical pairs are blocked, then apg may generate no paths. Generally, one must ascertain from studying the program that two segments cannot co-exist in any possible actual execution path before adding them to the file of blocked names.

4.2 cyclo (Cyclomatic Number Calculation)

The **cyclo** command is a utility that computes the cyclomatic complexity, sometimes referred to as the McCabe Metric, for the named digraph file.

The cyclomatic complexity is a characterization of the relative complexity of a digraph based on a specific count of the edges and nodes. The formula for the **cyclomatic complexity** is (this is how the output appears):

```
Cyclomatic Complexity
                    =  McCabe Metric
                    =  E(n)
                    =  edge - node + 2
                    =  <value> \f1
```

This metric is commonly used to assess the complexity of a module. If E(n) is over 10, then the module is normally considered ‘‘too complex''. However, in some cases E(n) >> 10 for] ‘‘easy to test modules'', and E(n) is very small for ‘‘hard to test modules''. User caution is advised.

Syntax:

```
cyclo   name.dig  [-q]
```

where,

name.dig is the name of the digraph file for which the cyclomatic number is to be computed. The file is assumed to be in standard digraph format.

[**-q**] This switch is used to quiet down the output produced to just the character string (without carriage return or newline) representing the computed cyclomatic number.

This switch allows the output of cyclo to be combined in expressions. For example, on UNIX systems one could use the command fragment:

```
expr ‘cyclo -q file1‘  + ‘cyclo -q file2‘
```

Note: There is a supplied script, **DoCYC** that calculates the cyclomatic number for each ***.dig** file in the current directory.

4.3 digpic Digraph Display (Digraph Picture)

This command displays a digraph in a visible format; it is useful for manual checking and analysis of generated digraphs.

Syntax:

```
digpic name.dig [-B filename] [-C value] [-R value]
[-W value]
```

where,

name.dig — The file **name.dig** that contains the digraph to be displayed. The format for a digraph file is described below; this is the output of **`tp-i<lang>`**.

Note that in this case the filename must be used explicitly; no automatic extension to the basename is performed.

[**-B** *filename*] — Defines the basis path to be used to draw the picture. The default is the "natural" basis path, computed by using the set of supplied node names in their natural order of first occurrence. This normally produces a fairly well-organized and complete picture.

A source of possible basis paths is the output of apg; one of these paths could be used as the basis for drawing pictures of the digraph.

[**-C** *value*] — The value to use as the centerline of the digraph picture. The default value is 0, which tells digpic to use the left-adjusted version of the picture.

[**-R** *value*] — The number of rows to leave between node names. The default value is 1.

[-W value]

The maximum width of the picture; digpic issues an error message if the maximum value is exceeded. The default value for the width is 80; the maximum assignable value is 128. A picture cannot be constructed if the maximum width is less than the longest node name.

Note: In some cases digpic will give a misleading “picture" of the corresponding program. One case is when there are multiple switch blocks that do not each contain a break. Also, because conditional expressions are not supported constructs, any programs that contain instances of them may also produce misleading digpic results.

Other Notes: There is a supplied script, DoPIC that generates a ***.pic** file for each ***.dig** in the current directory.

4.3.1 Sample Outputs:

digpic generates a representation of the structure of the program. The format generated by digpic varies with the available output media. For character based systems the format appears as shown below.

The input digraph file (format "node1 node2 edge"; node and edge names are strings of 16 bytes each or less):

```
0 1 Enter
1 2 A
2 Exit B
2 1 C
2 Exit D
The corresponding path set (as generated by apg):
Enter A B
 Enter A C <{ A C }> D
 Enter A C <{ A C }> B
 Enter A D

-------------------------------------------------------
Total of 4 paths for this function.
Here is a sample output from digpic:

                  [[0    ]] 0       - Enter
                  [[     ]] |
                > [[1    ]] <  0    - A
                | [[     ]]    |
                0 [[2    ]] 0  <  0 - C B D
                  [[     ]] |     |
                  [[Exit ]] <     <
Here is a more complex example digraph:
                Node-9     Node-11   E1
                Node-11    Node-12   E2
                Node-11    Node-12   E3
                Node-12    Node-10   E4
                Node-12    Node-13   E5
                Node-13    Node-14   E6
                Node-14    Node-15   E7
                Node-15    Node-16   E8
                Node-16    Node-17   E9
                Node-17    Node-15   E10
                Node-17    Node-15   E11
                Node-16    Node-15   E12
                Node-15    Node-18   E13
                Node-18    Node-19   E14
                Node-19    Node-19   E15
                Node-19    Node-10   E16
                Node-18    Node-10   E17
                Node-14    Node-10   E18
                Node-13    Node-10   E19
```

The path set for this graph is as follows:

```
E1 E2 E4
E1 E2 E5 E6 E7 E8 E9 E10 <{E8 E9 E10 E11 E12 }> E13 E14
    E15 <{E15 }> E16
E1 E2 E5 E6 E7 E8 E9 E10 <{E8 E9 E10 E11 E12 }> E13 E14
    E16
E1 E2 E5 E6 E7 E8 E9 E10 <{E8 E9 E10 E11 E12 }> E13 E17
E1 E2 E5 E6 E7 E8 E9 E11 <{E8 E9 E10 E11 E12 }> E13 E14
    E15 <{E15 }> E16
E1 E2 E5 E6 E7 E8 E9 E11 <{E8 E9 E10 E11 E12 }> E13 E14
    E16
E1 E2 E5 E6 E7 E8 E9 E11 <{E8 E9 E10 E11 E12 }> E13 E17
E1 E2 E5 E6 E7 E8 E12 <{E8 E9 E10 E11 E12 }> E13 E14 E15
    <{E15 }> E16
E1 E2 E5 E6 E7 E8 E12 <{E8 E9 E10 E11 E12 }> E13 E14 E16

   ...(some paths have been eliminated for clarity)...

E1 E3 E5 E6 E7 E8 E12 <{E8 E9 E10 E11 E12 }> E13 E17
E1 E3 E5 E6 E7 E13 E14 E15 <{E15 }> E16
E1 E3 E5 E6 E7 E13 E14 E16
E1 E3 E5 E6 E7 E13 E17
E1 E3 E5 E6 E18
E1 E3 E5 E19

    ----------------------------------------------------
    Total of 30 paths for this function.
```

The digpic-produced picture is as follows. The S symbol is used for a “self-loop".

```
Digpic

          Node-9  0       E1
                  |
          Node-11 < 0 0   E2 E3
                    | |
          Node-12 0 < < 0 E4 E5
                  |     |
  > > > > Node-10 <     |
  | | | |               |
  0 | | | Node-13 0     < E19 E6
    | | |         |
    0 | | Node-14 < 0     E18 E7
      | |           |
> > > | | Node-15 0 < 0   E8 E13
| | | | |         |   |
0 | | | | Node-16 < 0 |   E12 E9
  | | | |           | |
  0 0 | | Node-17   < |   E10 E11
      | |             |
```

```
0 | Node-18 0   <   E17 E14
  |         |
S 0 Node-19 <       E15 E16
```

4.4 pathcon Utility

The purpose of the **pathcon** utility is to extract and display the logical conditions (predicates) for a particular path given the sequence of segments in the path (which could be a complete path), the digraph file (***.dig** file), and the reference listing file (***.i.A** or ***.iA file**).

4.4.1 Invocation Syntax

Syntax:

```
pathcon      -A ref-listing   -D dig-file   [-g]
[-P path-file]   [-N number [number]]
```

where,

-A ref-listing ref-listing, produced by the **tp-ic** instrumentor, is used for predicate referencing. This file has **.i.A** or **.iA** extension.

-D dig-file **dig-file** is a digraph file for a module that specifies the set of segments in "tail-node head-node segment-name" format.

This file is produced by the tp-ic instrumentor and is normally named **module-name.dig**, where **module-name** is the module in question.

-g **pathcon** output normally goes to standard output. If this option is specified, the output goes to a file named **module-name.con**.

-N *number* *number* specifies the path number which logical conditions are to be displayed. The path number is relative to the beginning of the path file. The user can specify one or more path numbers by supplying the numbers as part of arguments to **pathcon**. If this option is not specified, then pathcon will display all the paths in the specified path file.

-P *path-file* *path-file* is a file that contains a set of paths from the module-name module. If this option is not specified, then pathcon will get the paths from the module-name.pth file. This file is normally produced by the apg utility of *TCAT-PATH*. The file can contain all or a subset of the paths that **apg** generates.

4.4.2 Example Invocation

For example, using the example restaurant program in the full *TCAT-PATH* example chapter, the command:

```
pathcon -D main.dig -A example.i.A
```

would instruct pathcon to generate the set of logical conditions for each generated path in the main.pth file using the information in the main.dig digraph file and example.i.A reference listing file. The output will go to standard output.

The following command:

```
pathcon -D proc_input.dig -A example.i.A -g -N 3 169
```

would instruct pathcon to generate the set of logical conditions for paths number 3 and 169 in the **proc_input.pth** file using the information in the **proc_input.dig** file and the same reference listing as previously. The output will go to the file named **proc_input.con**.

4.4.3 Output format

pathcon gives a detailed output for each path requested. Each path is printed, along with the path number relative to the beginning of the path file. Segments in the path are listed in rows. Segments that are inside the <{ ... }> iteration symbols are not included, however, segments that are inside the [{ ... }] iteration symbols are included. The latter indicates 1 or more iterations of the loop and thus need to be included in the output.

The output format is shown below. Entries in italics are the entries that pathcon generates. Each segment occupies a row and has the following information:

```
PATH #: path-string
-----------------------------------------------------------------
#      Segment    Cycle   Sense   Predicate
-----------------------------------------------------------------
22    2       Entry   TRUE    while(isspace(in_str[char_index]))
```

"#" Indicates the number of lines in the report. Each line corresponds to one segment, however, a segment may be listed more than once if it is part of a 1 or more iteration loop.

Segment Lists the segment name as in the digraph file.

Cycle Includes: Entry, Exit, Loop, AbExit, Ex/Ent.Indicates which part of the loop this segment belongs to. If this entry is left blank, it indicates that the current segment does not belong to a loop.

entry The current segment is hit before the loop is executed.

Exit The current segment is hit after the loop is exited.

Loop The current segment is inside the loop.

AbExit The current segment is hit when loop is exited "abnormally". This is the case when the segment comes after a loop with one (1) or more iterations (the [{ ... }]symbols).

Ex/Ent This segment comes between two loops (e.g. segment 3 is such a loop in the following path: 1 2 <{ 2 }> 3 <{ 4 5 }> 7)

Sense	Includes: TRUE, FALSE, CASE. Indicates which sense of evaluation of the predicate that will cause this segment to be hit. TRUE The current segment is hit if the evaluation of the predicate is TRUE. FALSE The current segment is hit if the evaluation of the predicate is FALSE. CASE The current segment is hit if the evaluation of the **switch** statement is as indicated in the predicate (for “C" language only).
Predicate	Includes: NONE, ***, and predicate string. The logical condition(s) that need to evaluated if the current segment is to be hit. Predicate for the first segment in a module is indicated by the string NONE. If no predicate is encountered, pathcon will output ***.

4.4.4 Example Output

The output from the second example invocation from the previous section is shown in the following figure.

```
PATH 3: 1 2 <{ 2 }> 3 4 5 6 7 8 9 10 11 12 13 14 [{ 4 5 6 7 8 9 10 11 12 13 14
        6 7 8 9 10 11 12 13 14 }] 15 17 18 20 21 <{ 20 21 22 }> 23
------------------------------------------------------------------------------
#    Segment    Cycle   Sense   Predicate
------------------------------------------------------------------------------
1       1               TRUE    NONE
2       2       Entry   TRUE    while(isspace(in_str[char_index]))
3       3       Exit    FALSE   while(isspace(in_str[char_index]))
4       4               TRUE    for( ; char_index <= strlen(in_str);
char_index++) {
5       5               CASE    switch(in_str[char_index]) {
6       6               TRUE    case '1':
7       7               TRUE    case '2':
8       8               TRUE    case '3':
9       9               TRUE    case '4':
10      10              TRUE    case '5':
11      11              TRUE    case '6':
12      12              TRUE    case '7':
13      13              TRUE    case '8':
14      14      Entry   TRUE    case '9':
15      4       Loop    TRUE    for( ; char_index <= strlen(in_str);
char_index++) {
16      5       Loop    CASE    switch(in_str[char_index]) {
17      6       Loop    TRUE    case '1':
18      7       Loop    TRUE    case '2':
19      8       Loop    TRUE    case '3':
20      9       Loop    TRUE    case '4':
21      10      Loop    TRUE    case '5':
22      11      Loop    TRUE    case '6':
23      12      Loop    TRUE    case '7':
24      13      Loop    TRUE    case '8':
25      14      Loop    TRUE    case '9':
26      6       Loop    TRUE    case '1':
27      7       Loop    TRUE    case '2':
28      8       Loop    TRUE    case '3':
29      9       Loop    TRUE    case '4':
30      10      Loop    TRUE    case '5':
31      11      Loop    TRUE    case '6':
32      12      Loop    TRUE    case '7':
33      13      Loop    TRUE    case '8':
34      14      Loop    TRUE    case '9':
35      15      AbExit  CASE    switch(in_str[char_index]) {
36      17              FALSE   if(chk_char(in_str[char_index])) {
37      18              TRUE    if(char_index > 0 && got_first)
38      20              TRUE    while(char_index <= strlen(in_str)) {
39      21      Entry   TRUE    if(chk_char(in_str[char_index]))
40      23      Exit    FALSE   while(char_index <= strlen(in_str)) {
==============================================================================
```

```
PATH 169: 1 3 4 15 16
------------------------------------------------------------------------
#    Segment    Cycle   Sense   Predicate
------------------------------------------------------------------------
1       1               TRUE    NONE
2       3               FALSE   while(isspace(in_str[char_index]))
3       4               TRUE    for( ; char_index <= strlen(in_str);
char_index++) {
4       15              CASE    switch(in_str[char_index]) {
5       16              TRUE    if(chk_char(in_str[char_index])) {
========================================================================
```

4.5 pathcover Utility

The purpose of the **pathcover** utility is to extract path and segment information from a set of paths supplied in the input file. **pathcover** allows the user to get “essential" paths, i.e. a minimal subset of paths from the input file that would guarantee 100% C1 (branch or segment) level coverage.

It is assumed that the input file supplied to pathcover is the path file produced by the **apg** (all path generator) utility of *TCAT-PATH*.

pathcover will gives several sets of “essential" paths depending on user's options. The user can get “essential" paths based on the order of occurrence of the segments in the original path file (“first" or “last" instance found algorithm), or the user can rearrange the path file in certain order and then apply the search algorithm. Finally, the user can also get information on which segments are encountered most often in the input file.

4.5.1 Invocation Syntax

Syntax:

```
pathcover path-file [-c] [-f] [-fi] [-fl] [-fs]
    [-g] [-l] [-li] [-ll] [-ls] [-n] [-q] [-r]
```

where,

path-file A file that contains a set of paths for a particular module. This file is normally produced by the apg utility and named **module-name.pth**.

-c Prints out the population statistics on each segment encountered in the path file. It reports on the number of paths that contain a particular segment.

-f Prints out the “essential" paths based on the "first instance found" algorithm. The search is done on the original input path set (input file). This is pathcover output if no options are specified.

-fi Prints out the “essential" paths based on the “first instance found" algorithm. The search is done on the paths sorted by iteration. The sorted paths are obtained by ordering the non-iterative paths first and then the iterative paths.

Iteration is indicated by the <{ ... }> (0 or more iterations) and the [{ ...}] (1 or more iterations) symbols.

-fl Prints out the “essential" paths based on the “first instance found" algorithm. The search is done on the paths sorted by the length (segment counts) of the paths. The sorted paths are obtained by ordering the paths in ascending order based on the segment counts in the path. Segments that are inside the <{ ...}> (0 or more) iteration symbol are excluded from the segment counts.

-fs Prints out the “essential" paths based on the “first instance found" algorithm.The search is done on the paths sorted by the segment. The sorted paths are obtained by ordering the paths in ascending lexicographic order.

-g Prints pathcover output to a file called *module-name.cov*, where module-name is the particular module in question. pathcover output normally goes to standard output.

-l	Prints out the “essential" paths based on the “last instance found" algorithm. The search is done on the original input path set (input file).
-li	Prints out the “essential" paths based on the “last instance found" algorithm. The search is done on the paths sorted by iteration. The sorted paths are obtained by ordering the non-iterative paths first and then the iterative paths. Iteration is indicated by the <{ ... }> (0 or more iterations) and the [{ ...}]
-ll	Prints out the “essential" paths based on the “last instance found" algorithm. The search is done on the paths sorted by the length (segment counts) of the paths. The sorted paths are obtained by ordering the paths in ascending order based on the segment counts in the path. Segments that are inside the <{ ...}>\(0 or more) iteration symbol are excluded from the segment counts.
-ls	Prints out the “essential" paths based on the “last instance found" algorithm. The search is done on the paths sorted by the segment. The sorted paths are obtained by ordering the paths in ascending lexicographic order.
-n	Each path is preceded by a path number. For example,@2 : 1 3 4 <{ 4 }> 5 6. The number between the @ and : is the path number.
-q	The quiet switch will suppress version number and other extraneous outputs.
-r	Prints out the “essential" paths randomly. First and last algorithms are ignored.

4.5.2 Example Invocation

For example, using the same example “restaurant" program, the command sequence:

```
cc -P example.c
tp-ic example.i
apg main.dig -g
pathcover main.pth -c -f -l -g
```

would instruct pathcover to generate a report on the segment population statistics and two sets of “essential" paths for the main module in example.c file: one with “first instance found” algorithm and another with “last instance found" algorithm.

The output is written to a file called **main.cov**, and it is shown in the next figure.

```
pathcover -- Path Coverage Utility. [Release 1.1 -- 6/91]
(c) Copyright 1991 by Software Research, Inc.

Selected PATHCOVER Options:

[-c]   Population Statistics    -- YES
[-f]   First Found              -- YES
[-l]   Last Found               -- YES
[-fi]  First Found (Iteration)  -- NO
[-li]  Last Found  (Iteration)  -- NO
[-fl]  First Found (Length)     -- NO
[-ll]  Last Found  (Length)     -- NO
[-fs]  First Found (Segment)    -- NO
[-ls]  Last Found  (Segment)    -- NO


pathcover:  POPULATION STATISTICS BY SEGMENT
Module:: “main”         Option:: “-c”
------------------------------------------------------------------------
Segment     # of paths
------------------------------------------------------------------------
  1            155
  2            154
  3            77
  4            154
  5            140
  6            14
  7            14
  8            14
  9            14
  10           14
  11           14
  12           14
  13           14
  14           28
  15           14
  16           14
```

```
  17          154
  18          132
  19          110
  20          22
  21          44
  22          22
  23          44
  24          22
  25          132
  26          154
  27          155
pathcover:  FIRST INSTANCE FOUND BY SEGMENT
Module:: "main"         Option:: "-f"
-------------------------------------------------------------------------
#       Path#   Path
-------------------------------------------------------------------------
1      1      1 2 3 <{ 3 }> 4 5 6 <{ 5 6 7 8 9 10 11 12 13 14 15 16 }> 17 18
              19 20 21 <{ 19 20 21 21 22 23 23 24 }> 25 <{ 18 24 23 23 22 21
               21 20 19 25 }> 26 <{ 2 3 4 16 15 14 13 12 11 10 9 8 7 6 5 17
                  25 19 20 21 21 22 23 23 24 18 26 }> 27
2      3      1 2 3 <{ 3 }> 4 5 6 <{ 5 6 7 8 9 10 11 12 13 14 15 16 }> 17 18
              19 22 23 <{ 19 20 21 21 22 23 23 24 }> 25 <{ 18 24 23 23 22 21
               21 20 19 25 }> 26 <{ 2 3 4 16 15 14 13 12 11 10 9 8 7 6 5 17
                  25 19 20 21 21 22 23 23 24 18 26 }> 27
3      5      1 2 3 <{ 3 }> 4 5 6 <{ 5 6 7 8 9 10 11 12 13 14 15 16 }> 17 18
              19 24 <{ 19 20 21 21 22 23 23 24 }> 25 <{ 18 24 23 23 22 21 21
               20 19 25 }> 26 <{ 2 3 4 16 15 14 13 12 11 10 9 8 7 6 5 17 25
                  19 20 21 21 22 23 23 24 18 26 }> 27
4      8      1 2 3 <{ 3 }> 4 5 7 <{ 5 6 7 8 9 10 11 12 13 14 15 16 }> 17 18
              19 20 21 <{ 19 20 21 21 22 23 23 24 }> 25 <{ 18 24 23 23 22 21
               21 20 19 25 }> 26 <{ 2 3 4 16 15 14 13 12 11 10 9 8 7 6 5 17
                  25 19 20 21 21 22 23 23 24 18 26 }> 27
5      15     1 2 3 <{ 3 }> 4 5 8 <{ 5 6 7 8 9 10 11 12 13 14 15 16 }> 17 18
              19 20 21 <{ 19 20 21 21 22 23 23 24 }> 25 <{ 18 24 23 23 22 21
               21 20 19 25 }> 26 <{ 2 3 4 16 15 14 13 12 11 10 9 8 7 6 5 17
                  25 19 20 21 21 22 23 23 24 18 26 }> 27
6      22     1 2 3 <{ 3 }> 4 5 9 <{ 5 6 7 8 9 10 11 12 13 14 15 16 }> 17 18
              19 20 21 <{ 19 20 21 21 22 23 23 24 }> 25 <{ 18 24 23 23 22 21
               21 20 19 25 }> 26 <{ 2 3 4 16 15 14 13 12 11 10 9 8 7 6 5 17
                  25 19 20 21 21 22 23 23 24 18 26 }> 27
7      29     1 2 3 <{ 3 }> 4 5 10 <{ 5 6 7 8 9 10 11 12 13 14 15 16 }> 17
              18 19 20 21 <{ 19 20 21 21 22 23 23 24 }> 25 <{ 18 24 23 23 22
               21 21 20 19 25 }> 26 <{ 2 3 4 16 15 14 13 12 11 10 9 8 7 6 5
                  17 25 19 20 21 21 22 23 23 24 18 26 }> 27
8      36     1 2 3 <{ 3 }> 4 5 11 <{ 5 6 7 8 9 10 11 12 13 14 15 16 }> 17
              18 19 20 21 <{ 19 20 21 21 22 23 23 24 }> 25 <{ 18 24 23 23 22
               21 21 20 19 25 }> 26 <{ 2 3 4 16 15 14 13 12 11 10 9 8 7 6 5
                  17 25 19 20 21 21 22 23 23 24 18 26 }> 27
9      43     1 2 3 <{ 3 }> 4 5 12 <{ 5 6 7 8 9 10 11 12 13 14 15 16 }> 17
              18 19 20 21 <{ 19 20 21 21 22 23 23 24 }> 25 <{ 18 24 23 23 22
               21 21 20 19 25 }> 26 <{ 2 3 4 16 15 14 13 12 11 10 9 8 7 6 5
                  17 25 19 20 21 21 22 23 23 24 18 26 }> 27
10     50     1 2 3 <{ 3 }> 4 5 13 <{ 5 6 7 8 9 10 11 12 13 14 15 16 }> 17
              18 19 20 21 <{ 19 20 21 21 22 23 23 24 }> 25 <{ 18 24 23 23 22
```

```
                21 21 20 19 25 }> 26 <{ 2 3 4 16 15 14 13 12 11 10 9 8 7 6 5
                   17 25 19 20 21 21 22 23 23 24 18 26 }> 27
11      57      1 2 3 <{ 3 }> 4 5 14 15 <{ 5 6 7 8 9 10 11 12 13 14 15 16 }>
                17 18 19 20 21 <{ 19 20 21 21 22 23 23 24 }> 25 <{ 18 24 23 23
                22 21 21 20 19 25 }> 26 <{ 2 3 4 16 15 14 13 12 11 10 9 8 7 6
                   5 17 25 19 20 21 21 22 23 23 24 18 26 }> 27
12      64      1 2 3 <{ 3 }> 4 5 14 16 <{ 5 6 7 8 9 10 11 12 13 14 15 16 }>
                17 18 19 20 21 <{ 19 20 21 21 22 23 23 24 }> 25 <{ 18 24 23 23
                22 21 21 20 19 25 }> 26 <{ 2 3 4 16 15 14 13 12 11 10 9 8 7 6
                   5 17 25 19 20 21 21 22 23 23 24 18 26 }> 27
pathcover:  LAST INSTANCE FOUND BY SEGMENT
Module:: "main"         Option:: "-l"
------------------------------------------------------------------------
#       Path#   Path
------------------------------------------------------------------------
1       77      1 2 3 <{ 3 }> 4 17 26 <{ 2 3 4 16 15 14 13 12 11 10 9 8 7 6 5
                   17 25 19 20 21 21 22 23 23 24 18 26 }> 27
2       84      1 2 4 5 6 <{ 5 6 7 8 9 10 11 12 13 14 15 16 }> 17 26 <{ 2 3 4
                16 15 14 13 12 11 10 9 8 7 6 5 17 25 19 20 21 21 22 23 23 24
                   18 26 }> 27
3       91      1 2 4 5 7 <{ 5 6 7 8 9 10 11 12 13 14 15 16 }> 17 26 <{ 2 3 4
                16 15 14 13 12 11 10 9 8 7 6 5 17 25 19 20 21 21 22 23 23 24
                   18 26 }> 27
4       98      1 2 4 5 8 <{ 5 6 7 8 9 10 11 12 13 14 15 16 }> 17 26 <{ 2 3 4
                16 15 14 13 12 11 10 9 8 7 6 5 17 25 19 20 21 21 22 23 23 24
                   18 26 }> 27
5       105     1 2 4 5 9 <{ 5 6 7 8 9 10 11 12 13 14 15 16 }> 17 26 <{ 2 3 4
                16 15 14 13 12 11 10 9 8 7 6 5 17 25 19 20 21 21 22 23 23 24
                   18 26 }> 27
6       112     1 2 4 5 10 <{ 5 6 7 8 9 10 11 12 13 14 15 16 }> 17 26 <{ 2 3 4
                16 15 14 13 12 11 10 9 8 7 6 5 17 25 19 20 21 21 22 23 23 24
                   18 26 }> 27
7       119     1 2 4 5 11 <{ 5 6 7 8 9 10 11 12 13 14 15 16 }> 17 26 <{ 2 3 4
                16 15 14 13 12 11 10 9 8 7 6 5 17 25 19 20 21 21 22 23 23 24
                   18 26 }> 27
8       126     1 2 4 5 12 <{ 5 6 7 8 9 10 11 12 13 14 15 16 }> 17 26 <{ 2 3 4
                16 15 14 13 12 11 10 9 8 7 6 5 17 25 19 20 21 21 22 23 23 24
                   18 26 }> 27
9       133     1 2 4 5 13 <{ 5 6 7 8 9 10 11 12 13 14 15 16 }> 17 26 <{ 2 3 4
                16 15 14 13 12 11 10 9 8 7 6 5 17 25 19 20 21 21 22 23 23 24
                   18 26 }> 27
10      140     1 2 4 5 14 15 <{ 5 6 7 8 9 10 11 12 13 14 15 16 }> 17 26 <{ 2
                3 4 16 15 14 13 12 11 10 9 8 7 6 5 17 25 19 20 21 21 22 23 23
                   24 18 26 }> 27
11      147     1 2 4 5 14 16 <{ 5 6 7 8 9 10 11 12 13 14 15 16 }> 17 26 <{ 2
                3 4 16 15 14 13 12 11 10 9 8 7 6 5 17 25 19 20 21 21 22 23 23
                   24 18 26 }> 27
12      148     1 2 4 17 18 19 20 21 <{ 19 20 21 21 22 23 23 24 }> 25 <{ 18 24
                23 23 22 21 21 20 19 25 }> 26 <{ 2 3 4 16 15 14 13 12 11 10 9
                   8 7 6 5 17 25 19 20 21 21 22 23 23 24 18 26 }> 27
13      149     1 2 4 17 18 19 21 <{ 19 20 21 21 22 23 23 24 }> 25 <{ 18 24 23
                23 22 21 21 20 19 25 }> 26 <{ 2 3 4 16 15 14 13 12 11 10 9 8 7
                   6 5 17 25 19 20 21 21 22 23 23 24 18 26 }> 27
14      150     1 2 4 17 18 19 22 23 <{ 19 20 21 21 22 23 23 24 }> 25 <{ 18 24
```

```
                23 23 22 21 21 20 19 25 }> 26 <{ 2 3 4 16 15 14 13 12 11 10 9
                   8 7 6 5 17 25 19 20 21 21 22 23 23 24 18 26 }> 27
15      151     1 2 4 17 18 19 23 <{ 19 20 21 21 22 23 23 24 }> 25 <{ 18 24 23
                23 22 21 21 20 19 25 }> 26 <{ 2 3 4 16 15 14 13 12 11 10 9 8 7
                   6 5 17 25 19 20 21 21 22 23 23 24 18 26 }> 27
16      152     1 2 4 17 18 19 24 <{ 19 20 21 21 22 23 23 24 }> 25 <{ 18 24 23
                23 22 21 21 20 19 25 }> 26 <{ 2 3 4 16 15 14 13 12 11 10 9 8 7
                   6 5 17 25 19 20 21 21 22 23 23 24 18 26 }> 27
17      153     1 2 4 17 18 25 <{ 18 24 23 23 22 21 21 20 19 25 }> 26 <{ 2 3 4
                16 15 14 13 12 11 10 9 8 7 6 5 17 25 19 20 21 21 22 23 23 24
                   18 26 }> 27
18       154     1 2 4 17 26 <{ 2 3 4 16 15 14 13 12 11 10 9 8 7 6 5 17 25 19
                   20 21 21 22 23 23 24 18 26 }> 27
19        155     1 27
```

CHAPTER 5

Coverage Analyzer

This chapter covers the **ctcover** utility that generates *Ct* coverage report for a module. It analyzes a ***.pth** and a ***.trc** file to produce a ***.rpt** report file. This is the command a user will employ most often to check *Ct* coverage.

5.1 'ctcover' Syntax

Syntax:

```
ctcover   name  tracefile  [-f shortname]
```

where,

name is the name of the module to be analyzed.This name can be of any length (but see below).

tracefile is the full trace file name to be analyzed.

-f shortname is used to permit the module name to be of any length, but the output file is named shortname.rpt anyway. You must make sure that **shortname.pth** exists and contains the correct path information.

This option is primarily used for DOS, where filenames are limited to eight characters.

Produces:

name.seg File with the extractions from the trace file for the named module.

name.rpt File containing the Ct coverage report for the name module.

Examples:

The command:

```
ctcover  verylongmodulename trace.trc -f long
```

specifies that **verylongmodulename** is the name of the module to be analyzed. The output of the above should be the file **long.rpt**.

Notes: The script **DoRPT** will run the command:

```
ctcover <name> *.trc
```

for all of the ***.pth** files that it finds. This will have the effect of producing all of the ***.rpt** files possible within the current directory. However, this script only works for modules that have the same name as the path file-name (i.e. does not use the **-f** option).

Sample Output:

Here are sample outputs from **ctcover**:

Example 1:

```
Ct Test Coverage Analyzer Version 1.8
        (c) Copyright 1990 by Software Research, Inc.
Module “getfil”: 5 paths, 3 were hit in 227 invocations.
 60.00% Ct coverage
Test descriptor: Coverage report for module boxes.<lang>

HIT/NOT-HIT REPORT
------------------
P#   Hits       Path text

1      90       1 2
2      22       1 3 4 <{4 }> 5 6
3     115       1 3 4 <{4 }> 5 7
4    None       1 3 5 6
5    None       1 3 5 7
```

Example 2:

```
Ct Test Coverage Analyzer Version 1.8
        (c) Copyright 1990 by Software Research, Inc.
Module “putfld”: 6 paths, 6 were hit in 115 invocations.
100% Ct Coverage!
Test descriptor: Coverage report for module boxes.<lang>

HIT/NOT-HIT REPORT
------------------
P#   Hits       Path text

1       1       1 2 3
2      10       1 2 4 5
3      22       1 2 4 6
4      10       1 7 8 9
5      71       1 7 8 10
6       1       1 7 11
```

Example 3:

```
Ct Test Coverage Analyzer Version 1.8
```

```
        (c) Copyright 1990 by Software Research, Inc.
Module "putbox": 192 paths, 7 were hit in 22 invocations.
  3.65% Ct coverage
Test descriptor: Coverage report for module boxes.<lang>

HIT/NOT-HIT REPORT
------------------
P#   Hits       Path text

1    None       1 2 3 6 7 9 17 <{17 }> 18 19 20 21 22 23 <{22 23
24 }> 25 \
                        <{21 24 23 22 25 }> 26
2    None       1 2 3 6 7 9 17 <{17 }> 18 19 20 21 22 24 <{22 23
24 }> 25 \
                        <{21 24 23 22 25 }> 26
3    None       1 2 3 6 7 9 17 <{17 }> 18 19 20 21 25 \
                        <{21 24 23 22 25 }> 26

                ...(intervening paths deleted for clarity)

159  None       1 11 13 17 <{17 }> 18 19 20 21 25 <{21 24 23 22
25 }> 26
160  None       1 11 13 17 <{17 }> 18 19 20 26
161  None       1 11 13 17 <{17 }> 18 19 27
162    15       1 11 13 17 <{17 }> 18 28
163  None       1 11 13 18 19 20 21 22 23 <{22 23 24 }> 25 \
                        <{21 24 23 22 25 }> 26
164  None       1 11 13 18 19 20 21 22 24 <{22 23 24 }> 25 \
                        <{21 24 23 22 25 }> 26
165  None       1 11 13 18 19 20 21 25 <{21 24 23 22 25 }> 26
166  None       1 11 13 18 19 20 26
167  None       1 11 13 18 19 27
168  None       1 11 13 18 28
169  None       1 14 15 17 <{17 }> 18 19 20 21 22 23 <{22 23 24
}> 25 \
                        <{21 24 23 22 25 }> 26
170  None       1 14 15 17 <{17 }> 18 19 20 21 22 24 <{22 23 24
}> 25 \
                        <{21 24 23 22 25 }> 26
171  None       1 14 15 17 <{17 }> 18 19 20 21 25 \
                        <{21 24 23 22 25 }> 26
172  None       1 14 15 17 <{17 }> 18 19 20 26
173     1       1 14 15 17 <{17 }> 18 19 27
174  None       1 14 15 17 <{17 }> 18 28

                ...(intervening paths deleted for clarity)

192  None       1 14 16 18 28
```

CHAPTER 6

TCAT-PATH Menus

The second way to access *TCAT-PATH* is with menus and this chapter will explain how to do so. If you would rather use command line invocation, you may skip this chapter and go on to Chapters 8 and 9 or the full *TCAT-PATH* example in Chapter 10.

6.1 TCAT-PATH ASCII Menus

Menus help users in two ways: by providing a fixed structure for collecting test coverage information and by providing a convenient way to customize a sequence of operations.

6.1.1 Invoking TCAT-PATH

Start up *TCAT-PATH* in interactive mode with the command:

```
tcatpath [-r file]
```

where,

file is the optional configuration file (**rc** file) name. The default name for the configuration file is *tcatp.rc*. If you don't specify a configuration file, or if *TCAT-PATH* doesn't find the file **tcatp.rc** in the current directory, then *TCAT-PATH* issues a warning message and continues processing, using default values.

Remember that the content of the *TCAT-PATH* configuration file, tcatp.rc, always overrides the internally supplied (default) values of all parameters.

6.1.2 TCAT-PATH Menu Tree

The organization and structure of the menus for the interactive *TCAT-PATH* is shown in the diagram on the following page:

```
TCAT-PATH:
|               Selects ACTIONS or FILES or OPTIONS menus
|               Shows option settings
|               Shows current execution statistics
|               Saves option settings
|               Exit from TCAT-PATH system
|               On-line help frames
|               !<system commands>
|
+----ACTIONS:
|               Selects basic TCAT-PATH operations
|               Shows option settings
|               Return to prior menu
|               On-line help frames
|               !<system commands>
|
+----OPTIONS:
|               Helps select all user-settable options
|               Shows option settings
|               Return to prior menu
|               On-line help frames
|               !<system commands>
|
+----FILES:
                Shows all current options settings
                Allows changing file settings
                Return to prior menu
                On-line help frames
                !<system commands>
```

After *TCAT-PATH* starts, you will see the title information, version control indication,

and the prompt

```
"TCAT-PATH:MAIN:" .
```

To see the available menu options, type from any prompt within *TCAT-PATH*:

```
?
```

and then

```
[RETURN].
```

TCAT-PATH then displays the available options for that menu. This feature works for all menus throughout *TCAT-PATH*.The current menu is redrawn whenever you give an unrecognized command.

6.1.2.1 Issuing Commands

You can issue commands by typing the first few letters of each command's name. The only requirement is that the letter sequence be unique to that command. *TCAT-PATH* will inform you when a command you issue matches two or more possible commands.

To set variables (see the options menu description, below) you must type the entire variable name. This is done in order to be consistent with configuration file processing.

Displaying Current Parameter Settings

You can display the current settings (options and filenames) known to *TCAT-PATH* at any time using the settings command, get on-line help with the help \f1 command, and exit the current menu using exit. The configuration file reading in the settings is automatically used. However, the settings can be changed if required.

TCAT-PATH Menu ‘Stack'

You can move from the MAIN menu to any other menu at will. *TCAT-PATH* remembers the sequence of your choice of menus in an internal “stack". This means that when switching from one menu to another, you can return to the immediately prior menu with the exit command. This feature is provided to prevent you from entering conflicting or incorrect data during a run.

If you wish, you can issue a series of exit commands that will eventually return you to the MAIN menu to exit the system. That is, your moves between the three subsidiary menus are “stacked" and must be “unstacked" before returning to the MAIN menu.

If you press the DEL key, you return immediately to the MAIN menu.

6.1.3 Main Menu

All commands may be abbreviated when no ambiguity exists, e.g. “options" can be shortened to “o” because no other command in the *TCAT-PATH* menu starts with “o”.

When *TCAT-PATH* is activated the following menu options are displayed:

```
TCAT-PATH:MAIN:
Options:
                save     -- Save the current settings
                              for TCAT-PATH.
               stats     -- Show current usage values
                              for TCAT-PATH.

             actions      -- Go to the ACTIONS menu.
               files      -- Go to the FILES menu.
             options      -- Go to the OPTIONS menu.

            settings     -- List the current settings
                              for TCAT-PATH options.
          help [opt]      -- Display HELP text for a
                              command.
             release      -- Show release and version
                              numbers for this TCAT-
                              PATH copy.
              exit    -- Exit from TCAT-PATH to system
```

6.1.4 Actions Menu

The Actions menu is displayed below:

```
TCAT-PATH:ACTIONS:
Options:
         preprocess     -- Runs the preprocessor
                           command on the program.
       instrument     -- Instrument/generate digraph
                           of program.
              apg     -- Run apg on *.dig file.
            cyclo     -- Compute cyclomatic number
                           on *.dig file.
           digpic     -- Run digpic on *.dig file.
          ctcover     -- Compute Ct path cover.
            files     -- Go to the FILES menu.
          options     -- Go to the options menu.
         settings     -- Display current runtime
                           settings.
       help [opt]    -- Display HELP text for command
             exit     -- Exit current level
```

6.1.5 Files Menu

The Files menu is displayed below:

```
TCAT-PATH:FILES:
Options:
      prefix <name>      -- Base name of module being
                            processed.
     digraph <name>      -- Name of digraph file
                            (default 'prefix'.dig).
       path <name>       -- Name of path file (default
                            'prefix'.pth).
   tracefile <name>      -- Name of trace file (default
                            'prefix'.trc).
     report <name>       -- Name of report file (default
                            'prefix'.rpt).
       basis <name>      -- Name of basis path file
                            (default 'prefix'.bas).

            actions      -- Go to the ACTIONS menu.
            options      -- Go to the OPTIONS menu.

           settings      -- Display current runtime
                            settings.
       help [opt]        -- Display HELP text for command
               exit      -- Exit current level
```

6.1.6 Options Menu

The Options menu is displayed below:

```
TCAT-PATH:OPTIONS:
Options:
      maxnodes  <#>     -- Maximum number of nodes
                           digraph will process.
      maxedges  <#>     -- Maximum number of edges
                           digraph will process.
     loopcount  <#>     -- Value of K to use in apg
                           executions; default K = 1.
      maxprint  <#>     -- Maximum number of paths
                           apg prints; default 300.
       maxpath  <#>     -- Maximum number of paths
                           apg calculates.
         basis  <#>     -- Basis path default number
                           if non-zero.
    centerline  <#>     -- Centerline offset for a
                           digraph picture.
         space  <#>     -- Spaces between nodes in
                           digraph picture, default
                           1
        width  <#>     -- Maximum width of the image
                           produced; default = 80.
      maxcalls  <#>     -- Maximum number of calls
                           ctseg produces.
    chnghelp <file>     -- Specify a new on-line
                           documentation <file>.
            actions     -- Go to the ACTIONS menu.
              files     -- Go to the FILES menu.
           settings     -- Display current runtime
                           settings.
        help [opt]      -- Display HELP text for a
                           command
               exit     -- Exit to the system
```

6.1.7 Saving Changed Option Settings

Before leaving *TCAT-PATH*, or before running a digraph analysis, instrumentation, path generation, and / or coverage analysis session, the user will be prompted to save the current option settings (unless this has already been done in the current execution of *TCAT-PATH* and the options have not been changed since they were saved).

This part of an interactive session appears as follows (assuming you wish to save all current options in the file example.rc):

```
TCAT-PATH:
Do you want to save the current parameter settings
(y/n):
y
Do you want to use the default filename ("tcatp.rc")
(y/n):
n
Specify filename:
example.rc
Parameter settings saved in "example.rc".
```

Note that *TCAT-PATH* will normally prompt you about saving current settings when you finally exit the system (via an exit command in the *TCAT-PATH* MAIN Menu).

6.1.8 Running System Commands

You may execute a command available to the underlying operating system by using the "!" symbol, as follows:

```
TCAT-PATH:!<command>
```

Control is returned to *TCAT-PATH* after the command is executed.

This feature is useful for editing files and other activity within a *TCAT-PATH* session.

6.1.9 Settings Command Output

The current set of options values is available from ALL *TCAT-PATH* menus, using the settings command.

An example of the output produced by the settings command is shown below. The values shown are the actual default values assigned as if there were NO configuration file present. This is also the set of values that will be written during *TCAT-PATH* exit if you choose to save the values.

```
Current TCAT-PATH Options Settings Are:
   Parameters:
        maxnodes        = 500
        maxedges        = 1000
        loopcount       = 1
        maxprint        = 300
        maxpath         = 4800
        basis           = 300
        centerline      = 0
        space           = 1
        width           = 80
        maxcalls        = 10000
        documentation   = /usr/tcatpath/tcatpath.hlp
   Files:
        prefix          = example
        digraph         = example.dig
        path            = example.pth
        tracefile       = example.trc
        report          = example.rpt
        basis_file      =
        config_file     = tcatp.rc
```

6.2 TCAT-PATH Configuration File

This section describes how to construct or edit *TCAT-PATH* configuration files. A sample file is shown at the end of this chapter.

All the commands in the *TCAT-PATH* system can read a configuration file (the default name is tcatp.rc) before starting processing.

This feature allows the user to set various run-time parameters automatically. Command-line parameters, however, override the configuration file settings when command-line parameters are present.

The *TCAT-PATH* configuration file is a simple ASCII text file that can be created with an editor.

Alternatively, you can create this file, and give it any name you like, by using the save option from within an interactive invocation of *TCAT-PATH*.

6.2.1 Configuration File Syntax

The following run-time parameters can be set from the configuration file. These parameters are shown here in the same order as they are displayed with a "settings" command within the interactive menus of *TCAT-PATH*.

<any comment> A line that begins with a # is treated as a comment.

maxnodes=<number>

The maximum number of nodes tp-i<lang> will process.Default is 500. An impractical limit is probably 2500.

maxedges=<number>

The maximum number of edges tp-i<lang> will process. Default is 1000. An impractical limit is probably 2500.

loopcount=<number>

The value of K to use in apg executions; default K = 1. (At present, only K = 1 can be used.)

maxprint=<number>

The maximum number of paths for apg to print. Default is 300. A practical limit is probably 1000.

maxpath=<number>

The maximum number of paths for apg to calculate. apg gives a message at the end of execution to show the total number of paths it would have printed; or it issues an error message when the "maxpath" parameter is exceeded.The default value is 4800. A practical limit is probably around 10,000.

basepath=<filename>

The name of the filename that contains the alternative definition of the basis path for drawing digraph pictures with digpic. The default value is " ", meaning that the "naturally generated" picture basis path should be used.

centerline=<number>

The centerline offset for a digraph picture, in bytes (default value 0). A typical value (for interactive screens) is 40.

space=<number> The number of spaces up / down between nodes. The default is 1. The maximum value is 5 (larger values are treated as errors).

width=<number> The maximum width of the image produced; The default is 80. The maximum possible value is 128. If the graph is large than this value then it is “clipped" at that value.

maxcalls=<number>

The maximum number of calls to be processed by ctseg, which is called by ctcover. The default is 10000. This is probably a practical limit.

help=<pathname>

The fully specified path name for the file containing the *TCAT-PATH* help frame information (interactive operation only). The default location is:

```
/bin/tcatpath/helpframes
```

This location is installation-dependent. If this filename is not specified correctly then the *TCAT-PATH* online help frames will not work correctly.

prefix=<name> The module or function name to be used as the base name or filename prefix for all subsequent processing. This is referred to in the following option descriptions as “<*>”.

If no prefix is specified then *TCAT-PATH* will not be able to process any files, generate any digraphs, or analyze path coverage. Accordingly, the default assigned value for the prefix is example.

digraph=<name.dig>

The name of the digraph file. If not specified, *TCAT-PATH* assumes you mean <*>.dig. The default value is example.dig

path=<name.pth> The name of the file of paths. If not specified, *TCAT-PATH* assumes you mean <*>.pth. The default value is example.pth.

tracefile=<name.trc>The name of the trace file (generated during your program execution). If not specified, *TCAT-PATH* assumes you mean <*>.trc. The default value is example.trc.

report=<name.rpt> The name into which to write the Ct coverage report.The default name is example.rpt.

6.2.2 Configuration File Processing

Lines in the configuration file can contain any of these commands in any order. Comment lines must have a "#" as the first character.

All white space (i.e. tabs and blanks) in the configuration file is ignored.

All arguments (when appropriate) are treated as character string tokens (i.e. no internal white space).

The latest-occurring command in case there are duplicate commands prevails. (This feature may be useful when handling several configuration files that differ only slightly.)

6.2.3 Sample TCAT-PATH Configuration File

Below is an example of a typical *TCAT-PATH* configuration file.

```
# Sample options setting commands (configuration file)
width=20
=example
basis_file = example.basis

# Redefine the maxima for “apg” operation...
maxprint= 1000
maxedges=10000

# Value to keep updated archive records (C1 analysis)...
report=my.archive
# End of example configuration file
```

CHAPTER 7

Source Viewing Utility

This utility is only available on X Window System environments.

7.1 Introduction

Source viewing associates a segment or node with its corresponding source code. By simply clicking the mouse, the user is able to see source relating to a node or segment.

For the purpose of source viewing, nodes are indicated by circles. A segment (or edge) is a directed line connecting two nodes (or circles).

7.2 Invocation Syntax

Source viewing is invoked with the following command:

```
Xdigraph dig-file-S2ref-listing[-SC number]
```

where,

dig-file — The dig-file is the file that specifies the set of segments in “tail-node head-node segment-name" format. This is what is normally produced by **tp-ic** and named **module-name.dig**. This is the source file that the user can view.

-S *ref-listing* — The Reference Listing file (that is **filename.i.A**) is produced by the instrumentor.

[**-SC** *number*] — This switch is optional. *number* specifies the number of lines of source code above and below the clicked segment or node that are to be displayed. The default value is 10.

7.3 Example Invocation

This section refers to the full *TCAT-PATH* example chapter. For *TCAT-PATH*, the digraph files will be one of the following modules: **main**, **proc_input**, or **chk_chr**.

The reference listing is always **example.i.A**, which is the "C" program.

The following two pages show an example of source viewing, using main module. Figure 8 shows the mouse pointer (indicated by an arrow in the display) selecting a segment. For edges, the segment number must be clicked on. For nodes, the pointer must click somewhere inside the circle.

Figure 9 shows the result. For this, hold down the mouse button, and the source code will be displayed for as long as the button is held down.

NOTE: If the node / segment numbers are not visible, it is probably because the window size is too small. In this case, increase its size.

The main module on the following pages is invoked with the following:

```
Xdigraph main.dig -S example.i.A
```

To source view with graphical user interfaces, see Chapter 11.

Below is an example of the mouse pointer clicking on Segment 2.

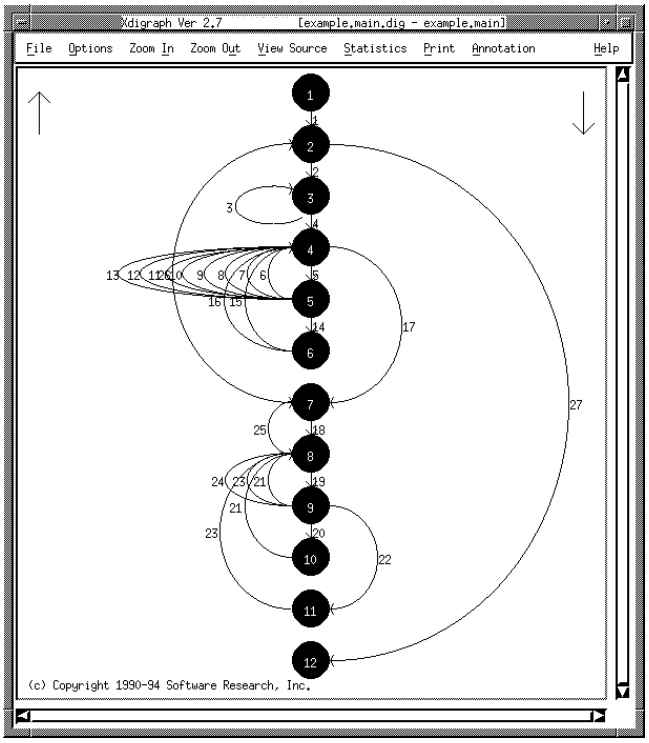


FIGURE 2 Source Viewing (Part 1 of 2)

Below is an example of the source code displayed as the mouse button is held down.

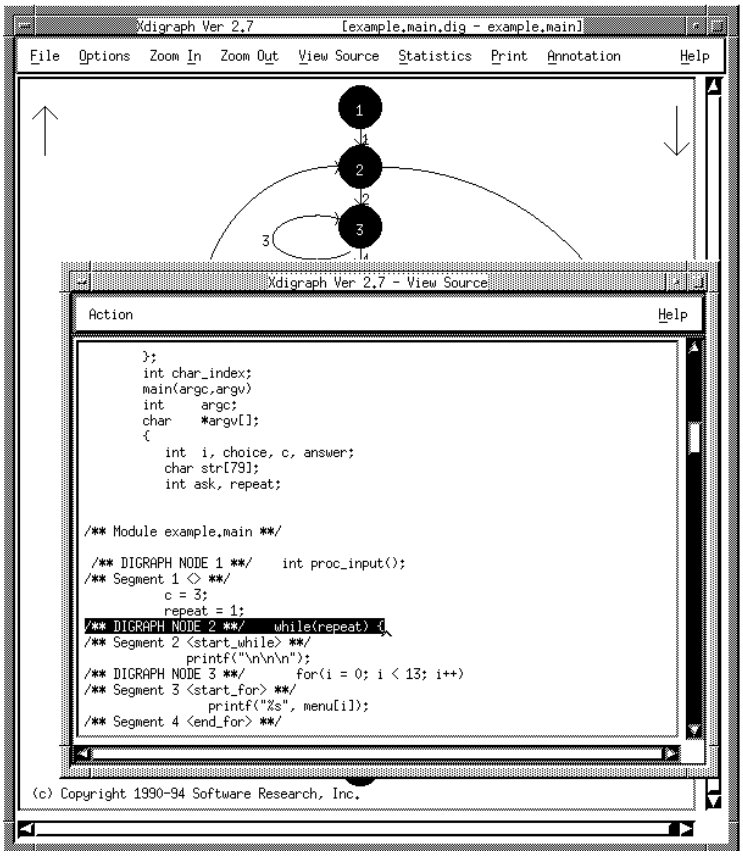


FIGURE 3 Source Viewing (Part 2 of 2)

CHAPTER 8

TCAT-PATH Command Summary for MS-DOS

This chapter gives a short command summary for *TCAT-PATH* for “C" running under MS-DOS.

8.1 Instrumentation, Compilation and Linking

The user is required to preprocess the source file through a “C" preprocessor before putting it to tp-ic instrumentor. The instrumented program is then compiled and linked with the appropriate runtime module. Depending on the size of your program and the development method used, the following subsections describe how it is done.

8.1.1 Stand-Alone Files

Here are the commands you would use with the Microsoft C 6.0 compiler on MS-DOS:

```
Preprocess: cl /P <filename>.c /* to produce <filename>.i */
Instrument: tp-ic -m6 <filename>.i /* to produce <filename>.ic */
Compile:    cl /c /Tc <filename>.ic/* to produce <filename>.obj  */
Link:       cl <filename>.obj ctrun1s.obj/* to produce <file-
                                              name>.exe */
Execute:(Run your program as usual.  Press RETURN
              twice to accept the default values for
              trace file message and name.)
```

Note that **-m6** is the **tp-ic** switch for Microsoft C 6.0 compiler. **/Tc** is a Microsoft C 6.0 option that allows for compilation of files with extensions other than **.c**.

Also, note that **ctrun1s.obj** is the runtime object module that comes with *TCAT-PATH*. There are various runtime object files, depending on compiler, runtime level, and memory model used. For more runtime descriptions on MS-DOS runtimes, turn to “Runtime Descriptions” on page 24, Section 3.1.

8.1.2 Systems with 'make' Files

1. In systems that have 'make' files where **.obj** files are explicitly listed as targets, add the following built-in rule before other targets:

```
# Built in rule for TCAT instrumentation...
 .c.obj:
        cl $(CFLAGS) /P $*.c            cl. $(CFLAGS) /P $*.c
        tp-ic -m6 $*.i              or  tp-ic -m6 $*.i
        ren $*.i temp.c                 cl $(CFLAGS) /c /Tc
$*.ic
        cl $(CFLAGS) /c temp.c
        ren temp.o $*.obj
sample.obj: sample.c
        ...
```

2. Add .**cM ctrun<level><model>.obj** to the list of linked object modules. You must choose the version of runtime to use, based on the runtime level and the memory model (small, compact, medium, large or huge).
3. Run the 'make' file to produce the instrumented program.

8.1.3 'make' With 'cl', 'msc'

This section deals with situations that involve 'make' files for commonly available PC-based compilers, such as Microsoft C, where compile statements are explicitly mentioned.

1. Replace 'cl' (or 'msc') with the following lines:

```
cl $(CFLAGS) /P <filename>.c
tp-ic -m6 <filename>.i
ren <filename>.i temp.c
cl $(CFLAGS) /c temp.c
ren temp.o <filename>.o
```

2. Add **`ctrun<level><model>.obj`** to the list of linked object modules.
3. Run the make file to produce the instrumented program.

8.1.4 Systems without ‘make' Files

Go to the directories with the source code and follow the method for stand alone files with each source code file (preprocess, instrument, compile). Finally, link all the object files with the appropriate runtime object file.

8.1.5 Program Execution

Run your program as usual.

NOTE: With the default runtimes (runtime level 1), the instrumented program will add two prompts when the first instrumented code is executed. You may fill in a value or press return each time. The prompts may be suppressed by changing the provided runtime. Refer to Section 3.1 for a more detailed description of runtimes available.

CHAPTER 9

TCAT-PATH Command Summary for UNIX

This chapter summarizes commands you use with *TCAT-PATH* for “C" in UNIX and UNIX-like environments.

9.1 Instrumentation, Compilation and Linking

The user is required to preprocess the source file through a “C" preprocessor before putting it to tp-ic instrumentor. The instrumented program is then compiled and linked with the appropriate runtime modules.

Depending on the size of your program and the development method that you use, the following subsections describe how it is done.

9.1.1 Stand-Alone Files

The commands used are:

```
Preprocess:    cc -P <filename>.c   /* to produce <filename>.i */
Instrument:    tp-ic  <filename>.i  /* to produce <filename>.i.c */
Compile:       cc -c <filename>.i.c /* to produce <filename>.i.o  */
Link:          cc <filename>.i.o ctrun1.o  /* to produce a.out */

Execute:       (Run your program as usual.  Press RETURN
                      twice to accept the default values for
                      trace file message and name.)
```

1. If you have ‘make' files where ***.o** files are created with built-in rules, add the following built-in rule before other targets:

```
# Built in rule for TCAT-PATH instrumentation...
 .c.o:
    cc $(CFLAGS) -P $*.c
    tp-ic $*.i
    cc $(CFLAGS) -c $*.i.c
    mv $*.i.o $*.o

sample.o: sample.c
    ...
 # The above will depend on which one invokes built
in rules.
```

2. Add **ctrun<level>.o** to the list of linked object modules.
3. Then run the ‘make' file to produce the instrumented version of the software.

9.1.2 'make' files with cc called in directives

When cc is explicitly called in directives, then add tp-ic commands to the cc commands within the 'make' file.

1. Replace cc with the following lines:

```
cc $(CFLAGS) -P <filename>.c
tp-ic <filename>.i
cc $(CFLAGS) -c <filename>.i.c
mv <filename>.i.o <filename>.o
```

2. Add **ctrun<level>.o** to the list of linked object modules.
3. Finally, run the make file to produce the instrumented version of the software.

9.1.3 A System Which Does Not Use ‘make' Files

(Or which will not allow ‘make' file changes)

Go to the directories that contain the source code.

There, type the following commands:

```
cc -P *.c
tp-ic *.i
cc -c *.i.c
cc *.i.o ctrun<?>.o
```

to create the instrumented source, objects and executable.

9.2 Program Execution

Run your program as usual.

NOTE: With the default runtimes (runtime level 1), the instrumented program will add two prompts when the first instrumented code is executed. You may fill in a value or press return each time. The prompts may be suppressed by changing the provided runtime. Refer to Section 3.1 for a more detailed description of runtimes available.

CHAPTER 10

Full TCAT-PATH Example

This chapter describes a full *TCAT-PATH* example that includes a sample “C" program, instrumented program, referenced listing, digraph files for each module, cyclomatic number calculations, digraph pictures, and coverage reports.

10.1 Introduction

It is assumed that *TCAT-PATH* will be used on syntactically correct programs, that is programs that will compile cleanly before instrumentation. Of course, *TCAT-PATH* will be used to verify that each program segment or logical branch executes correctly under typical operating conditions.

Figures 2 and 3 show a sample “C" program with three function modules.

This example program will be used throughout the chapter to describe each component of *TCAT-PATH* to better aid the user.

```
/* EXAMPLE.C --example file for use with TCAT, STCAT, TCAT-PATH.
*/
#include "stdio.h"
#include <ctype.h>

#define INPUTERROR     -1
#define INPUTDONE       0
#define MENU_CHOICES   13
#define STD_LEN        79
#define TRUE  1
#define FALSE 0
#define BOOL  int
#define OK      TRUE
#define NOT_OK  FALSE

char menu[MENU_CHOICES][STD_LEN] = {
   "SOFTWARE RESEARCH'S RESTAURANT GUIDE \n”),
   “     What type of food would you like”),
   “\n”,
   “     1        American 50s   \n”),
   “     2        Chinese    - Human Style \n”),
   “     3        Chinese    - Seafood Oriented \n”),
   "     4        Chinese    - Conventional Style \n”),
   "     5        Danish          \n”),
```

```
   "      6        French          \n"),
   "      7        Italian         \n"),
   "      8        Japanese        \n"),
   "\n\n"
};
int char_index;

main(argc,argv)      /* simple program to pick a restaurant */
int   argc;
char   *argv[];
{
   int  i, choice, c,answer;
   char str[STD_LEN];
   BOOL ask, repeat;
   int proc_input();
   c = 3;
   repeat = TRUE;
   while(repeat) {
      printf("\n\n\n");
      for(i = 0; i < MENU_CHOICES; i++)
         printf("%s", menu[i]);
      gets(str);
      printf("\n");
      while(choice = proc_input(str)) {
         switch(choice) {
         case 1:
           printf("\\tFog City Diner 1300 Battery 982-2000 \n");
            break;
         case 2:
           printf("\tHunan Village Restaurant 839 Kearney 956-
7868 \\n");
            break;
         case 3:
          printf("\tOcean Restaurant 726 Clement 221-3351 \n");
            break;
         case 4:
           printf("\tYet Wah 1829 Clement  387-8056 \n");
            break;
         case 5:
           printf("\tEiners Danish Restaurant 1901 Clement 386-
9860 \n");
            break;
         case 6:
           printf("\tChateau Suzanne 1449 Lombard 771-9326 \n");
            break;
         case 7:
         printf("\tGrifone Ristorante 1609 Powell 397-8458 \n");
            break;
         case 8:
```

```
            printf("\tFlints Barbecue 4450 Shattuck, Oakland
\n");
            break;
         default:
            if(choice != INPUTERROR)
              printf("\t>>> %d: not a valid choice.\n", choice);
            break;
         } }

      for(ask = TRUE; ask; ) {
         printf("\n\tDo you want to run it again? ");
         while((answer = getchar()) != '\\n') {
            switch(answer) {
            case 'Y':
            case 'y':
               ask = FALSE;
               char_index = 0;
               break;
            case 'N':
            case 'n':
               ask = FALSE;
               repeat = FALSE;
               break;
            default:
               break;
            } } } } }

int proc_input(in_str)
char *in_str;
{
   int tempresult = 0;
   char bad_str[80], *bad_input;
   BOOL got_first = FALSE;
   bad_input = bad_str;

   while(isspace(in_str[char_index]))
      char_index++;
   for( ; char_index <= strlen(in_str); char_index++) {
      switch(in_str[char_index]) {
      case '0':
      case '1':
      case '2':
      case '3':
      case '4':
      case '5':
      case '6':
      case '7':
      case '8':
      case '9':
```

```
            /* process choice */
            tempresult = tempresult * 10 + (in_str[char_index] -
'0');
            got_first = TRUE;
            break;

         default:
            if(chk_char(in_str[char_index])) {
               return(tempresult);
            }
            else {
               if(char_index > 0 && got_first)
                  char_index--;
               while(char_index <= strlen(in_str)) {
                  if(chk_char(in_str[char_index]))
                     break;
                  else
                     *bad_input++ = in_str[char_index];
                  char_index++;
               }
               *bad_input = '\\0';
               printf("\t>>> bad input: %s\n", bad_str);
               char_index++;
               return(INPUTERROR);
            } } }
      return(INPUTDONE);
}

BOOL chk_char(ch)
char ch;
{
   if(isspace(ch) || ch == '\0')
      return(OK);
   else
      return(NOT_OK);
```

10.2 Preprocess, Instrument, Compile and Link

The first step in *TCAT-PATH* is to prepare your "C" program to provide segment coverage data. You start by:

1. Pre-processing the program. Most "C" compilers have this facility.
2. Instrumenting it to insert markers at every segment position.

The program on the next pages shows, in bold, the effects of *TCAT-PATH* instrumentation on your "C" program:

```
------------------------------------------------------------------
-- C1 instrumentation by TCAT-PATH/C instrumenter:
--
-- Program tp-ic, Release 8

-- Instrumented on Wed Jan 30 14:21:08 1991

-- SR Copy Identification No. 0.
--
------------------------------------------------------------------
-- (c) Copyright 1990 by Software Research, Inc.  All Rights
Reserved.
--
-- This program was instrumented by SR proprietary software,
-- for use with the SR proprietary TCAT runtime package.
-- Use of this program is limited by associated software
-- license agreements.
------------------------------------------------------------------
*/

extern SegHit();
extern Strace();
extern Ftrace();
extern EntrMod();
extern ExtMod();
\

char menu[13][79] = {
   "SOFTWARE RESEARCH'S RESTAURANT GUIDE \\n",
   "      What type of food would you like?\n",
   "\n",
   "      1        American 50s   \n",
   "      2        Chinese    - Hunan Style \n",
   "      3        Chinese    - Seafood Oriented \n",
   "      4        Chinese    - Conventional Style \n",
   "      5        Danish          \n",
   "      6        French          \n",
   "      7        Italian         \n",
```

```
   "      8        Japanese        \\n",
   "\n\n"
};

int char_index;

main(argc,argv)
int   argc;
char   *argv[];
{
   int  i, choice, c,answer;
   char str[79];
   int ask, repeat;
   int proc_input();

   \Strace("IC",0x7504,0,0);
   \EntrMod(27,"main",-1);

   SegHit(1);
   c = 3;
   repeat = 1;
   { while(repeat) { SegHit(2);
      {
        printf("\n\n\n"); {
        for(i = 0; i < 13; i++) { SegHit(3);
           printf("%s", menu[i]); }
        SegHit(4); };
        gets(str);
        printf("\n");

        { while(choice = proc_input(str)) { SegHit(5);
           {
             { switch(choice)
           {
                  case 1:SegHit(6);
                     printf("\tFog City Diner1300 Battery  982-
2000 \n");
                     break;
                  case 2: SegHit(7);
                   printf("\tHunan Village Restaurant  839 Kear-
ney   956-7868 \n");
                     break;
                  case 3: SegHit(8);
                     printf("\tOcean Restaurant 726 Clement 221-
3351 \n");
                     break;
                  case 4: \SegHit(9);\
                   printf("\tYet Wah 1829 Clement 387-8056 \n");
                     break;
```

```
                  case 5: \SegHit(10);
                   printf("\tEiners Danish Restaurant 1901 Clem-
ent   386-9860 \n");
                     break;
                  case 6: SegHit(11);
                     printf("\tChateau Suzanne 1449 Lombard 771-
9326\n");
                     break;
                  case 7: SegHit(12);
                     printf("\tGrifone Ristorante1609 Powell397-
8458 \\n");
                     break;
                  case 8: SegHit(13);
                   printf("\tFlints Barbecue 4450 Shattuck, Oak-
land \n");
                     break;
                  default: \SegHit(14);
                     if(choice != -1) { SegHit(15);
                      printf("\t>>> %d: not a valid choice.\n",
choice);
                     } else SegHit(16);
                     break;
                     } } } } SegHit(17); };


    { for(ask = 1; ask; ) { SegHit(18);
           {
             printf("\n\tDo you want to run it again? ");
             {while((answer = getchar()) != '\n') { SegHit(19);
                {
                  {\ switch(answer)
                {
                       case 'Y': SegHit(20);
                       case 'y': SegHit(21);
                          ask = 0;
                          char_index = 0;
                          break;
                       case 'N': \SegHit(22);\
                       case 'n': \SegHit(23);\
                          ask = 0;
                          repeat = 0;
                          break;
                       default: \SegHit(24);\
                          break;
                           } \}\
                } \} SegHit(25); };\
            } \} SegHit(26); };\
        } \} SegHit(27); };\

  \ExtMod("main"); \
```

```
  \Ftrace(0);\
}

int proc_input(in_str)
char *in_str;
{
   int tempresult = 0;
   char bad_str[80], *bad_input;
   int got_first = 0;

   \EntrMod(24,"proc_input",-1);\
   \SegHit(1);\

   bad_input = bad_str;

   \{\ while(isspace(in_str[char_index])) \{ SegHit(2);\
         char_index++; \} SegHit(3); };\


   \{\ for( ; char_index <= strlen(in_str); char_index++) \{
SegHit(4);\
      {
        \{\ switch(in_str[char_index])
          {
              case '0': \SegHit(5);\
              case '1': \SegHit(6);\
              case '2': \SegHit(7);\
              case '3': \SegHit(8);\
              case '4': \SegHit(9);\
              case '5': \SegHit(10);\
              case '6': \SegHit(11);\
              case '7': \SegHit(12);\
              case '8': \SegHit(13);\
              case '9': \SegHit(14);\
                 tempresult = tempresult * 10 +
(in_str[char_index] - '0');
                 got_first = 1;
                 break;
             default: \SegHit(15);\
               if(chk_char(in_str[char_index])) \{ SegHit(16);\
                  { \{ExtMod("proc_input");\
                    return(tempresult); \}\
                  } \}\
                else \{ SegHit(17);\
                  {
               if(char_index > 0 && got_first) \{ SegHit(18);\
                   char_index--; \} else SegHit(19);\
                    \{\ while(char_index <= strlen(in_str)) \{
SegHit(20);\
```

```
                        {
                        if(chk_char(in_str[char_index])) \{ Seg-
Hit(21);\
                             break; \}\
                          else \{ SegHit(22);\
                          *bad_input++ = in_str[char_index]; \}\
                          char_index++;
                        } \} SegHit(23); };\

                      *bad_input = '\0';
                      printf("\\t>>> bad input: %s\\n", bad_str);
                      char_index++;
                      \{ ExtMod("proc_input");\
                      return(-1); \}\
                    } \}\
                } \}\
            } \} SegHit(24); };\


   \{ ExtMod("proc_input");\
   return(0); \}\

   \ExtMod("proc_input");\
}

int chk_char(ch)
char ch;
{
   \EntrMod(3,"chk_char",-1);\
   \SegHit(1);\

   if(isspace(ch) || ch == '\0') \{ SegHit(2); { Ext-
Mod("chk_char");\
      return(1); \} }\
   else \{ SegHit(3); { ExtMod("chk_char");\
      return(0); \} }\

   \ExtMod("chk_char");\
}
```

10.3 Reference Listing

The Reference Listing file (that is filename.i.A or filename.ia for DOS) is produced by the instrumentor and is used for manual cross-referencing during a series of tests. The Reference Listing is a version of your "C" program with segments (or edges) and nodes marked.

You will use this report by gathering the "Not Hit" paths from report files, and then looking up the related code in the Reference Listing. After reviewing the exercised and not-exercised parts of the program, you can design subsequent test cases to exercise more paths.

Extensive segment, node and module notation have also been embedded and the segment and node sequence numbers are listed along the left-most column.

The header identifies the file as a Reference Listing and includes the Release number plus a copyright notice.

The code that tp-ic adds appears in bold in the following program.

```
------------------------------------------------------------------
-- TCAT-PATH/C, Release 8


--
-- (c) Copyright 1990 by Software Research, Inc.  ALL RIGHTS
RESERVED.
--
-- SEGMENT REFERENCE LISTING
--
-- Instrumentation date: Wed Jan 30 14:21:08 1991
--
-- Separate modules and segment definitions for each module are
-- indicated in this commented version of the supplied source
file.
------------------------------------------------------------------
\

        char menu[13][79] = {
           "SOFTWARE RESEARCH'S RESTAURANT GUIDE \n",
           "      What type of food would you like?\n",
           "\n",
           "      1        American 50s   \n",
           "      2        Chinese    - Hunan Style \n",
           "      3        Chinese    - Seafood Oriented \n",
           "      4        Chinese    - Conventional Style \n",
           "      5        Danish          \n",
           "      6        French          \n",
           "      7        Italian         \n",
           "      8        Japanese        \n",
```

```
            "\n\n"
         };
         int char_index;
         main(argc,argv)
         int argc;
         char*argv[];
         {
            int  i, choice, c,answer;
            char str[79];
            int ask, repeat;


\/** Module main **\
\/* DIGRAPH NODE 1 *\

    int proc_input();
\/** Segment 1 <> **\
           c = 3;
           repeat = 1;
\/* DIGRAPH NODE 2 *\ while(repeat) {
\/** Segment 2 <start while> **\
              printf("\n\n\n");
\/* DIGRAPH NODE 3 *\ for(i = 0; i < 13; i++)
\/** Segment 3 <start for> **\
                 printf("%s", menu[i]);
\/** Segment 4 <end for> **\
              gets(str);
              printf("\n");
\/* DIGRAPH NODE 4 *\ while(choice = proc_input(str)) {
\/** Segment 5 <start while> **\
\/* DIGRAPH NODE 5 *\ switch(choice) {
                    case 1:
\/** Segment 6 <case alt> **\
                      printf("\tFog City Diner 1300 Battery
982-2000 \n");
                      break;
                     case 2:
\/** Segment 7 <case alt> **\
                      printf("\tHunan Village Restaurant  839
Kearney   956-7868 \n");
                      break;
                     case 3:
\** Segment 8 <case alt> **\
                      printf("\tOcean Restaurant 726 Clement 221-
3351 \n");
                      break;
                     case 4:
\** Segment 9 <case alt> **\
                      printf("\tYet Wah 1829 Clement 387-8056
\n");
```

```
                    break;
                  case 5:
\/** Segment 10 <case alt> **\
                    printf("\tEiners Danish Restaurant 1901
Clement   386-9860 \n");
                    break;
                  case 6:
\/** Segment 11 <case alt> **\
                    printf("\tChateau Suzanne          1449
Lombard   771-9326 \n");
                    break;
                  case 7:
\/** Segment 12 <case alt> **\
                    printf("\tGrifone Ristorante       1609
Powell 397-8458 \n");
                    break;
                  case 8:
\/** Segment 13 <case alt> **\
                    printf("\tFlints Barbecue          4450
Shattuck, Oakland \n");
                    break;
                  default:
\/** Segment 14 <case alt> **\
\/* DIGRAPH NODE 6 *\ if(choice != -1)
\/** Segment 15 <if> **\
                       printf("\t>>> %d: not a valid
choice.\n", choice);
\** Segment 16 <implied else> **\
                    break;
               }
            }
/** Segment 17 <end while> **\
/* DIGRAPH NODE 7 *\ for(ask = 1; ask; ) {
/** Segment 18 <start for> **\
               printf("\n\tDo you want to run it again? ");
/* DIGRAPH NODE 8 */ while((answer = getchar()) != '\n') {
/** Segment 19 <start while> **\
/* DIGRAPH NODE 9 *\ switch(answer) {
                    case 'Y':
/** Segment 20 <case alt> **\
/* DIGRAPH NODE 10 *\ case 'y':
/** Segment 21 <case alt> **\
                       ask = 0;
                       char_index = 0;
                       break;
                    case 'N':
/** Segment 22 <case alt> **\
/* DIGRAPH NODE 11 *\ case 'n':
/** Segment 23 <case alt> **\
                       ask = 0;
```

```
                        repeat = 0;
                        break;
                      default:
/** Segment 24 <case alt> **\
                        break;
                   } } } } }
/** Segment 25 <end while> **\
/** Segment 26 <end for> **\
/** Segment 27 <end while> **\
/* DIGRAPH NODE 12 *\ int proc_input(in_str)
       char *in_str;
       {
          int tempresult = 0;
          char bad_str[80], *bad_input;


/** Module proc_input **\

/* DIGRAPH NODE 1 */ int got_first = 0;
/** Segment 1 <> **\/
          bad_input = bad_str;
/* DIGRAPH NODE 2 */ while(isspace(in_str[char_index]))
/** Segment 2 <start while> **/
             char_index++;
/** Segment 3 <end while> **/
/* DIGRAPH NODE 3 */ for( ; char_index <= strlen(in_str);
char_index++) {
/** Segment 4 <start for> **/
/* DIGRAPH NODE 4 */ switch(in_str[char_index]) {
                case '0':
/** Segment 5 <case alt> **/
/* DIGRAPH NODE 5 */ case '1':
/** Segment 6 <case alt> **/
/* DIGRAPH NODE 6 */ case '2':
/** Segment 7 <case alt> **/
/* DIGRAPH NODE 7 */ case '3':
/** Segment 8 <case alt> **/
/* DIGRAPH NODE 8 */ case '4':
/** Segment 9 <case alt> **/
/* DIGRAPH NODE 9 */ case '5':
/** Segment 10 <case alt> **/
/* DIGRAPH NODE 10 */ case '6':
/** Segment 11 <case alt> **/
/* DIGRAPH NODE 11 */ case '7':
/** Segment 12 <case alt> ***/
/* DIGRAPH NODE 12 */ case '8':
/** Segment 13 <case alt> **/
/* DIGRAPH NODE 13 *\/ case '9':
/** Segment 14 <case alt> **/
```

```
                    tempresult = tempresult * 10 +
(in_str[char_index] - '0');
                    got_first = 1;
                    break;
                 default:
/** Segment 15 <case alt> **/
/* DIGRAPH NODE 14 */ if(chk_char(in_str[char_index])) {
/** Segment 16 <if> **/
                       return(tempresult);
                    }
                    else {
/** Segment 17 <else> **/
/* DIGRAPH NODE 15 */ if(char_index > 0 && got_first)
/** Segment 18 <if> **/
                          char_index--;
/** Segment 19 <implied else> **/
/* DIGRAPH NODE 16 */ while(char_index <= strlen(in_str)) {
/** Segment 20 <start while> **/
/* DIGRAPH NODE 17 */ if(chk_char(in_str[char_index]))
/** Segment 21 <if> **/\
                             break;
                          else
/** Segment 22 <else> **/
                             *bad_input++ = in_str[char_index];
                          char_index++;
                       }
/** Segment 23 <end while> **/
                       *bad_input = '\0';
                       printf("\t>>> bad input: %s\n", bad_str);
                       char_index++;
                       return(-1);
                    } } }
/** Segment 24 <end for> **/
             return(0);
/* DIGRAPH NODE 18 */ }
           int chk_char(ch)
           char ch;


/** Module chk_char **/

 {
/* DIGRAPH NODE 1 */
/** Segment 1 <> **/
/* DIGRAPH NODE 2 */
             if(isspace(ch) || ch == '\0')
/** Segment 2 <if> **/
                return(1);
```

```
          else
/** Segment 3 <else> **/
             return(0);
/* DIGRAPH NODE 3 */
       }
\------------------------------------------------------------
TCAT-PATH/C,  Release 8

END OF TCAT-PATH/C SEGMENT REFERENCE LISTING
------------------------------------------------------------------
```

10.4 Instrumentation Statistics

The instrumentor also produces program statistics. They are organized module-by-module.

```
------------------------------------------------------------
-- TCAT-PATH/C, Release 8.

--
-- (c) Copyright 1990 by Software Research, Inc.  ALL RIGHTS
RESERVED.
--
-- INSTRUMENTATION STATISTICS
--
-- Instrumentation date: Wed Jan 2 15:23:28 1991
--
------------------------------------------------------------
----
MODULE 'main':
               statements = 42
               compound statements = 7

               branching nodes = 12
               segments instrumented = 27

               conditional statements (if, switch) = 3
                   if statement = 1
                   else statement added = 1
                   switch statements = 2
                   switch statement cases = 14
                   default statement added = 0

               iterative statements (for, while, do) = 5
                   for statements = 2
                   while statements = 3
                   do statements = 0

               exit statement = 0
               return statement = 0

MODULE 'proc_input':
               statements = 22
               compound statements = 6

               branching nodes = 18
               segments instrumented = 24

               conditional statements (if, switch) = 4
                   if statements = 3
                   else statement added = 1
```

```
                    switch statement = 1
                    switch statement cases = 11
                    default statement added = 0
                iterative statements (for, while, do) = 3
                    for statement = 1
                    while statements = 2
                    do statements = 0

                exit statement = 0
                return statements = 3

MODULE 'chk_char':
                statements = 2
                compound statement = 1

                branching nodes = 3
                segments instrumented = 3

                conditional statement (if, switch) = 1
                    if statement = 1
                    else statement added = 0
                    switch statement = 0
                    switch statement case = 0
                    default statement added = 0

                iterative statements (for, while, do) = 0
                    for statements = 0
                    while statements = 0
                    do statements = 0

                exit statement = 0
                return statements = 2

-----------------------------------------------------------------
----
-- TCAT-PATH/C,  Release 8.

-- END OF TCAT-PATH/C INSTRUMENTATION STATISTICS
-----------------------------------------------------------------
```

10.5 Path Generation

The next step is to generate a complete set of paths for all modules of interest. **apg** processes a digraph file (*.*dig* file) into a path file (*.*pth* file). This path information is needed for generating a coverage report, which will be discussed in the next section.

The example program has three modules, and thus has three digraph files resulting from the instrumentation. The three digraph files are shown below:

```
# digraph for ‘main.dig'
        1   2   1
        2   3   2
        3   3   3
        3   4   4
        4   5   5
        5   4   6
        5   4   7
        5   4   8
        5   4   9
        5   4   10
        5   4   11
        5   4   12
        5   4   13
        5   6   14
        6   4   15
        6   4   16
        4   7   17
        7   8   18
        8   9   19
        9   10  20
        10  8   21
        9   8   21
        9   11  22
        11  8   23
        9   8   23
        9   8   24
        8   7   25
        7   2   26
        2   12  27
```

```
# digraph for 'proc_input.dig'
        1   2   1
        2   2   2
        2   3   3
        3   4   4
        4   5   5
        5   6   6
        4   6   6
        6   7   7
        4   7   7
        7   8   8
        4   8   8
        8   9   9
        4   9   9
        9   10  10
        4   10  10
        10  11  11
        4   11  11
        11  12  12
        4   12  12
        12  13  13
        4   13  13
        13  3   14
        4   3   14
        4   14  15
        14  18  16
        14  15  17
        15  16  18
        15  16  19
        16  17  20
        17  16  21
        17  16  22
        16  18  23
        3   18  24
```

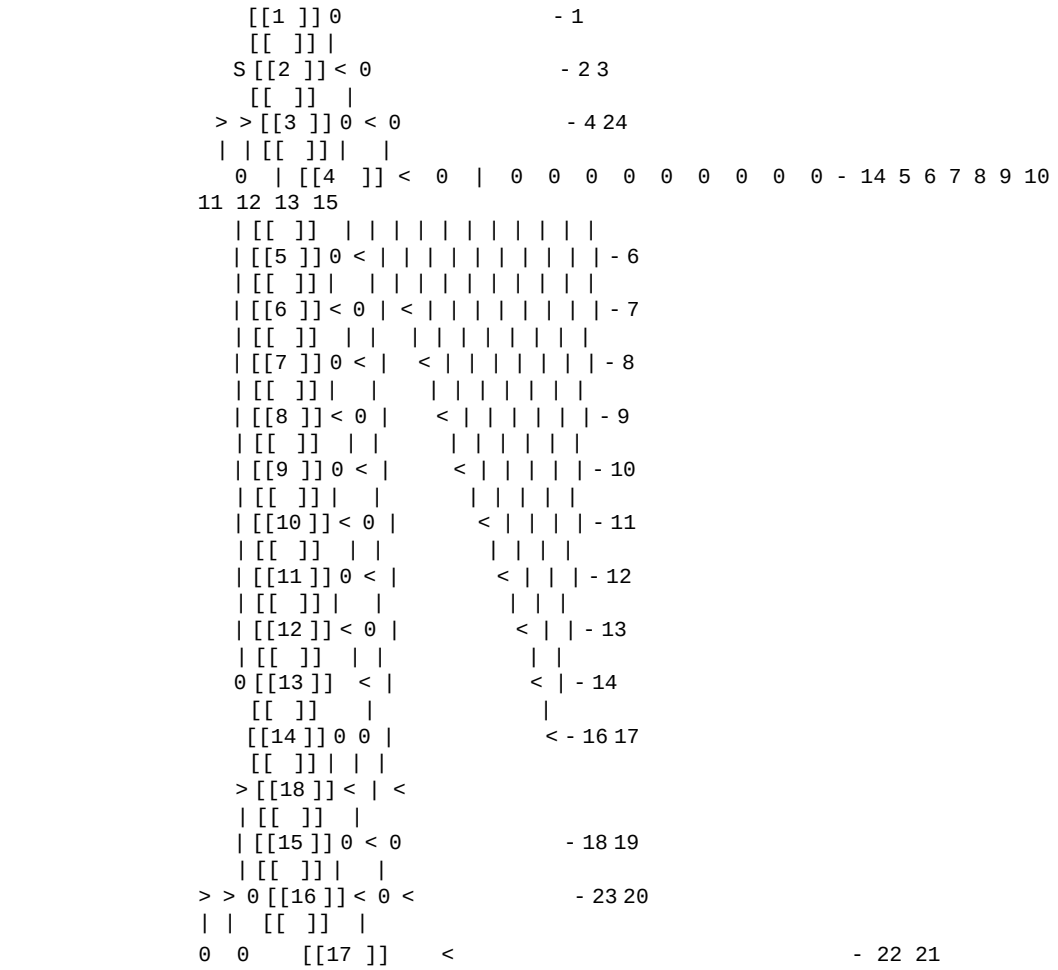
```
# digraph for ‘chk_char.dig'
       1   2   1
       2   3   2
       2   3   3
```

At this time you can also run cyclo and digpic on the digraph files and study the structures and properties of the modules in question. If any of the modules appears to be too “complex", you can break up the module into smaller and easier to test modules.

The cyclomatic numbesr for those three modules mentioned above are shown below:

```
Module main
cyclo  [Release 3]

Cyclomatic Number  =  Edges - Nodes + 2  =  29 - 12 + 2  =  19

cyclo  [Release 3]
Module proc_input

Cyclomatic Number  =  Edges - Nodes + 2  =  33 - 18 + 2  =  17

cyclo  [Release 3]
Module chk_char

Cyclomatic Number  =  Edges - Nodes + 2  =  3 - 3 + 2  =  2
```

The cyclomatic number for module *main* and proc_input is quite large. The digraph display of module *proc_input* below suggests that the module is quite complex.



apg generates more than 100 paths for both of the modules mentioned above. The paths are not reproduced here, but the user can refer to them in the next section.

10.6 TCAT-PATH Reports

The last and most important step in test analysis is to obtain test coverage analysis reports. This section details how to read reports generated by **ctcover**.

The commands on the following page are to be executed to get the coverage reports for all three modules.

```
ctcover main Trace.trc
ctcover proc_input Trace.trc(for UNIX)
ctcover chk_char Trace.trc

or

ctcover main Trace.trc
ctcover proc_input Trace.trc -f proc_inp(for DOS)
ctcover chk_char Trace.trc
```

The following are the coverage reports for all three modules from the example program. The reports for *main* and *proc_input* modules are intentionally truncated due to the the size of the reports.

10.6.1 Report for 'main' Module

```
Ct Test Coverage Analyzer
       (c) Copyright 1997 by Software Research, Inc.
Module "main": 155 paths, 1 were hit in 1 invocations.
  0.65% Ct coverage
Test descriptor: sample restaurant program run

HIT/NOT-HIT REPORT
------------------
P#   HitsPath text

1    None 1 2 3 <{ 3 }> 4 5 6 <{ 5 6 7 8 9 10 11 12 13 14 15 16
}> \
       17 18 19 20 21 <{ 19 20 21 21 22 23 23 24 }> 25 <{ 18 24
\\
       23 23 22 21 21 20 19 25 }> 26 <{ 2 3 4 16 15 14 13 12 11
\\
       10 9 8 7 6 5 17 25 19 20 21 21 22 23 23 24 18 26 }> 27
2    None 1 2 3 <{ 3 }> 4 5 6 <{ 5 6 7 8 9 10 11 12 13 14 15 16
}> \
       17 18 19 21 <{ 19 20 21 21 22 23 23 24 }> 25 <{ 18 24 23
23 \\
       22 21 21 20 19 25 }> 26 <{ 2 3 4 16 15 14 13 12 11 10 9 8
7 \
       6 5 17 25 19 20 21 21 22 23 23 24 18 26 }> 27
```

```
3    None 1 2 3 <{ 3 }> 4 5 6 <{ 5 6 7 8 9 10 11 12 13 14 15 16
}> \\
       17 18 19 22 23 <{ 19 20 21 21 22 23 23 24 }> 25 <{ 18 24
23 \
       23 22 21 21 20 19 25 }> 26 <{ 2 3 4 16 15 14 13 12 11 10
9 8 \
       7 6 5 17 25 19 20 21 21 22 23 23 24 18 26 }> 27
4    None 1 2 3 <{ 3 }> 4 5 6 <{ 5 6 7 8 9 10 11 12 13 14 15 16
}> \
       17 18 19 23 <{ 19 20 21 21 22 23 23 24 }> 25 <{ 18 24 23
23 \
       22 21 21 20 19 25 }> 26 <{ 2 3 4 16 15 14 13 12 11 10 9 8
7 \
       6 5 17 25 19 20 21 21 22 23 23 24 18 26 }> 27
5     1 1 2 3 <{ 3 }> 4 5 6 <{ 5 6 7 8 9 10 11 12 13 14 15 16
}> \\
       17 18 19 24 <{ 19 20 21 21 22 23 23 24 }> 25 <{ 18 24 23
23 \
       22 21 21 20 19 25 }> 26 <{ 2 3 4 16 15 14 13 12 11 10 9 8
7 \
       6 5 17 25 19 20 21 21 22 23 23 24 18 26 }> 27
       ...(intervening paths deleted for clarity)...

152  None 1 2 4 17 18 19 24 <{ 19 20 21 21 22 23 23 24 }> 25 <{
18 \
       24 23 23 22 21 21 20 19 25 }> 26 <{ 2 3 4 16 15 14 13 12
11 \
       10 9 8 7 6 5 17 25 19 20 21 21 22 23 23 24 18 26 }> 27
153  None 1 2 4 17 18 25 <{ 18 24 23 23 22 21 21 20 19 25 }> 26
<{ 2 \
       3 4 16 15 14 13 12 11 10 9 8 7 6 5 17 25 19 20 21 21 22
23 23 \
       24 18 26 }> 27
154  None 1 2 4 17 26 <{ 2 3 4 16 15 14 13 12 11 10 9 8 7 6 5 17
25 19 \
       20 21 21 22 23 23 24 18 26 }> 27
155  None 1 27
```

10.6.2 Report for 'proc_input' Module

```
Ct Test Coverage Analyzer
       (c) Copyright 1997 by Software Research, Inc.
Module "proc_input": 176 paths, 12 were hit in 12 invocations.
  6.82% Ct coverage
Test descriptor: sample restaurant program run

HIT/NOT-HIT REPORT
------------------
P#   HitsPath text

1    None 1 2 <{ 2 }> 3 4 5 6 7 8 9 10 11 12 13 14 <{ 4 5 6 7 8
9 \
       10 11 12 13 14 6 7 8 9 10 11 12 13 14 }> 24
2    None 1 2 <{ 2 }> 3 4 5 6 7 8 9 10 11 12 13 14 [{ 4 5 6 7 8
9 \
       10 11 12 13 14 6 7 8 9 10 11 12 13 14 }] 15 16
3    None 1 2 <{ 2 }> 3 4 5 6 7 8 9 10 11 12 13 14 [{ 4 5 6 7 8
9 \
       10 11 12 13 14 6 7 8 9 10 11 12 13 14 }] 15 17 18 20 21 \
           <{ 20 21 22 }> 23

       ...(intervening paths deleted for clarity)...

17   None 1 2 <{ 2 }> 3 4 7 8 9 10 11 12 13 14 <{ 4 5 6 7 8 9 10
11 \
       12 13 14 6 7 8 9 10 11 12 13 14 }> 24
18    1 1 2 <{ 2 }> 3 4 7 8 9 10 11 12 13 14 [{ 4 5 6 7 8 9 10
11 \
       12 13 14 6 7 8 9 10 11 12 13 14 }] 15 16
19   None 1 2 <{ 2 }> 3 4 7 8 9 10 11 12 13 14 [{ 4 5 6 7 8 9 10
11 \
       12 13 14 6 7 8 9 10 11 12 13 14 }] 15 17 18 20 21 <{ 20
21 22 }> 23

       ...(intervening paths deleted for clarity)...

25   None 1 2 <{ 2 }> 3 4 8 9 10 11 12 13 14 <{ 4 5 6 7 8 9 10 11
12 \\
       13 14 6 7 8 9 10 11 12 13 14 }> 24
26    1 1 2 <{ 2 }> 3 4 8 9 10 11 12 13 14 [{ 4 5 6 7 8 9 10 11
12 \\
       13 14 6 7 8 9 10 11 12 13 14 }] 15 16
27   None 1 2 <{ 2 }> 3 4 8 9 10 11 12 13 14 [{ 4 5 6 7 8 9 10 11
12 \\
       13 14 6 7 8 9 10 11 12 13 14 }] 15 17 18 20 21 <{ 20 21
22 }> 23

       ...(intervening paths deleted for clarity)...
```

```
167  None 1 3 4 14 [{ 4 5 6 7 8 9 10 11 12 13 14 6 7 8 9 10 11 12
13 \\
       14 }] 15 17 19 20 22 <{ 20 21 22 }> 23
168  None 1 3 4 14 [{ 4 5 6 7 8 9 10 11 12 13 14 6 7 8 9 10 11 12
13 \\
       14 }] 15 17 19 23
169     1 1 3 4 15 16
170  None 1 3 4 15 17 18 20 21 <{ 20 21 22 }> 23
171  None 1 3 4 15 17 18 20 22 <{ 20 21 22 }> 23
172  None 1 3 4 15 17 18 23
173  None 1 3 4 15 17 19 20 21 <{ 20 21 22 }> 23
174     1 1 3 4 15 17 19 20 22 <{ 20 21 22 }> 23
175  None 1 3 4 15 17 19 23
176     1 1 3 24
```

10.6.3 Report for 'chk_char' Module

```
Ct Test Coverage Analyzer
       (c) Copyright 1997 by Software Research, Inc.
Module "chk_char": 2 paths, 2 were hit in 20 invocations.
100% Ct Coverage!
Test descriptor: sample restaurant program run

HIT/NOT-HIT REPORT
------------------
P#   HitsPath text

1      11 1 2
2       9 1 3
```

10.7 Summary

After reviewing the coverage reports you will typically rerun the tests with different or additional test cases, designed to exercise previously not-hit paths and achieve a higher *Ct* value. The higher the *Ct* value, the more complete your testing. When you achieve a satisfactory value for *Ct*, you can stop testing.

CHAPTER 11

Understanding the Graphical User Interface (GUI)

This chapter demonstrates using *TCAT-PATH* in the OSF / Motif X Window System environment.

11.1 Invocation

To invoke, type:

```
Xtcatpath
```

The result is the main menu (shown below). This window has a window menu button (available for all windows) that allows the user to restore, move, size, minimize, lower and close the window. This menu button can be used at any time during the X Window System program. For closing main application windows, however, it is best to use the “System" menu's “Exit" option to prevent any system crashes. The two buttons in the upper right hand corner of the window allow the user to maximize or minimize the window size.

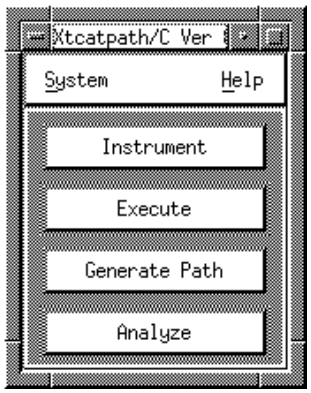


FIGURE 5 Main Menu

To invoke with *STW/Coverage*, click first on Coverage and then on *TCAT-PATH*. The *TCAT-PATH* main menu pops up.

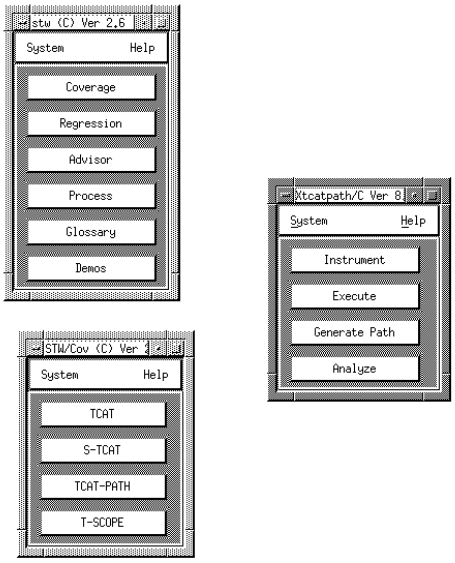


FIGURE 6 STW/Coverage Invocation

11.2 Using TCAT-PATH

For first time use, always read the help menus. Below is main menu's help, explaining *TCAT-PATH* four stages of testing: **Instrument**, **Execute**, **Generate Path**, and **Analyze**.

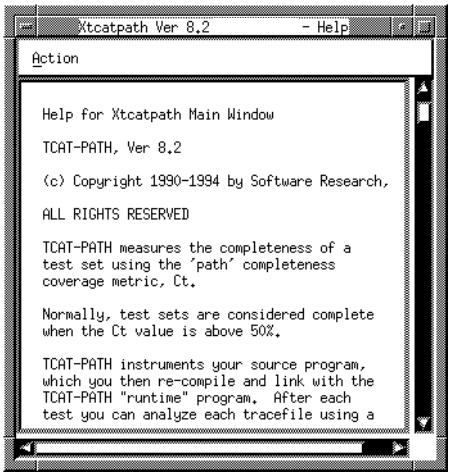


FIGURE 7 Main Menu Help

11.2.1 Instrumentation

TCAT-PATH instruments the source code of the program to be tested, that is it inserts function calls at each logical branch. Click twice on **Instrument** in order to begin testing.

There are a variety of options which can be selected with the menu in Figure 8 on page 127:

- **Preprocessing** can be turned on or off. If it is turned off, then the instrumentor will not preprocess.
- **Preprocessor output suffix** is set to **.i**, which is normally the extension for preprocessed ‘‘C" programs. This option is user editable.
- **Preprocessor Command** is set to **cc -P**. Refer to Chapter 9 for further information. This option is user editable.
- **Preprocessor options** are options in addition to the ‘‘Preprocessor command" previously specified.
- **Instrumentor Command** is set to **tp-ic**. This option is user editable.
- **Instrumentor options**
  - **Recognize _exit as keyword** corresponds to the command line **-u** switch. Refer to Section 2.2.2 on page 11.
  - **Do not recognize exit as keyword** corresponds to the command line **-x** switch. Refer to Section 2.2.2 on page 11.
  - **Do not instrument functions in file** corresponds to the **-DI** deinst switch. Specify a filename that contains lists of modules that are to be instrumented. Refer to Section 2.2.2 on page 11.
  - **Specify maximum file name length** corresponds to the **-f1** value switch. Specify a number that will correspond to the maximum number of characters. Refer to Section 2.2.2 on page 11.
  - **Specify maximum function name length** corresponds to the **fn** value switch. Specify a number that will correspond to the maximum number of characters. Refer to Section 2.2.2 on page 11.

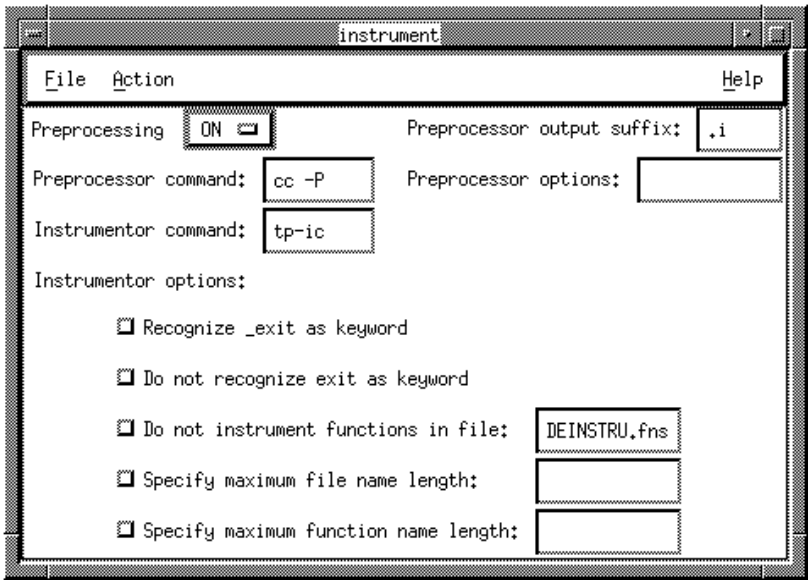


FIGURE 8 Instrument Menu

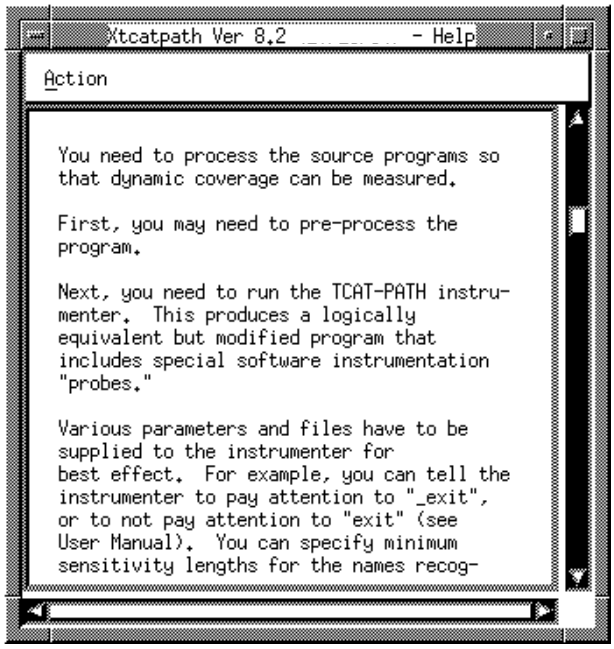


FIGURE 9 Instrument Help Menu

After selecting instrumentor options, do the following:

1. Make sure the **Preprocessing** switch is ON.
2. Click on the **File** pull-down menu. Drag the mouse down and select **Set File Name**. A file pop-up window appears. (Refer to the picture below.) Select the file to be instrumented by either highlighting or typing it into the Selection Box. Press **OK**.
3. After establishing the file to be instrumented, click on the **Action**pull-down menu. Drag the mouse down and select **Preprocess** and then **Instrument**.

NOTE: Instrument cannot be selected until preprocessing has been completed. When both preprocessing and instrumenting are in progress, the menu's optionsare grayed out and the cursor becomes a stop watch. When the options are darkened, then you can progress to the next step.

NOTE: Current status and errors are displayed in the invocation box from time to time. Frequently refer to the box while testing to see where system crashes, errors and passes occur.

When finished, click on **Exit** on the **File** pop-up menu.

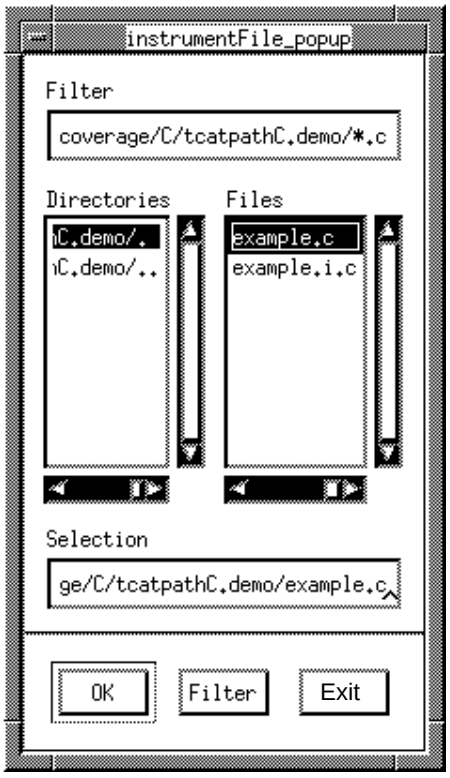


FIGURE 10 File Pop-Up Menu

11.2.2 Execute

The **Execute** menu compiles, links and executes the program. Normally, you compile the instrumented source file and then link all the source files with the runtime object module (which is specified under the **File** pull-down menu). The user can also use the Make file. Both methods are explained in this section.

Click once on **Execute** to begin. The menu below appears.

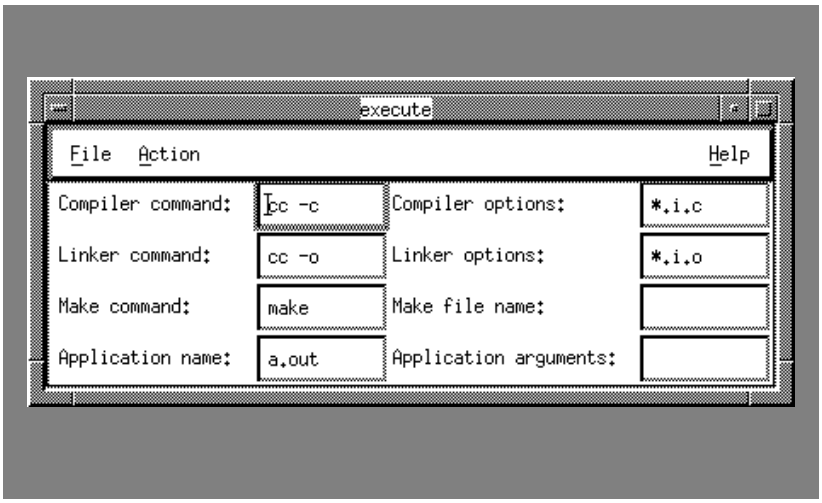


FIGURE 11 Execute Menu

There are a variety of options which can be selected from the **Execute** menu.

1. "Compiler command" is used to invoke the compiler on the system. It is set to **cc-c** but is user-editable.
2. "Compiler options" are the options for the compiler. It is set to ***.i.c** but is user-editable.
3. "Linker command" is used to invoke link. It is set to **cc-o** but is user-editable.
4. **Linker options** are the options used in order to link. It is set to ***.i.o** but is user-editable.
5. **Make command** is used to invoke the **make** utility.
6. **Make file name** is where the make file is specified. It is fixed to **Makefile** but is user-editable.
7. **Application name** is the command used to invoke the instrumented program. It is fixed to **a.out** but is user-editable.

8. **Application argument** is where command line arguments are specified. It is user-editable.

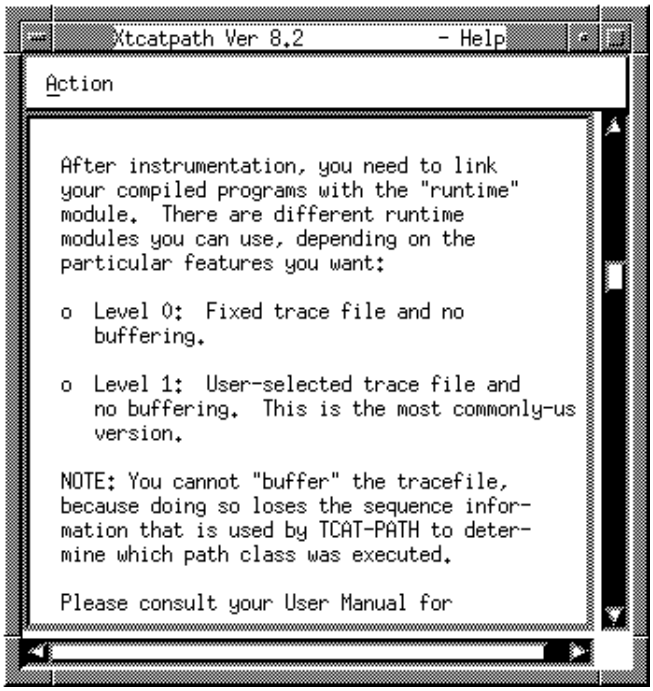


FIGURE 12 Execute Help Menu

Execute one of two ways:

1. Without Make File

   Click on the ‘‘File'' pull-down menu, drag the mouse to ‘‘Set Runtime Object Module'' and click. A pop-up window appears (shown in Figure 13 on page 133).

   - Highlight or type in (the Selection box) the necessary file. Click **OK**. Refer to the help frame and to Section 3.1 on page 24f or SR-supplied runtime object modules.
   - Set the compiler and linker commands (that is **Compiler command**, **Compiler options**, **Linker command** and **Linker options**) as appropriate.
   - Click on the **Action** pull-down menu and select **Compile**. When completed, the invocation window will state so.
   - Click on **Link**. Invocation window will indicate when linking has occurred.
   - Click on **Run Application**.

2. With Make File: make organizes all compiler and linker commands and files.
   - Click on the **File** pull-down menu, drag the mouse to **Set Runtime Object Module** (shown in Figure 13) and click.
   - Highlight or type in (the Selection box) the necessary file. Click **OK**. Refer to the help frame and to Section 3.1 on page 24 for SR-supplied runtime object modules.
   - Set the make commands (that is **Make Command**, **Make file name**, **Application name** and **Application arguments**) as appropriate.
   - Click on the **Action** pull-down menu and select **Make**. When completed the invocation window will state so.
   - Click on **Run Application**.

Whichever methods is chosen, the trace file is created.

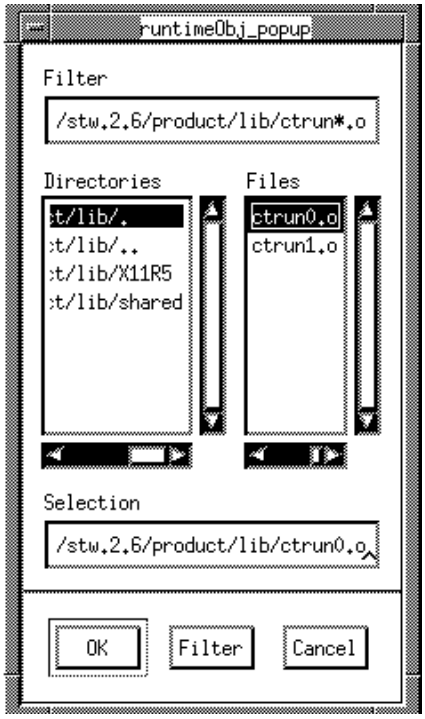


FIGURE 13 Runtime Object Module Pop-Up Menu

11.2.3 Generate Paths

After executing your program, you need to generate a set of paths for any module. **apg** processes a digraph file (***.dig** file) into a path file (***.pth** file). This path information is necessary for generating a coverage report.

To begin, click once on **Generate Path** and this menu appears:

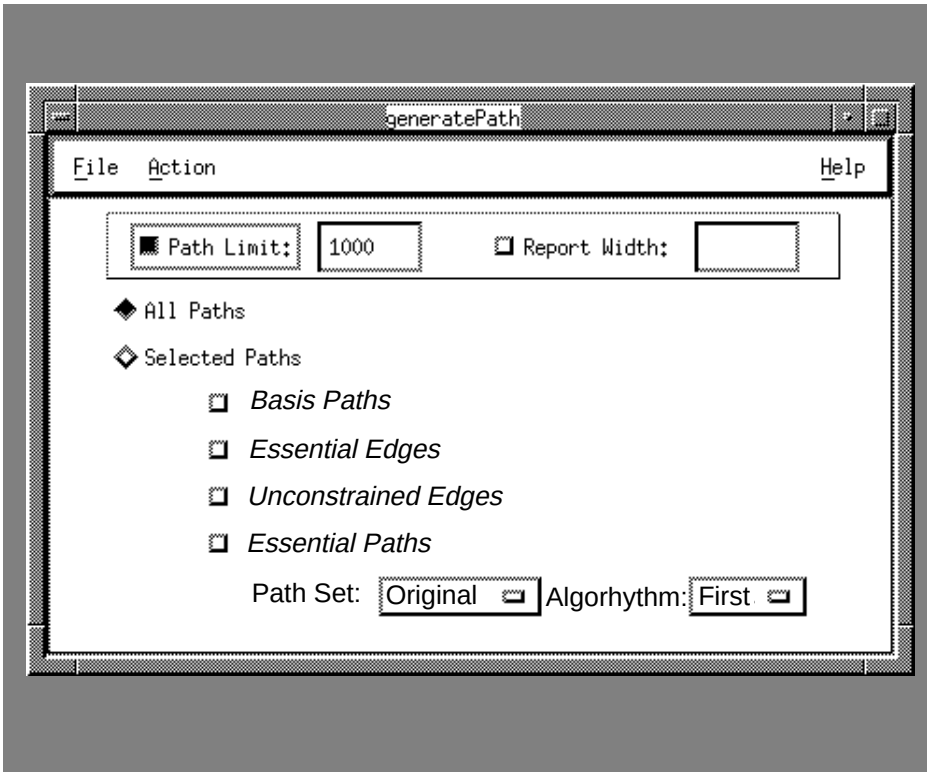


FIGURE 14 Generate Paths Menu

11.2.3.1 Help

To access a help screen for Generating Paths, select the Help menu. The following screen will appear:

11.2.3.2 "Generate Paths" Help Frame

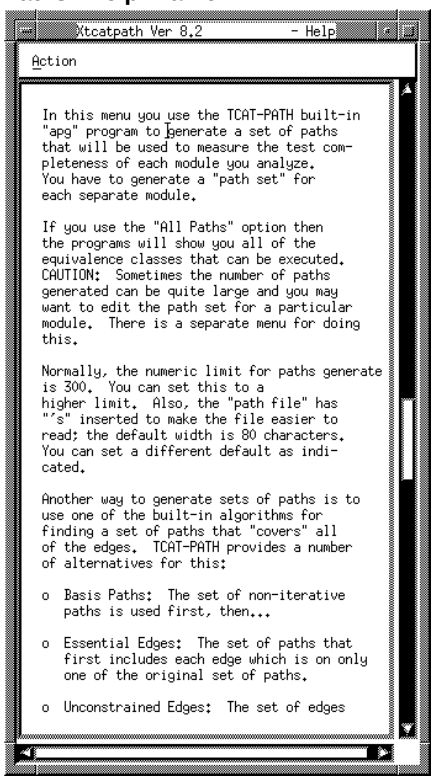


FIGURE 15 Generate Paths Help Frame

There are a variety of options which are available from the **Generate Path** menu:

1. **Path Limit** is the the (integer) maximum number of paths to generate. It corresponds to the command line [**-p limit**] option. Refer to Section 4.1.1 for further information.
2. **Report Width** specifies that the report is never to be wider than width characters. It corresponds to the command line [**-w width**] option. Refer to Section 4.1.1 for further informations.
3. **All Paths** are all the structurally visual paths. It is equivalent to running apg on a ***.dig** file. Refer to Chapter 4 for further information.
4. **Selected Paths** selects paths from **All Paths**. Because **All Paths** can be overwhelmingly large, you may want to select only particular paths from the **Selected Paths** option. The following paths may be selected:
    - **Basis Paths**: The set of non-iterative paths. It corresponds to the apg's **-b** command line switch. See Section 4.1.1 for further information).

- **Essential Edges**: The set of paths that first includes each edge which is on only one of the original set of paths. It has not been implemented at this time.
- **Unconstrained Paths**: The set of edges that will imply execution of other edges in the program. It has not been implemented at this time.
- **Essential Paths**: The set of paths that include one essential edge, that is an edge that lies on no other path.

  •**Path Set** and **Algorithm**: Paths can be arranged in the following ways:

  Original refers to the original path set that was generated by apg. (This is accomplished when you select **All Paths** from the **Generate Paths** menu. It corresponds to the**-f** (first found alogorithm) or the **-l** (last found algorithm) in pathcover. Refer to Section 4.1.1 “apg” on page 31 for further information.

  Iteration arranges the path set in terms of increasing iteration complexity.

  It corresponds to the **-fi** (first found alogorithm) or the **-li** (last found algorithm) in pathcover. Refer to Section 4.1.1 “apg” on page 31 for further information.

  Length arranges the path set in ascending order. It corresponds to the **-fl** (first found alogorithm) or the **-ll** (last found algorithm) in pathcover. Refer to Section 4.1.1 “apg” on page 31 for further information.

  Sorted arranges the path set in natural order according to the names of the segments. It corresponds to the **-fs** (first found alogorithm) or the **-ls** (last found algorithm) in pathcover. Refer to Section 4.1.1 “apg” on page 31 for further information.

  Random arranges the path set in random order. It corresponds to the **-r** switch. First found and last found algorithms are ignored. Refer to Section 4.1.1 “apg” on page 31 for further information.

1. To generate **All Paths**:
   - Click on the **All Paths** option.
   - Specify the **Path Limit** and **Report Length** if desired.
   - Click on the **File** pull-down menu and click on **Set Module Name**. The menu below pops up. Highlight or type in the module in the **Selection Box** and click on **OK**.

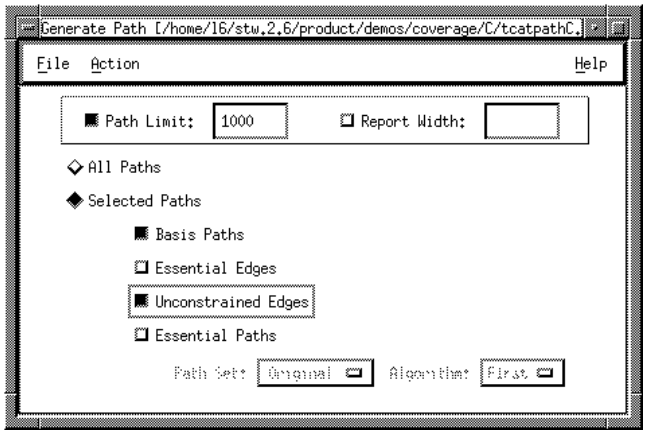


FIGURE 16 Generate Paths Pop-Up Menu

   - Click on the "Action" pull-down menu. Drag the mouse to "Generate Paths" and click.
   - Click on the "Action" pull-down menu. Drag the mouse to "Generate Path Statistics" and the menu below pops up. View the reports by using the menu's scroll bars. After viewing, click on "Action" and "Exit".

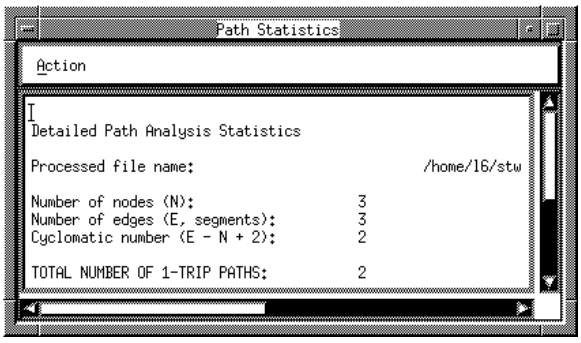


FIGURE 17 Generate Path Statistics Pop-Up Menu

At this time, you can use other available utilities with the **Action** pull-down menu. These utilities are optional, not necessary.

Click on **Edit Paths** and the window below pops up.

Note: If you do not use the ‘‘Selected Paths" option, then the ‘‘All Path List" and the ‘‘Selected Path List" scrolled text windows will contain the same paths.

11.2.3.3 "Edit Paths" Menu

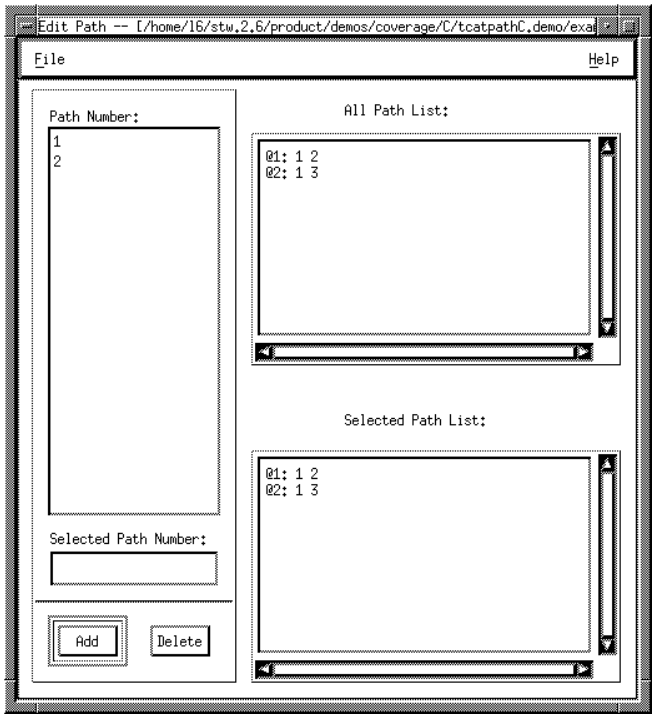


FIGURE 18 Edit Paths Menu

11.2.3.4 "Edit Paths" Help Frame

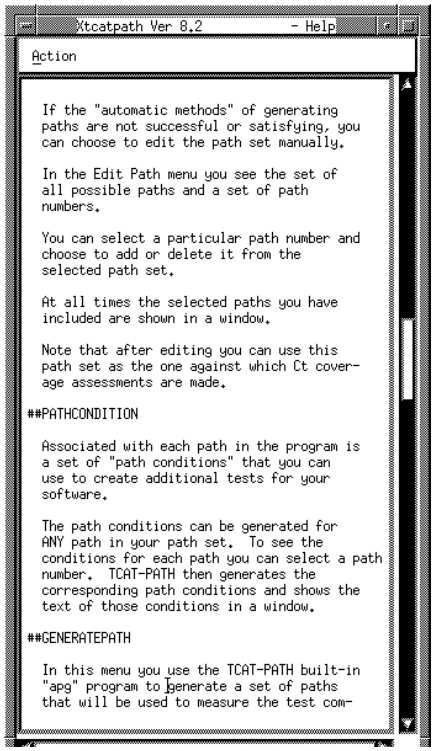


FIGURE 19 Edit Paths Help Frame

11.2.3.5 "Set Path" File Pop-up Menu

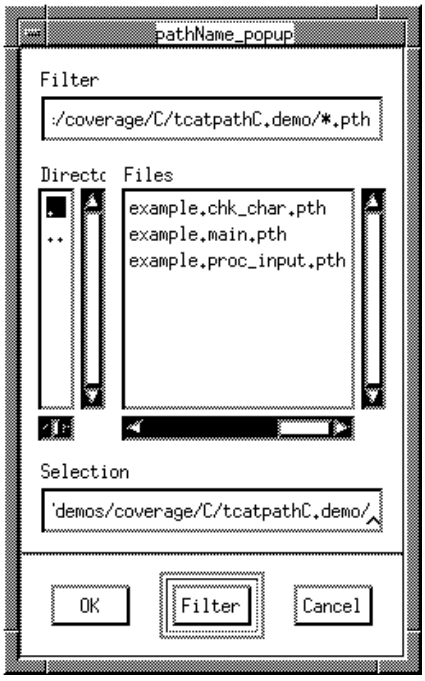


FIGURE 20 Set Path File Pop-Up Menu

1. Your module name should be carried over from the Generate Paths menu. If not or to select a different module (assuming you have already generated paths for it), then click on the "File" pop-up menu and select "Set Path File". A window similar to the one in Figure 34 pops up. Select a file by highlighting or typing in the path (*.pth) file.
2. To add or delete a path in the "Edit Paths" menu, simply type in or highlight the number in the "Select Path Number" Selection Box.
3. Click "Add" or "Delete" and the "Selected Path List" will change according. (See the note under1).
4. If you wish to save the path (*.pth) file, then select the "Saved to New Path File" under the "Action" pop-up menu. A window like the one below will appear. Select a file in the usual manner.

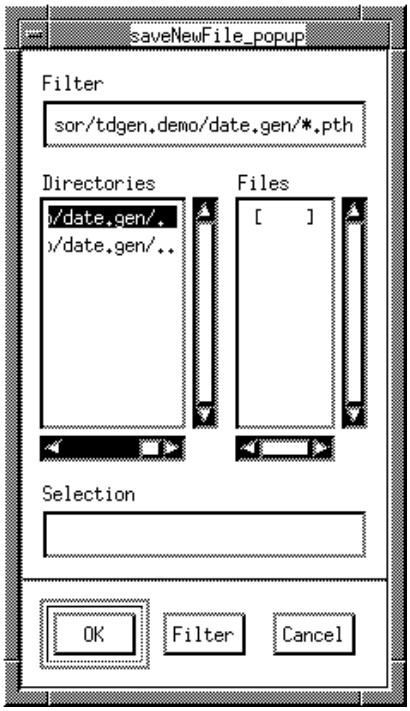


FIGURE 21 Save New Path File Pop-Up Menu

Click on "Display Paths" and the window below pops up. It allows you to view source. For further information, see Chapter 7, **SOURCE VIEWING**.

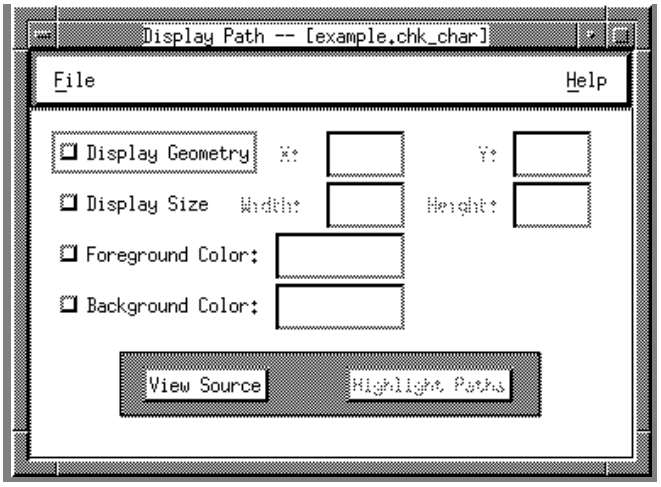


FIGURE 22 Display Path Menu

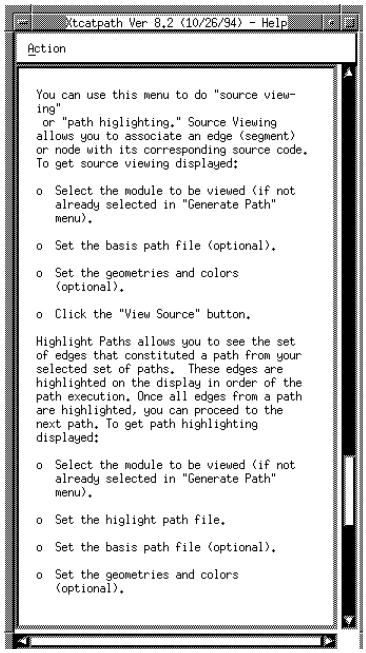


FIGURE 23 Display Path Help Frame

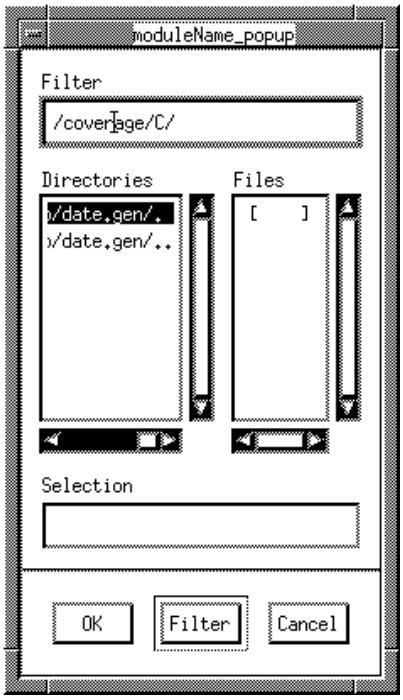


FIGURE 24 Set Module File Pop-Up Menu

- Select the module to be viewed (if not already selected in the Generate Path menu). Do so by clicking on the **File** pull-down menu's **Set Module Name**. Choose the file in the usual manner.
- **Set Basis Path File** under the **File** pop-up menu, if necessary. The basis path file establishes the set of nodes that appear on the vertical axis.
- To choose where to geometrically view source, select the x and y coordinates with the **Display Geometry** option. This is optional. Click first on the button and then type in the desired coordinates' positions. If not used, the display will pop up based on the default established for *T-SCOPE*'s (Test Data Observation and Analysis Tool) **Xdigraph** syntax.
- The display's width and height can be selected from the "Display Size" option. Click first on the button and then type in the desired width and height. If not used, the display will pop up based on the default established for *T-SCOPE* (Test Data Obervation and Analysis Tool) .

- You can also choose the foreground and background colors with this menu.
- After making selections, click on **View Source**. Based on your selections or the defaults, the module's display pops up.
- If the display is not the size you want or placed not where you want, you can resize or move as needed.
- Source view by clicking on a node or a segment and holding down the mouse button.
- When finished, press any key, and the display is deleted.

NOTE: **Highlight Paths** is used only with **Selected Paths**.

- To exit from the **Display Path** menu, click on **Exit** under the **File** pull-down menu.

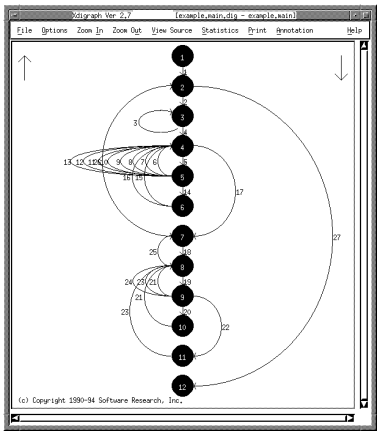


FIGURE 25 Source Viewing

‘‘Generate Path Condition" is the other option. Click on it, and the menu below pops up. It extracts and displays the logical conditions for a particular path given the sequence of segments in the path (which could be a complete path), the digraph file (*.dig), and the reference listing file (*.i.A or *.iA). See Section 7.3 on page 81 for further information.

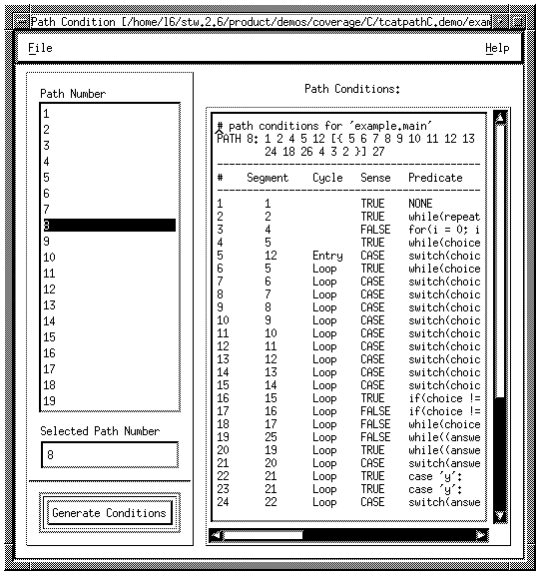


FIGURE 26 Path Condition Menu

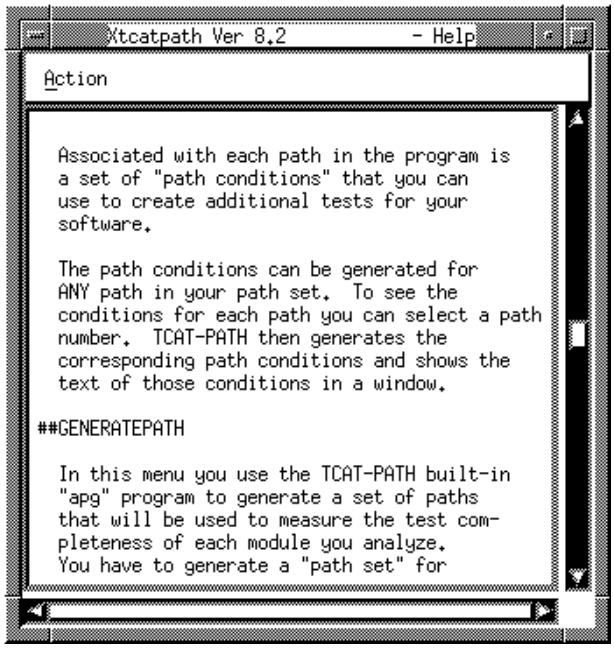


FIGURE 27 Path Condition Help Frame

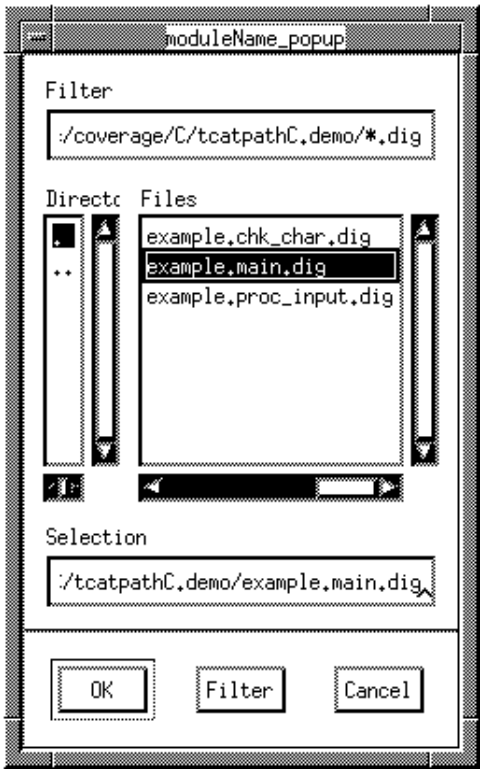


FIGURE 28 Set Module File Pop-Up Menu

- Your module name should be carried over from the **Generate Paths** menu. If not or to select a different module (assuming you have already generated paths for it), then click on the **File** pop-up menu and select **Set Module**. A window like the one in Figure 28 pops up. Select a file in the usual way.
- Select a path number by either clicking (and, thus, highlighting) the number or typing in a number in the **Selected Path Number Box**.
- Click **Generate Conditions**.
- *TCAT-PATH* then generates the corresponding path conditions and shows the text of those conditions in the **Path Conditions** scrolled text window.
- To view the text, use the scoll bars.

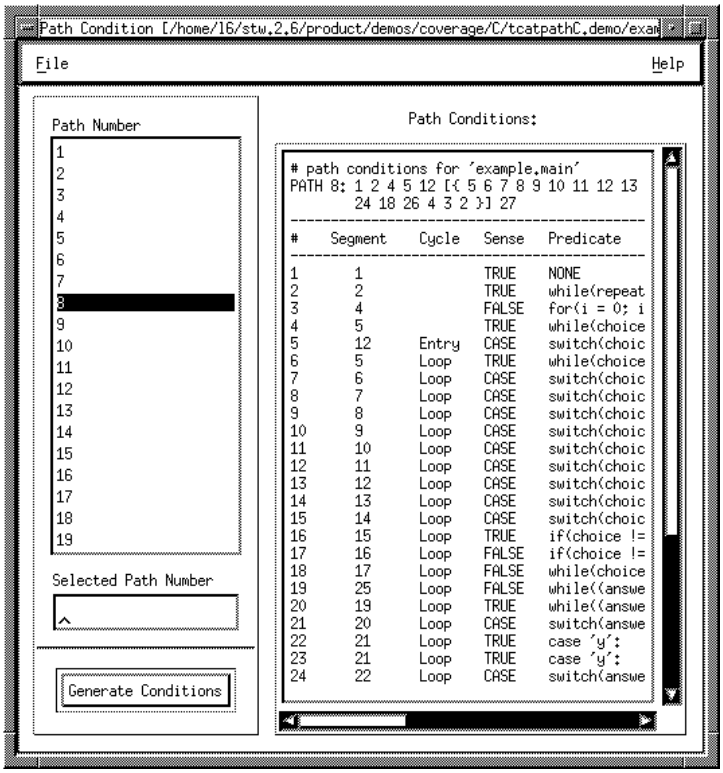


FIGURE 29 Path Condition Menu

The path conditions will automatically be saved to **<module_name>.con.#**, where # corresponds to the module number. If you wish to save the file to a different **pathcon** file, click on the **Save to Path Conditions File**. A window similar to the one below pops up. Select a file in the usual manner.

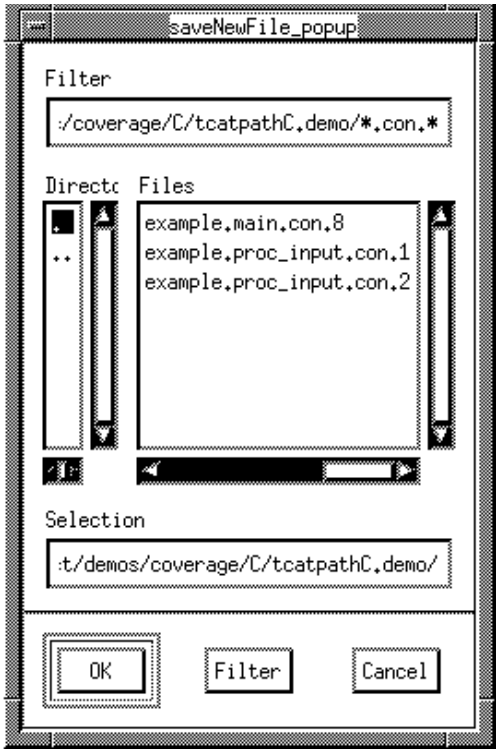


FIGURE 30 Save New Pathcon File Pop-Up Menu

To generate **Selected Paths**:

NOTE: Because **Selected Paths** is very similar to **All Paths**, this section will be in summary form.

---

1. Click on the **Selected Paths** option.
2. Select any or all of the paths and arrange the **Path Set** to your specifications.

NOTE: **Essential Edges** and **Unconstrained Edges** have not been implemented at this time.

---

3. Specify the **Path Limit** and **Report Length**, if desired.
4. If you haven't already set the module name, click on **Set Module Name**, then do so now.

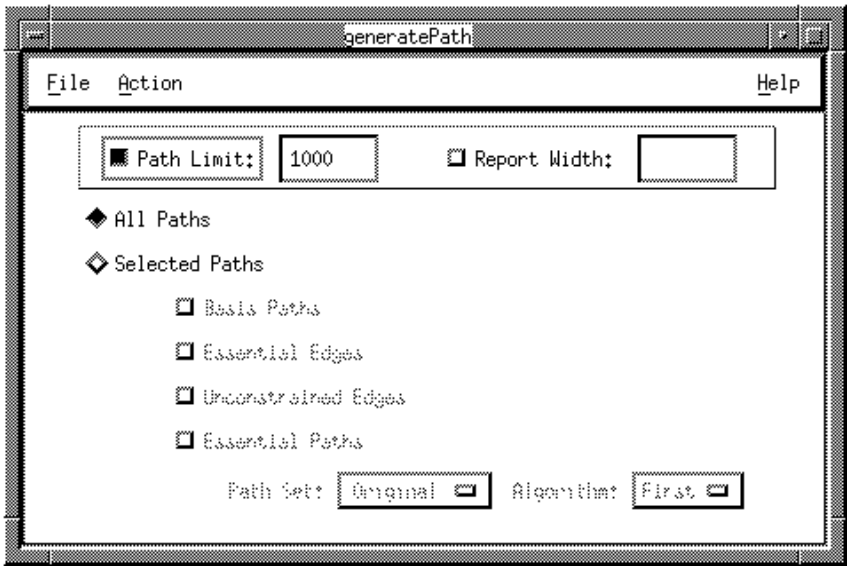


FIGURE 31 Generate Path Statistics Pop-Up Menu

- Click on the **Action** pull-down menu. Drag the mouse to **Generate Paths**. Then select **Generate Path Statistics**. When generated, **Selected Paths** will automatically generate path statistics for **All Paths**, whether you generated **All Paths** or not.
- The available options are the same as **All Paths**.
  - **Edit Paths** : All additions and deletions appear in the **Selected Path List** scrolled text windows.

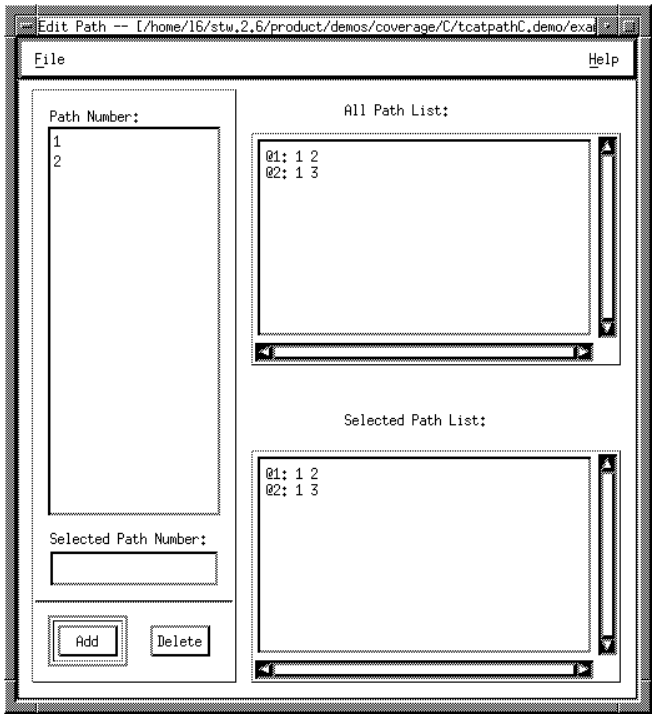


FIGURE 32 Edit Paths Window

- **Display Paths** generates for **All Paths**, whether you selected **All Paths** or **Selected Paths**. Source view the same way youwould for **All Paths.**
  - **Selected Paths**, however, allows you to highlight particular edges. To activate, select **Set Highlight File** from the **File** pull-down menu. Select the file (***.pth** file) in the usual manner. See the Figure on the following page.

Note: The module name and the highlight file name must be from the same module.

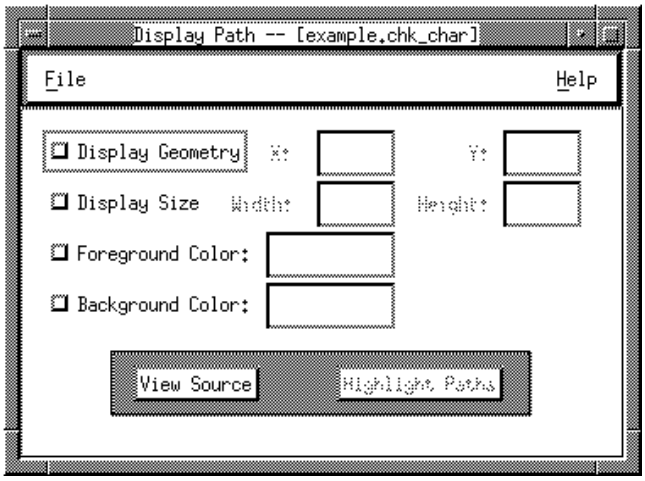


FIGURE 33 Display Paths Menu

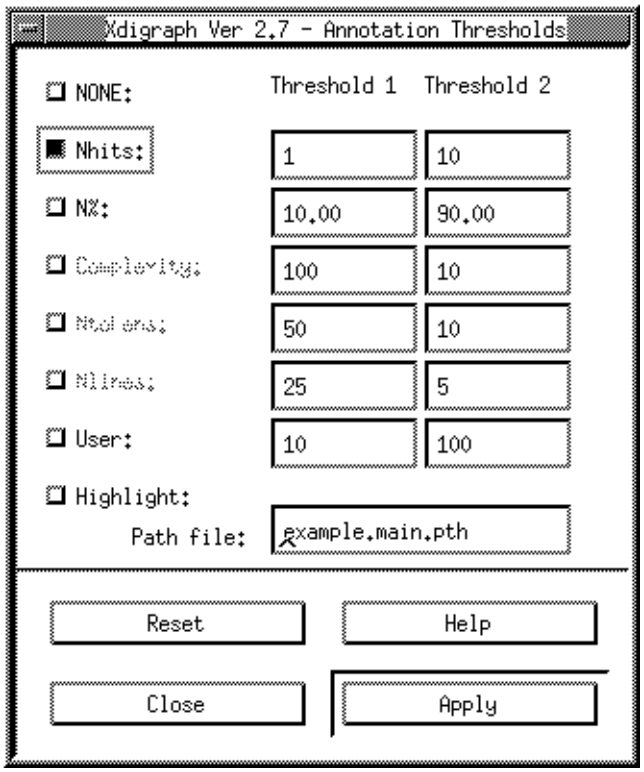


FIGURE 34 Set Highlight File Pop-Up Menu

- After selecting the highlight file, click on **Highlight** and the appropriate path(s') edges are highlighted in the display.
- If you have more than one path in your file click on the mouse button and the next highlighted path is displayed.
- When finished, press any key and the display will disappear.

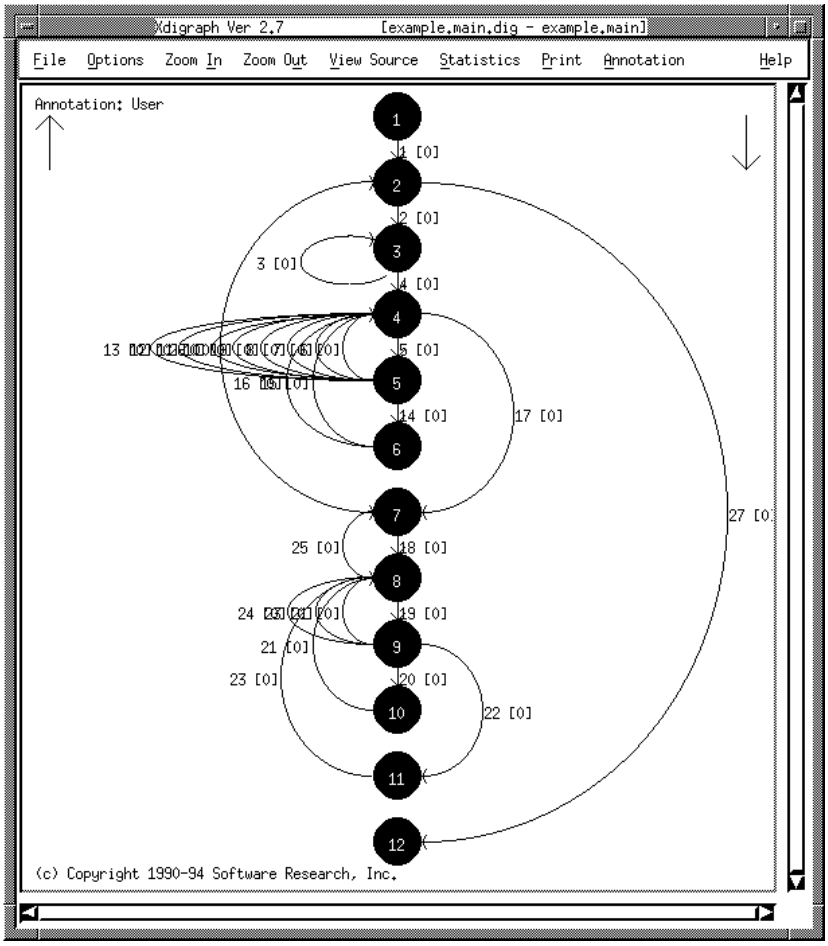


FIGURE 35 Highlighted Path Display

11.2.4 Analyze

After generating paths, you can analyze the trace file using the **ctcover** command. Click on **Analyze** and the menu below pops up.

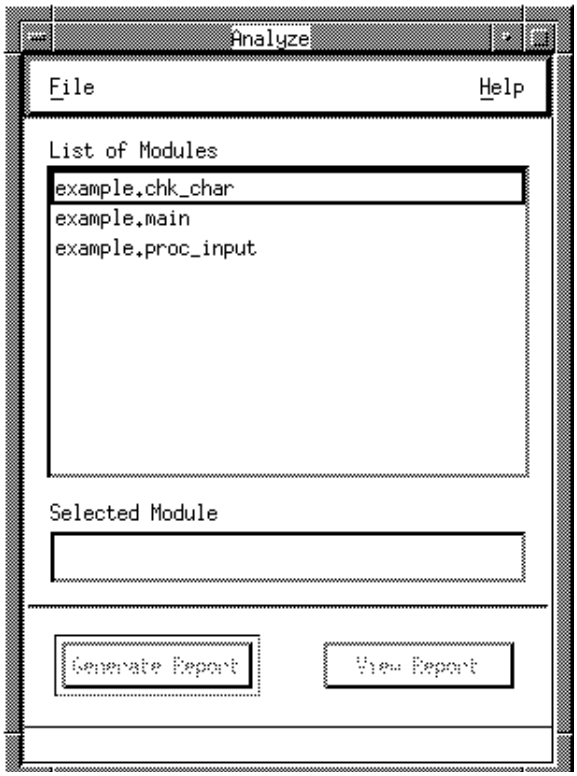


FIGURE 36 Analyze Menu

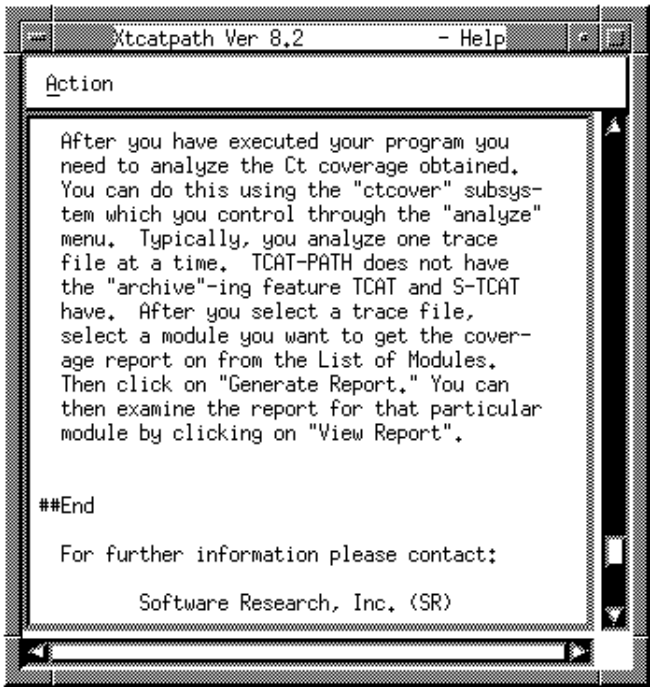


FIGURE 37 Analyze Help Frame

To use:

1. Click on the **File** pull-down menu and select **Set Trace File**. A pop-up window like the one below appears. Highlight or type in the file of your choice.

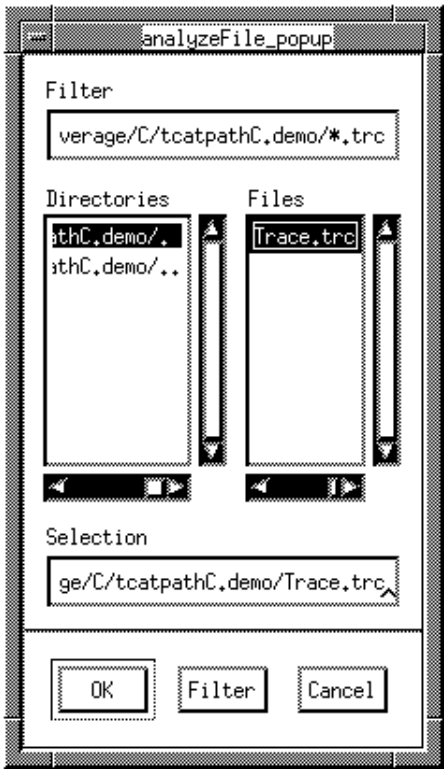


FIGURE 38 Set Trace File Pop-Up Menu

2. Select the module. This accomplished by clicking on the module (and, thus, highlighting it) or manually typing in the module.
3. Click on **Generate Report**.
4. Click on **View Report**.
5. You can view the report by using the scoll bars.
6. When finished, select **Exit** under the **Action** menu.

   At this point, you have successfully used *TCAT-PATH*.

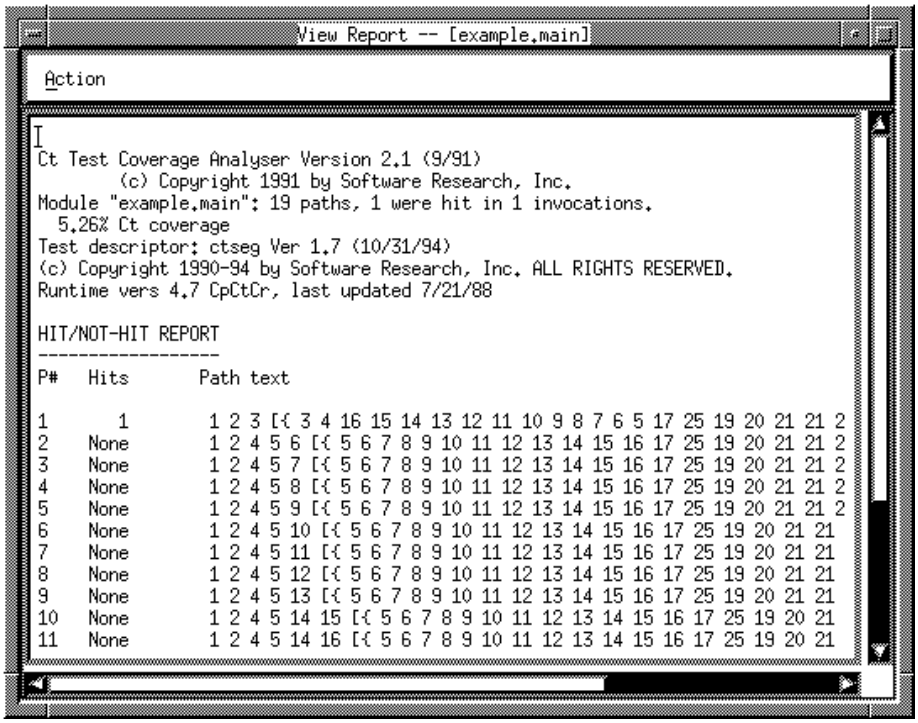


FIGURE 39 View Report Window

CHAPTER 12

System Restrictions and Dependencies

It is important to recognize that *TCAT-PATH* can only be used with “legal" programs. Non-legal constructions may pass through *TCAT-PATH*, but results cannot be guaranteed.

The *TCAT-PATH* package can measure very complex programs. In some cases, however, programs are so complex that analysis of them will be too time consuming or will require too much execution space.

TCAT-PATH has certain pre-defined limits to prevent “overload" of the system components. An example of such limits is the following set, defined for the language. Other limits may be in effect for other languages.

- **tp-i<lang>** gives up processing beyond a threshold number of program edges or nodes. This limit is defaulted at 5000 nodes per invocation.
- **apg** has limits on the total number of paths emitted,
- and on the total number of paths computed without being printed. This threshold is defaulted at 300 printed paths (or 4800 computed paths).
- **ctcover** has limits on the total number of records processed (after which it ceases processing paths). This threshold is defaulted at 100,000 segments per call. Also, path processing is memory limited; an error message is issued in case the limits are exceeded.
- ctcover analysis system allocates memory dynamically and can run out of memory. When it does it indicates when, and what caused the overload. The stack sizes within the system are chosen to represent
- a capacity that should not be exceeded in practice, except for extremely complex (or intentionally complex) programs.

Certain restrictions exist in *TCAT-PATH* instrumentor (language processor). They are summarized here.

"C" Language: tp-ic

- The function names **EntrMo**,**ExtMod**,**SegHit**, **Strace** , and **Ftrace** are reserved for the runtime calls.
- The instumentor (**tp-ic**) can take identifiers (function or variable names) that are up to 128 characters long.
- Conditional expressions in "C" (of the form "expr ? expr : expr") are not supported; they must be expanded to the explicit "if...[else]..." form.
- The **tp-ic** language analyzer in *TCAT-PATH* does not support switch statement instrumentation in exactly the same way as does TCAT. The difference is due to special handling of empty "case:" statements. Generally, *TCAT-PATH* is a more complete model of program flow.
- Conditional expressions are not processed by *TCAT-PATH*. Conditional expressions should be converted at the source level to simple "if ... else" statements, which will have the same effect and which are processed by *TCAT-PATH*.
- For various reasons, "goto" statements are not processed by *TCAT-PATH*; their presence in a program could cause misunderstandings about Ct coverage.

Ada Language: tp-iada

No restrictions exist for processing of Ada programs.

FORTRAN Language: tp-if77

FORTRAN statements such as ASSIGN and GOTO-ASSIGN are not supported.

CHAPTER 13

On-Line Help Frames

The interactive mode of *TCAT-PATH* provides the user with an on-line help frame facility.

From any interactive mode menu, you can obtain help by typing:

```
help
```

or

```
help?
```

or

```
help <command-name>
```

TCAT-PATH responds by showing the user a screen of data describing how to use the selected commands. The available help frames are shown on the following pages.

Note: the actual help frames will vary slightly with the particular version of the *TCAT-PATH* system that you have. This is done to ensure that the on-line assistance exactly matches that needed for your system.

##tcatpath.h00

```
                                                                        HELP
     Usage:  help [opt]

      Help is available for the following commands and categories.
Substitute |
    any of the words below in place of [opt] to get its help screen.
    Abbreviations are acceptable, as long as they are not ambiguous.

          apg                rcfile             !
          ctruntime          release            !!
          cyclo              save
         digpic             settings
          digraph            tcatpath
          exit              terminology
          menu               trace file
```

##tcatpath.h01

```
 > TCAT-PATH -- Path Test Coverage Analysis Tool                    HELP
>>> General Information

    TCAT-PATH provides commands that measure the pat coverage
     of instrumented programs.

     TCAT-PATH commands can generate a program digraph, can generate a
full set of equivalence classes of flow (the
     path set), can instrument a program, and can measure
     how many paths are executed in a test that involves one
     or more invocations of the test object.
```

##tcatpath.h02

```
> TCAT-PATH -- Path Test Coverage Analysis Tool                    HELP
>>> ACTIONS Menu
>>>>> apg

     The apg command generates sets of paths from the digraph file
      derived from a source program.

      The syntax for the apg command is as follows:
                 apg  name
      where,
      name    is the basename of the module/function being
              analyzed.  The filename name.dig must exist
               in the local directory.

     Paths are expressed as a sequence of segments; the notation
     <{a, b, c}> is used to designate zero or more repetitions, in any
      order, of the named segments.
|

      See also:  digraph, cover, ctcover.
```

##tcatpath.h03

```
> TCAT-PATH -- Path Test Coverage Analysis Tool                    HELP
>>> ACTIONS Menu
>>>>> cyclo

    This command computes the cyclomatic number (McCabe metric) for the
      underlying program.

       The cyclomatic number is given by the formula:                  |

             E(n)  =  e  - n  + 2

     where  e  is the number of edges in the program, and  n  is the
       number of nodes in the program.  Generally, but with some
exceptions, |
     programs with a cyclomatic number greater than 10 present unusually
      difficult test situations.

      See also, apg.
```

##tcatpath.h04

```
> TCAT-PATH -- Path Test Coverage Analysis Tool                    HELP
>>> ACTION Menu
 >>>>> digpic
 This command reads a digraph file and generates a picture of the
 program structure you are analyzing.  If you wish to vary the
 picture you must alter the "basis path."  The command syntax is:

           digpic    name    [-B 'file']    [-C center]
                             [-R rows]  [-W width]
      where,
            name    is the name of the file for which you want a picture
             center  is the column number you wish to use
            rows    is the number of rows (default = 1) between nodes
            width   is the width of the picture (default = 80)

     The default basis path is simply the sorted list of names of nodes
      in the digraph file.
```

##tcatpath.h05

```
> TCAT-PATH -- Path Test Coverage Analysis Tool                    HELP
>>> ACTIONS Menu
>>>>> digraph

     This command generates a digraph file from the specified file
      and also instruments the program.

      The syntax for this command is as follows:

               digraph  name.c

      where,

     name.c    is the name of the program you wish processed

     In interactive mode, type "help <command name>" to get help
     screens (like this one) on most topics.  For detailed information
      please consult the TCAT-PATH User's Manual.
```

##tcatpath.h06

```
> TCAT-PATH -- Path Test Coverage Analysis Tool                    HELP
>>> TCAT-PATH -- Menu Descriptions

   TCAT-PATH has four basic interactive menus:

            TCAT-PATH menu:  used to select submenus

           ACTIONS menu:  used to decide on operating modes

           OPTIONS menu:  used to choose execution options

            FILES menu:  used to define file names
```

##tcatpath.h07

```
 > TCAT-PATH -- Path Test Coverage Analysis Tool                   HELP
 >>> All Menus
 >>>>> settings

     The TCAT-PATH system permits you to specify a number of options.
     Many of these options are specified via the TCAT-PATH configuration
       file.

      Options that you can include in this file, for later use or for
       editing, include:

      basename of files to be used (must be specified)
      maximum number of digraph nodes to process (default 500)
      maximum number of paths to generate (default 4800)
      maximum number of paths to display (default 300)
       basis path to be used in digraph display
      maximum number of module invocations (default 1000)

     For more information about user settable options please consult
 TCAT-PATH User's Manual.
```

##tcatpath.h08

```
> TCAT-PATH -- Path Test Coverage Analysis Tool                    HELP
>>> ACTIONS Menu
>>>>> ctruntime

     Once your program is instrumented (see the "digraph" command) you
     need to recompile it and link it with the supplied runtime library.

     The particular version of runtime you use may change depending
      on the language of the programs you are processing.

     The runtime programs capture essential trace file data from the
     system you are testing.  The ctruntime generates a standard
      trace file, ready for processing by "ctcover".
```

##tcatpath.h09

```
 > TCAT-PATH -- Path Test Coverage Analysis Tool                    HELP
>>> Terminology

     TCAT-PATH measures the Ct coverage value of programs under test.
      Here are terms used during TCAT-PATH operation.

     digraph (directed graph) -- The flowchart for the function or
               procedure being studied.

     segment --  A part of the flowchart (digraph) that connects one node
              to another; a decision-to-decision path.

      path -- A sequence of segments within the program.  A path may be
             structurally infeasible but logically unexecutable due to
               data flow within the program.

    tracefile --  The record or sequence of segments hit during a test.
             The trace file is generated with the instrumented program.

    Ct coverage --  The percentage of paths executed in one test or many
             tests from the Ct path set generated by "apg".

   cyclomatic number (McCabe metric) -- A measure of internal complexity
             of a module based on properties of its digraph.
```

##tcatpath.h10

```
> TCAT-PATH -- Path Test Coverage Analysis Tool                    HELP
>>> Trace File Description

     The trace file contains data about all of the functions that
     were executed during the current test.  You need to process it
     with the "ctcover" command to learn what path coverage level you
      have obtained.
```

##tcatpath.h11

```
> TCAT-PATH -- Path Test Coverage Analysis Tool                    HELP
>>> TCAT-PATH Menu
>>>>> save

     The save command permits you to save the values of options
     that you may have chosen during a TCAT-PATH execution.

     When you type save the system prompts you for information
     about whether, and where, you wish parameter values to be
      saved.
```

##tcatpath.h12

```
> TCAT-PATH -- Path Test Coverage Analysis Tool                    HELP
>>> TCAT-PATH Menu
>>>>> release

     The release command causes TCAT-PATH to display release
     and version information.  This information may be useful
      in identifying system problems.
```

##tcatpath.h13

```
> TCAT-PATH -- Path Test Coverage Analysis Tool                    HELP
>>> All Menus
>>>>> exit

     The exit command causes control to return to the TCAT-PATH
     menu, or, if you are in the TCAT-PATH mene, to return to
      the system.
```

##tcatpath.h14

```
> TCAT-PATH -- Path Test Coverage Analysis Tool                    HELP
>>> rcfile

|         The rcfile communicates option values to TCAT-PATH
      at startup time.  It also is used by the main TCAT-PATH verbs:
       digraph, ic, apg, and ctcover.

      Values can be set and switches chosen permanently.  Values
       set during execution can be saved for later use.
```

##tcatpath.h15

```
> <CHANGE THE HEADER..............>                                 HELP
>>> !

    The "!" command allows you to invoke and execute programs at
    system level from within SMARTS's internal menus.  This will,
| example, permit you to send text files to the printer or
     call up a system directory.
```

##tcatpath.h16

```
> <CHANGE THE HEADER..............>                                 HELP
>>> !!

   The "!!" command permits execution of the previous system level
from within SMARTS's internal menus. For example, if the
   previous system level command was to print out a text file, typing
    "!!" will print the text file out again.
```

Coverage Measure Explained

14.1 Introduction

Coverage measures describe the effect of a test - or a set of tests - has on exercising the structure of a software system. The goal of a test coverage metric is to ensure tests are as diverse as possible. The objective is to ensure that a test is more diverse than those which are chosen by reference to functional specifications alone, or are chosen based on a programmer's intuition.

For example, the popular *C1* test metric describes the percentage of program segments that a test exercises. A segment is a part of the program with the property that if any part of it is executed, then all parts of it are executed.

Similarly, the *S1* test metric is a system test completeness measure that calculates the percentage of possible call-pairs that a test - or a group of tests - exercises.

Here is a formal definition of the *Ct* test metric:

> *Ct* Test Coverage Metric: The *Ct* test coverage metric measures the number of times each path or path class in a program is exercised, expressed as a percentage of the total number of paths, calculated up to a specified iteration count K, within the program.

Note that the *Ct* metric depends on the user specifying a minimum iteration count value, K. Normally we keep K = 1, but *Ct* can be defined for other values of K as well.

The key to understanding *Ct* is to understand how a “path is calculated". This will be explained by using some example program passages.

14.2 Example Paths

A path is a sequence of logical segments that can occur in a program. A path exists for each invocation (i.e., execution) of a program. Paths can be classified according to whether or not they have possible repetitions. Programs that do have potential repetition are called iterative programs; otherwise the program is called noniterative.

Noniterative programs have a fixed finite numbers of paths; the number may be large if the program is complex.

Iterative programs have a countably finite number, but without details of program data flow we have to assume that iterations can be of any repetition count. The problem with iteration in terms of path calculations is to know when to “stop" the iteration.

14.3 Noniterative Programs

Consider the program passage shown below. (The example is not intended to be in the syntax of any particular programming language, and should be understandable independent of language.) The lower-case letters a, b, c... represent sequences of statements. The predicates x, y,... are functions that return logical values of some kind.

PROGRAM ONE:

```
a
IF (x)
        b
ELSE
        c
END
d
IF (y)
        e
ELSE
        f
END
g
```

In the example below, a, b,... represent fixed sequences of statements, called “segments". Depending on what the values for the predicates x and y are, the program can take any one of these paths, i.e. sequences of segments (the notation will be explained in more detail on the following page):

PROGRAM ONE:

```
K = 0:
        1:      a b d e g
        2:      a c d e g
        3:      a b d f g
```

In this case there are only four possible paths, numbered above. There is no chance for repetition, so the iteration count value, K = 0, tells us all there is to know about this program's behavior. For K = 1, there are no added paths because there is no iteration

possible in the program.

For a noniterative program, the number of possible paths is a combinatorial function that is computable in advance. There may be a large number of paths but which ones are is known by analysis of the structure of the program and can be computed in advance.

It is important to note that some structurally suggested paths may be logically infeasible. In the example above this means that even though there is a structurally possible sequence “a c d f g”, it is not know for certain that the actual predicates “x” and “y” will permit edge c and edge f to be “co-executed". To determine this requires knowledge of the details of the program.

14.4 Iterative Programs, Various Values of K"

For iterative programs, one must keep track of the number of times each loop is traversed. This is illustrated in the example below, in which paths with varying values of K are calculated.

PROGRAM TWO:

```
a
WHILE (x)
       b
END WHILE
c
WHILE (y)
       d
ENDWHILE
e
```

In the previous program, the paths are a function of the minimum number of times the program traverses each loop. Hence, the paths have to be shown in terms of the loop count, maximum, K.

The notation **`.. <{edge}> ...`** is used to indicate that the edge is executed at least once and possibly more times. It is important to note that the paths are not inclusive upward; that is, even when K = 2, for example, the notation **`... <{a}> ...`** still means exactly one or more repetitions of edge a. To show that a path is supposed to have two repetitions of a particular edge, write **`...a <{a}> ....`**

Here are the paths in the example program, stated in terms of the various possible values of K:

PROGRAM TWO:

```
K = 0:
       1:     a  c  e

.ne 8
K = 1:
       1:     a     c         e
       2:     a     c <{d}>   e
       3:     a <{b}> c       e
       4:     a <{b}> c <{d}> e

K = 2:
       1:     a       c         e
       2:     a       c d       e
       3:     a       c d <{d}> e

       4:     a b     c         e
       5:     a b     c d       e
       6:     a b     c d <{d}> e
```

```
        7:      a b <{b}> c         e
        8:      a b <{b}> c d       e
        9:      a b <{b}> c d <{d}> e


K = 3:
        1:      a         c             e
        2:      a         c d           e
        3:      a         c d d         e
        4:      a         c d d <{d}>   e

        5:      a b       c             e
        6:      a b       c d           e
        7:      a b       c d d         e
        8:      a b       c d d <{d}>   e

        9:      a b b     c             e
        10:     a b b     c d           e
        11:     a b b     c d d         e
        12:     a b b     c d d <{d}>   e

        13:     a b b <{b}> c           e
        14:     a b b <{b}> c d         e
        15:     a b b <{b}> c d d       e
        16:     a b b <{b}> c d d <{d}> e
```

As noted on the previous page, the notation **... <{b}> ...** means that the edge b is executed one or more times. Note that the order of these path classes is grouped to make it easy to see what the sequence actually is. Automatic generation of the paths may result in a different order.

It is important to understand the set of paths varies significantly as the value of K varies. For example, note that when K = 2 you have to include three paths that involve various repetition counts of the edge b, as follows:

PROGRAM TWO:

```
K = 2:

        1:      a         c  e
and
        2:      a b       c  e
and
        3:      a b <{b}> c  e
```

Here Path 1 requires that edge b is used zero times; Path 2 requires that it be used exactly one time; and, Path 3 requires that it be used two or more times.

When you increase the value of K, the growth in path groups is evident:

```
PROGRAM TWO:

K = 3:

        1:      a           c  e
and
        2:      a b         c  e
and
        3:      a b b       c  e
and
        4:      a b b <{b}> c  e
```

Note that Path 3 now loses its **<{b}>** term, only to have it installed again in Path 4.

It should be easy to see that a large value for K will produce a very large set of paths. However, the programs that generate the path class groups will always generate a set of paths that is universal in the sense that every actual program execution will fall into a single, unique class.

14.5 The Exact Meaning of K

From these examples we can begin to understand the intended meaning of the value of K:

> The minimum iteration count, K, is a requirement on a set of actual test paths of a program. The value of K is intended to be the threshold value above which iterations are grouped into equivalence classes which include multiple instances of iteration.
>
> K = 0
>
> means that the test set will map paths that include any repetitions of an edge or node as an equivalence class. (This is a degenerate case that is included for consistency.)
>
> K = 1
>
> means that the test set must include some paths that involve NO repetition of edges or nodes, and will map paths that involve one or more repetitions of an edge or node as an equivalence class.
>
> K = 2
>
> means that the test set must include some paths that involve NO repetition of edges or nodes, some that involve SINGLE repetitions of edges or nodes, and will map paths that involve two or more repetitions of an edge or node as an equivalence class. And so forth...

While all of the paths for some value of K are larger than one may be very interesting theoretically,in practice it is usually enough just to deal with paths generated when K = 1.

14.6 Complex Looping Structures

Sometimes programs have structures that make the processing and representation of the paths very complicated. Consider the following:

PROGRAM BIG:

```
                a
                IF (x)
                        b
                        WHILE (x)
                                c
                                IF (x)
                                        e
                                ELSE
                                        d
                                        IF (x)
                                                e
                                        ELSE
                                                f
                                        ENDIF
                                ENDIF
                                IF (y)
                                        h
                                        EXIT
                                ELSE
.bp
                                        i
                                ENDIF
                        END WHILE
                        j
                ELSE
                        k
                ENDIF
                EXIT
                END PROGRAM
```

In this program the loop has two possible exits: One is the normal exit, g, and the other is the abnormal exit e, which is the last fragment executed before the RETURN statement. It is best if programs did not have such multiple entry and / or multiple exit statements; but, in practical reality they do.

To do so involves using the notation **..<..>.**., which means that the contents of the **<..>**'s can be any path composed of any sequence of the segments named.Using this new notation here is the generated path set for this program.

PROGRAM BIG:

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

V

X