

USER'S GUIDE

METRIC

Version 1.3

Software Metrics
Processor/Generator



SOFTWARE RESEARCH, INC.

This document property of:

Name: _____

Company: _____

Address: _____

Phone _____



SOFTWARE RESEARCH, INC.

1663 Mission Street, Suite 400

San Francisco, CA 94103

Tel: (415) 861-2800

Toll Free: (800) 942-SOFT

Fax: (415) 861-9801

E-mail: support@soft.com

<http://www.soft.com>

ALL RIGHTS RESERVED. No part of this document may be reproduced, stored in a retrieval system or transmitted, in any form or by any means, photocopying, recording or otherwise without prior written consent of Software Research, Inc. While every precaution has been taken in the preparation of this document, Software Research, Inc. assumes no responsibility for errors or omissions. This publication and features described herein are subject to change without notice.

TOOL TRADEMARKS: CAPBAK/MSW, CAPBAK/UNIX, CAPBAK/X, CBDIFF, EXDIFF, SMARTS, SMARTS/MSW, S-TCAT, STW/Advisor, STW/Coverage, STW/Coverage for Windows, STW/Regression, STW/Regression for Windows, STW/Web, TCAT, TCAT C/C++ for Windows, TCAT-PATH, TCAT for JAVA, TCAT for JAVA/Windows, TDGEN, TestWorks, T-SCOPE, Xdemo, Xflight, and Xvirtual are trademarks or registered trademarks of Software Research, Inc. Other trademarks are owned by their respective companies. METRIC is a trademark of SET Laboratories, Inc. and Software Research, Inc. and STATIC is a trademark of Software Research, Inc. and Gimpel Software.

Copyright ©2004 by Software Research, Inc

Table of Contents

Preface	xiii
Congratulations!	xiii
Audience	xiii
Format of Chapters	iv
Identifying Special Text	xv
CHAPTER 1 Introduction to METRIC	1
1.1 The Maintenance Problem	1
1.2 The Solution	2
1.2.1 Developing Metrics	3
1.2.2 Assessing Metric Accuracy	5
Determining Validity	5
Statistical Techniques	6
1.3 METRIC and Software Complexity Measures	8
1.4 Main System Features	9
CHAPTER 2 Quick Start	11
2.1 Instructions	11
2.1.1 STEP 1: Setting Up METRIC	12
2.1.2 STEP 2: Invoking METRIC	13
2.1.3 STEP 3: Selecting a Source Code File	14
2.1.4 STEP 4: Analyzing the Complexity Report	16
2.1.5 STEP 5: Analyzing Multiple Files	18
2.1.6 STEP 6: Determining the Most Complex Procedures	20
2.1.7 STEP 7: Viewing a Summary Report	22
2.1.8 STEP 8: Viewing an Exception Report	24
2.1.9 STEP 9: Viewing an Error Report	26
2.1.10 STEP 10: Using Kiviat Charts to See Complexity	28
2.1.11 STEP 11: Looking at the Type II Chart	30

TABLE OF CONTENTS

2.1.12	STEP 12: Looking at the Type III Chart	32
2.1.13	STEP 13: Signoff and Cleanup	34
2.2	Summary36
CHAPTER 3	System Introduction39
3.1	Overview of METRIC39
3.2	How to Use METRIC40
3.3	The Complexity Report41
3.3.1	Fields n1, n2, N1, N2, and N	43
3.3.2	Software Science Counting Rules	44
3.3.3	Fields N^ and P/R	46
3.3.4	Field V	48
3.3.5	Field E	49
3.3.6	Field VG1	50
3.3.7	Field VG2	52
3.3.8	Fields LOC, BLK, CMT, and <;>	53
3.3.9	Fields SP and VL	54
3.4	The Summary Report55
3.4.1	Field B^	58
3.4.2	Field T^	60
3.5	The Exception Report61
3.6	Accuracy of the Reports63
3.7	Using Metrics in Software Development64
3.7.1	Producing Less Complex Code	65
3.7.2	Allocating Testing Resources	67
3.7.3	Managing Maintenance	68
3.7.4	Expecting Too Much	69
CHAPTER 4	System Operation71
4.1	Using this Chapter71
4.2	User Interface72
	File Selection Box	72
	Help Boxes	74
	Message Boxes	75
	Pull-Down Menus	76
4.3	Invoking METRIC78
4.4	Processing a Source Code File80
4.4.1	Selecting a Language	81
4.4.2	Writing Reports to a File	82
4.4.3	Selecting a Source Code File	84

4.4.4	Selecting Multiple Source Code File	85
4.5	Looking at the Reports	86
4.5.1	Looking at a Complexity Report	87
4.5.2	Re-Ordering Procedures/Functions	89
4.5.3	Looking at a Summary Report	91
4.5.4	Looking at an Exception Report	93
4.5.5	Looking at an Error Report	94
4.5.6	Setting Report Threshold Values	95
4.6	Graphically Viewing Complexity	98
4.6.1	Looking at a Type I Kiviat Chart	99
	Setting Type I Chart Parameters	101
4.6.2	Looking at a Type II Kiviat Chart	103
	Setting Type II Chart Parameters	104
4.6.3	Looking at a Type III Kiviat Chart	107
	Setting Type III Chart Parameters	109
4.6.4	Customizing Your Own Kiviat Chart	113
4.7	Exiting METRIC	114
CHAPTER 5	Helpful Hints	115
5.1	Measuring Program Complexity	115
5.1.1	Set-Up Suggestions	115
5.1.2	Software Development	116
	Using Metrics as a Feedback Tool	116
	Using Metrics in the Review Process	118
	Using Metrics in Estimation	119
	Metrics in Software Testing	121
5.1.3	Software Maintenance	123
	Apportioning Duties	123
	Controlling Entropy	124
CHAPTER 6	Graphical User Interface	125
6.1	About the Main Window	125
6.2	Main Features of METRIC	126
6.2.1	Display Area	127
6.2.2	File Pull-Down Menu	128
6.2.3	Options Pull-Down Menu	131
6.2.4	Report Pull-Down Window	137
6.2.5	Charts Pull-Down Menu	140
6.2.6	Help Button	144

TABLE OF CONTENTS

CHAPTER 7	Command Line Activation	145
7.1	Command Line Usage	145
7.2	'Xmetric' Command	146
	Options and Parameters:	146
7.3	'langmetric' Command	147
	Options and Parameters	148
7.4	'Xkiviat' – Static Metrics Display System	153
	Options and Parameters	154
7.5	Configuration File Processing	156
Appendix A	"C" Notes	171
	cmetric	171
	cresword.tab	171
	cnonexe.tab	172
	The Complexity Report	173
	The Summary Report	175
	The Exception Report	177
	The Error Report	178
	Reserved Word/Nonexecutable Word File	180
	Miscellaneous Operator/Operand Rules	181
	Cyclomatic Complexity	181
	Span of Reference	182
	Executable Semi-colons	182
	Average Variable Name Length	182
	Lines of Code	182
	Comments	182
	Creating a Shell Script File	183
	Conditional Compilation Directives	184
	Comments About METRIC	186
Appendix B	"C++" Notes	187
	cppmetric	187
	cppresword.tab	187
	cppnonexe.tab	188
	The Complexity Report	189
	The Summary Report	191
	The Exception Report	193
	The Error Report	194
	C++ Class Report	196
	Class Summary	196
	Class Hierarchy	197
	Class Exception Report	197
	Counting Rules	198

Reserved Word/Nonexecutable Word Files	198
Miscellaneous Operator/Operand Rules	199
Cyclomatic Complexity	200
Span of Reference	200
Executable Semi-colons	200
Average Variable Name Length	200
Lines of Code	200
Comments	201
Creating a Shell Script File	202
Conditional Compilation Directives	203
Comments About METRIC	205

Appendix C Ada Notes 207

adametric	207
adaresword.tab	207
Description of the Reports	208
The Complexity Report	208
The Summary Report	209
The Exception Report	211
The Error Report	211
The Generic Report	212
The Package Exception Report	214
The Package Intermediates Report	215
Counting Rules	216
Reserved Word File	216
Miscellaneous Operator/Operand Rules	216
Cyclomatic Complexity	217
Span of Reference	217
Executable Semi-colons	217
Average Variable Name Length	218
Lines of Code	218
Comments	218
Creating a Shell Script File	219
Comments About METRIC	219

Appendix D FORTRAN Notes 221

fmetric	221
forreswo.tab	221
Description of the Reports	223
The Complexity Report	223
The Summary Report	225
The Exception Report	227
The Error Report	228
Counting Rules	230
Reserved Word File	230
Miscellaneous Operator and Operand Rules	231

TABLE OF CONTENTS

Cyclomatic Complexity	231
Span of Reference	232
Executable Carriage Returns	232
Average Variable Name Length	232
Lines of Code	232
Comments	232
Creating a Shell Script File	234
Comments About METRIC	235
Index	237

List of Figures

FIGURE 1	METRIC System Flow Chart	8
FIGURE 2	Setting Up the Display Options (Initial Condition)	12
FIGURE 3	Invoking METRIC	13
FIGURE 4	Selecting a Source Code File	15
FIGURE 5	Analyzing the Complexity Report	17
FIGURE 6	Selecting Multiple Source Code Files	19
FIGURE 7	Controlling Complexity Order	21
FIGURE 8	Analyzing the Summary Report	23
FIGURE 9	Analyzing the Exception Report	25
FIGURE 10	Viewing the Errors Report	27
FIGURE 11	Type I Kiviat Chart	29
FIGURE 12	Type II Kiviat Chart	31
FIGURE 13	Type III Kiviat Chart	33
FIGURE 14	Completing a METRIC Session	35
FIGURE 15	metric.ksv Setup	37
FIGURE 16	Sample Complexity Report	41
FIGURE 17	Flow Graph	51
FIGURE 18	Sample Summary Report	55
FIGURE 19	Sample Exception Report	61
FIGURE 20	Using a File Selection Dialog Box	73
FIGURE 21	Using the Help Dialog Box	75
FIGURE 22	Using a Dialog Box	76
FIGURE 23	Using a Pull-down Menu	77
FIGURE 24	Invoking the Main Window	78
FIGURE 25	Invoking METRIC from STW	79
FIGURE 26	Selecting a Language	81

List of Figures

FIGURE 27 Saving Reports to a Common Basename	83
FIGURE 28 Selecting a Source Code File	84
FIGURE 29 Selecting Multiple Source Code Files	85
FIGURE 30 Selecting the Complexity Report	87
FIGURE 31 Complexity Report	88
FIGURE 32 Sort Report By Window	90
FIGURE 33 Re-Ordered Complexity Report	90
FIGURE 34 Selecting the Summary Report	91
FIGURE 35 Summary Report	92
FIGURE 36 Exception Report	93
FIGURE 37 Configuration Options Window	95
FIGURE 38 Type I Kiviat Chart	100
FIGURE 39 Xmetric.I.def Configuration File	101
FIGURE 40 Type I Configuration Window	102
FIGURE 41 Type II Kiviat Chart	104
FIGURE 42 Xmetric.II.def Configuration File	105
FIGURE 43 Type II Configuration Window	106
FIGURE 44 Type III Kiviat Chart	109
FIGURE 45 Xmetric.III.def Configuration File	110
FIGURE 46 Type III Configuration Window	112
FIGURE 47 Exiting METRIC	114
FIGURE 48 Display Area	127
FIGURE 49 Load Single File Selection	128
FIGURE 50 Load Multiple Files Selection	129
FIGURE 51 Setting the Report Files Basename	129
FIGURE 52 File Pull-Down Window Help	130
FIGURE 53 File Pull-Down Menu	130
FIGURE 54 Configuration Options Window	131
FIGURE 55 Type I Configuration Window	133
FIGURE 56 Type II Configuration Window	134
FIGURE 57 Type III Configuration Window	135
FIGURE 58 Select Language Window	136
FIGURE 59 Sort Report By Window	137
FIGURE 60 Report Pull-Down Menu	139
FIGURE 61 Type I Kiviat Chart	140
FIGURE 62 Type II Kiviat Chart	141

FIGURE 63	Type III Kiviat Chart	142
FIGURE 64	Type User Kiviat Chart Selection	143
FIGURE 65	Charts Pull-Down Menu	143
FIGURE 66	Help Window for METRIC	144
FIGURE 67	Example Kiviat Diagram	153
FIGURE 68	Complexity Report for "C"	174
FIGURE 69	Summary Report for "C"	176
FIGURE 70	Exception Report for "C"	177
FIGURE 71	Complexity Report for "C++"	190
FIGURE 72	Summary Report for "C++"	192
FIGURE 73	Exception Report for "C++"	193
FIGURE 74	Complexity Report for Ada	209
FIGURE 75	Summary Report for Ada	210
FIGURE 76	Exception Report for Ada	211
FIGURE 77	Generic Report	213
FIGURE 78	Package Exceptions Report	214
FIGURE 79	Package Intermediates Report	215
FIGURE 80	Complexity Report for FORTRAN	224
FIGURE 81	Summary Report for FORTRAN	226
FIGURE 82	Exception Report for FORTRAN	227

Preface

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

Congratulations!

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

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

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

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

Audience

This manual is intended for software testers who are using Metric tools. You should be familiar with the X Window System and your workstation.

Format of Chapters

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

This manual is divided into the following sections:

- | | |
|-----------|---|
| Chapter 1 | <i>INTRODUCTION TO METRIC</i> explains the basic functions of METRIC™ and its role in Quality Assurance/Quality Control. |
| Chapter 2 | <i>QUICK START</i> is a tutorial and shows step-by-step how to run a basic METRIC™ test session. |
| Chapter 3 | <i>SYSTEM INTRODUCTION</i> is an overview of the METRIC™ system. |
| Chapter 4 | <i>SYSTEM OPERATION</i> covers the basic X Window System graphical user interface operations of METRIC™. |
| Chapter 5 | <i>HELPFUL HINT</i> offers recommendations and principles of operation for METRIC™ in an automated testing environment. |
| Chapter 6 | <i>GRAPHICAL USER INTERFACE</i> defines and explains the content of the Main window that makes up the METRIC™ product. |
| Chapter 7 | <i>COMMAND LINE ACTIVATION</i> disrobes in detail the various command line switches which perform tasks very similar to the graphical user interface. |

Identifying Special Text

This section explains the typographical conventions that are used throughout this manual.

boldface Introduces or emphasizes a term that refers to TestWorks' window, its sub-menus and its options.

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

"Double Quotation Marks"

Indicates chapter titles and sections. Words with special meanings may also be set apart with double quotation marks the first time they are used.

`courier` Indicates system output such as error messages, system hints, file output, and CAPBAK/X's keysave file language.

Boldface Courier

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



Introduction to METRIC

This chapter explains the basic functions of *METRIC*TM and its role in Quality Assurance/Quality Control.

1.1 The Maintenance Problem

In the life cycle of software, much effort and time goes into maintenance. All changes and enhancements made from the time the product has been conceived to the time it first delivered involves a great deal of maintenance. During this maintenance period, software can become victim to **complexity**. Complexity is a measure of how difficult or how complex a program may be. If a program is becomes too complex after several changes, you can lose control over maintenance. This is known as **entropy**.

Many managers rely on their programmers' common sense to know when entropy is affecting the quality of the code. Today, this kind of ad hoc maintenance can spell disaster for mid- to large-scale programs. Many of today's applications contain dozens of user-selectable functions, each of which can have several changes made to it before it is shipped.

If the complex parts of a program could be identified in a reliable, quantitative manner, programmers could more easily determine when program entropy has taken it toll and adjust the overly complex modules accordingly. In other words, if you could identify the most of program that are most likely to be cause maintenance problems and the measure the degree of possible entropy, you could more easily maintain those problem areas.

One such method is the field of **software metrics**. Software metrics can be used to develop QA procedures and measure a program's quality (and its potential entropy) before it is released. This chapter addresses the field of software metrics.

1.2 The Solution

The field of software metrics grew out of a need by programmers and managers to be able to express the characteristics of a piece of software in quantitative terms. The first “software metric” was simply a count of the number of lines in a program. This was sufficient when attempting to express how “big” a program was. However, a simple count of lines of code turned out to be much too simplistic for many other purposes.

Clearly there are lines of code, and then there are lines of code. For example, consider the following two C program segments:

```
for (j=1;j<21;j++) {a=10;
  for (i=2;i<21;i++)  b=10;
  m[i-1]=m[i];d=b+a
  fscanf(MF,"%d",m[20]);b=0;
}a=0;
for (j=1;j<11;j++) {d+=2;
  s=s+m[j]+m[j*2];o=d*2;
}printf("%d",d);
printf("%d",s);printf("%d",0);
```

It seems clear that the segment on the left has much more going on within it than the one on the right, yet they are both nine lines long.

Computer Science researchers and practitioners alike searched for an alternative measure that would distinguish programs which were of similar length, but differed in terms of “how much was going on” within them. This property came to be known as software complexity. A method of measurement is often called a “metric”. Therefore, a measure of software complexity is usually known as a software complexity metric.

Intuitively, software complexity can be viewed as “how much is going on” within the code. Obviously, complexity can have a significant impact on how hard it is to understand and work with the code. Software complexity metrics allow us to objectively express how complex a piece of code is. This can help identify the parts of a program which might benefit from rewriting. Metrics can also help those managing a project assess how difficult a program will be to work with (and hence how much time might be required to perform some activity upon it).

It has been shown that a program which is more complex than another is also likely to have more errors. Project managers can thus use complexity metrics to gauge how many errors various procedures or programs within a system might have. This information can then be used to allocate resources for testing.

1.2.1 Developing Metrics

As we noticed before, the number of lines of code is not the only program characteristic contributing to complexity. A number of other such characteristics exist. For example, a program that exhibits excessive decision making can be hard to follow. Likewise, a program that makes use of many global variables can be extremely hard to modify because of the dreaded "ripple effect". The list of characteristics which can contribute to software complexity goes on and on.

Researchers trying to measure software complexity approach the problem not by measuring complexity itself but rather by measuring the degree to which those characteristics thought to lead to complexity exist within a program. Thus, for example, a classical measure of software complexity is the number of decision statements in the code.

Unfortunately, it seems at times that every researcher with an interest in software complexity has his own pet characteristic which he thinks is the factor that contributes most to software complexity. As a consequence, the number of measures which were suggested mushroomed as interest began to grow in this area of research in the late 1970's and early 1980's. Many different complexity metrics were suggested, but soon two major categories of metrics became recognized: **size metrics** and **control flow metrics**.

Size metrics were developed on the premise that the "larger" the program, the more complex it is. Size in this context is not necessarily based strictly on the number of lines of code in the program. For example, common size measures are the number of procedures, number of variables, or number of operators used.

For example, common size measures are the number of procedures, number of variables, or number of operators used. Control flow metrics attempt to address the issue of numerous decision points located in the program. Some popular measures simply count the number of `IF` statements within the program, while others include a count of all decision points (i.e., `IF`, `WHILE`, `FOR`, `CASE`, etc.), others attempt to assess the level of control flow nesting (i.e., an `IF` within the scope of another `IF`) and yet others attempt to relate the selected measures to graph theory by viewing the abstract flow of program control as a directed graph.

No consensus has been reached as to which approach is correct. Because of this, some researchers have developed a special set of metrics referred to as hybrid complexity metrics. Hybrid metrics take (hopefully) the best from each category. For example, a popular hybrid measure involves assessing both the control flow and program size. However, since the complexity contribution of each of the characteristics often is not equal, it is not always clear how to combine the measurements of different characteristics.

1.2.2 Assessing Metric Accuracy

Anyone can say that a given characteristic contributes to software complexity, and therefore measuring the degree to which that characteristic exists in a program should provide a measure of how complex that program is. However, people are generally more comfortable using metrics if they have some evidence that this is in fact, the case.

Determining Validity

To obtain evidence supporting the validity of a metric, it is useful to be able to establish two facts:

- Does the characteristic(s) the metric is built around actually contribute to complexity?
- Does the method of measuring the presence of the characteristic actually reflect how much complexity is being contributed?

Often, researchers will combine these two goals by using a *proxy* for complexity, and testing to see if that proxy is related to the measurements provided by the metric in question.

Some activity is chosen that can be objectively measured, and that we are reasonably sure is impacted by complexity. For example, popular proxies are time to develop the program being analyzed, the number of programming errors made during development of the program, etc. These are sometimes referred to as process metrics since they measure aspects of the programming process. Measures of software characteristics, such as complexity metrics, are known as product metrics since they measure aspects of the product.

By using this approach to validating a particular software metric, the question at hand can be restated as:

Is the product measure of interest correlated with the process metric of interest?

In other words, suppose we look at several different programs. As the product metric (e.g., number of IF statements) in the different programs increase, does the corresponding process metric (e.g., number of errors) increase as well?

Statistical Techniques

If the value of the product metric *does* increase as the process metric is found to increase, then we say we have a positive correlation. The degree of correlation may range from -1 to +1. This measure of correlation is known as the correlation coefficient. A correlation coefficient of zero means there is no relationship. A 0.5 correlation coefficient means some relationship between the two metrics exist, but there are other factors which account for the increase in the process metric besides an increase in the product metric. A correlation coefficient of +1 means that it is likely that the only factor which accounts for the increase in the process metric is an increase in the product metric.

Negative correlations coefficients mean similar things, except the relationship is such that a decrease in the product metric accounts for an increase of the process metric, or vice-versa. For example, a correlation coefficient of -1 means that probably the only factor which accounts for an increase in the process metric is a decrease in the product metric. A high correlation does not necessarily imply that one variable is equal, or even close to equal to the other variable. What it means is that one variable seems to increase and decrease in relation to the other variable being studied. For example, the following columns of numbers have a perfect +1.0 correlation:

1020
1530
2040
2550
3060

Note that the numbers in the right column are always twice those in the left column - therefore, as the numbers in one column increase or decrease, so do the numbers in the other column, and at the same rate! This is what is meant by a “perfect correlation”.

Formulas to compute the correlation coefficient for two variables can be found in any introductory statistics text book. However, if a high correlation exists between two metrics, plotting their values on an X-Y graph will allow the relationship to be recognized without going through the calculations to determine the correlation coefficient.

Other approaches may also be taken to show that a product metric is related to a process metric. For example, a set of programs can be split into two experimental groups based on some property such as number of decisions. All the programs with 10 or fewer decisions would be assigned to one group, and all the programs with more than 10 decisions assigned to the second group. The average number of programming errors in each group would then be compared to see if the difference between the two groups is *statistically significant*.

If there is a statistically significant difference, then we can conclude the metric in question may be valid. This is reliable only if the entire set of programs are homogeneous in regards to other characteristics, such as size, application, etc. If this is not true, differences in one of the other characteristics might account for the differences in the process metric.

1.3 METRIC and Software Complexity Measures

As mentioned earlier, numerous metrics exist. Perhaps the two most popular families of metrics currently in use are the **Software Science** of the late Maurice Halstead, and the **Cyclomatic Complexity**.

In addition to being the two most popular, they are also good representatives of the two categories of metrics mentioned earlier. Software Science is based on the “size” of the software, while Cyclomatic Complexity is based on the flow of control within the code.

METRICTM resolves the maintenance issue by automatically computing several complexity metrics for the program, including Software Science and Cyclomatic Complexity measures. These measures identify the most complex, error prone, and maintenance effort parts of the program. Because *METRICTM* identifies the most complex modules, you can concentrate your testing resources on just those complex parts.

The diagram below illustrates the *METRICTM* process. You should study this diagram carefully so that you see the natural structure and rhythm of the *METRICTM* use.

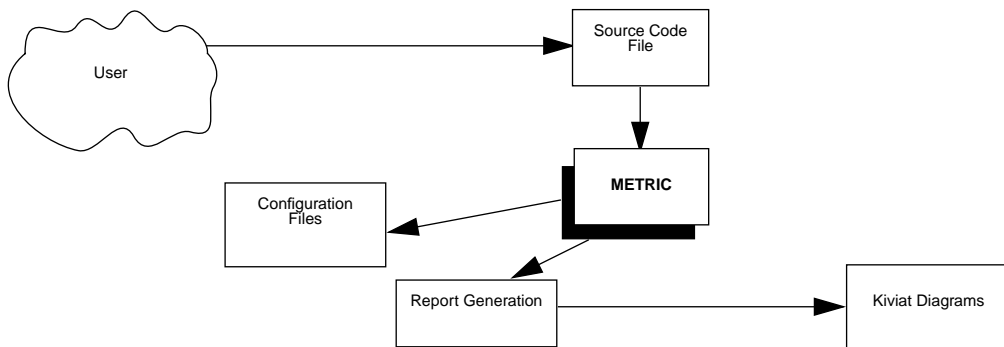


FIGURE 1 METRIC System Flow Chart

1.4 Main System Features

Here is a list of the most important features the *METRIC*TM offers:

- Reads any compilable source code text file.
- Automatically computes Software Science and Cyclomatic Complexity measures for the specified program.
- Creates a Complexity report, which lists the complexity measures for each program module.
- Creates an Exception reports, which lists the program's modules that exceed default complexity measure threshold values.
- Generates Kiviat diagrams, which allows you to view the impact of multiple metrics on the program.
- Allows you to analyze entire groups of source code files whose names match some sort of pattern.
- Allows you to set your complexity values.
- Functions accessed through a X Window System graphical user interface (GUI).
- Supports code written in Ada, C, C++, and FORTRAN.

Quick Start

This chapter is a tutorial and shows step-by-step how to run a basic *METRIC*TM test session.

LEVEL: If you are an advanced *METRIC*TM user, you may skip this chapter. This chapter is intended for beginning and intermediate users.

2.1 Instructions

It is recommended that you complete the instructions in this chapter *before* continuing to other sections. This chapter will give you a feel for how the system is organized and will permit you to create more efficient and effective tests.

For best results, follow the instructions very carefully. When you have completed this chapter, you should be familiar with the main activities involved in using *METRIC*TM, including selecting a program to analyze, processing complexity metrics for that program, viewing resulting reports, and using Kiviat diagrams to visualize the impact of several metrics at one time.

If you are a first-time *METRIC*TM user, this chapter is best used if you make reference to the introductory chapters (See CHAPTER 3 - "System Introduction" on page 39.) (See CHAPTER 4 - "System Operation" on page 71.) If you are an intermediate user, this chapter is best used if you make reference only to those menu definitions which need further explanation (see the *SYSTEM OPERATION* and *GRAPHICAL USER INTERFACE* chapters for further information).

If you have available the **xplabak** utility (playback utility for *CAPBAK/X*TM, you may, when you are done, run the supplied `metric.ksv` file to see an example of how this session works. The instructions are at the end of this chapter (See Section 2.2 - "Summary" on page 36.).

2.1.1 STEP 1: Setting Up METRIC

You should start with the screen organized in a particular way, as shown in the figure (See Figure 2 "Setting Up the Display Options (Initial Condition)" on page 12.).

Initialize an xterm-type window by using the mouse to click on **New Windows** or issuing the command `xterm&` from an existing window. The xterm window will serve as the *METRICTM* invocation window.

Move the window to the upper left of the screen. Go to the `$SR/demos` directory. The `demos` directory is supplied with the product, and it consists of several language dependent files, which you can be used with *METRICTM* for practice. This **QUICK START** will use `xcalc.c` and `sr.c` to demonstrate *METRICTM* usage.

When initiating this quick start session, your display should look something like this:

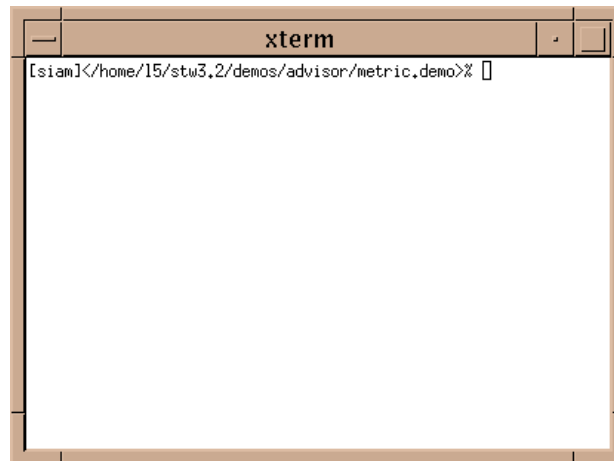


FIGURE 2 Setting Up the Display Options (Initial Condition)

2.1.2 STEP 2: Invoking METRIC

Now, invoke *METRIC*TM.

1. Position the mouse so that it is located in the invocation window.
2. Activate it by clicking the mouse button on it. This window becomes the main control window. During your session, all status messages and warnings are displayed in this window.
3. Start *METRIC*TM from your working directory by typing in:

xmetric

xmetric is the GUI-version of *METRIC*TM.

4. When you type in the command, the **Main** *METRIC*TM window pops up. All operations for *METRIC*TM can be performed from this window.
5. Move the **Main** window to the lower right of the screen. You can move a window by clicking on its title bar and dragging it.
6. If you want to start over, you can terminate from the **Main** window by clicking on the **File** pull-down menu and selecting **Exit**.

After invoking *METRIC*TM, your display should look something like this:

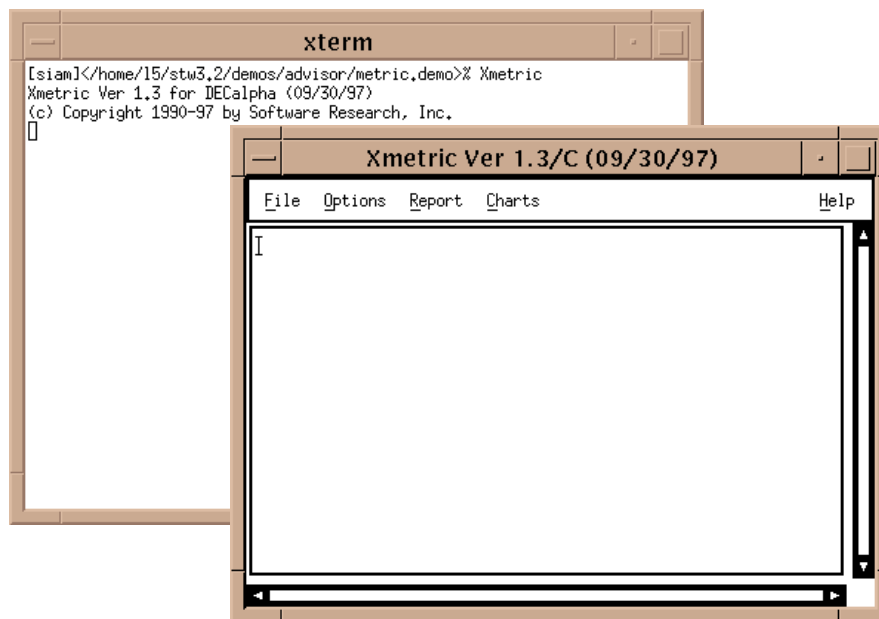


FIGURE 3 Invoking METRIC

2.1.3 STEP 3: Selecting a Source Code File

To obtain complexity measures for a source code file, all you have to do is select any compilable file. *METRICTM* is a static code analyzer, so you do not have to do anything special to a program's code. For this demo, select the file named `sr.c`:

1. Click on the **File** pull-down menu.
2. Select the **Load Single File** option.
3. A file selection dialog box pops up.
4. To select `sr.c`, do one of three things:
 - Double click on `sr.c` in the File selection window,
 - Highlight `sr.c` in the File selection window or type in the file name in the Selection entry box and click on OK, or
 - Highlight or type in `sr.c` and press the <ENTER> key.
5. *METRICTM* automatically processes complexity measures for `sr.c`. These measures are displayed in a **Complexity** report.

When selecting a source code file, your display should like this:

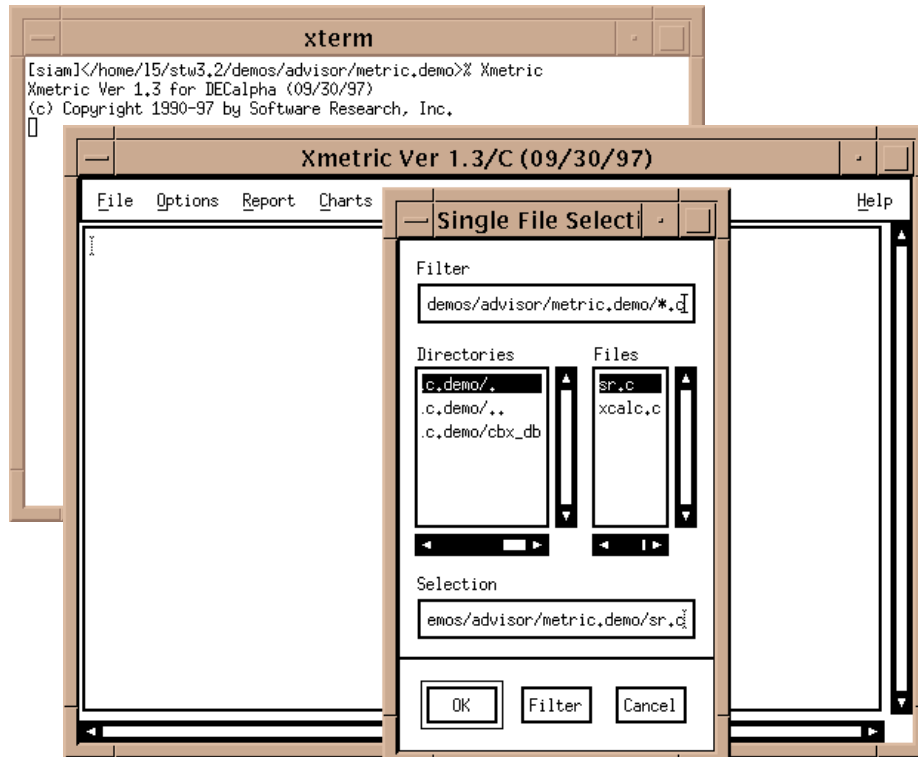


FIGURE 4 Selecting a Source Code File

2.1.4 STEP 4: Analyzing the Complexity Report

After selecting the source code file, *METRICTM* automatically processes complexity metrics for it in a **Complexity** report. Because this report is important to your understanding of *METRICTM*, this step concentrates on analyzing this report.

1. When *METRICTM* automatically created the **Complexity** report for `sr.c`, the **Main** window may not reveal the entire report. In such a case, simply resize the window to fit the report's size or use the scroll bars to move up/down and side/side.
2. The **Complexity** reports lists the program's encountered procedures and lists Software Science metrics (which are concerned with the size of the software) and Cyclomatic Complexity measures (which are concerned with the flow of control within the program's code). The report includes the following fields:
 - Procedure Name
 - Unique Operators (n1)
 - Unique Operands (n2)
 - Total Operators (N1)
 - Total Operands (N2)
 - Length (N)
 - Predicted Length (N[^])
 - Purity Ratio – estimated length divided by length (P/R)
 - Volume (V)
 - Effort (E)
 - Cyclomatic Complexity (VG1)
 - Extended Cyclomatic Complexity (VG2)
 - Lines of Code (LOC)
 - Number of Comment Lines (CMT)
 - Number of Blank Lines (BLK)
 - Number of Executable Semi-Colons (< ; >)
 - Average Maximum Span of Reference of Variables (SP)
 - Variable Name Length (VL)

Please refer to (See CHAPTER 7 - "Command Line Activation" on page 145.) for complete information on these fields of complexity measures.

- Those procedures with the highest field values are the most complex values. You may consider analyzing procedures with particularly high field values. In the case of `sr.c`, `do_sr` may need further analysis.

When analyzing the **Complexity** report, your display should look like the one below:

```

[siam]~/home/15/stw3.2/demos/advisor/metric.demo>% Xmetric
Xmetric Ver 1.3 for DECalpha (09/30/97)
(c) Copyright 1990-97 by Software Research, Inc.

```

Procedure	n1	n2	N1	N2	N	N^	P/R
do_sr	59	103	635	403	1038	1036	1.00 7
XSRError	6	4	9	4	13	24	1.81
rescale	24	26	120	71	191	232	1.22 1
drawmark	9	8	19	16	35	53	1.50
dolabel	10	12	20	15	35	76	2.18
drawframe	29	38	199	146	345	340	0.99 2
doscale	25	20	106	79	185	203	1.09 1
dotenths	28	28	204	141	345	269	0.78 2
drawslide	27	32	134	98	232	288	1.24 1
redrawslide	11	10	22	14	36	71	1.98
redrawframe	11	10	22	14	36	71	1.98
drawhair1	6	4	11	6	17	24	1.38
drawnums	12	20	148	91	239	129	0.54 1

FIGURE 5 Analyzing the Complexity Report

2.1.5 STEP 5: Analyzing Multiple Files

In most cases, you will most probably analyze more than one source code file at once. For the remainder of this demo, you will be analyzing `sr.c` and `xcalc.c`. To select multiple files:

1. Click on the **File** pull-down menu.
2. Select the **Load Multiple Files** option.
3. A file selection dialog box pops up.
4. To select `sr.c` and `xcalc.c`, do one of two things:
 - Highlight `sr.c` and `xcalc.c` in the File selection window.
 - Select `Select All` and then select `OK`.
5. *METRICTM* automatically processes complexity measures for `sr.c` and `xcalc.c` into a **Complexity** report.

When selecting multiple source code files, your display should like this:

```

[siam]~/home/15/stw3.2/demos/advisor/metric.demo>% Xmetric
Xmetric Ver 1.3 for DECalpha (09/30/97)
(c) Copyright 1990-97 by Software Research, Inc.
□
  
```

Procedure	n1	n2	N1	N2	N	N^	P/R	V
do_sr	59	103	635	403	1038	1036	1.00	7619
XSRError	6	4	9	4	13	24	1.81	43
rescale	24	26	120	71	191	232	1.22	1078
drawmark	9	8	19	16	35	53	1.50	143
dolabel	10	12	20	15	35	76	2.18	156
drawframe	29	38	199	146	345	340	0.99	2093
doscale	25	20	106	79	185	203	1.09	1016
dotenths	28	28	204	141	345	269	0.78	2004
drawslide	27	32	134	98	232	288	1.24	1365
redrawslide	11	10	22	14	36	71	1.98	158
redrawframe	11	10	22	14	36	71	1.98	158
drawhair1	6	4	11	6	17	24	1.38	56
drawnums	12	20	148	91	239	129	0.54	1195

Procedure	n1	n2	N1	N2	N	N^	P/R	V
parse_double	8	6	11	6	17	40	2.32	65
open_the_display	22	15	61	29	90	157	1.74	469

FIGURE 6 Selecting Multiple Source Code Files

2.1.6 STEP 6: Determining the Most Complex Procedures

When dealing with mid- to large- scale programs, you may initially find it difficult to determine the most complex procedures, or modules. The procedures are ordered according to how they are encountered in the program. You can, however, avoid scanning by ordering the procedures according to ascending or descending order based on one of the 17 complexity measures. For the sake of this demo, you are going to order the procedures according to the number of unique operators (`n1`). Here's how:

1. Click on the **Report** pull-down menu.
2. Select **Order Complexity**.
3. The **Sort Report By** window pops up. It lists all of the complexity fields. The default is set to **procedure**, which is reflected in the **Complexity** report.
4. Click on the corresponding radio button for `n1`.
5. You can order the procedures in ascending or descending order. For this demo, please click on the **Descend** button.
6. *METRICTM* automatically reorders the procedures according to the number of unique operators (`n1`). Those modules with the highest number of unique operators are listed first. For `sr.c`, `do_sr` has the highest number of unique operators; for `xcalc.c`, `main` has the highest number of unique operators.
7. You may want see the effect of other fields on these two source code files. Simply follow 1 through 5 again, but select different fields.

When ordering procedures complexity based on the number of unique operators, your display should like this:

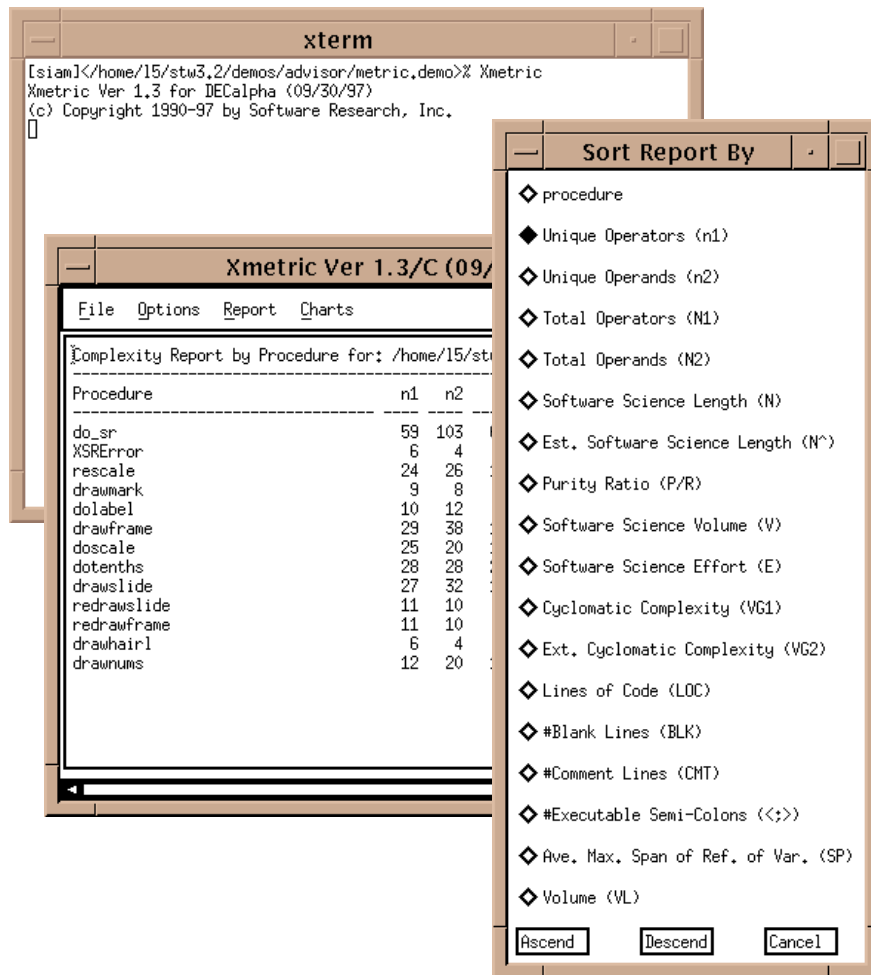


FIGURE 7 Controlling Complexity Order

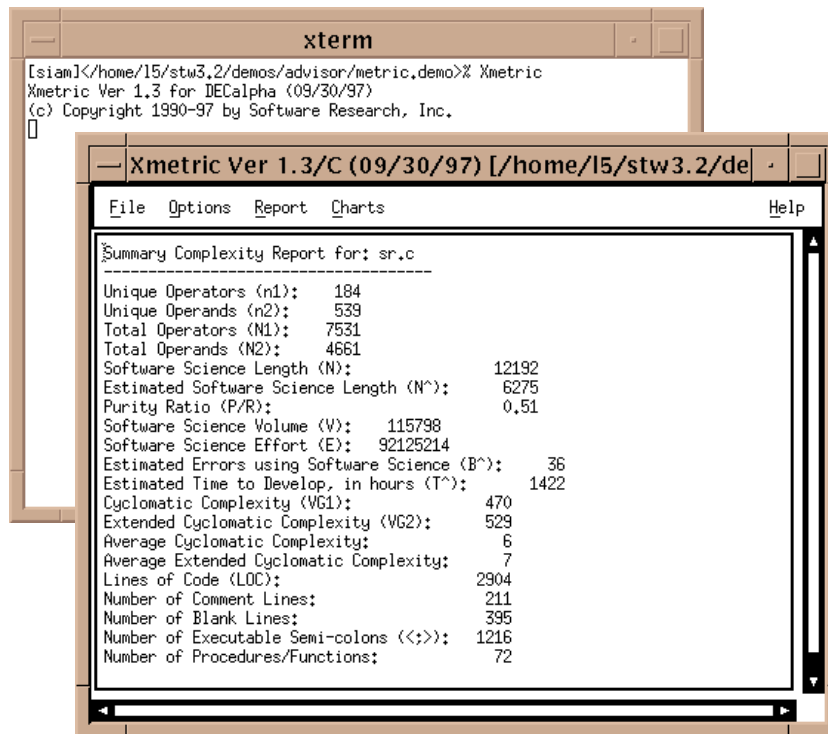
2.1.7 STEP 7: Viewing a Summary Report

The **Complexity** report lists the complexity measures for each procedure. *METRICTM* also offers a **Summary** report, which is an accumulated account of the complexity measures for the entire source code file or files. To look at a **Summary** report:

1. Click on the **Report** pull-down menu.
2. Select **Summary Only**.
3. *METRICTM* automatically creates an accumulated summary for `src.c` and `xcalc.c`.
4. You may want to resize the window to fit the size of the report, or you can use the scroll bars to move side/side or up/down.
5. The **Summary** report consists of the following fields:
 - Unique Operators (n1)
 - Unique Operands (n2)
 - Total Operators (N1)
 - Total Operands (N2)
 - Software Science Length (N)
 - Estimated Software Science Length (N[^])
 - Purity Ratio (P/R)
 - Software Science Volume (V)
 - Software Science Effort (E)
 - Estimated Errors Using Software Science (B[^])
 - Estimated Time to Develop, in hours (T[^])
 - Cyclomatic Complexity (VG1)
 - Extended Cyclomatic Complexity (VG2)
 - Average Cyclomatic Complexity
 - Average Extended Cyclomatic Complexity
 - Lines of Code (LOC)
 - Number of Comment Lines (CMT)
 - Number of Blank Lines (BLK)
 - Number of Executable Semi-Colons (< ; >)
 - Number of Procedures/Functions

For complete information on these fields of complexity measures, see the correct section (See CHAPTER 4 - "System Operation" on page 71.).

After obtaining a **Summary** report your display should like this:



```
[siam]~/home/l5/stw3.2/demos/advisor/metric.demo>% Xmetric
Xmetric Ver 1.3 for DECalpha (09/30/97)
(c) Copyright 1990-97 by Software Research, Inc.
[]

Xmetric Ver 1.3/C (09/30/97) [~/home/l5/stw3.2/de
File Options Report Charts Help
Summary Complexity Report for: sr.c
-----
Unique Operators (n1): 184
Unique Operands (n2): 539
Total Operators (N1): 7531
Total Operands (N2): 4661
Software Science Length (N): 12192
Estimated Software Science Length (N^): 6275
Purity Ratio (P/R): 0,51
Software Science Volume (V): 115798
Software Science Effort (E): 92125214
Estimated Errors using Software Science (B^): 36
Estimated Time to Develop, in hours (T^): 1422
Cyclomatic Complexity (VG1): 470
Extended Cyclomatic Complexity (VG2): 529
Average Cyclomatic Complexity: 6
Average Extended Cyclomatic Complexity: 7
Lines of Code (LOC): 2904
Number of Comment Lines: 211
Number of Blank Lines: 395
Number of Executable Semi-colons (<>): 1216
Number of Procedures/Functions: 72
```

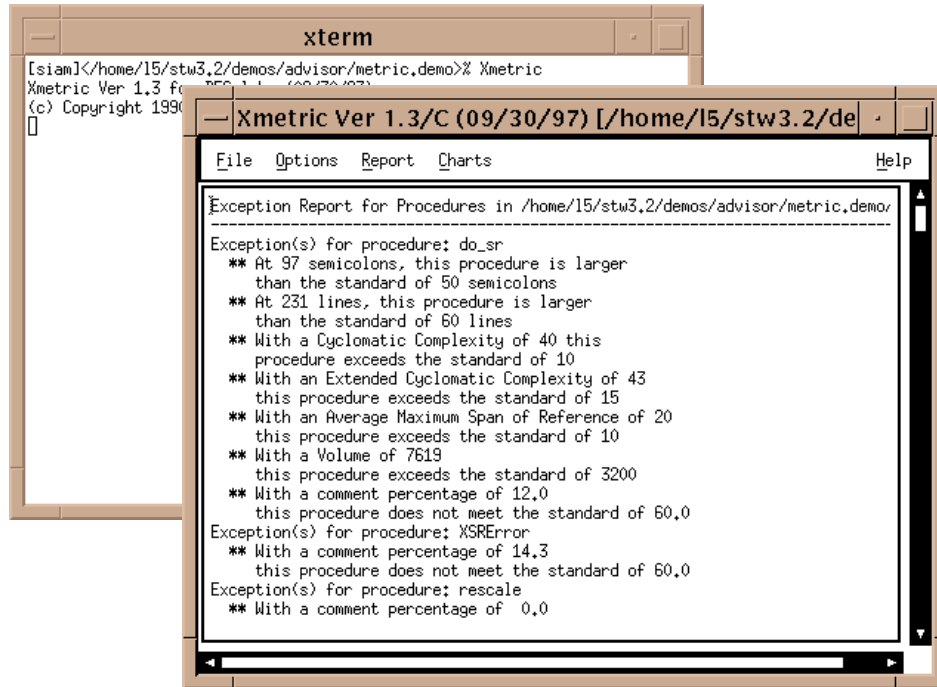
FIGURE 8 Analyzing the Summary Report

2.1.8 STEP 8: Viewing an Exception Report

*METRIC*TM also produces an **Exception** report. Each procedure in the source files which exceeds a set of predefined complexity maximums is included in this report. These complexity standards can be set in the configuration file, `.uxmetriccfg`, or with the GUI's **Configuration Options** window (under the **Options** pull-down menu). To look at an **Exception** report:

1. Click on the **Report** pull-down menu.
2. Select **Exceptions**.
3. *METRIC*TM automatically generates an **Exception** report.
4. It lists each "violation" for each procedure encountered in `sr.c` and `xcalc.c`. In other words, the **Exception** report lists where code exceeds complexity measures' threshold limits.
5. For module `do_sr`, for instance, seven of the complexity measures go over their threshold limits. This module may be troublesome. In a testing situation, you would design your test cases to some way more thoroughly cover the more complex modules.

After obtaining an **Exception** report your display should look like this:



```
xterm
[lsiam]~/home/15/stw3.2/demos/advisor/metric,demo% Xmetric
Xmetric Ver 1.3 for PC (09/30/97)
(c) Copyright 1990

Xmetric Ver 1.3/C (09/30/97) [/home/15/stw3.2/de
File Options Report Charts Help
-----
Exception Report for Procedures in /home/15/stw3.2/demos/advisor/metric,demo/
Exception(s) for procedure: do_sr
** At 97 semicolons, this procedure is larger
than the standard of 50 semicolons
** At 231 lines, this procedure is larger
than the standard of 60 lines
** With a Cyclomatic Complexity of 40 this
procedure exceeds the standard of 10
** With an Extended Cyclomatic Complexity of 43
this procedure exceeds the standard of 15
** With an Average Maximum Span of Reference of 20
this procedure exceeds the standard of 10
** With a Volume of 7619
this procedure exceeds the standard of 3200
** With a comment percentage of 12,0
this procedure does not meet the standard of 60,0
Exception(s) for procedure: XSRError
** With a comment percentage of 14,3
this procedure does not meet the standard of 60,0
Exception(s) for procedure: rescale
** With a comment percentage of 0,0
```

FIGURE 9 Analyzing the Exception Report

2.1.9 STEP 9: Viewing an Error Report

METRICTM also produces an **Error** report. If an error is created during processing and analysis, those errors will be listed in this report. To look an **Errors** report:

1. Click on the **Report** pull-down menu.
2. Select **Errors**.
3. *METRICTM* automatically generates an **Errors** report.
4. The error message is

Error file.

<<< Unable to open include file icon >>>

This is not a serious error, so we don't have to worry about it.

After obtaining an **Errors** report your display should like this:

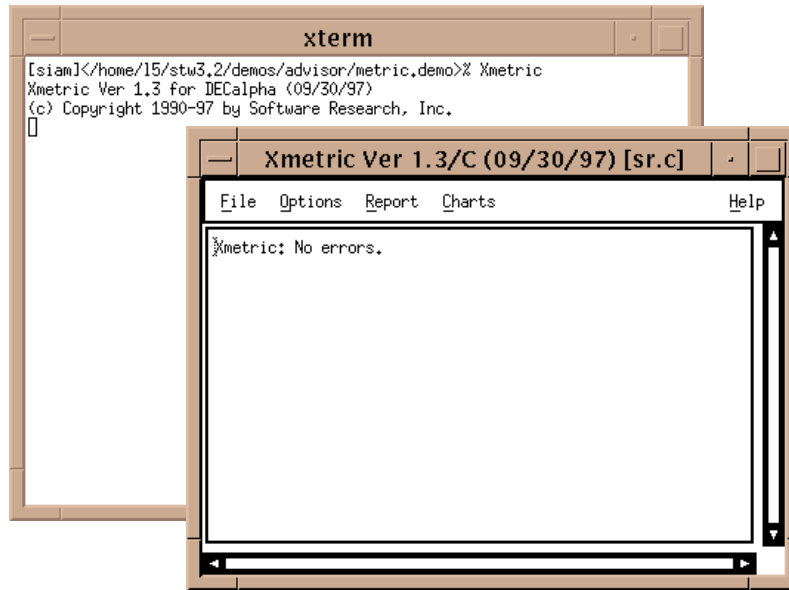


FIGURE 10 Viewing the Errors Report

2.1.10 STEP 10: Using Kiviat Charts to See Complexity

With *METRICTM*, you can look at Kiviat diagrams. Kiviat diagrams provide a graphical means to view the impact of multiple metrics on source code files. *METRICTM* produces three types of Kiviat diagrams. These diagrams represent information from the **Summary** report. This step looks at the first type.

To look at a Kiviat chart:

1. Click on the **Charts** option.
2. Select **Type 1**.
3. A window displaying a Kiviat chart pops up. **You may have to resize it.** Move the window to the lower left of the screen.
4. The inner circle represents minimum values, the outer circle represents maximum values and radii through the circles represent the metrics of interest.

In this Kiviat chart, the metrics of interest are

- Unique Operators (n1).
- Unique Operators (n2).
- Total Operators (N1).
- Total Operands (N2).
- Lines of Code (LOC).
- Number of Comment Lines (CMT).
- Number of Blank Lines (BLK).
- Number of Executable Semi-Colons (<i>).
- Number of Functions.

These values are defined in a file named `xmetric.I.def` and can also be set in the **Type I** window with the `xmetric` GUI (See Section 4.6 - “Graphically Viewing Complexity” on page 100.)

Observed values are plotted on the radii and connected. From this, metrics that are not within the acceptable range of values can be easily identified. In this example, some of the complexity measure fall above the upper threshold.

5. To close the Kiviat chart, click on the **File** pull-down menu and select **Exit**. The Kiviat chart closes.

When looking at the **Type I** Kiviat chart, your display should like this:

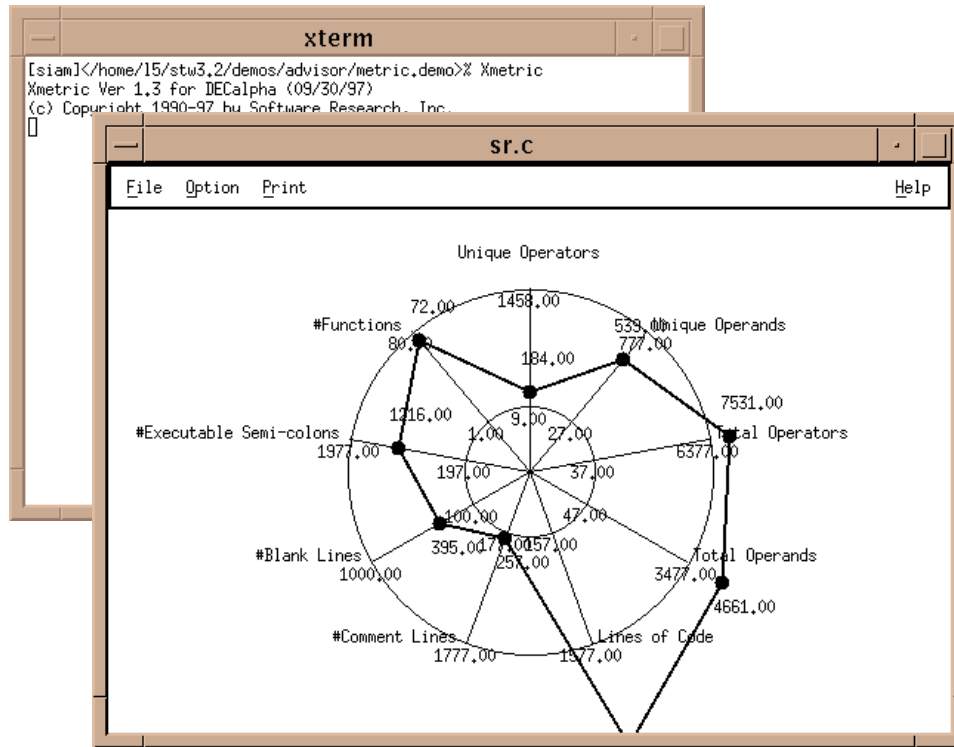


FIGURE 11 Type I Kiviat Chart

2.1.11 STEP 11: Looking at the Type II Chart

*METRIC*TM also offers two other Kiviat charts. These charts offer different metrics than the **Type I** chart, and may give us more insight into the programs. This step looks at the **Type II** chart.

1. Click on the **Charts** pull-down menu.
2. Select **Type II**.
3. A window with the Kiviat chart pops up. **You may have to resize it.** Move it to the lower left of the screen.
4. In this Kiviat chart, the metrics of interest are
 - Length (N).
 - Predicted Length (N[^]).
 - Purity Ratio (P/R).
 - Estimated Effort (E).
 - Estimated Errors (E[^]).
 - Estimated Time to Develop (T[^]).
 - Cyclomatic Complexity (VG1).
 - Extended Cyclomatic Complexity (VG2).
 - Average Cyclomatic Complexity.
 - Average Extended Cyclomatic Complexity.

These values are defined in a file named `xmetric.II.def` and can also be set in the **Type II** window with the `xmetric` GUI (See Section 4.6 - “Graphically Viewing Complexity” on page 100.).

5. To close the Kiviat chart, click on the **File** pull-down menu and select **Exit**. The Kiviat chart closes.

When looking at the **Type II** Kiviati chart, your display should like this:

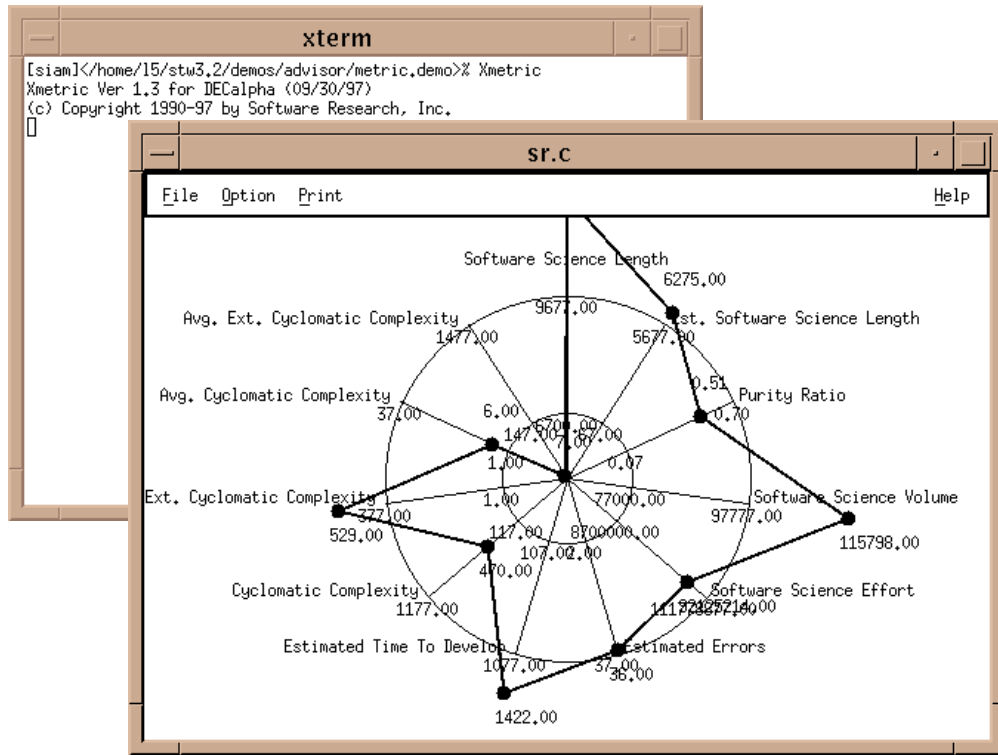


FIGURE 12 Type II Kiviati Chart

2.1.12 STEP 12: Looking at the Type III Chart

The most comprehensive Kiviat chart is the **Type III** chart. This step looks at the **Type III** chart.

1. Click on the **Charts** pull-down menu.
2. Select **Type III**.
3. A window with the Kiviat chart pops up. **You may have to resize it.** Move it to the lower left of the screen.
4. This chart graphically displays all of the metrics from the **Summary** report. (See Section 2.1.7 - "STEP 7: Viewing a Summary Report" on page 17.)

These values are defined in a file named `xmetric.III.def` and can also be set in the **Type III** window with the `xmetric` GUI (See Section 4.6 - "Graphically Viewing Complexity" on page 100.).

In the case of this example, note how some of the metrics values are plotted above and below the threshold circles.

At this point, it is useful to determine which metrics do not fall within the threshold limits, then look at a **Complexity** report and order procedures according to the metrics in violation. In so doing, you can easily determine which modules are causing the complexity.

5. To close the Kiviat chart, click on the **File** pull-down menu and select **Exit**. The Kiviat chart closes.

When looking at the **Type III** Kiviat chart, your display should look like this:

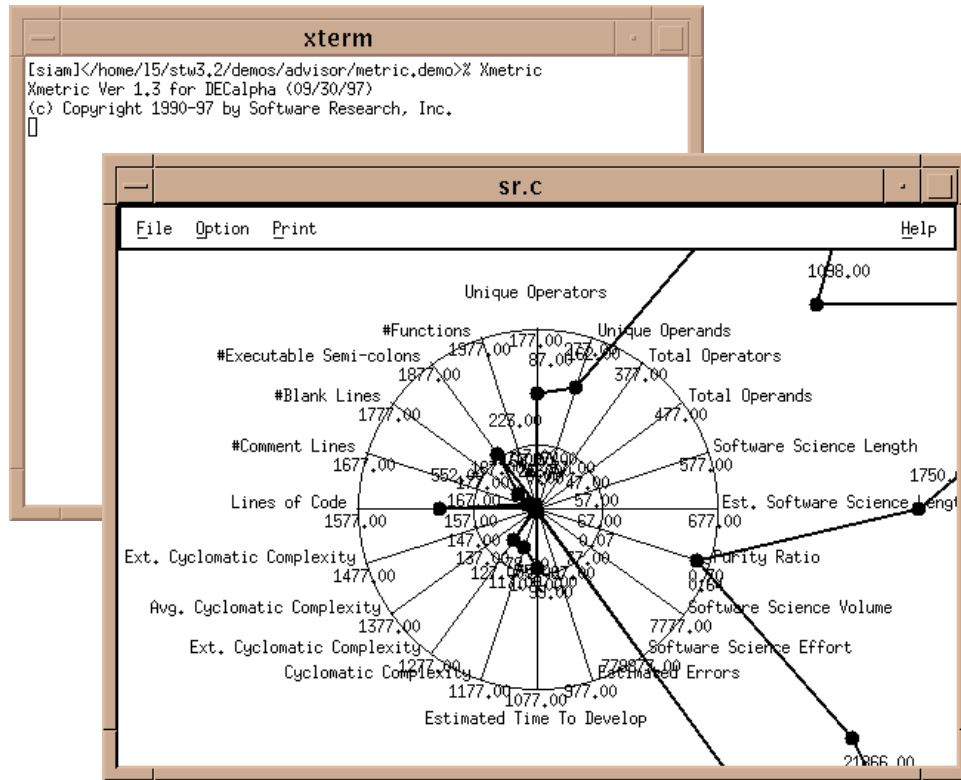


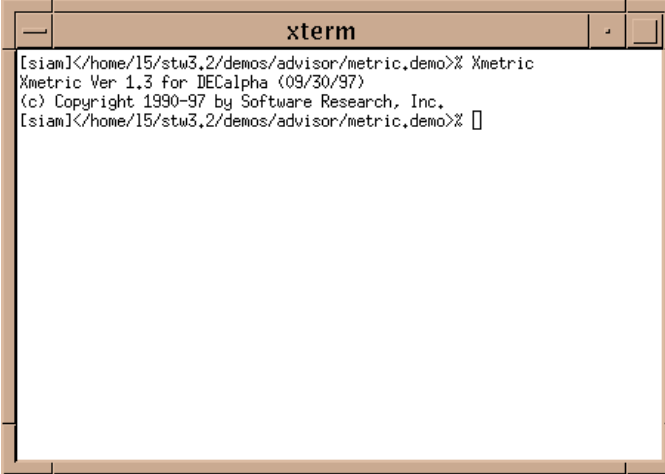
FIGURE 13 Type III Kiviat Chart

2.1.13 STEP 13: Signoff and Cleanup

To complete this session:

1. Click on **Main** window's **File** pull-down window.
2. Select **Exit**.

After you finish a *METRICTM* session, your display should look like this:



```
[siam]~/home/15/stw3,2/demos/advisor/metric.demo% Xmetric
Xmetric Ver 1.3 for DECalpha (09/30/97)
(c) Copyright 1990-97 by Software Research, Inc.
[siam]~/home/15/stw3,2/demos/advisor/metric.demo% █
```

FIGURE 14 Completing a METRIC Session

2.2 Summary

If you successfully completed the preceding 13 steps, you've seen and practiced the basic skills you need to use *METRICTM* productively. In this chapter you should have learned how to invoke *METRICTM*, how to load single file and multiple files, how to analyze a **Complexity** report, how to obtain a **Summary**, **Exception** and **Errors** reports, and how to study a program's complexity with Kiviat charts.

For best learning, you may want to

- Repeat STEPS 1 - 13 without the manual.
- Repeat STEPS 1 - 13 with your application.
- Turn to the chapters on system operation reference and GUI reference where you had difficulties and to learn about other features. (See CHAPTER 4 - "System Operation" on page 71.) (See CHAPTER 6 - "Graphical User Interface" on page 125.) The table of contents and the index can help you locate the topic you want.
- Use the supplied `metric.ksv` file to watch the session run:

To use the supplied `metric.ksv` file, initialize two xterm-type windows by using the mouse to click on **New Windows** or issuing the command `xterm &` from an existing window. Use the mouse to move one to the upper left corner and the other to lower left corner (as shown on the following page).

Then type the command:

```
xplabak -S -k metric.ksv
```

in the lower left xterm window. This command will issue a call to *CAPBAK/XTM* to playback the same 13 steps you went through. While `xplabak` is playing back the session, do not interrupt the keyboard and mouse input. Playback is done when you see the message, "Playback complete." appearing on the lower left window.

When using the supplied `metric.ksv` file to playback a *METRICTM* session, your display should look like this:

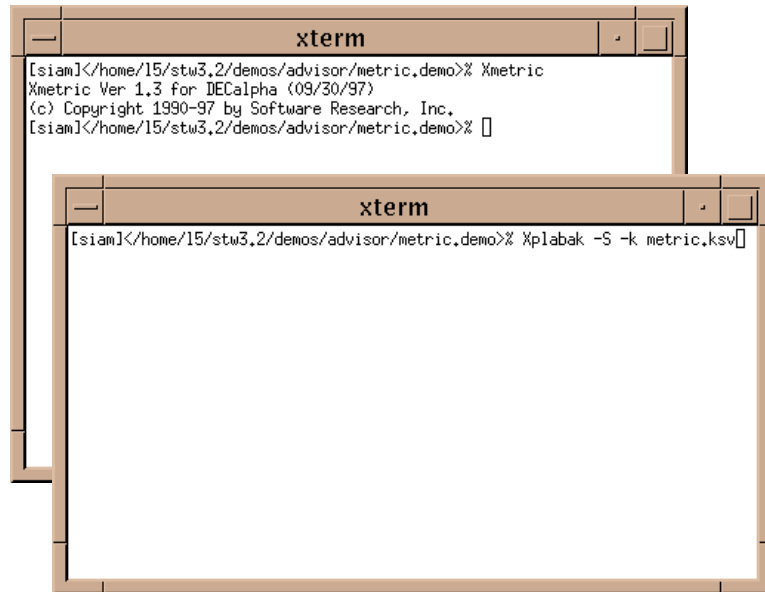


FIGURE 15 metric.ksv Setup

System Introduction

This chapter is an overview of the *METRICTM* system. It explains the overall operation of *METRICTM*, and in particular, it explains the complexity measures and what their values mean.

LEVEL: All level users may want to refer to this chapter for reviews of the complexity measures. Beginners may find the introductory material useful.

3.1 Overview of METRIC

METRICTM takes your program and computes several complexity metrics, including the metrics known as Software Science and Cyclomatic Complexity. These metrics have been researched in dozens of major studies over the last ten years and found to be highly correlated to programming errors and maintenance effort.

METRICTM identifies the most complex modules through its reports: the **Complexity** report, the **Summary** report, and the **Exception** report. The **Complexity** report provides a set of complexity metrics for each of the modules in a given source code file. The **Summary** report reports the complexity measurements for the entire source code program, not for individual procedures or functions. The **Exception** report lists where code exceeds set threshold limits of complexity metric standards. These reports identify the most complex modules for you.

In a testing situation this is ideal, because you can design your test cases to more thoroughly test the most complex modules. You can then use *STW/CoverageTM* to test the thoroughness of your test cases.

3.2 How to Use METRIC

To use *METRICTM* all you have to do is specify the source code file(s) you want *METRICTM* to process. *METRICTM* is a static code analyzer, so you do not have to alter or change your code. *METRICTM* automatically creates reports and kiviats charts for the file(s) being analyzed. This chapter will deal specifically with the reports and the kiviats charts.

NOTE: Please note that there are differences between the languages. Please refer to "APPENDIX A" for "C" information, "APPENDIX B" for "C++" information, "APPENDIX C" for Ada information, and "APPENDIX D" for FORTRAN information.

3.3 The Complexity Report

The **Complexity** report provides a set of complexity metrics for each of the modules in a given source code file(s). It uses two sets of complexity metric; the **Software Science** family of metrics and the **Cyclomatic Complexity** metrics. Software metrics are based on the “size” of the software, while Cyclomatic Complexity metrics are based on the flows of control within the code.

Below is a **Complexity** report generated for a “C” program.

Procedure	n1	n2	N1	N2	N	N [^]	P/R	V	E	VG1	VG2	LOC
parse_double	8	6	11	6	17	40	2.32	65	259	1	1	11
open_the_display	21	15	57	26	83	151	1.82	429	7810	6	7	28
main	108	211	1404	840	2244	2359	1.05	18664	4012377	83	113	412
Syntax	6	18	67	34	101	91	0.90	463	2624	1	1	21
XCalcError	6	7	12	7	19	35	1.85	70	211	1	1	8
SetupTICalc	28	53	174	131	305	438	1.44	1934	66912	20	20	40
SetupHPCalc	28	56	175	133	308	460	1.49	1969	65464	23	23	40
DrawDisplay	24	48	191	132	323	378	1.17	1993	65765	12	12	30
DrawKey	22	22	49	37	86	196	2.28	470	8686	3	5	18
InvertKey	9	9	17	13	30	57	1.90	125	813	1	1	5
LetGoKey	28	21	100	52	152	227	1.49	853	29586	8	10	60
digit	29	26	114	66	180	263	1.46	1041	38304	9	15	30
bkspf	18	6	33	14	47	91	1.93	215	4525	4	5	12
decf	13	11	32	15	47	86	1.83	215	1910	4	4	18
eef	15	12	36	19	55	102	1.85	262	3106	5	5	18
clearf	13	12	29	16	45	91	2.03	209	1811	2	3	14
negf	18	14	60	31	91	128	1.41	455	9068	6	6	28
twoop	33	23	162	67	229	271	1.18	1330	63921	13	13	60
twof	28	22	87	42	129	233	1.80	728	19459	10	10	28
entrf	13	12	31	19	50	91	1.82	232	2390	4	4	18
equf	29	22	104	46	150	239	1.59	851	25797	10	10	40

FIGURE 16 Sample Complexity Report

The report includes the following fields:

- Procedure Name
- Unique Operators (n1)
- Unique Operands (n2)
- Total Operators (N1)
- Total Operands (N2)
- Length (N)
- Predicted Length (N[^])
- Purity Ratio – estimated length divided by length (P/R)

- Volume (V)
- Effort (E)
- Cyclomatic Complexity (VG1)
- Extended Cyclomatic Complexity (VG2)
- Lines of Code (LOC)
- Number of Comment Lines (CMT)
- Number of Blank Lines (BLK)
- Number of Executable Semi-Colons (< ; >)
- Average Maximum Span of Reference of Variables (SP)
- Variable Name Length (VL)

Those procedures with the highest field values are the most complex values. The more complicated fields are discussed next.

3.3.1 Fields n_1 , n_2 , N_1 , N_2 , and N

In the early 1970s, Maurice Halstead, of Purdue University, observed that all programs were made up of **operators** and **operands**. He recognized that simply counting the number of lines in a program did not accurately measure how difficult it might be to work with. Therefore, by measuring the number of operators and operands used in the program, one could better measure the complexity of the code.

Halstead defined the following four parameters upon which the rest of his theoretical framework was built:

- n_1 number of unique operators
- n_2 number of unique operands
- N_1 number of total operators
- N_2 number of total operands

In addition, he defined the *vocabulary* of the program, n as being the number of unique operators and operands used, or:

$$n = n_1 + n_2$$

Likewise, he defined the *Length* of the program, N to be the total number of operators and operands used, or:

$$N = N_1 + N_2$$

3.3.2 Software Science Counting Rules

Halstead originally felt that the executable statements were the most significant in terms of complexity, and so did not include declarations and specification statements within his counts. The following Pascal procedure would possess the indicated Software Science values:

```
function Fold(ch:char):ch;
begin
  if ch in ['A'..'Z'] then
    Fold:=chr(ord(ch)+ord('a'))
  else
    Fold:=ch;
end; (* Fold *)
```

```
n1 = 12          n2 = 5
N1 = 17          N2 = 8
n = 17           N = 25
```

These measures were arrived at in the following manner:

```
Operators#Operands#
begin/end1ch3
if/then1'A'1
in1'Z'1
[]1Fold2
..1'a'1
:=2
chr1
()i3
ord2
+1
else1
;2
```

Note that those items which are always paired together (e.g., (), [], begin/end, etc.) are counted as a single item. Likewise, note that procedure and function invocations are treated as operators.

Exactly what is to be counted as an operator and an operand is in many cases left to an individual's opinion, since some things could reasonably be counted as either. For example, how does one count the statement `goto xyz`? Is `goto` an operator and `xyz` an operand, or is the entire `goto xyz` an operator?

Luckily, most work in this field suggests that minor differences in counting rules have little impact upon the effectiveness of Software Science as long as the rules are consistent. It is usually best if a specific set of counting rules can be formulated, and then those rules used for all subsequent work. A study by Nancy Currans of Hewlett-Packard Corporation, presented at the *1986 Pacific Northwest Software Quality Conference* supports this view. Currans computed various Software Science measures using three different sets of counting rules for a software system consisting of over 30,000 lines of C. She found that as long as the rules were applied consistently, the results were consistent among the three strategies.

Specific counting rules have been formulated for many languages. For example, Norman Salt formulated a set of counting rules for Pascal which were published in the March 1982 issue of the *ACM SIGPLAN Notices*. An alternate set of counting rules for Pascal were proposed by Melton and Ramamurthy in 1984. Yet another set of counting rules for Pascal can be found in the book *Software Engineering Metrics and Models* by Conte, Dunsmore and Shen. The particular set of rules selected probably does not matter as long as they are applied consistently.

3.3.3 Fields N^{\wedge} and P/R

Halstead was able to derive a large number of interesting relationships from the four basic parameters that Software Science is built upon. Perhaps one of the most interesting is the *Predicted Length*, N^{\wedge} (pronounced “N-hat” - throughout Software Science, estimated parameters are suffixed by the “hat” character to indicate they are an estimate). Halstead theorized that a well-written program with $n1$ unique operators and $n2$ unique operands should have a length of approximately:

$$N^{\wedge} = [n1 \times \log_2(n1)] + [n2 \times \log_2(n2)]$$

This is known as the *length equation*. Halstead suggested that programs which are not the same length as predicted by N^{\wedge} are victims of *impurities*. Six classes of impurities exist:

1. Canceling of operators,
2. Ambiguous operands,
3. Synonymous operands,
4. Common subexpressions,
5. Unnecessary replacements,
6. Unfactored expressions.

The *Purity Ratio* is the ratio of N^{\wedge} to N (ie, N^{\wedge}/N). This is a rough measure of the degree to which impurities exist in a piece of code. A Purity Ratio of 1 suggests few impurities exist. This measure has been the subject of few empirical studies, however it seems in theory to be a reasonable measure of style.

The Length Equation is interesting because it suggests that the major factor which determines program length is the number of unique operators and operands. This is important because often we might have an idea of how many variables we are going to use in a program *before* we write it. Since the number of operators available in most programming languages is finite, and most programmers have their own small set of operators that they regularly use anyway, the unique number of operators can be treated as a constant. Therefore, if we can come up with an accurate estimate of how many operands are going to be used in our program, a reasonable prediction of how large it is going to be can be made.

High correlations between N^{\wedge} and N have been reported in study after study. Correlations of .90 and above are not uncommon. The reported correlation seldom drops below .80 in studies involving traditional programming languages.

A nice property of variables which are highly correlated is that another number may be added to or multiplied by the numbers in one of the columns (as long as it is done to all the numbers in the column), without changing the relationship. The length equation can benefit from this property. The N^{\wedge} value can be multiplied by some "correction factor" to come up with a better absolute predictor.

Even though the length equation is highly correlated with the observed length, like all the techniques we are going to be examining in this tutorial, it may not be a satisfactory predictor for any specific program. Rather, its predictions only hold in a statistical sense over a large number of programs. In this respect, software metrics and their application are very much like the actuarial tables used by insurance companies to assess life insurance premiums.

For example, if a 35 year old male has a life expectancy of 71 years, it does not necessarily mean all, or for that matter, any particular 35 year old male will die at age 71. However, if the insurance company charges premiums that will fully cover the amount of the policy within 36 years they will not lose money in the long run. Some of the 35 year old males they insure will die before they reach 71, and hence before paying off the policy (a loss for the insurance company) while others will live beyond 71, and will continue to pay their premiums even after they have fully paid the policy amount (a gain for the insurance company).

Software metrics must be applied in the same spirit. They may not work for a particular program, but if applied over a large number of programs, they will be right more often than not.

3.3.4 Field V

Another interesting relationship Halstead hypothesized is one called *Volume* or V . The idea behind program Volume is simple. If a program has n unique operators and operands, then it would take $\log_2(n)$ “bits” to uniquely represent each. If there are N total usages of those operators and operands, then the number of “bits” to represent the program is:

$$V = N \times \log_2 n$$

Halstead felt that this would be a reasonable measure of program size. He suggested it as an alternative to a simple count of operators and operands (N) since it tends to “penalize” programs with a large number of unique operands and/or operators.

For example, consider two programs, each which consist of 100 uses of operators and operands (ie, $N = 100$). However, the first program only uses 8 different unique operators and operands, while the second uses 64 different operators and operands. The Volume of the first program is:

$$100 \times \log_2(8) = 300$$

while the Volume of the second program is:

$$100 \times \log_2(64) = 600$$

Thus, the second program could be considered twice as complex as the first. This seems to agree with our intuition since a program with 64 different operators and operands (especially if most of them are variables) to keep straight in our minds would seem to be more difficult to work with and understand than a program with only 8.

3.3.5 Field E

Another measure suggested by Halstead is the abstraction level of a program. The abstraction level, L (sometimes referred to as the *Program Level* in the literature) of a program is calculated as:

$$L = V^*/V$$

where V is the Volume measure, and V^* is called the *Potential Volume*. A program's Potential Volume is the Volume it would have if it were implemented as a library function within the programming language (in other words, a procedure call). The highest possible level of abstraction would result in an L of 1 (the actual Volume is equal to the Potential Volume), with programs of lower levels of abstraction having an L of less than 1.

As an example, consider the following invocation of a library routine which is passed an unsorted array, and returns an ordered array:

```
SORT(Numbers)
```

This invocation would have a Potential Volume of:

$$3 \times \log_2(3) = 4.74$$

Naturally, the implementation of the procedure would yield a much larger actual Volume, resulting in a Program Level of much less than 1.

Unfortunately, for large programs, especially those analyzed after the fact, the Program Level can be difficult to arrive at. For this reason, Halstead proposed an estimation (L^{\wedge}) that could be calculated from the basic parameters, n_1 , n_2 and N_2 .

$$L^{\wedge} = 2/n_1 \times n_1/N_2$$

Because L^{\wedge} can be derived from a simple analysis of the source program without having to know much about the design or application, it is usually used in most studies in place of L .

Yet another measure derived from the four Software Science parameters is something called *Effort*, or E . Effort is based somewhat on Volume, but is "adjusted" to account for the level of abstraction at which the program is written. Effort is calculated in the following manner:

$$E = V / L$$

with most researchers using L^{\wedge} in place of L .

Halstead suggested that the Effort measure reflected the number of *mental discriminations* that a programmer would have to perform in order to write the program.

3.3.6 Field VG1

Software Science tends to be based on program size. An alternative approach to assessing program complexity is to consider the program's *flow of control*. A program's flow of control is of course based on the number and arrangement of decision statements within the code.

A measure known as the *Cyclomatic Complexity* measures a program's control flow graph. It determines the complexity of a program's control flow.

A control flow graph is simply a variant of a program flowchart. The major difference is that while the ordinary flowchart might have a box, or *node* in the terminology of flowgraphs, for every statement in the program, a flowgraph only contains nodes for each *basic block* within the code. A basic block is a segment of code which is entered only at one point (the top), exited at one point (the bottom), and which has no transfers of control within it. Basic blocks often begin with decision-making statements, and end immediately before another decision-making statement. For example, the following C program segment:

```
scanf( "%d%d%d%d", a, b, c, d );      [A]
  if a==b {
    c=10*d;                          [B]
    scanf( "%d", d );                [B]
    c=10*d;                          [B]
  }
  else {
    scanf( "%d", d );                [C]
    c=25*d;                          [C]
  }
printf( "%d", c );                   [D]
```

can be represented by the flow graph on the next page (the statements that make up each node are shown in []):

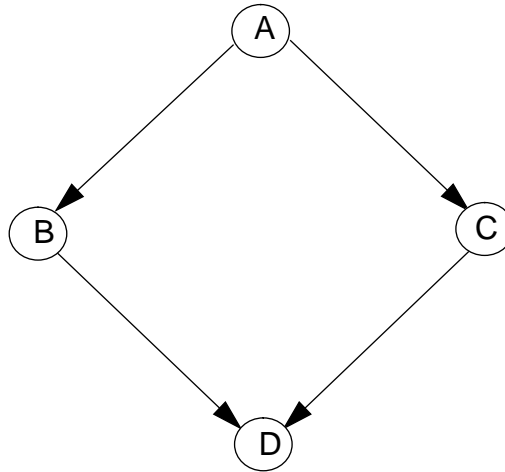


FIGURE 17 Flow Graph

The cyclomatic number of a flowgraph can be calculated as follows:

$$V(g) = e - n + 2$$

where n is the number of nodes in the graph, and e is the number of *edges* or lines connecting each node. In the example, n is 4, and e is 4, giving a cyclomatic number of 2.

The great popularity of this measure of complexity stems from the fact that one need not create the program flowgraph in order to compute the Cyclomatic Complexity, but instead can simply sum the number of decision making statements (ie, IF, WHILE, FOR, REPEAT, etc.), and add 1. In the case of the previous example, there is 1 IF statement, and adding 1 gives 2. This “shortcut” method of calculation has made the Cyclomatic Complexity easy to calculate, and thus it appears in almost every complexity metric study ever performed.

3.3.7 Field VG2

Suggestions have been made to extend the Cyclomatic Complexity so that it includes not just decision making statements, but also decision making *predicates*. The first such suggestion was made by Glenford Myers of IBM in the October 1977 issue of *ACM SIGPLAN Notices*. Myers suggested that two numbers be provided in the measure, the first reflecting the number of decisions, and the second reflecting the number of simple conditions. This measure is sometimes referred to as the *Extended Cyclomatic Complexity*. This is illustrated by the following Pascal program segment:

```
function Type(ch:char):ch;
begin
  if(ch in ['A'..'Z'])or
    (ch in ['a'..'z']) then
    Type:='c'
  else
    Type:='n';
end;

Myers' Extended
Complexity: 2:3
```

Thus, the corresponding short cut calculation would be:

(number of decisions) + 1 : (the number decisions & ANDs & ORs) + 1

Regardless of its exact form, numerous studies have assessed the validity of the Cyclomatic Complexity. Most results suggest that it is highly correlated with selected process metrics such as time to find errors, number of errors, effort to maintain, effort to test, etc.

3.3.8 Fields LOC, BLK, CMT, and <;>

LOC includes all lines, including blank and comment lines, from the procedure declaration to the last statement of the procedure, }.

BLK includes the total number of blank lines of code.

CMT includes the count of all comments on a line by themselves encountered within the body of the procedure. If a single comment spans multiple lines, the number of lines that it spans is added to the comment count.

<;> begins counting with the first executable line of code. Hence, all declarations are not included. FOR loops will contribute two executable semicolons.

3.3.9 Fields SP and VL

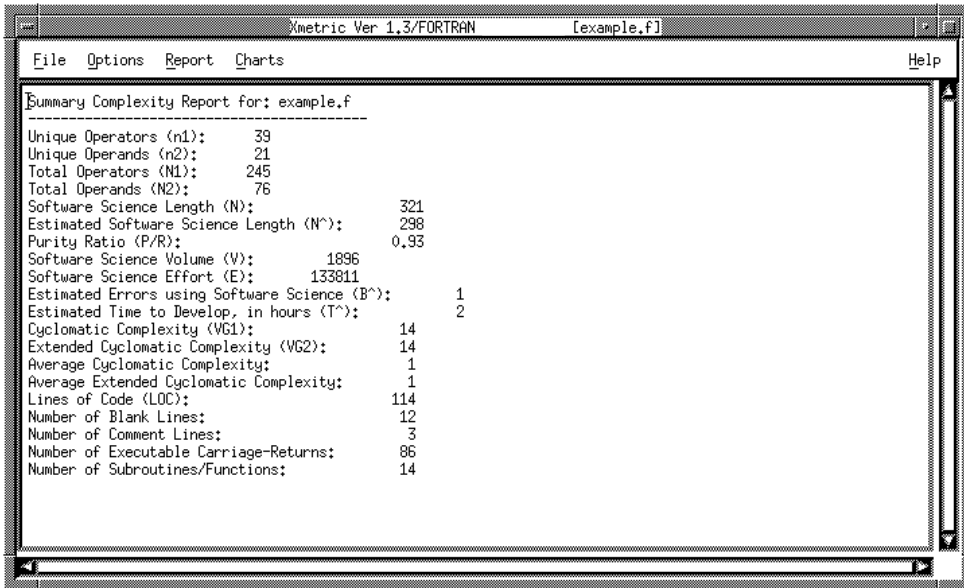
The average maximum span of reference *SP* counts the maximum number of lines between references to each variable in a procedure (either use or assignment). The average of all the maximum references is then compared. This average is listed in the report.

The variable name length (*VL*) value can be calculated in two ways, based on the configuration file entry `UNIQUE_VARIABLES`. If `UNIQUE_VARIABLES` is set to 1, then the length of all unique variable names (used at least once) is divided by the number of unique variables. If `UNIQUE_VARIABLES` is set to 0, then a weighted calculation is done.

With this method, for all variables, the length of the variable name is multiplied by the number of times the variable is used and this sum is divided by the total count of variable usage.

3.4 The Summary Report

The **Summary** report provides a set of complexity metrics for the entire source code file. Below is a sample FORTRAN **Summary** report. Like the **Complexity** report, there are slight differences between the languages. Please refer to the appropriate language appendix.



```

Metric Ver 1.3/FORTRAN [example.f]
File Options Report Charts Help
-----
Summary Complexity Report for: example.f
-----
Unique Operators (n1):      39
Unique Operands (n2):     21
Total Operators (N1):     245
Total Operands (N2):      76
Software Science Length (N):      321
Estimated Software Science Length (N^): 298
Purity Ratio (P/R):        0.93
Software Science Volume (V):    1896
Software Science Effort (E):    133811
Estimated Errors using Software Science (B^): 1
Estimated Time to Develop, in hours (T^): 2
Cyclomatic Complexity (VG1):    14
Extended Cyclomatic Complexity (VG2): 14
Average Cyclomatic Complexity:  1
Average Extended Cyclomatic Complexity: 1
Lines of Code (LOC):          114
Number of Blank Lines:       12
Number of Comment Lines:     3
Number of Executable Carriage>Returns: 86
Number of Subroutines/Functions: 14

```

FIGURE 18 Sample Summary Report

The report includes the following fields:

- Unique Operators (n1)
- Unique Operands (n2)
- Total Operators (N1)
- Total Operands (N2)
- Software Science Length (N)
- Estimated Software Science Length (N[^])
- Purity Ratio (P/R)
- Software Science Volume (V)
- Software Science Effort (E)

- Estimated Errors Using Software Science (B^{\wedge})
- Estimated Time to Develop, in hours (T^{\wedge})
- Cyclomatic Complexity (VG1)
- Extended Cyclomatic Complexity (VG2)
- Average Cyclomatic Complexity
- Average Extended Cyclomatic Complexity
- Lines of Code (LOC)
- Number of Comment Lines (CMT)
- Number of Blank Lines (BLK)
- Number of Executable Semi-Colons ($\langle ; \rangle$)
- Number of Procedures/Functions

The **Summary** report consists of the same complexity measures as the **Complexity** report (See Section 3.3 - “The Complexity Report” on page 41.), with the exception of the average minimum span of reference of variables (SP) and variable name length (VL). In addition, it includes the values:

- Estimated errors using Software Science (B^{\wedge})
- Estimated time to develop in hours (T^{\wedge})
- Average Cyclomatic Complexity
- Average Extended Cyclomatic Complexity
- Number of Procedures/Functions

Note that Software Science length (N) and Software Science predicted length (N^{\wedge}) should not be confused with lines of code. The Software Science lengths are based on the number of operators and operands whereas lines of code are the actual number of physical lines in the source file.

Lines of code for the **Summary** report contains all lines including blank and comment lines from the beginning of the file to the end (including lines between procedures). Because of this, lines of code in the **Summary** report will often be greater than the sum of lines of code listed with procedures.

The sum of certain parameters listed in the procedure list will not usually match the corresponding values listed in the summary report. For example, $n1$, $n2$, $VG1$, $VG2$, etc. This is to be expected and should not alarm the user. This phenomenon is due to the mathematical definitions of the measures.

The fields in the reports that should match between the **Summary** report and the **Complexity** report are total operators, total operands, and executable semi-colons.

B^{\wedge} and T^{\wedge} take the **Complexity** report's Volume and Effort values and go even further. These fields are discussed next.

3.4.1 Field B^{\wedge}

You may want to read the section on Volume (See Section 3.3.4 - "Field V" on page 48.) prior to reading about B^{\wedge} .

Another popular interpretation of Volume suggests a program possessing n unique operators and operands, and N total operators and operands would require at most

$$N \times \log_2 (n)$$

mental lookups to fully read and understand the entire program. This is because each of the N tokens encountered in the code would require $\log_2 (n)$ mental lookups to recognize using the most efficient search we know.

People tend to make mistakes, on the average, every E_0 *mental comparisons*. Therefore the number of errors B^{\wedge} , that would be expected in a program would be:

$$B^{\wedge} = [N \times \log_2 (n)] / E_0$$

Independent work by psychologists, as well as empirical studies carried out by Halstead and his students, suggest that an appropriate value for E_0 is around 3000-3200.

Thus, Software Science can predict, with some confidence, the number of *coding* errors that can be expected in a computer program, if the number of unique and total operators and operands used in the program are known.

Naturally, every program will not have *exactly* the number of errors suggested by the B^{\wedge} relation. By the same token, the number of errors predicted by B^{\wedge} is simply an estimate of how many errors existed in the code upon *completion of the coding phase*. Because B^{\wedge} predicts the number of errors which existed in the code at the completion of coding, removal of errors during testing will not, in itself, change the value of B^{\wedge} .

Obviously, other factors besides the number of operators and operands used in a program affect the number of errors in a piece of code. The B^{\wedge} relation rests on a number of assumptions, including a familiarity of the programmer with the programming language, system, and application area, as well as what might be characterized as "average ability". Very good (or very poor) programmers will obviously produce code that is less error-prone (or more error-prone) than suggested by B^{\wedge} . By the same token, an application which is new to a programmer might well be more error prone than suggested by B^{\wedge} . The B^{\wedge} relation likewise cannot reflect requirement or specification errors since they typically do not manifest themselves in the actual code (though the idea of applying these techniques to the actual requirements and/or specification documents is intriguing).

However, B^{\wedge} gives one a reasonable starting point to work from. By adjustments to the $E_{sub 0}$ factor, one may customize the relation for a particular environment.

3.4.2 Field T[^]

You may want to read the section on Effort prior to reading about T[^] (See Section 3.3.5 - “Field E” on page 49.).

Independent work by a psychologist named Stroud found that humans are capable of making up to 20 mental discriminations per second. This suggests that by dividing Effort by the *Stroud Number*, we can estimate how long the program *should* have taken to write.

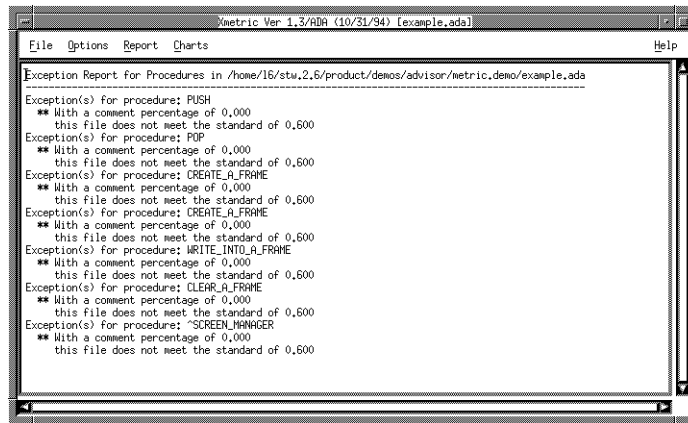
In its general form, the estimated time, in seconds, is calculated as:

$$T^{\wedge} = E/S$$

where *s* is the Stroud Number. Naturally, depending on the environment, the Stroud Number may vary greatly. Stroud’s studies found a range of 5 to 20, in situations not involving programmers. Studies by Halstead and his colleagues involving programmers found a value of 18 seemed to work best.

3.5 The Exception Report

The **Exception** report lists each procedure in the source code file which exceeds a set of predefined complexity maximums. This report uses either the default standards or standards specified in the configuration file, `.uxmetriccfg`, to determine what the maximum complexities are. You may edit the configuration file. Certain of the defaults are also specified in the GUI's **Configuration Options** window (use the **Options** pull-down menu to initiate), which is user-editable. Below is a sample Ada **Exception** report.



```
uxmetric Ver 1.3/ADR (10/31/94) [example.ada]
File Options Report Charts Help
-----
Exception Report for Procedures in /home/15/stw,2.6/product/demos/advisor/metric_demo/example.ada
-----
Exception(s) for procedure: PUSH
** With a comment percentage of 0,000
   this file does not meet the standard of 0,600
Exception(s) for procedure: POP
** With a comment percentage of 0,000
   this file does not meet the standard of 0,600
Exception(s) for procedure: CREATE_R_FRAME
** With a comment percentage of 0,000
   this file does not meet the standard of 0,600
Exception(s) for procedure: WRITE_INTO_R_FRAME
** With a comment percentage of 0,000
   this file does not meet the standard of 0,600
Exception(s) for procedure: CLEAR_R_FRAME
** With a comment percentage of 0,000
   this file does not meet the standard of 0,600
Exception(s) for procedure: ^SCREEN_MANAGER
** With a comment percentage of 0,000
   this file does not meet the standard of 0,600
```

FIGURE 19 Sample Exception Report

The possible messages that will appear in this report when a standard is not met are:

- ** At x lines, this procedure is larger than the standard of n lines
- ** At x executable semicolons, this procedure is larger than the standard of n executable semicolons
- ** With a Cyclomatic Complexity of x this procedure exceeds the standard of n
- ** With an extended Cyclomatic Complexity of x this procedure exceeds the standard of n
- ** With an Average Maximum Span of Reference of x this procedure exceeds the standard of n
- ** With a Comment Percentage of x this procedure does not meet the standard of n
- ** With a Volume of x this procedure exceeds the standard of n
- ** This procedure contains x gotos

3.6 Accuracy of the Reports

Because of the significant effect environment, application, programmer ability, etc. can have on the act of programming, software measures are of little use out of the box. In other words, they can be used to rank software implementations in terms of complexity and other associated characteristics, but the numbers themselves have little meaning.

However, by having historical data available, many of the metrics can be tuned to reflect the environment in which they are being used. For example, Software Science B^{\wedge} and T^{\wedge} measures are most meaningful if data for past projects is available and can be used to tailor error rate and programmer speed. Likewise, excessive complexity levels for a given installation can be determined by identifying the most complex 5% of all programs. Thus, before a complete software metrics effort can be put into place, a software data collection effort must first be initiated. Among other things, development time and number of errors should be recorded for every major system.

If software metrics are approached in this manner, without expecting any “magic numbers” to fall out of the sky, we think you’ll find *METRICTM* a valuable addition to your set of programming and management tools.

METRICTM applies state of the art methods for objectively measuring the complexity of software. **Software Research** cannot guarantee that these methods are foolproof. In studies conducted by our staff, as well as others, the metrics included in our tool have been shown to perform well most of the time. However, differences from installation to installation can reduce the usefulness of measuring software complexity.

3.7 Using Metrics in Software Development

Up until now, we have had (hopefully) an interesting discussion about software metrics. However, your reaction might very well be:

How can I use software metrics?

This section will attempt to address this question.

3.7.1 Producing Less Complex Code

Perhaps the most directly applicable use of metrics is to help programmers produce code that is easier to understand and work with. This can be done by using metrics as a *feedback tool*, just like writers might use one of the style analysis packages currently on the market.

Thus, as the programmer codes, the complexity of each module completed can be measured. If the module exceeds some previously set limit of complexity, remedial action *might* be appropriate. Remedial action could take several forms:

- Recognizing that the module is overly complex, the programmer might consider using a different approach or algorithm that will result in less complex code. The use of complexity metrics can make the selection of alternate approaches easier since only one of the metrics might be above the pre-set complexity threshold. For example, an unacceptable Cyclomatic Complexity suggests that the control flow is too complex. Thus, an alternate algorithm should be selected that will address this problem. Naturally if several metrics are above the acceptable level, or no acceptable alternatives exist, other actions might be appropriate.
- If the complexity of the solution cannot be reduced, it might be beneficial to consider dividing the module up into several smaller ones, each of which may individually possess an acceptable level of complexity. This is of particular importance if the module appears to perform more than a single “function”.
- Naturally, one will probably never reduce the complexity of every module to acceptable limits. Some problems and their solutions are necessarily complex. In this case, complexity metrics can help identify modules which need especially thorough commenting.

Further, the metric or metrics which are beyond an acceptable level can suggest the aspects to address in the commenting. This approach ensures that the modules which need extra documentation are identified, and valuable time is not wasted doing extra documentation of modules which may be adequately explained using a minimum of commenting.

The main point is that *local* standards of “acceptable” complexity levels be established. The question is not *if* code is complex (all code will exhibit some complexity), but rather *how* complex code is. Numerous “magic numbers” representing maximum accepted complexity have been suggested in the literature. For example, a Cyclomatic Complexity of 10 is the maximum complexity a module should exhibit.

More important than some arbitrary number suggested by an “expert” is a level of maximum complexity that your entire programming staff can agree on and live with. Perhaps the best way to do this is through experimentation. Examine the complexity of existing modules and reflect on which ones were the most difficult to test, debug or maintain.

Another approach is followed by least one large organization which has created baseline complexity measures based on their best programmers’ code. Any programmer who submits a piece of code more than a standard deviation off the baseline for a code review must explain why the additional complexity is necessary. If a satisfactory answer is not provided, the programmer must rewrite the code.

3.7.2 Allocating Testing Resources

Typically, a small percentage of the modules in a code system have an inordinate percentage of the errors. Likewise, a small percentage of the modules account for fewer errors than would be expected. Often, a 15-70-15 rule is accepted by software testers. That is to say, 15% of the modules account for perhaps 25% of the errors, another 15% of the modules account for 5% of the errors, and the remaining 70% of the modules account for the remaining 70% of the errors.

To ensure effective allocation of testing resources, identifying those top 15% and bottom 15% of the modules is important. Then, the bulk of the resources allocated to test the less error-prone modules can be shifted to help test the more error-prone modules. Often, complexity metrics can be used as *one* of several tools to identify those modules whose testing resources can be beneficially redistributed. A good description and evaluation of this process can be found in "Using Software Metrics to Allocate Testing Resources" by Warren Harrison in the Spring 1988 issue of *The Journal of Management Information Systems*.

3.7.3 Managing Maintenance

After a program is completed, a process of on-going maintenance begins. Often, the amount of time (both chronological and total) required to make the specified changes must be estimated. Metrics can serve as *one* of many tools that can be used to help make such estimates. Metrics can be used to help identify other modules which share common characteristics, such as a V(g) of 10, or a Software Science Effort of 100,000. These modules can be referred to as *baseline* modules. Then, experience gained from the maintenance of the baseline modules can be applied to the new module to help assess the effort required to maintain it.

Another use of metrics is in what is called *preemptive rewriting*. Often, a decision must be made as to whether the entire module should be rewritten, or a modification should be made to the existing code. It is recognized by most that as changes are made to a piece of code, it becomes more and more complex. This phenomenon is usually referred to as *entropy*. At some point, it might be worth rewriting to avoid having to work with code that has been continuously patched and re-patched over time. Again, metrics can help identify those modules which should be rewritten, and those which could be profitably modified without a major rewrite.

Yet another use of metrics in software maintenance is their use in assigning modules to maintainers. Often a system which consists of multiple modules will be maintained by a group of programmers. The parts of the system each maintainer is responsible for can be assigned to level the complexity encountered by each person using metrics. This can be done by dividing the sum of the modules' complexities by the number of maintenance programmers. The resulting number should approximately equal the sum of the complexities of the modules assigned to each maintainer.

3.7.4 Expecting Too Much

Metrics should be viewed as one of *many* tools used to help manage the programming process. In all cases, *common sense* will aid you in determining if metric results seem reasonable.

As we have earlier remarked, applying metrics to one or two individual programs will be disappointing. Metrics work best when they can be applied to a large set of programs. Some good code will be flagged as bad by the metrics, and some bad code will not be identified by the metrics. This is to be expected.

System Operation

This chapter covers the basic X Window System graphical user interface operations of *METRICTM*. It demonstrates how to operate the user interface, how to invoke *METRICTM*, how to look at the available reports, set complexity measures thresholds, and how to read the Kiviat charts.

4.1 Using this Chapter

Use this chapter to look up answers to questions about *METRICTM*. The table of contents or the index can help you locate the topic you want. If you prefer to work from the command line, please refer to the chapter on *COMMAND LINE ACTIVATION*.

The first part of this chapter discusses the basics of the *METRICTM* graphical user interface. If you are familiar with the OSF/Motif GUI, you may go on to the next section.

4.2 User Interface

If you are familiar with the OSF/Motif style graphical user interface, you can go on to the next section. This section demonstrates using file selection dialog boxes, help menus, message dialog boxes, option menus, and pull-down menus.

File Selection Box

You must use file selection boxes to select the file(s) you want *METRIC*TM to analyze.

Refer to the next figure for each of the dialog box's components:

Filter entry box	Specifies a directory mask. When you click the Filter push button, the directory mask is used to filter files or directories that match this mask (or pattern).
Directories	Lists directories in path defined in the Filter entry box.
Files	Lists files in path defined in the Filter entry box.
Scroll Bars	Move up/down and side/side in the Directories and Files selection windows. You use them to search for the appropriate directory or file.

Selection entry box

Selects and enter file name.

Use the three push buttons at the bottom of the dialog box to issue commands:

OK	Accepts the file in the Selection entry box as the new file or the file to be opened and then exits the dialog box.
Filter	Applies the pattern you specified in the Filter entry box. It lists the directories and files that match that pattern.
Cancel	Cancel any selections made and then exits the dialog box. No file is selected as a result.

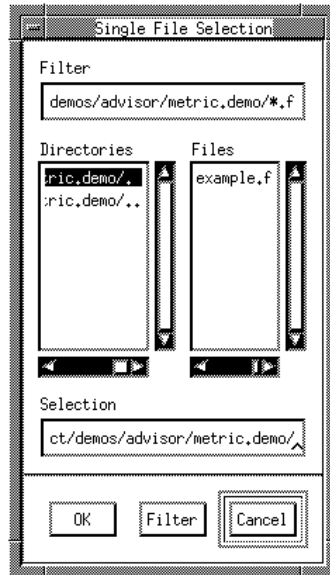


FIGURE 20 Using a File Selection Dialog Box

To use a file selection dialog box, follow these steps:

7. You can restrict the file selection operation to a named region (directory path) by typing in a directory path name in the **Filter** entry box or by clicking on a path name in the **Directories** selection window. Then click on the **Filter** push button.
8. Select a file by clicking on an already existing source file you want *METRICTM* to process in the **Files** selection window or type in the file name in the Selection entry box, with no limit on character length.
9. To select a source file name, do one of these three things:
 - Double click on the file in the File selection window,
 - Highlight the file in the File selection window or type in the file name in the Selection entry box and click **OK**, or
 - Highlight or type in the file name and press the <ENTER> key.

Help Boxes

*METRIC*TM provides one on-line help frame for its **Main** window and all dependent windows. This on-line help will automatically bring up the text corresponding to where you invoke it at. In other words, if you invoke it at the **Options** pull-down window's **Language** window, the **Help** window will automatically display information pertinent to the **Language** window.

Here's how to use a help frame:

1. Once it is invoked, the text should correspond to the window at which you invoke it.
2. You can use the scroll bars to move up/down and side/side.
3. If you don't see what you need, you can search for specific text. To do this:
 - Click on the **Action** pull-down menu and select **Search**.
 - A dialog box (shown below) pops up.
 - Type in the pattern you want to search for and then click on **OK** or press the <ENTER> key.
 - If the pattern is found, the help frame will automatically scroll to the location of the pattern.
4. If you select another **Help** option from another window, while the current one is displayed, the **Help** window will automatically scroll to the context of the new window.
5. To exit, click on **Quit**.

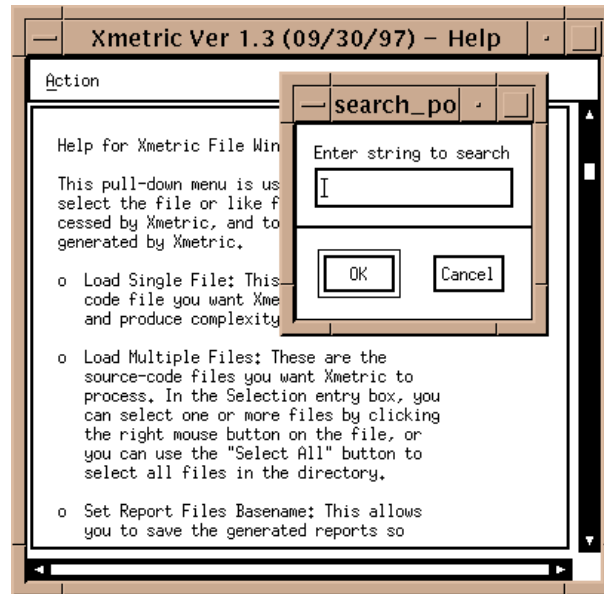


FIGURE 21 Using the Help Dialog Box

Message Boxes

Pop-up message dialog boxes have three purposes:

1. They display warnings and error information.
2. They ask you to verify that you want to perform a task.
3. They ask to enter a command.

To remove a message box after you have read it or to tell *METRICTM* to go ahead with a command, click the **OK** push button. If you want to cancel a command, click the **Cancel** push button.



FIGURE 22 Using a Dialog Box

Pull-Down Menus

Pull-down menus are located within the menu bar. They often contain several options. To use pull-down menus and their options, follow these steps:

1. Move the mouse pointer to the menu bar and over the menu containing the item.
2. Hold the left mouse button down. This displays the items on the menu.
3. While holding down the left mouse button, slide the mouse pointer to the menu item you want to select. The menu item is highlighted in reverse shadow.

Three dots at the right of the menu item indicates that selecting the item will bring up a pop-up window.

An arrow to the right of the menu item indicates that the item is a submenu (or cascading menu).

To display the submenu, slide the mouse pointer over the arrow. You can then select an item on the submenu.

4. Release the mouse button while the desired item is highlighted to activate the command. To the function exit without selecting anything, simply drag the mouse pointer off the menu before releasing the mouse button to not activate anything.

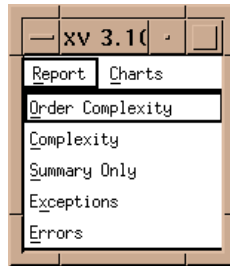


FIGURE 23 Using a Pull-down Menu

4.3 Invoking METRIC

To start *METRIC*TM from your working directory, type this command:

```
xmetric -L lang
```

where the default is set to the “C” language. The **Main** window (See Figure 24 “Invoking the Main Window” on page 78.) pops up, ready to process the language of your source code file(s).

If you invoke **xmetric** for the wrong language, you don’t have to quit your session. The **Option** pull-down menu has a **Language** window which allows you to select languages (See Section 4.4.1 - “Selecting a Language” on page 81.).

Note: *METRIC*TM can be customized through a combination of keyword files and configuration file parameters. This allows most language dialects to be accommodated. You can find keyword file information in the **APPENDIX** for the language you are working with. Configuration file information can be found in the appropriate section. (See Section 5.1.1 - “Set-Up Suggestions” on page 115.)

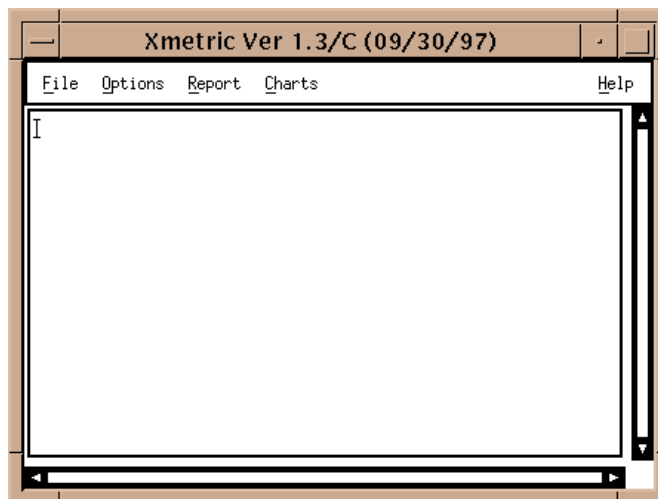


FIGURE 24 Invoking the Main Window

If you have the *STW*TM product tool set, you can invoke *METRIC*TM by typing the command:

stw

1. The *STW*TM window (shown in the next figure) pops up.
2. Click on the **Advisor** activation button.
3. The *STW/ADVTM* window pops up.
4. Click on **METRIC**. The *METRIC*TM window pops up.

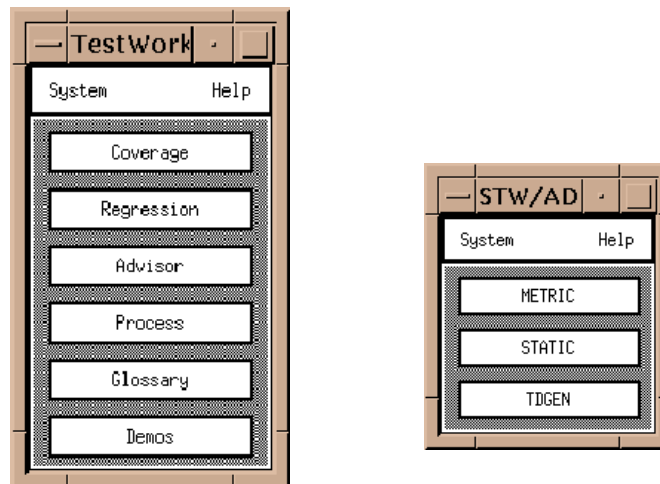


FIGURE 25 Invoking METRIC from STW

4.4 Processing a Source Code File

Because *METRICTM* is a static code analyzer, you do not have to do anything special to the code. To use *METRICTM*, all you have to do is make sure you have the language set, select a source code file name, and processing is automatic.

4.4.1 Selecting a Language

If you did not invoke *METRIC*TM with the correct language, you can easily change it. Here's how:

1. Click on the **Options** pull-down menu.
2. Select **Language**.
3. The **Select Language** window (shown below) pops up.
4. To change the language, simply click on the corresponding language radio button.
5. Click on **OK** to change the language *METRIC*TM will process or click on **Cancel**.

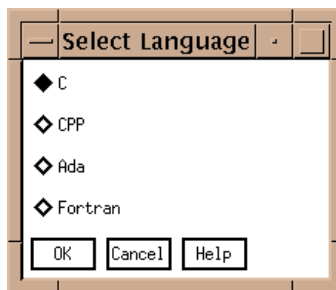


FIGURE 26 Selecting a Language

4.4.2 Writing Reports to a File

Before you select a source code file, you may want to first save the reports to files so that they share a specific prefix. If you choose not to do this, your reports will not be saved.

To be consistent with the command line, reports will be saved in the following manner:

- For “C”:
 - *filename.rpt* - The **Complexity** and the **Summary** reports.
 - *filename.exp* - The **Exception** report.
 - *filename.err* - The **Error** report.
- For “C++”:
 - *filename.rpt* - The **Complexity** and the **Summary** reports.
 - *filename.exp* - The **Exception** report.
 - *filename.err* - The **Error** report.
 - *filename.cls* - The **C++ Class** and **Class Summary** reports.
 - *filename.cht* - The **Class Hierarchy** report.
 - *filename.cex* - The **Class Exception** report.
- For Ada:
 - *filename.rpt* - The **Complexity**, the **Summary**, and the **Package Intermediates** reports.
 - *filename.exp* - The **Exception** report.
 - *filename.err* - The **Error** report.
 - *filename.gen* - The **Generic** report.
 - *filename.pex* - The **Package Exception** report.
- For FORTRAN:
 - *filename.rpt* - The **Complexity** and **Summary** reports.
 - *filename.exp* - The **Exception** report.
 - *filename.err* - The **Error** report.

Note: Customization features are available when generating reports. Please refer to Section 7.5 - “Configuration File Processing” for more information

To save the reports to a set of files sharing a common *basename*:

1. Click on the **File** pull-down menu.
2. Select **Set Report Files Basename**.
3. The **Set Report Files Basename** window (shown below) pops up.
4. Click the mouse pointer in the specification region. When a cursor appears, type in the report file prefix.
5. Click on **OK**. When files are processed by *METRICTM*, they will automatically be saved to files with the base name you specified.
6. The following Kiviat definition files will also be generated:
 - *basename.I.kvi*
 - *basename.II.kvi*
 - *basename.III.kvi*

These files allow you to use the command line invocation of **Xkiviat** (See Section 7.4 - “Xkiviat’ - Static Metrics Display System” on page 153.) to display the various metrics in a Kiviat chart.

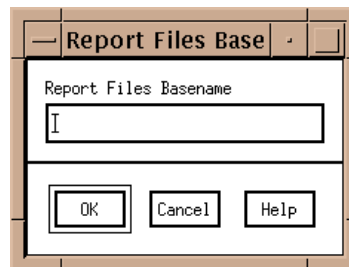


FIGURE 27 Saving Reports to a Common Basename

4.4.3 Selecting a Source Code File

*METRIC*TM will automatically process report information for a source code file. Here's how to select a source code file:

1. Click on the **File** pull-down menu.
2. Select the **Load Single File** option. The file selection dialog box below pops up. For further information on using the file selection dialog box, please refer to that section (See Section 4.1 - "Using this Chapter" on page 71.).
3. Select a source code file.
4. *METRIC*TM is fast. It can process over 4,000 lines of code well under a minute on a 386.
5. When it has processed the source code file, it will automatically create the reports. The **Complexity** report is the default report, and should be displayed in the window display area.

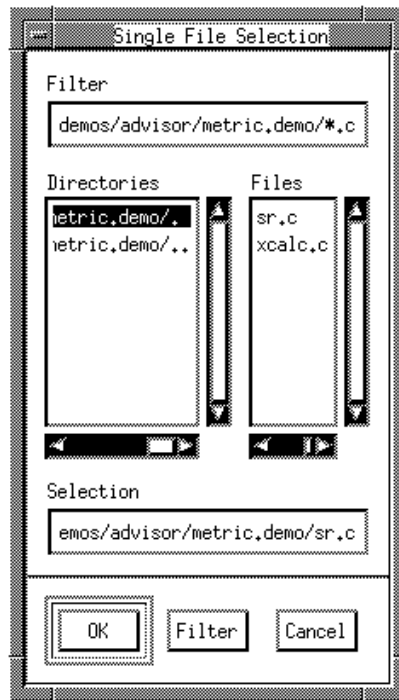


FIGURE 28 Selecting a Source Code File

4.4.4 Selecting Multiple Source Code File

*METRIC*TM also allows you to select more than one file for analysis. Here's how to select multiple source code files:

1. Click on the **File** pull-down menu.
2. Select the **Load Multiple Files** option. The file selection dialog box below pops up. For further information on using the file selection dialog box, please refer to that section (See Section 4.1 - "Using this Chapter" on page 71.).
3. To select more than one file, do one of two things:
 - Highlight the files in the File selection window by clicking on the actual file names.
 - You can select all of the files by clicking on the **Select All** button.
4. Click on **OK**.
5. When *METRIC*TM has processed the source code files, it will automatically create the reports. The **Complexity** report is the default report, and should be displayed in the window display area.

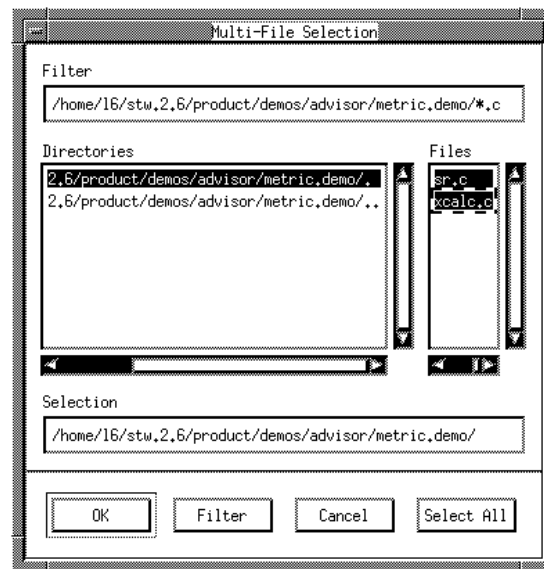


FIGURE 29 Selecting Multiple Source Code Files

4.5 Looking at the Reports

After a source code file or multiple files have been loaded into *MET-RICTM*, a **Complexity** report, a **Summary** report, an **Exception** report, and an **Error** report are generated for the language you specified.

Note for “C++” Users: Four additional reports are created for you: **C++ Class** report, **Class Summary** report, **Class Hierarchy** report, and the **Class Exception** report. Information on these reports is located in Appendix B.

Note for “Ada” Users: Three additional reports are created for you: **Generic** report, **Package Exceptions** report, and **Package Intermediates** report. Information on these reports is located in Appendix C.

4.5.1 Looking at a Complexity Report

When files are processed, the **Complexity** report should automatically be loaded into the display area. If you are looking at another report and want to go back to the **Complexity** report:

1. Click on the **Report** pull-down menu.
2. Select **Complexity**. The **Complexity** report is automatically loaded into the display area.
3. Use the scroll bars to move up/down and side/side to look at the report.

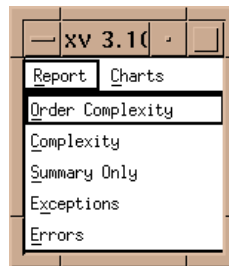


FIGURE 30 Selecting the Complexity Report

This report computes size, Software Science, control flow complexity, and data object metrics for each function/procedure encountered.

These are the measures computed:

Size	Total Lines of Code, Number of Blank Lines, Number of Comment Lines, and Number of Executable Statements.
Software Science	$n1$, $n2$, $N1$, $N2$, N , N^* , Purity Ratio (N^*/N), Volume, and Effort.
Control Flow	Cyclomatic Complexity (number of decision statements) and Extended Cyclomatic Complexity (number of decision statements, plus number of compound conditionals).
Data Objects	Mean Maximum Span of Reference (average number of maximum lines between subsequent variable references) and Average Variable Name Length.

In-depth discussion of these fields can be found in a previous section (See Section 3.3 - "The Complexity Report" on page 41.). You may also want to refer to **Description of the Reports** section in the **APPENDIXES** that applies to your language.

Procedure	n1	n2	N1	N2	N	N*	P/R	V	E	VG1	VG2	LOC
do_sr	59	103	628	336	1024	1036	1.01	7516	892446	40	43	211
XSRError	6	4	9	4	13	24	1.81	43	130	1	1	6
rescale	24	26	118	69	187	232	1.24	1055	33610	4	4	31
drawmark	8	8	19	16	35	53	1.50	143	1288	3	3	11
dolabel	10	12	20	15	35	76	2.18	156	976	2	2	10
drawframe	29	38	159	146	345	340	0.99	2093	118591	7	7	31
doscale	25	29	106	79	185	203	1.09	1016	59155	5	5	28
dotenths	28	28	204	141	345	269	0.78	2004	141249	15	15	50
drawslide	27	32	134	98	232	288	1.24	1365	56425	4	4	28
redrawslide	11	10	22	14	36	71	1.96	159	1218	2	2	11
redrawframe	11	10	22	14	36	71	1.38	159	1218	2	2	11
drawhair1	6	4	11	6	17	24	1.38	56	254	1	1	6
drawnuus	12	20	148	91	239	129	0.54	1195	36624	1	1	21

Procedure	n1	n2	N1	N2	N	N*	P/R	V	E	VG1	VG2	LOC
parse_double	8	6	11	6	17	40	2.32	65	259	1	1	11
open_the_display	21	15	57	26	83	151	1.82	429	7810	6	7	28
main	108	211	1404	840	2244	2359	1.05	18864	4012377	83	113	411

FIGURE 31 Complexity Report

4.5.2 Re-Ordering Procedures/Functions

By carefully analyzing this report, you can detect modules that are overly complex. Because the procedures/functions are ordered according to how they are encountered, not by complexity, this can be very time consuming for mid- to large- scale program.

METRICTM alleviates this problem by allowing you to order the procedures/functions in ascending or descending order according to one of the 17 complexity measures. For example, you can detect which one of your modules has the highest predicted length by ordering the function/procedure in ascending or descending order with the N^{\wedge} measures.

Here's how to order the **Complexity** report's procedures/functions according to a complexity measure:

1. Click on the **Report** pull-down menu.
2. Select **Order Complexity**.
3. The **Sort Report By** window pops up (See Figure 32 "Sort Report By Window" on page 90.). It lists all of the complexity measure fields. The default is set to **Procedure**.
4. Click on the complexity field by which you want your **Complexity** report ordered.
5. Now, you must decide if you want the order to be ascending or descending. For ascending order, click on the **Ascend** button. For descending order, click on the **Descend** button.
6. *METRICTM* automatically reorders the **Complexity** report. Your report should be ordered according to the complexity measure you selected.
7. We took the **Complexity** report from Section 4.5.1 and ordered it according to the predicted length (N^{\wedge}) measure in descending order (See Figure 33 "Re-Ordered Complexity Report" on page 90.)

Note: You can only use the **Sort Report By** window for 132 column reports.

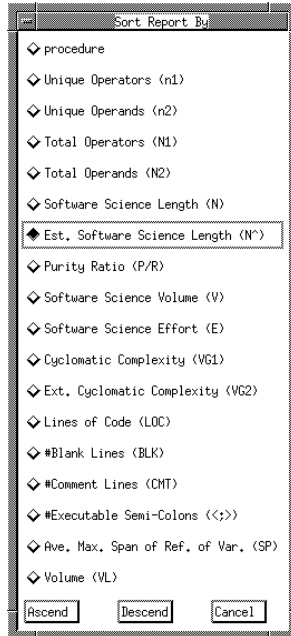


FIGURE 32 Sort Report By Window

Procedure	n1	n2	N1	N2	N	N*	P/R	V	E	VG1	VG2	LOC
XSRError	6	4	9	4	13	24	1.81	43	130	1	1	6
drawhair	6	4	11	6	17	24	1.39	56	254	1	1	6
drawmark	9	8	19	16	35	52	1.50	143	1288	3	3	7
redrawslide	11	10	22	14	36	71	1.98	158	1218	2	2	11
redrawframe	11	10	22	14	36	71	1.98	158	1218	2	2	6
dolabel	10	12	20	15	35	76	2.19	156	976	2	2	11
drawans	12	20	148	91	239	129	0.54	1195	32624	1	1	22
doscale	25	20	106	79	185	203	1.09	1016	50165	5	5	26
rescale	24	26	118	69	187	232	1.24	1055	33510	4	4	32
dotests	28	28	204	141	345	393	0.79	2004	141245	15	15	52
drawslide	27	32	134	98	232	288	1.24	1365	56425	4	4	24
drawframe	29	38	199	146	345	340	0.93	2093	116591	7	7	34
do_sr	59	103	628	396	1024	1036	1.01	7516	652446	40	45	211

Procedure	n1	n2	N1	N2	N	N*	P/R	V	E	VG1	VG2	LOC
onalarm	3	2	3	2	5	7	1.35	12	17	1	1	6
nps	5	2	5	2	7	14	1.94	20	49	1	1	4
fperr	5	2	5	2	7	14	1.94	20	49	1	1	7

FIGURE 33 Re-Ordered Complexity Report

4.5.3 Looking at a Summary Report

The **Summary** report provides a set of complexity metrics for the entire source code file. Please refer to the appropriate section for information on the **Summary** report (See Section 3.4 - "The Summary Report" on page 55.) and to section entitled **Description of the Reports** in the **APPENDICES** that applies to your language. To look at a **Summary** report:

1. Click on the **Report** pull-down menu.
2. Select **Summary Only**. The **Summary** report is automatically loaded into the display area.
3. Use the scroll bars to move up/down and side/side to view the report.

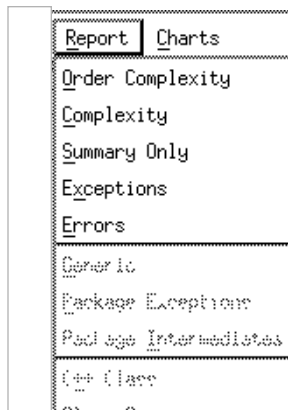


FIGURE 34 Selecting the Summary Report

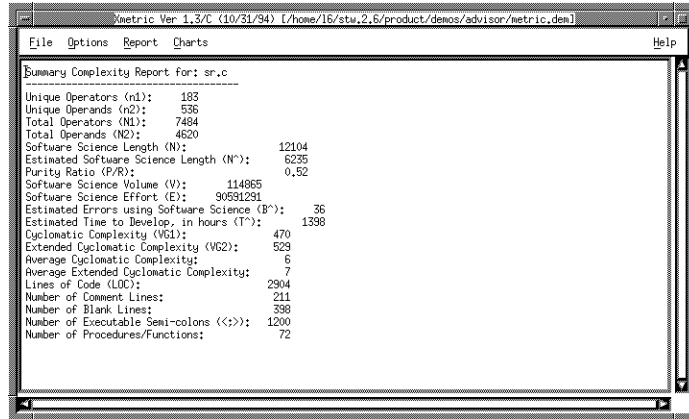


FIGURE 35 Summary Report

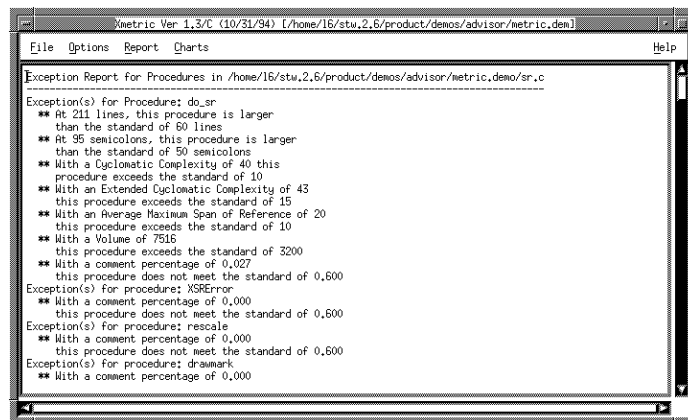
The metrics computed for the **Summary** report:

Size	Total Lines of Code, Number of Blank Lines, Number of Comment Lines, Number of Executable Statements, and number of procedures/functions.
Software Science	n1, n2, N1, N2, N, N^, Purity Ratio, Volume, Effort, estimated programming errors (B^), and estimated programming time (T^).
Control Flow	Cyclomatic Complexity, Extended Cyclomatic Complexity, Average Cyclomatic Complexity, and Average Extended Cyclomatic Complexity.

4.5.4 Looking at an Exception Report

The **Exception** report lists all functions/procedures that exceed user defined threshold values, such as Lines of Code, Number of Executable Statements, Cyclomatic Complexity, Extended Cyclomatic Complexity, Mean Maximum Span of Reference, Number of `goto` statements, and Comment Density. These threshold values are defined in the configuration file, `.uxmetriccfg`, and some can be set with the GUI (See Section 4.4.4 - "Selecting Multiple Source Code File" on page 85.). To look at an **Exception** report:

1. Click on the **Report** pull-down menu.
2. Select **Exceptions**. The **Exception** report is automatically loaded into the display area.
3. Use the scroll bars to move up/down and side/side to view the report.



```
Exception Report for Procedures in /home/16/stw,2,6/product/demos/advisor/metric.demo/sr.c
-----
Exception(s) for Procedure: do_ar
** At 211 lines, this procedure is larger
   than the standard of 60 lines
** At 95 semicolons, this procedure is larger
   than the standard of 50 semicolons
** With a Cyclomatic Complexity of 40 this
   procedure exceeds the standard of 10
** With an Extended Cyclomatic Complexity of 43
   this procedure exceeds the standard of 15
** With an Average Maximum Span of Reference of 20
   this procedure exceeds the standard of 10
** With a Volume of 7516
   this procedure exceeds the standard of 3200
** With a comment percentage of 0,027
   this procedure does not meet the standard of 0,600
Exception(s) for procedure: XSRError
** With a comment percentage of 0,000
   this procedure does not meet the standard of 0,600
Exception(s) for procedure: rescale
** With a comment percentage of 0,000
   this procedure does not meet the standard of 0,600
Exception(s) for procedure: drawmark
** With a comment percentage of 0,000
```

FIGURE 36 Exception Report

4.5.5 Looking at an Error Report

The **Error** report lists any errors encountered during processing and analysis. Errors differ with each language. See the section entitled **Description of Report** in the **APPENDIXES** for the language with which you are working.

To look at an **Error** report:

1. Click on the **Report** pull-down menu.
2. Select **Errors**. The **Error** report is automatically loaded into the display area.
3. Use the scroll bars to move up/down and side/side.

4.5.6 Setting Report Threshold Values

*METRIC*TM offers a **Configuration Options** window, which allows you to set the report parameters. To invoke it:

1. Click on the **Options** pull-down menu.
2. Select **Report**.
3. The **Configuration Options** (shown below) window pops up.

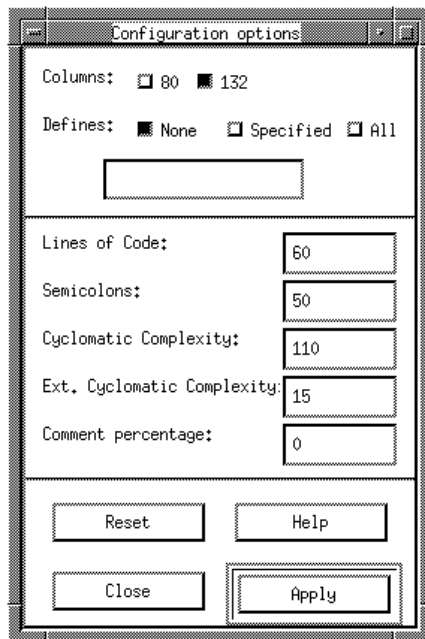


FIGURE 37 Configuration Options Window

4. You can set the following parameters:

- **Columns:** Simply click on the corresponding radio button to change the column width of the **Complexity** report.

We recommend that use the 132-column report when you have very large procedures in the files you are analyzing and wider printer is available.

An 80-column report does *not* have the following complexity metrics: P/R (Purity Ratio), BLK (Number of Blank Lines), CMT (Number of Comment Lines), and VL (Average Variable Name Length). BLK and CMT are replaced with B/C, which is a combination calculation for BLK and CMT.

- **Defines:** Allows you to turn on **None** (the default) of the source code file(s) conditional compilation directives, **Specific** directives, or **All** of the directives during `Xmetric` processing. Click on the corresponding radio button.

For **Specify** you must specify the directives in the specification region. Click in the specification region and type when the cursor appears.

This option only applies to “C” and “C++”.

- **Semicolons:** Allows you set the maximum threshold for the number of semi-colons a procedure can have. Those procedures that exceed this threshold are listed in the **Exception** report. The threshold is set at 50.

Click in the specification region. When the cursor appears, edit accordingly.

- **Lines of Code:** Allows you to set the maximum threshold for the number of lines of code a procedure/function can have. Those procedures/functions that exceed this threshold are listed in the **Exception** report. The threshold is set at 62.

Click in the specification region. When the cursor appears, edit accordingly.

- **Cyclomatic Complexity:** Allows you set the maximum threshold for the cyclomatic complexity number a procedure/function can have. Those procedures/functions that exceed this threshold are listed in the **Exception** report. The threshold is set at 10.

Click in the specification region. When the cursor appears, edit accordingly.

- **Extended Cyclomatic Complexity:** Allows you set the maximum threshold for the extended cyclomatic complexity number a procedure/function can have. Those procedures/functions that exceed this threshold are listed in the **Exception** report. The threshold is set at 15.
Click in the specification region. When the cursor appears, edit accordingly.
 - **Comment Percent:** Allows you set the maximum threshold for the comment percent (comment/lines of code - blank lines) a procedure/function can have. Those procedures/functions that exceed this threshold are listed in the **Exception** report. The threshold is set at 0.
Click in the specification region. When the cursor appears, edit accordingly.
5. After you have made your changes, click on the **Apply** button. The changes should be reflected in the reports.
 6. Exit the window by clicking on the **Close** button.

These threshold guidelines can also be set in the configuration file (See Section 7.5 - "Configuration File Processing" on page 156.).

4.6 Graphically Viewing Complexity

After looking at reports, you may want to look at *METRIC*TM's Kiviat diagrams. Kiviat diagrams provide a graphical means to view the impact of multiple metrics on a source code file or multiple files.

These diagrams represent information from the **Summary** report. *METRIC*TM provides three types of Kiviat diagrams, each with different metric than the previous one.

4.6.1 Looking at a Type I Kiviat Chart

The first Kiviat chart, **Type I**, displays the following software measures for the processed source code:

- Unique Operators (n1).
- Unique Operators (n2).
- Total Operators (N1).
- Total Operands (N2).
- Lines of Code (LOC).
- Number of Comment Lines (CMT).
- Number of Blank Lines (BLK).
- Number of Executable Semi-Colons (< ; >).
- Number of Functions.

To look at the Kiviat chart:

1. Click on the **Charts** pull-down menu.
2. Select **Type I**.
3. A window displaying a Kiviat chart pops up. You may need to resize it. See the example diagram. (See Figure 38 "Type I Kiviat Chart" on page 100.)
4. The inner circle represents minimum values, the outer circle represents maximum values and radii through the circles represent the metrics of interest.

These minimum/maximum thresholds are defined in a file named `.Xmetric.I.def` and also can be set in the **Type I Kiviat Chart** window with the `Xmetric` GUI (both are discussed in this section).

Observed values are plotted on the radii and connected. From this, metrics that are not within the acceptable range of values can be easily identified.

5. To close the Kiviat chart, click on the **File** pull-down menu and select **Exit**. The Kiviat chart closes.

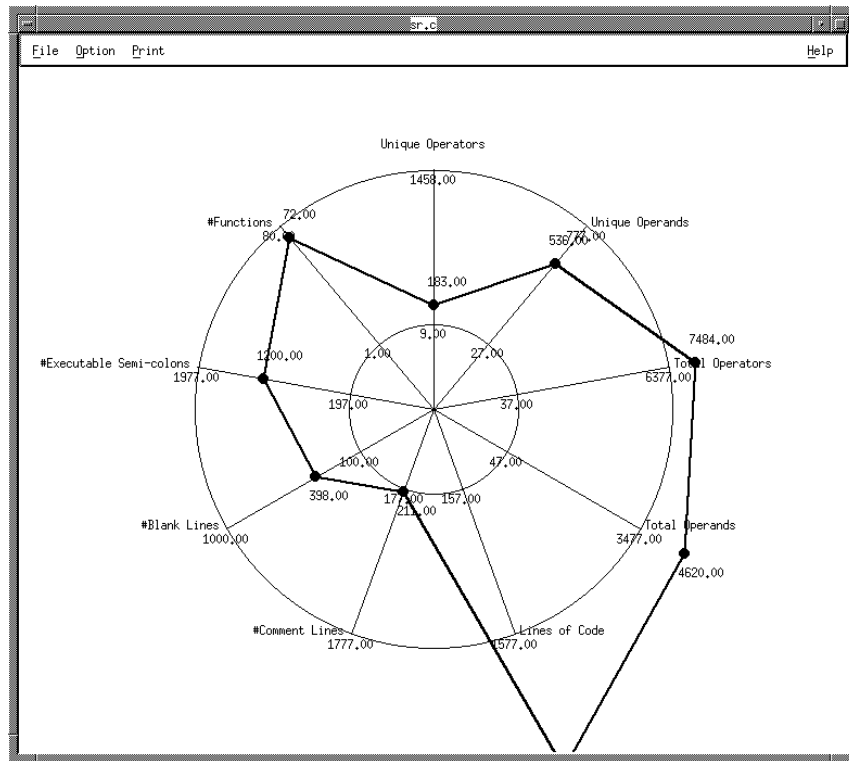


FIGURE 38 Type I Kiviat Chart

Setting Type I Chart Parameters

You can set your own **Type I** chart metric values. One way is to edit the `Xmetric.I.def` configuration file to fit your own needs, or you can edit the threshold parameters with from the GUI.

Below is the **Type I** configuration file, `Xmetric.I.def`:

```
#
# A Sample of Type-I Kiviat Chart Definition
#
# Min      Max      Value      Text
#   9      1458    100      Unique Operators
#   27      777     100      Unique Operands
#   37      6377    100      Total Operators
#   47      3477    100      Total Operands
#  157     1577    100      Lines of Code
#  177     1777    100      #Comment Lines
#  100     1000    100      #Blank Lines
#  197     1977    100      #Executable Semi-
colons
#   1       80     100      #Functions
```

FIGURE 39 Xmetric.I.def Configuration File

MIN	The minimum threshold parameter.
MAX	The maximum threshold parameter.
Value	The threshold values for the metrics. These values are overwritten by the actual values of the <code>\f6Summary\f1</code> report, so you do not need to edit this column.
Text	The complexity measure. Do not remove any of the metrics. If you want to customize your own report, please see the correct section (See Section 4.6.4 - "Customizing Your Own Kiviat Chart" on page 113.).

Simply use any ASCII file editor to edit the file.

To edit minimum/maximum thresholds from the GUI:

1. Click on the **Options** pull-down menu.
2. Drag the mouse to the **Charts** cascading menu.
3. Select **Type I**.
4. The **Type I Configuration** window pops up.
5. You can change the minimum and maximum parameters by clicking on the corresponding specification region. When a cursor appears, edit accordingly.
6. After you have made your changes, click on the **Apply** button. The Kiviart chart will be redrawn, reflecting the changes.
7. Exit the window by clicking on the **Close** button.

The screenshot shows a window titled "Type I Configuration". It contains a table with three columns: "Metric", "Min", and "Max". The table lists various software metrics and their corresponding minimum and maximum values. Below the table are four buttons: "Reset", "Help", "Close", and "Apply".

Metric	Min	Max
Software Science Length	6700	9677
Est. Software Science Length	67	9677
Purity Ratio	0,070000	0,700000
Software Science Volume	77000	97777
Software Science Effort	8700000	11178877
Estimated Errors	2	37
Estimated Time To Develop	107	1077
Cyclomatic Complexity	117	1177
Ext. Cyclomatic Complexity	1	377
Avg. Cyclomatic Complexity	1	37
Avg. Ext. Cyclomatic Complexity	147	1477

Buttons: Reset, Help, Close, Apply

FIGURE 40 Type I Configuration Window

4.6.2 Looking at a Type II Kiviat Chart

The second Kiviat chart, **Type II**, displays the following software measures for the processed source code:

- Length (N).
- Predicted Length (N^{\wedge}).
- Purity Ratio (P/R).
- Estimated Effort (E).
- Estimated Errors (E^{\wedge}).
- Estimated Time to Develop (T^{\wedge}).
- Cyclomatic Complexity ($VG1$).
- Extended Cyclomatic Complexity ($VG2$).
- Average Cyclomatic Complexity.
- Average Extended Cyclomatic Complexity.

To look at the Kiviat chart:

1. Click on the **Charts** pull-down menu.
2. Select **Type II**.
3. A window displaying a Kiviat chart pops up. You may need to resize the window. See the example diagram (See Figure 41 "Type II Kiviat Chart" on page 104.).
4. The inner circle represents minimum values, the outer circle represents maximum values and radii through the circles represent the metrics of interest.

These values are defined in a file named `.Xmetric.II.def` and can be set in the **Type II** window with the **Xmetric** GUI (both are described in this section).

Observed values are plotted on the radii and connected. From this, metrics that are not within the acceptable range of values can be easily identified.

5. To close the Kiviat chart, click on the **File** pull-down menu and select **Exit**. The Kiviat chart closes.

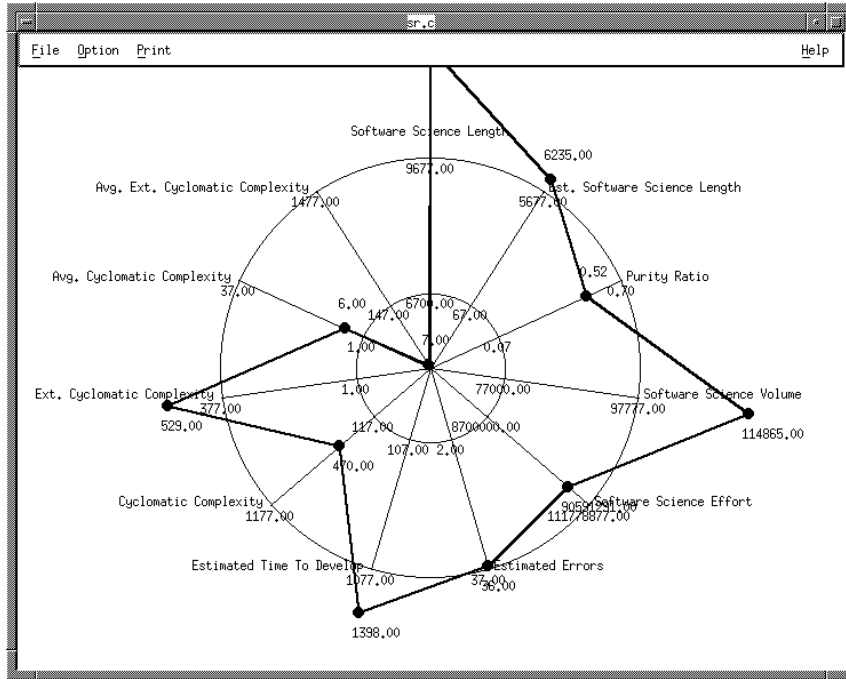


FIGURE 41 Type II Kiviat Chart

Setting Type II Chart Parameters

You can set your own **Type II** chart metric values. One way is to edit the `Xmetric.II.def` configuration file to fit your own needs or you can edit the threshold parameters with the GUI.

Below is the **Type II** configuration file, Xmetric.II.def:

```
#
# A Sample of Type-II Kiviat Chart Definition
#
6700      9677      100  Software Science Length
67        5677      100  Est. Software Science
Length
0.070     0.700     100  Purity Ratio
77000     97777      100  Software Science Volume
8700000   111778877      100  Software Science Effort
2         37         100  Estimated Errors
107       1077      100  Estimated Time To Develop
117       1177      100  Cyclomatic Complexity
1         377       100  Ext. Cyclomatic Complexity
1         37        100  Avg. Cyclomatic Complexity
147       1477      100  Avg. Ext. Cyclomatic Com-
plexity
```

FIGURE 42 Xmetric.II.def Configuration File

MIN	The minimum threshold parameter.
MAX	The maximum threshold parameter.
Value	The threshold values for the metrics. These values are overwritten by the actual values of the Summary report, so you do not need to edit this column.
Text	The complexity measure. Do not remove any of the metrics. If you want to customize your own report, please see the correct section (See Section 4.6.4 - "Customizing Your Own Kiviat Chart" on page 113.).

Simply use any ASCII file editor to edit the file.

To edit the minimum/maximum values from the GUI:

1. Click on the **Options** pull-down menu.
2. Drag the mouse to the **Charts** cascading menu.
3. Select **Type II**.
4. The **Type II Configuration** window pops up.
5. You can change the minimum and maximum parameters by clicking on the corresponding specification region. When a cursor appears, edit accordingly.
6. After you have made your changes, click on the **Apply** button. The Kiviatic chart will be redrawn, reflecting the changes.
7. Exit the window by clicking on the **Close** button.

The screenshot shows a window titled "Type II Configuration". It contains a table with three columns: "Metric", "Min", and "Max". The table lists various metrics and their corresponding minimum and maximum values. Below the table are four buttons: "Reset", "Help", "Close", and "Apply".

Metric	Min	Max
Unique Operators	3	1458
Unique Operands	27	777
Total Operators	37	6377
Total Operands	47	3477
Lines of Code	157	1577
#Comment Lines	177	1777
#Blank Lines	100	1000
#Executable Semi-colons	197	1977
#Functions	1	80

Buttons: Reset, Help, Close, Apply

FIGURE 43 Type II Configuration Window

4.6.3 Looking at a Type III Kiviat Chart

The third Kiviat chart, **Type III**, displays the following software measures for the processed source code:

- Unique Operators ($n1$).
- Unique Operands ($n2$).
- Total Operators ($N1$).
- Total Operands ($N2$).
- Software Science Length (N).
- Estimated Software Science Length (N^{\wedge}).
- Purity Ratio (P/R).
- Software Science Volume (V).
- Software Science Effort (E).
- Estimated Time to Develop (T^{\wedge}).
- Cyclomatic Complexity ($VG1$).
- Extended Cyclomatic Complexity ($VG2$).
- Average Cyclomatic Complexity.
- Average Extended Cyclomatic Complexity.
- Lines of Code (LOC).
- Number of Comment Lines (CMT).
- Number of Blank Lines (BLK).
- Number of Executable Semi-colons ($< ; >$).
- Number of Functions.

To obtain a Kiviat chart:

1. Click on the **Charts** option.
2. Select **Type III**.
3. A window displaying a Kiviat chart pops up. You may need to resize it. See the example diagram (See Figure 44 "Type III Kiviat Chart" on page 109.).
4. The inner circle represents minimum values, the outer circle represents maximum values and radii through the circles represent the metrics of interest.

Observed values are plotted on the radii and connected. From this, metrics that are not within the acceptable range of values can be easily identified.

5. To close the Kiviat chart, click on the **File** pull-down menu and select **Exit**. The Kiviat chart closes.

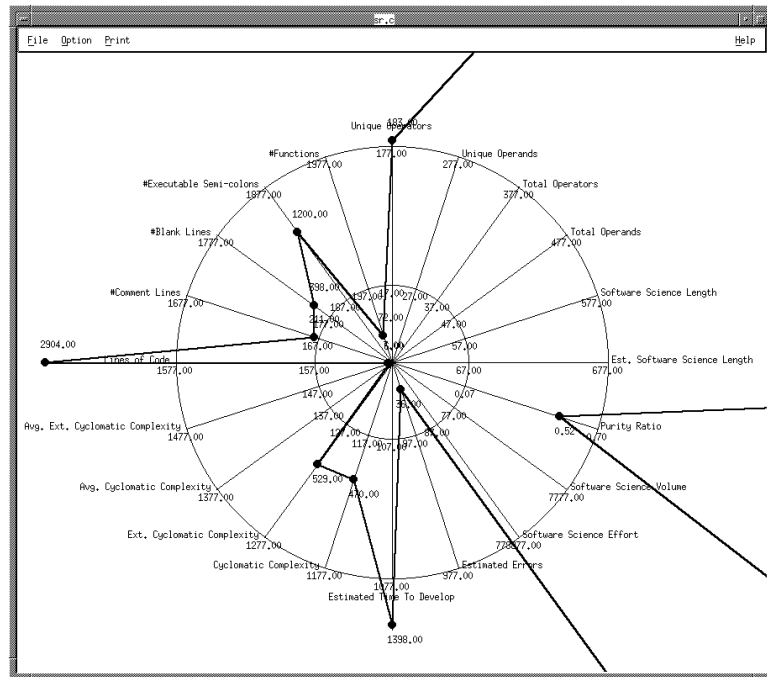


FIGURE 44 Type III Kiviati Chart

Setting Type III Chart Parameters

You can set your own **Type III** chart metric values. One way is to edit the `xmetric.III.def` configuration file to fit your own needs or you can edit the threshold parameters from the GUI.

Following is the Type III configuration file, Xmetric.III.def:

```
#  
# A Sample of Type-III Kiviat Chart Definition  
# Min      Max      Value  Text  
9          1458    100    Unique Operators  
27         777     100    Unique Operands  
37         6377    100    Total Operators  
47         3477    100    Total Operands  
6700      9677    100    Software Science Length  
67        5677    100    Est. Software Science  
          Length  
0.070     0.700   100    Purity Ratio  
77000     97777   100    Software Science Volume  
8700000   111778877 100    Software Science Effort  
2         37      100    Estimated Errors  
107       1077    100    Estimated Time To  
          Develop  
117       1177    100    Cyclomatic Complexity  
1         377     100    Ext. Cyclomatic Com-  
          plexity  
1         37      100    Avg. Cyclomatic Com-  
          plexity  
147       1477    100    Avg. Ext. Cyclomatic  
          Complexity  
157       1577    100    Lines of Code  
177       1777    100    #Comment Lines  
100       1000    100    #Blank Lines  
197       1977    100    #Executable Semi-colons  
1         80      100    #Functions
```

FIGURE 45 Xmetric.III.def Configuration File

MIN	The minimum threshold parameter.
MAX	The maximum threshold parameter.
Value	The threshold values for the metrics. These values are overwritten by the actual values of the Summary report, so you do not need to edit this column.
Text	The complexity measure. Do not remove any of the metrics. If you want to customize your own report, please see Section 4.6.4

Simply use any ASCII file editor to edit the file.

To edit the minimum/maximum values from the GUI:

1. Click on the **Options** pull-down menu.
2. Drag the mouse to the **Charts** cascading menu.
3. Select **Type III**.
4. The **Type III Configuration** window pops up.
5. You can change the minimum and maximum parameters by clicking on the corresponding specification region. When a cursor appears, edit accordingly.
6. After you have made your changes, click on the **Apply** button. The Kiviat chart will be redrawn, reflecting the changes.
7. Exit the window by clicking on the **Close** button.

The screenshot shows a window titled "Type III Configuration". It contains a table with three columns: "Metric", "Min", and "Max". The table lists various software metrics and their corresponding minimum and maximum values. At the bottom of the window, there are four buttons: "Reset", "Help", "Close", and "Apply".

Metric	Min	Max
Unique Operators	17	177
Unique Operands	27	277
Total Operators	37	377
Total Operands	47	477
Software Science Length	57	577
Est. Software Science Length	67	677
Purity Ratio	0.070000	0.700000
Software Science Volume	77	7777
Software Science Effort	87	778877
Estimated Errors	97	977
Estimated Time To Develop	107	1077
Cyclomatic Complexity	117	1177
Ext. Cyclomatic Complexity	127	1277
Avg. Cyclomatic Complexity	137	1377
Avg. Ext. Cyclomatic Complexity	147	1477
Lines of Code	157	1577
#Comment Lines	167	1677
#Blank Lines	177	1777
#Executable Semi-Colons	187	1877
#Functions	197	1977

FIGURE 46 Type III Configuration Window

4.6.4 Customizing Your Own Kiviat Chart

You do not have to use *METRIC*TM's available Kiviat charts. You can create your own. For example, you may want a Kiviat chart, using only Software Science measures or Control Flow measures.

To do this, simply create a file that follows the pattern of

`Xmetric.I.def`, `.Xmetric.II.def`, or `.Xmetric.III.def`.

Then you must load this file into *METRIC*TM. To do this:

1. Click on the **Charts** pull-down menu.
2. Select **Type User**.
3. A file selection dialog box pops up.
4. Select the file in which you have your own metrics and values listed.
5. Your own Kiviat diagram will be displayed.

4.7 Exiting METRIC

The **Exit** option allows you to close the **Main** window. Here's how:

1. Click on the **File** pull-down menu.
2. Select **Exit**.
3. You have exited *METRIC*TM.

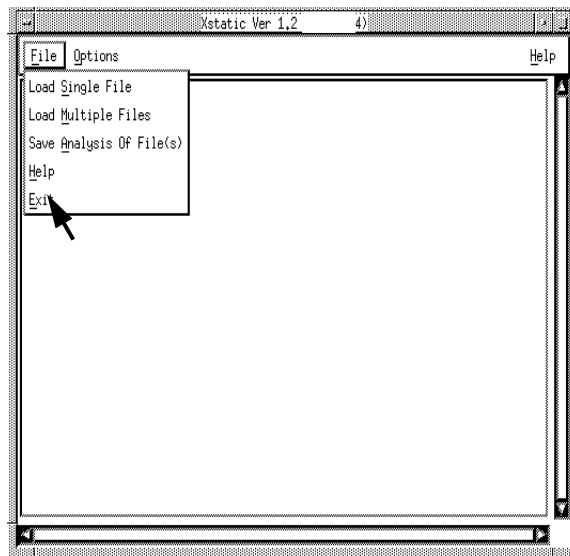


FIGURE 47 Exiting METRIC

Helpful Hints

READERS' GUIDE: This chapter offers recommendations and principles of operation for *METRICTM* in an automated testing environment.

LEVEL: This chapter is intended for all users.

5.1 Measuring Program Complexity

The *METRICTM* system for expressing the characteristics of a piece of software in quantitative software complexity measures is a powerful and effective tool for computing complexity measures. This chapter describes important set-up information and how you can use software metrics in general, and *METRICTM* to help you develop, test and maintain software.

5.1.1 Set-Up Suggestions

These are some common guidelines you should follow:

- Make sure the environment variable, *SR*, is set up correctly. **Xmetric** relies on this variable to locate various control files for different reports. Please see your *Installation Instructions* for further information.
- Before running **Xmetric**, you should put all the source files that need to be analyzed into one directory, and then invoke **Xmetric**.
- If you have different groups using **Xmetric**, then each of them should have their own configuration files in their home directory. This way they can customize their own environment without interfering with others' work.
- **Xmetric** will look first in the home directory for the various configuration files. If **Xmetric** cannot find the configuration files, it will look at the directory defined in the *SR* environment variable.
- If you run **Xmetric** on a source file and no reports are generated, look at the **Error** report. Most often this occurs when the *SR* file is not defined correctly.

5.1.2 Software Development

Software metrics can be used quite effectively during the development stage of the software life cycle. *METRICTM* can be introduced quite early in this phase as a feedback tool for programmers. Likewise, software metrics can be used during code reviews to help reviewers identify those portions of the code which may be the most difficult to work with, and hence most appropriate to cover in a review.

Using Metrics as a Feedback Tool

One of the most effective uses of metrics is as a feedback tool for programmers. In general, this is best accomplished by identifying a set of *complexity thresholds* beyond which code complexity must be addressed. The complexity may be addressed by a variety of actions, ranging from simply adding additional comments to procedures which violate the thresholds to an entire rewrite of the offending procedures.

One of the most difficult tasks is arriving at suitable threshold levels for a particular environment. Due to differences among personnel, applications, etc., installation standards must be developed on a case by case basis. Thus, the numbers used by the shop down the street may have little relevance to the numbers used in your environment. While we can't suggest specific numbers, we can outline some steps to follow to arrive at some standards for your installation.

Once installation thresholds are obtained, they can be enforced by specifying them in the configuration file, `.uxmetriccfg`. After setting up the configuration file, procedures which violate the standards will be flagged in the *METRICTM* exception report. This allows "remedy by exception", so the programmer can avoid addressing the complexity of those procedures which are within the predefined thresholds.

The installation standards can best be established by either (1) identifying your top programmer and analyzing a sample of his/her code using *METRICTM* or (2) simply analyzing a sample of all your programmers' code using *METRICTM*. In either case, the sample should consist of well over 100 different procedures.

After computing these threshold values, you would then use your favorite text editor to change the entries in the configuration file to flag procedures which exceed the threshold values.

After establishing these standards, every procedure would be analyzed by *METRICTM* before the programmer could consider it complete. A cursory examination of the exception report would identify those procedures which violate the standards.

A procedure which violates the threshold complexity is not necessarily poorly written. It may be the case that it cannot be improved upon. However, by carefully examining procedures which *do* violate the threshold complexity, and taking remedial action with those procedures which can be improved it is likely the programmer will deliver code with fewer bugs that is easier to test and maintain. *Remedial action* may consist of one or more of these activities:

1. The module may be rewritten (and perhaps decomposed into several less complex modules) to reduce complexity. Depending on the metric which exceeded the complexity threshold, either the control flow (VG), data structuring (SP) or size (LOC or < i >) of the original module should be changed.
2. The module may not need to be rewritten, but extra documentation may be added to the module to compensate for its additional complexity. As in the previous activity, the particular measure which exceeded the threshold may give some ideas on which items to stress in the documentation.

Other approaches to arriving at a complexity threshold include examination of selected programs and arriving at a consensus among the programming team. It is important that the software developers “buy into” the thresholds since complexity metrics can easily be circumvented by using pathological coding practices. For example, the lines of code measure can be minimized by writing several statements to a line, and the extended cyclomatic complexity can be bypassed by separating compound conditionals into individual groups of *if* statements.

Using Metrics in the Review Process

Many organizations make use of a formal software review process to identify potential problems and inconsistencies in the code before it can be integrated with the rest of the system. The greatest drawback with this activity is that preparation required by the reviewers can be quite time consuming since they have to carefully read every line of code to be reviewed. By using software metrics to identify overly complex parts of the code, the reviewers can easily locate the most risky parts of the code.

Two approaches can be taken by the reviewers. In the first approach, threshold limits can be identified, and procedures which exceed these thresholds can be flagged using the **Exception** report generated by *METRICTM*. The second approach recognizes the fact that the reviewer has only a fixed amount of time to devote to working with the reviewed code. Using the Kiviat diagrams, reviewers can graphically identify those procedures which contribute the bulk of the complexity to the overall file under review.

In either case, the reviewer can then use this knowledge to help allocate their attention to specific parts of the code. Additionally, the programmer can be asked to justify the complexity of the procedures during the review process. This would ensure the programmer has examined the overly complex procedures and satisfied him/herself that they could not be written in a less complex manner.

Using Metrics in Estimation

One of the most interesting measures produced by *METRIC*TM is the Software Science based T^{\wedge} measure, estimated development time. The utility of this measure is often not appreciated since it is an estimate of how much time is required to develop the software. Unfortunately, this appears to be of little use to many developers once the code has been developed.

In fact, there are times when a programmer needs to know how long it took to develop a piece of code. For example, custom programming houses usually charge clients based on the amount of time a programmer spends writing the program, but a single programmer may be spread among several projects at once. At the end of the projects, how much time was spent on each? Likewise, if a program to be developed is similar to an existing program, it would be helpful to know how much effort went into developing the original version when presenting a bid or asking for a budget to develop the new version.

Unfortunately, if the developers were not careful in recording the time spent on the program it may require spending a great deal of effort attempting to “guesstimate” how much time was actually spent on the program. This is even more difficult if the original developer is not available. Even if a good estimate *is* formulated, it can be difficult to substantiate the estimate.

The T^{\wedge} measure can be used in these situations. It possesses many useful qualities. It is totally automated so little programmer effort need be expended manually “guesstimating” development time after the fact. Additionally, Software Science is well accepted by many researchers and practitioners, so there should be relatively little effort required to justify the estimate.

The T^{\wedge} measure is computed by dividing Software Science Effort (E) by programmer speed. Naturally, even though Halstead suggested a preliminary value of 18, programmer speed will vary from programmer to programmer. Therefore, this parameter should be adjusted for individual programmers.

Individual adjustments can be made by analyzing a large sample of a programmer's code with *METRIC*TM and summing the reported development time estimates (T^{\wedge}). This will yield the estimated development time in hours for the set of programs analyzed. Divide the sum of the T^{\wedge} measures by the sum of the actual observed development time, and multiply the current programmer speed parameter in the configuration file (*SPEED*) by the resulting ratio. If the *SPEED* parameter is not included in the current configuration file use 18, and insert the resulting value in the configuration file.

Chances are good that the resulting estimate will not be exactly correct for most programs, but it should yield close estimates for sets of programs. For example, we may be disappointed with the estimate obtained for a single program written for a client. However, the sum of the estimates for a set of programs written for a client will usually be fairly close to the time actually spent.

In effect, the T^{\wedge} measure can be viewed as an automated analogue to the mechanics “flat rate manual”, which indicates how much time should be billed for a variety of repairs. While seldom exactly correct for any particular repair performed by a garage, chances are good that the flat rate estimates will be accurate over time.

Metrics in Software Testing

Software metrics can be used even more effectively during the software testing activity than they can during the development phase. One of the most common dilemmas faced by a software project manager is the lack of resources available for testing. Usually by the time the testing phase is reached, any slack that was in the schedule and budget is used up. Thus, the manager must determine where to allocate his or her scarce testing resources.

A useful measure obtained from Software Science is B^{\wedge} , estimated number of coding errors. The B^{\wedge} measure is computed by dividing Software Science Volume (V) by *programmer error rate* (this is how it is computed by METRICTM; other definitions have been suggested which use Effort (E) in place of Volume). Naturally, even though Halstead suggested a preliminary value of 3200, error rate will vary from programmer to programmer. Therefore, this parameter should be adjusted for individual programmers.

Individual adjustments can be made by analyzing a large sample of a programmer's code with METRICTM and summing the reported bug estimates (B^{\wedge}). This will yield the estimated number of coding bugs for the set of programs analyzed. Divide the sum of the B^{\wedge} measures by the sum of the actual observed number of bugs, and multiply the current error rate parameter in the configuration file ($E0$) by the resulting ratio. If the $E0$ parameter is not included in the current configuration file use 3200, and insert the resulting value in the configuration file.

The exact number of errors predicted by the B^{\wedge} measure is unlikely to be precisely correct. However, recent studies (see for example, "Using Software Metrics to Allocate Testing Resources" by Warren Harrison in the Spring 1988 issue of *The Journal of Management Information Systems*) suggest that the proportion of the total errors encountered in each module is reflected well by Software Science.

In general, by allocating testing resources based on the proportion of the total number of predicted errors estimated for each module, a more effective testing program can be carried out.

For example, consider the following set of modules, associated B^{\wedge} values, and amount of testing resources allocated to each module assuming we have 100 units (a “unit” could be dollars, hours of tester’s time, etc.) of resources to go around:

Module	B^{\wedge}	% of B^{\wedge}	Test Resources
A	51	40%	40 units
B	13	10%	10 units
C	16	13%	13 units
D	6	5%	5 units
E	2	1%	1 units
F	20	16%	16 units
G	3	1%	1 units
H	11	9%	9 units
I	6	5%	5 units
Total	128	100%	100 units

As can be seen, the percentage of testing resources allocated to each procedure is proportional to the contribution the procedure has made to the estimated programming errors total.

As pointed out at the beginning of the chapter, other characteristics besides just those captured by software metrics impact the number of bugs a module will have. Thus, this technique may not work in every situation. As we suggested earlier, we recommend using this technique in parallel with whatever approach you use currently to allocate testing resources among modules, and compare them.

5.1.3 Software Maintenance

Software metrics can also be used during program maintenance. By using metrics as a feedback tool during development, they have an eventual impact on software maintenance. However, metrics can have an even more direct impact on the maintenance phase of the software life cycle if used properly.

Apportioning Duties

Large systems are quite often maintained by a staff of several programmers. In such situations, it is common to assign different parts of the system (groups of procedures or programs) to each individual maintainer. After assignment, the maintainer will then be responsible for all maintenance that needs to be performed on the corresponding set of procedures.

An unenviable task charged to most project managers is assigning portions of a system to each maintainer. Most managers have their own private set of heuristics for assigning portions of a system to maintainers. A common heuristic is arranging assignments so each maintainer is responsible for approximately the same number of lines of code. In some cases however, this can be inequitable since two modules with the same number of lines of code can differ greatly in complexity (and hence maintenance difficulty).

An alternative to allocating portions of a system for maintenance based on lines of code is to apportion modules so each maintainer is responsible for approximately the same amount of complexity. For example, the Software Science Effort measure would be appropriate for this use. This is illustrated using the following example:

Module	Effort (10000s)	% of Effort
A	1570	68%
B	119	5%
C	142	6%
D	38	2%
E	8	0%
F	291	13%
G	7	0%
H	76	3%
I	41	2%

Thus, if a manager has three maintenance programmers available, it would be reasonable to assign two of them to module A, and apportion the remaining modules to the third programmer (assuming each individual was of equal ability).

Controlling Entropy

When maintaining a piece of software, it is common for the changed version of the software to be a little less clean and a little more complex than the original version. For the most part this is due to changes not fitting into the overall structure of the program, and simply being “patched in”. This gradual degradation of the software is called *entropy*. If it occurs only once or twice, it presents few problems. However, over its life a frequently used program may be modified dozens of times. Software metrics can be used to help minimize the effects of entropy.

This can be done by requiring all modified software to be analyzed by *METRICTM* before it can be put back into production. If the changed modules' complexity increased by more than 10% the programmer must either re-implement the change to achieve a complexity increase of less than 10% or explain why the additional complexity increase is necessary.

Naturally this does not address issues such as maintaining up to date comments, but it can help reduce entropy by ensuring changes fit better into the structure of the code.

In those rare instances when a maintenance programmer has some time on his or her hands, it may be profitable to attempt to rewrite certain modules to help reduce the entropy which has accrued. Metrics can be used in this situation to help identify the modules which should be rewritten. If the complexity levels for modules over time are maintained, the amount of complexity growth since development of each module can be tracked. Those modules whose complexity has grown 100% or more are prime candidates for a *preemptive rewrite*.

Graphical User Interface

This chapter defines and explains the content of the **Main** window that makes up the *METRICTM* product. If you have questions about operation, please refer to the appropriate chapter for further information. (See CHAPTER 4 - "System Operation" on page 71.)

This chapter is intended to act as a reference chapter.

LEVEL: This chapter is intended for all users.

6.1 About the Main Window

Once you have invoked *METRICTM*, all operations are accessible from this window, including:

- Selecting a source code file for *METRICTM* to process.
- Writing report files to a specific prefix.
- Looking at reports.
- Setting report thresholds.
- Looking at Kiviat diagrams.
- Setting Kiviat definition thresholds.

6.2 Main Features of METRIC

The window includes the following features:

- Display area where reports are loaded.
- **File** pull-down menu. You can use it to load a source code file, multiple source code files, save report files to files that share the same prefix, and exit *METRICTM*.
- **Options** pull-down menu. You can use it to set parameters for the reports and Kiviat charts and to set the language.
- **Report** pull-down menu. You can use it to select reports for *METRICTM*.
- **Charts** pull-down menu. You can use it to select different types of Kiviat charts.
- **Help** button. This button brings up the main **Help** window for *METRICTM*.

Each of these features will be discussed in the sections that follow.

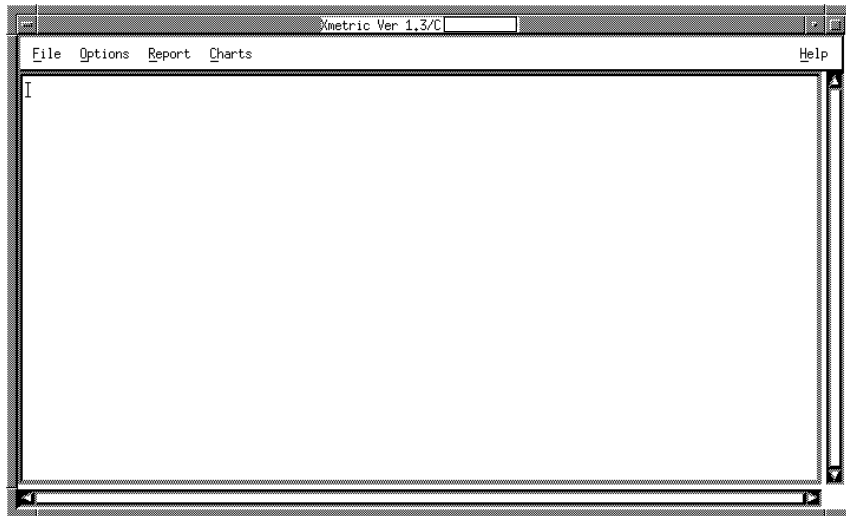


FIGURE 48 Display Area

6.2.1 Display Area

When reports are generated for *METRIC™*, they are displayed in the scrolled display area. You can use the scroll bars to move up/down or side/side. If you want the reports to fit the size of the window display, you must resize the window.

6.2.2 File Pull-Down Menu

File pull-down menu consists of these features:

Load Single File option brings up a selection dialog box that allows you to select an existing source code file. After clicking on **OK**, *METRIC™* automatically processes the file and generates reports. These reports are loaded into the display area. Please refer to the appropriate section for operation instructions (See Section 4.4.3 - "Selecting a Source Code File" on page 84.)

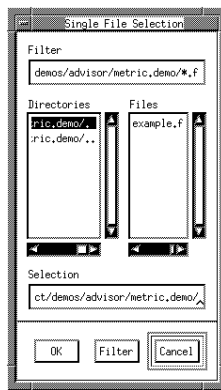


FIGURE 49 Load Single File Selection

Load Multiple Files option brings up a selection dialog box that allows you to select more than one source code file. After clicking on **OK**, *METRIC™* automatically processes the files and generates reports. These reports are loaded into the display area. Please refer to the appropriate section for operation instructions (See Section 4.4.4 - “Selecting Multiple Source Code File” on page 85.).

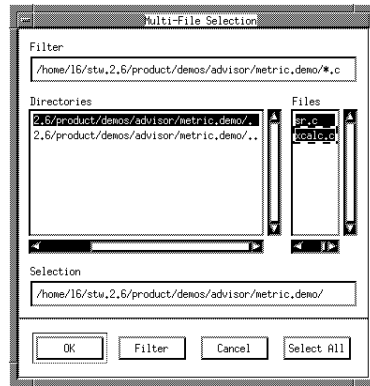


FIGURE 50 Load Multiple Files Selection

Set Report Files Basename option brings up the **Set Report Files Basename** window, where you type in the prefix that you want all of your reports to share. You must set the *basename* before you load in source code, if you want your report to be saved. Otherwise, the reports will not be saved. Please see Section 4.4.2 for operation instructions.



FIGURE 51 Setting the Report Files Basename

Help option brings up an on-line help window for the **File** pull-down menu. See Section 4.2 for information on using help windows.



FIGURE 52 File Pull-Down Window Help

Exit options terminates *METRIC™*.

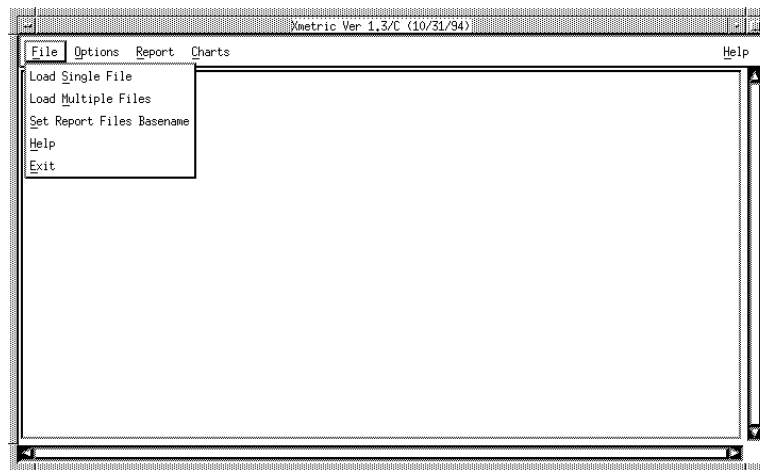


FIGURE 53 File Pull-Down Menu

6.2.3 Options Pull-Down Menu

The **Options** pull-down menu consists of these options:

Report option brings up the report **Configuration Options** window. Please refer to the appropriate section for operation instructions (See Section 4.4.2 - "Writing Reports to a File" on page 82.).

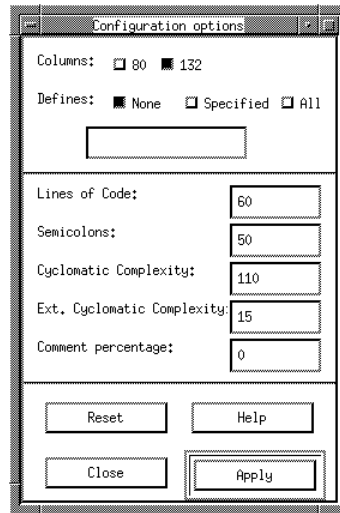


FIGURE 54 Configuration Options Window

Here you can adjust the following report parameters:

- **Columns:** Allows you to set the column width for the **Complexity** report. We recommend that use the 132-column report when you have very large procedures in the files you are analyzing and wider printer is available.

An 80-column report does **not** have the following complexity metrics: P/R (Purity Ratio), BLK (Number of Blank Lines), CMT (Number of Comment Lines), and VL (Average Variable Name Length). BLK and CMT are replaced with B/C, which is a combination calculation for BLK and CMT.

- **Defines:** Allows you to turn on **None** (the default) of the source code file(s) conditional compilation directives, **Specific** directives, or **All** of the directives during **Xmetric** processing. This option only applies to "C" and "C++".

- **Semicolons:** Allows you set the maximum threshold for the number of semi-colons a procedure can have. Those procedures that exceed this threshold are listed in the **Exception** report. The threshold is set at 50.
- **Lines of Code:** Allows you set the maximum threshold for the number of lines of code a procedure/function can have. Those procedures/functions that exceed this threshold are listed in the **Exception** report. The threshold is set at 62.
- **Cyclomatic Complexity:** Allows you to set the maximum threshold value for the cyclomatic number a procedures/functions can have. Those procedures/functions that exceed this threshold are listed in the **Exception** report. The threshold is set at 10.
- **Extended Cyclomatic Complexity:** Allows you set the maximum threshold for the extended cyclomatic complexity number a procedure/function can have. Those procedures/functions that exceed this threshold are listed in the **Exception** report. The threshold is set at 15.
- **Comment Percent:** Allows you set the maximum threshold for the comment percent (comment/lines of code - blank lines) a procedure/function can have. Those procedures/functions that exceed this threshold are listed in the **Exception** report. The threshold is set at 0.

Charts cascading menu offers has the following options:

Type I option brings up the Type I Configuration window. This window (See Figure 55 "Type I Configuration Window" on page 133.) consists of the minimum and maximum threshold complexity measure values for the **Type I** Kiviat chart. Remember, Kiviat diagrams provide a graphical representation of several metrics on a source code file or multiple files. Please refer to the appropriate section for further information on setting these threshold parameters (See Section 4.6.2 - "Looking at a Type II Kiviat Chart" on page 103.).

Metric	Min	Max
Unique Operators	8	1458
Unique Operands	27	777
Total Operators	37	6377
Total Operands	47	3477
Lines of Code	157	1577
#Comment Lines	177	1777
#Blank Lines	100	1000
#Executable Semi-colons	157	1577
#Functions	1	80

Reset Help
Close Apply

FIGURE 55 Type I Configuration Window

Type II option brings up the Type II Configuration window. This window (shown below) consists of the minimum and maximum threshold complexity measure values for the **Type II** Kiviat chart. The **Type II** Kiviat diagram offers different complexity measures than the **Type I** diagram. Please refer to Section 4.6.2 for further information on setting these threshold parameters.

Metric	Min	Max
Software Science Length	6700	9677
Est. Software Science Length	67	9677
Purity Ratio	0.070000	0.700000
Software Science Volume	77000	97777
Software Science Effort	8700000	111778877
Estimated Errors	2	37
Estimated Time To Develop	107	1077
Cyclomatic Complexity	117	1177
Ext. Cyclomatic Complexity	1	377
Avg. Cyclomatic Complexity	1	37
Avg. Ext. Cyclomatic Complexity	147	1477

Reset Help

Close Apply

FIGURE 56 Type II Configuration Window

Type III option brings up the Type III Configuration window. This window (shown below) consists of the minimum and maximum threshold complexity measure values for the **Type III** Kiviat chart. The **Type III** Kiviat diagram offers all of the complexity measures from the **Summary** report. Please refer to the appropriate section for further information on setting these threshold parameters (See Section 4.6.3 - "Looking at a Type III Kiviat Chart" on page 107.).

Metric	Min	Max
Unique Operators	17	177
Unique Operands	27	277
Total Operators	37	377
Total Operands	47	477
Software Science Length	57	577
Est. Software Science Length	67	677
Purity Ratio	0.070000	0.700000
Software Science Volume	77	7777
Software Science Effort	87	778877
Estimated Errors	97	977
Estimated Time To Develop	107	1077
Cyclomatic Complexity	117	1177
Ext. Cyclomatic Complexity	127	1277
Avg. Cyclomatic Complexity	137	1377
Avg. Ext. Cyclomatic Complexity	147	1477
Lines of Code	157	1577
#Comment Lines	167	1677
#Blank Lines	177	1777
#Executable Semi-Colons	187	1877
#Functions	197	1977

Reset Help

Close Apply

FIGURE 57 Type III Configuration Window

Language brings up the Select Language window (shown below). This window allows you to select the language you would like to work from: "C", "C++", Ada, and FORTRAN. The default is "C". You must have the proper language set, before you load a language source file. For example, you can only load an Ada file if the Ada language is set. Please refer to the appropriate section for operation instruction (See Section 4.4.1 - "Selecting a Language" on page 81.).



FIGURE 58 Select Language Window

6.2.4 Report Pull-Down Window

The window offers the following report options:

Order Complexity option brings up the Sort Report By window (shown below). This window lets you specify which complexity measure you would like your procedures/functions ordered by in the **Complexity** report. This is useful for mid- to large- scale programs, where it may be difficult to initially determine the most complex modules. The default is **procedure**.

Please see the appropriate section for operation instructions (See Section 4.5.2 - "Re-Ordering Procedures/Functions" on page 89.).

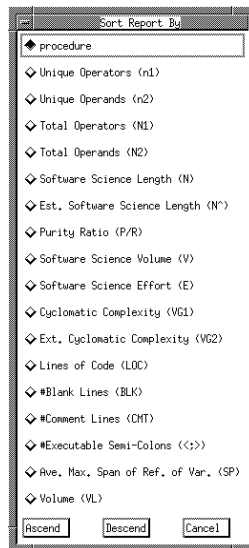


FIGURE 59 Sort Report By Window

Complexity option loads the **Complexity** report into the display area. Please see the appropriate section for instruction and background information (See Section 4.5.1 - "Looking at a Complexity Report" on page 87.).

Summary Only option loads the **Summary** report into the display area. Please see the appropriate section for instruction and background information (See Section 4.5.3 - "Looking at a Summary Report" on page 91.).

Exceptions option loads the **Exception** report into the display area. Please see the appropriate section for instruction and background information (See Section 4.5.4 - “Looking at an Exception Report” on page 93.).

Errors option loads the **Error** report into the display area. Please see the appropriate section for instruction and background information (See Section 4.5.5 - “Looking at an Error Report” on page 94.).

Generic option loads the **Generic** report into the display area. This report is available for Ada only. Please see Appendix C for instruction and background information.

Package Exceptions option loads the **Package Exception** report into the display area. This report is available for Ada only. Please see Appendix C for instruction and background information.

Package Intermediates option loads the **Package Intermediates** report into the display area. This report is available for Ada only. Please see Appendix C for instruction and background information.

C++ Class option loads the **C++ Class** report into the display area. This report is available for “C++” only. Please see Appendix B for instruction and background information.

Class Summary option loads the **Class Summary** report into the display area. This report is available for “C++” only. Please see Appendix B for instruction and background information.

Class Hierarchy option loads the **Class Hierarchy** report into the display area. This report is available for “C++” only. Please see Appendix B for instruction and background information.

Class Exceptions option loads the **Class Exception** report into the display area. This report is available for “C++” only. Please see Appendix B for instruction and background information.

Help brings up an on-line help window for the **File** pull-down menu.

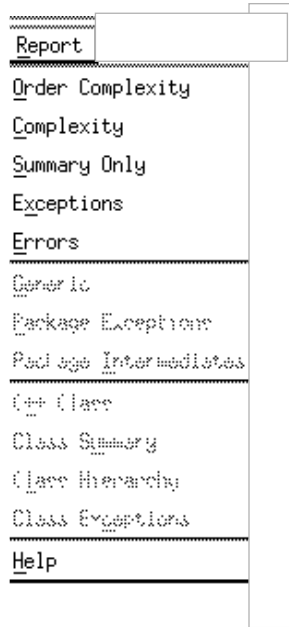


FIGURE 60 Report Pull-Down Menu

6.2.5 Charts Pull-Down Menu

Charts pull-down menu consists the following options:

Type I displays the first kind of Kiviati chart, which consists of the of the following software measures for the processed source code:

- Average Cyclomatic Complexity.
- Lines of Code (LOC).
- Software Science Length (N).
- Estimated Errors (E^).
- Purity Ratio (P/R).
- Number of Functions.

Below is a Type I Kiviati diagram. The inner circle represents minimum values, the outer circle represents maximum values and radii through the circles represent the metrics of interest.

Observed values are plotted on the radii and connected. From this, metrics that are not within the acceptable range of values can be easily identified. Please see the appropriate section for further information(See Section 4.6 - “Graphically Viewing Complexity” on page 98.).

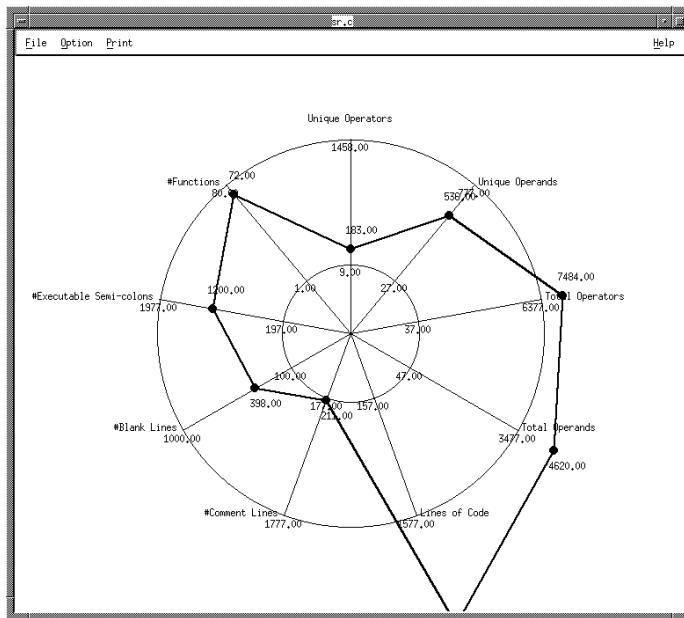


FIGURE 61 Type I Kiviati Chart

Type II displays the second kind of Kiviatic chart, which consists of the of the following software measures for the processed source code:

- Unique Operators (n1).
- Unique Operands (n2).
- Total Operators (N1).
- Total Operands (N2).
- Software Science Length (N).
- Estimated Errors (E^).
- Lines of Code (LOC).
- Number of Functions.

Below is a **Type II** Kiviatic diagram. The inner circle represents minimum values, the outer circle represents maximum values and radii through the circles represent the metrics of interest.

Observed values are plotted on the radii and connected. From this, metrics that are not within the acceptable range of values can be easily identified. Please see the appropriate section for further information (See Section 4.6.2 - "Looking at a Type II Kiviatic Chart" on page 103.).

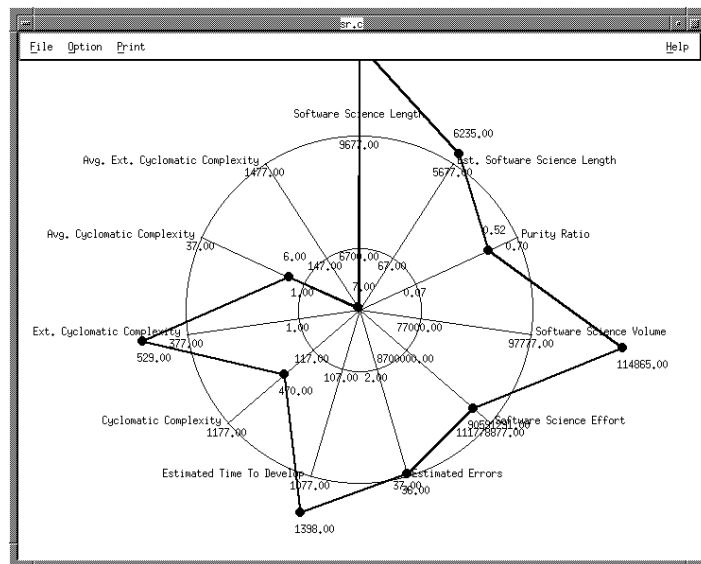


FIGURE 62 Type II Kiviatic Chart

Type III displays the third kind of Kiviat chart, which consists of all the software measures represented in the **Summary** report. Please see the appropriate section for further information (See Section 4.6.3 - "Looking at a Type III Kiviat Chart" on page 107.).

Below is a **Type III** Kiviat diagram.

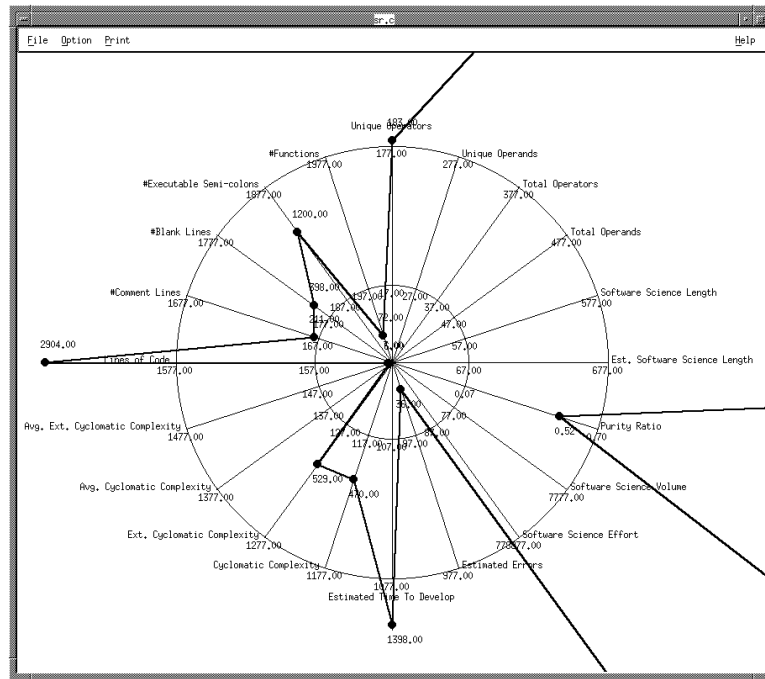


FIGURE 63 Type III Kiviat Chart

User brings up a file selection dialog box (shown below), which allows you to select a user-customized file. Please see the appropriate section for further information (See Section 4.6.4 - "Customizing Your Own Kiviat Chart" on page 113.).

When you select the file, a Kiviat diagram will pop up.

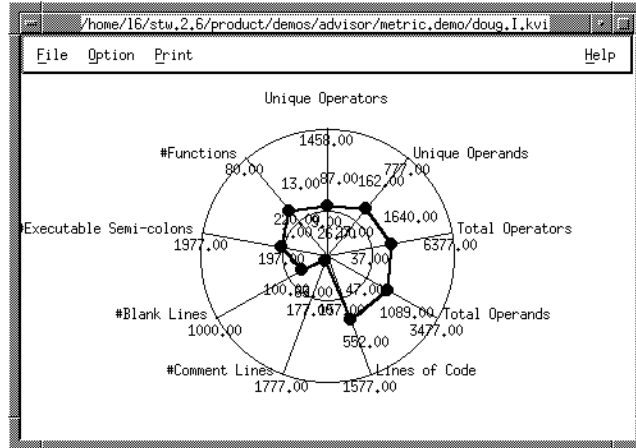


FIGURE 64 Type User Kiviatic Chart Selection

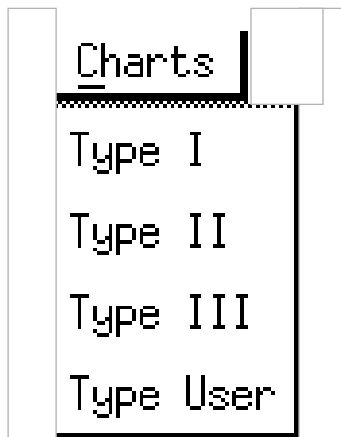


FIGURE 65 Charts Pull-Down Menu

6.2.6 Help Button

The **Help** activation button provides you with an on-line explanation of *METRIC*TM. Please refer to the appropriate section for usage of help windows (See Section 4.2 - "User Interface" on page 72.).

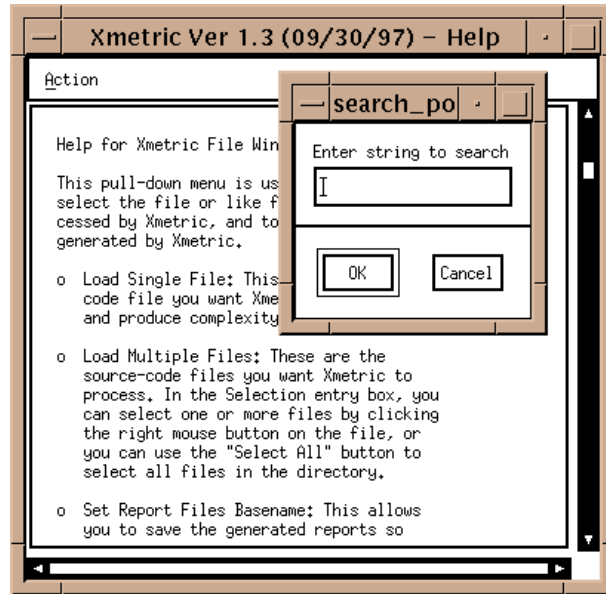


FIGURE 66 Help Window for METRIC

Command Line Activation

READERS' GUIDE: This chapter describes in detail the various command line switches which perform tasks very similar to the graphical user interface.

LEVEL: If you are a beginning or intermediate *METRIC™* user, you can skip this section on first reading. This chapter is intended for advanced users.

7.1 Command Line Usage

You may want to work from command line in situations where you will be analyzing several source files and just want to obtain complexity measures. This chapter describes the main operating modes of *METRIC™*, including obtaining reports and using Kiviat charts.

It also lists the configuration file, `.uxmetriccfg`, commands. Editing this configuration file can save you a lot of time. Rather than use special switches (for command line) or edit the GUI's parameters, you can edit this file to change a parameter.

You may find it helpful to read the *APPENDIX* for your specific language.

7.2 'Xmetric' Command

You invoke the *METRICTM* GUI with the command:

```
Xmetric [-L lang]
```

Options and Parameters:

<i>No Options</i>	Invokes <i>Xmetric</i> for the "C" language interactively.
<i>-L lang</i>	Specifies the language. The following languages are supported: <ul style="list-style-type: none"><i>-L C</i> - Supports the "C" language. This is the default.<i>-L C++</i> - Supports the "C++" language.<i>-L Ada</i> - Supports the Ada language.<i>-L F77</i> - Supports the FORTAN language.

Note: Please refer to the appropriate chapter for GUI usage (See CHAPTER 6 - Graphical User Interface" on page 125.).

7.3 'langmetric' Command

To execute in command line mode, at the system prompt enter

`cmetric <filename1> ... <filenamen> <options>` for "C".

`adametric <filename1> ... <filenamen> <options>` for Ada.

`cppmetric <filename1> ... <filenamen> <options>` for "C++".

`fmetric <filename1> ... <filenamen> <options>` for FORTRAN.

where *<filename>* is the name of the file(s) to be analyzed. Enter a path if the file(s) to be analyzed reside in a directory other than the current directory. Any number of files contained in any number of directories may be listed on the `i` command line (each file name must be separated by a space). Also, you may enter file name patterns (e.g. `*.c`, `*.ada`, `*.cpp`, `*.for`).

By default, the report files will be named after the first file name to be analyzed. You may specify your own report file name by entering `-obasename` anywhere on the command line after the `langmetric` command. If the report files are to be located in a directory other than the current directory, specify a path.

Do not enter an extension with the file name. The report files created will be `basename.rpt`, which contains a combined Complexity report by procedure and Summary report; `basename.exp`, which contains an Exception report; and `basename.err`, which contains any processing errors. If you analyze multiple files, the report files `basename` will be named after the first file specified for analysis.

NOTE: Additional reports are created for "C++" and Ada.

Options and Parameters

langmetric [basename]
[-bd]
[-bn]
[-cen]
[-chtn]
[-crn]
[-csym]
[-gn]
[-i]
[-ne]
[-pen]
[-pn]
[-prn]
[-s]
[-80|-132]

Additionally in the command line mode, the options *-bd*, *-bn*, *-csym*, *-cen*, *-chtn*, *-crn*, *gn*, *-i*, *-ne*, *pen*, *pn*, *prn*, *-s*, and *-80* or *-132* may be specified anywhere after the *langmetric* command. These are explained as follows:

- bd* Directs *METRICTM* to create an Intermediate Summary complexity report whenever a change in directory occurs. This may be helpful if different subsystems of an application being analyzed are placed in different directories and you want a summary complexity report for each subsystem.
This report is saved to *filename.rpt*.
- bn* Directs *METRICTM* to break whenever a change in the first *n* characters of the filenames occurs and creates an Intermediate Summary complexity report. This may be helpful if the files comprising the different subsystems of an application all have the same prefix and you want a summary complexity report for each subsystem.
This report is saved to *filename.rpt*.

<code>cht<i>n</i></code>	<p>Specifies whether or not the Class Hierarchy Table report, <i>filename.cht</i>, is to be generated. A value of 1 means to generate the report, a value of 0 will not generate the report. This value overrides the CLASSCHART entry in the configuration file, <i>.uxmetriccfg</i>.</p> <p>If this option is not specified in the command line, METRICTM will look for CLASSCHART in the configuration file. If it is not found, then this report will not be generated.</p> <p>This options applies only to “C++”.</p>
<code>-cr<i>n</i></code>	<p>Specifies whether or not the class report, <i>filename.cls</i>, is to be generated. A value of 1 means to generate the report, a value of 0 will not generate the report. This value overrides CLASSREPORT entry in the configuration file, <i>.uxmetriccfg</i>.</p> <p>If this option is not specified in the command line, METRICTM will look for CLASSREPORT in the configuration file. If it is not found, then this report will not be generated.</p> <p>This options applies only to “C++”.</p>
<code>-ce<i>n</i></code>	<p>Specifies whether or not the class exception report, <i>filename.cex</i>, is to be generated. A value of 1 means to generate the report, a value of 0 will not generate the report. This value overrides CLASSEXCEPTION entry in the configuration file, <i>.uxmetriccfg</i>.</p> <p>If this option is not specified in the command line, METRICTM will look for CLASSEXCEPTION in the configuration file. If it is not found, then this report will not be generated.</p> <p>This options applies only to “C++”.</p>
<code>-csym</code>	<p>The <code>-csym</code> option(s) specify the conditional compilation directives that are to be TRUE. Any number of <code>-csym</code> options can be used. If you want all directives to be TRUE, use the option <code>-call</code>.</p> <p>This option only applies to “C” and “C++”. Please refer to <i>Appendix A</i> for “C” compilation directives and <i>Appendix B</i> for “C++” compilation directives.</p>

- `-gn` Indicates whether or not you want the generic instantiated information printed in the `Generic` report. A value of 1 prints the information, a value of 0 does not. This value overrides the entry in the configuration file. Note that both the generic instantiated information and the interface pragma information are printed in the report file `filename.gen`.
- Disabling one portion of the report does not stop the report from being generated, but only a portion of the report. The report will not be generated only if both portions have been disabled.
- This option applies only to Ada.
- `-i` Directs `METRICTM` to display all processing messages on the screen. This option cannot be used when redirecting or piping input and output.
- `-ne` Directs `METRICTM` not to produce an `Exception` report, `filename.exp`. This overrides the `PRINTEXP` entry in the configuration file.
- `-pen` Indicates whether the `Package Exception` report, `filename.pex` should be printed or not. A value of 1 prints the report, a value of 0 does not. This option overrides the `PRINTPKGEXP` entry in the configuration file.
- This option applies only to Ada.
- `-pn` Specifies the package break level. Use this option if you wish to produce an **Intermediate Summary Complexity** report at the package level. Package breaks are only performed at the outermost level nested packages will not have a break. A value of 0 indicates that no package breaks are to be performed and a value of 1 indicates that package breaks are to be done. This overrides the package summary level parameter in the configuration file.
- This option applies only to Ada.

<code>-prn</code>	<p>Indicates whether or not you want the Interface Pragma information printed in the Generic report, <i>filename.gen</i>. A value of 1 prints the information, a value of 0 does not. This value overrides the entry in the configuration file.</p> <p>Note that both the generic instantiated information and the interface pragma information are printed in the Generic report file.</p> <p>Disabling one portion of the report does not stop the report from being generated, but only a portion of the report. The report will not be generated only if both portions have been disabled.</p> <p>This option applies only to Ada.</p>
<code>-s</code>	<p>Directs <i>METRICTM</i> to produce only the Summary report portion of <i>filename.rpt</i> (i.e., the procedure by procedure report Complexity is not generated). This overrides the SUMMARY ONLY entry in the configuration file.</p>
<code>-80 -132</code>	<p>These parameters indicate the column width of the complexity report. This overrides the 'report width' entry in the configuration file. The 132 column report is the default.</p> <p>Please note that the 80 column report does not have the following complexity measures found in the 132 column report: P/R (Purity Ratio), BLK (Blank Lines), CMT (Comment Lines), and VL (Variable Name Length). The fields BLK and CMT are combined into one field, B/C.</p>

If all options are used for each language, the following reports are generated:

- For “C”:
 - *filename.rpt* - The Complexity and the Summary reports.
 - *filename.exp* - The Exception report.
 - *filename.err* - The Error report.
- For “C++”:
 - *filename.rpt* - The Complexity and the Summary reports.
 - *filename.exp* - The Exception report.
 - *filename.err* - The Error report.
 - *filename.cls* - The C++ Class and Class Summary reports.
 - *filename.cht* - The Class Hierarchy report.
 - *filename.cex* - The Class Exception report.
- For Ada:
 - *filename.rpt* - The Complexity, the Summary, and the Package Intermediates reports.
 - *filename.exp* - The Exception report.
 - *filename.err* - The Error report.
 - *filename.gen* - The Generic report.
 - *filename.pex* - The Package Exception report.
- For FORTRAN:
 - *filename.rpt* - The Complexity and Summary reports.
 - *filename.exp* - The Exception report.
 - *filename.err* - The Error report.

7.4 'Xkiviat' - Static Metrics Display System

This command produces an X Window display that shows the relationships of a set of input factors, taken from a simple input file format, in a standard kiviati diagram format.

The diagram has:

1. A minimum threshold inner circle.
2. A maximum outer circle.
3. Radii through the circles represent complexity metrics.
4. Plotted dots along the radii represent the observed values.
5. Text annotation of each radii in standard locations.
6. The set of dots is connected with a double-width line to illustrate the relationships.

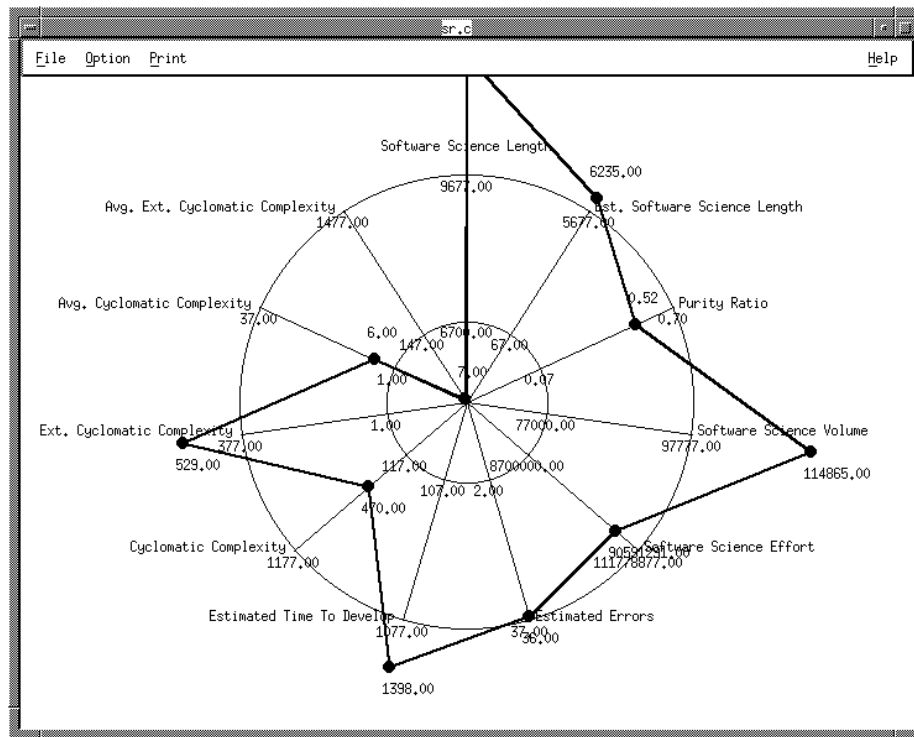


FIGURE 67 Example Kiviati Diagram

Options and Parameters

The `xkiviat` function is called as follows:

```
xkiviat filename
        [-n value]
        [-q]
        [-s x y]
        [-v]
```

where the parameters and switches have the following values:

`filename` Input Data File Name. This file is “white-space” delimited and contains the following information for each radius, or “arm” of the diagram: This argument is required to be present.

1. The minimum value for the minimum arm threshold.
2. The maximum value for the maximum arm threshold.
3. The actual value to be displayed on the arm.
4. The text (maximum 60 characters) to be displayed on the diagram.

`-n value` Number of Arms Value. This is the number of “arms” in the kiviatic chart. This number must be 4, 6, 8, 10, or 12. The default value is 8. (Actually, it would be nice if it were able to do *any* number of arms, say even 24(!), but that is not a hard requirement.)

If there is insufficient data to fill up all of the specified arms then the dots are placed at the *midpoints* of non-specified arms.

Extra lines in the input file, beyond the number of arms specified by `-n`, are ignored.

If the `-n` switch is absent, the Kiviatic chart is drawn with one arm for each non-blank or non-comment line in the file.

`-q` Quiet Operation Switch. The version and compilation date and other information is *not* displayed.

`-s x y` Diagram Size Switch. The size of the display in pixels. Default is 250 x 250. The actual diagram is scaled to fit into the specified area.

`-v` Display Value Switch. The complexity measures and the upper and lower threshold values are displayed in Kiviatic

Here is an example input file (lines with # are treated as comments):

```
# This is a sample of the input file for
"Xkiviat".
#
10      100      45      Number of State-
ments
# Min   Max     Value   Text
0.2     0.7     0.75   S1 Coverage Value
#
10      100      45      Number of State-
ments
#
0.5     0.9     0.75   C1 Coverage Value
#
0.2     0.7     0.75   S1 Coverage Value
#
10      100      45      Number of State-
ments
#
```

7.5 Configuration File Processing

The configuration file, `.uxmetriccfg`, can be used to change various guideline parameters supplied with the program. All configurations for all language versions of *METRICTM* are contained in this file. The group of entries pertaining to the “C” version of *METRICTM* must be preceded by a single line containing the entry `{C}`. The end of the “C” entries occurs when either a new *METRICTM* version is encountered (e.g., `{C++}`) or the end of file is reached. The Metric Configuration File can be updated through the use of your favorite UNIX editor.

It may be placed in any directory you wish. *METRICTM* first checks the user’s home directory for the file. If it is not found, then the environment variable `METRICCFG` is checked for the location of `.uxmetriccfg`. To set the environment variable, it is most convenient to add an entry to your `.login` file indicating where the configuration file is at. For example:

```
setenv METRICCFG
/usr/staff/develop/.uxmetriccfg
```

You may change values for some or all of the defaults so as to tailor them to your specifications.

The entries `LOC`, `SEMI`, `SPAN`, `VARIABLE_LENGTH`, `VG`, `VG+`, and `VOLUME` refer to threshold values which *METRICTM* uses to determine if a procedure exceeds user-defined complexity maximums. All procedures, functions, or subprograms that exceed one or more of these values will be listed in the `Exception` report. The default thresholds are general guidelines, but it is important that these values are tailored to the environment and installation norms.

The parameters `EO` and `SPEED` are used in the Software Science calculations for predicted number of bugs (B^{\wedge}) and predicted development time (T^{\wedge}). Again, the default values are general guidelines.

These parameters should be tailored to the environment *METRICTM* is being used in. This can be done by retroactively applying *METRICTM* to past code development projects. Assuming the database is representative of current projects, these values should provide reasonably accurate estimates in most cases.

The following parameters are available:

<code>ADVLENGTH</code>	Exception Page Length. This parameter indicates how many lines should be printed in a report before breaking for a new page. The default is 53 lines.
------------------------	---

ANALYZEINCLUDE	<p>Analyze Includes. This parameter indicates whether include files should be automatically analyzed (a value of 1) or not analyzed (a value of 0). The default is to analyze the included files (1). If you chose to not have include files automatically picked up, you will have to explicitly list them in the files to be analyzed.</p> <p>This parameter is not available for Ada.</p>
CLASSCHART	<p>Generate Class Chart. This parameter indicates whether or not the <i>filename.cht</i> report is to be generated. A value of 1 will generate a report, a value of 0 will not generate the report. These values can be overridden with the command line parameter -cht. The default is 1.</p> <p>This parameter applies only to "C++".</p>
CLASSEXCEPTION	<p>Generate Class Exception. This parameter indicates whether or not the <i>filename.cex</i> report is to be generated. A value of 1 will generate a report, a value of 0 will not generate the report. These values can be overridden with the command line parameter -ce. The default is 1.</p> <p>This parameter applies only to "C++".</p>
CLASSREPORT	<p>Generate Class Report. This parameter indicates whether or not the <i>filename.cls</i> report is to be generated. A value of 1 will generate a report, a value of 0 will not generate the report. These values can be overridden with the command line parameter -cr. The default is 1.</p> <p>This parameter applies only to "C++".</p>
COLSTART	<p>Starting Column. This parameter indicates in which column the FORTRAN code starts - including line labels. Generally this value will be 1. However, if you have some old FORTRAN code to be analyzed that has the sequencing numbers appearing in column 1 rather than after column 72, you will need to change this value. The default value is 1.</p> <p>This parameter is available for FORTRAN only.</p>

COMMENT_PERCENT	<p>Comment Percentage. This parameter indicates what percentage of a procedure's lines of code should be comprised of comments. A procedure's comment percentage is calculated as</p> $\text{Comments} / (\text{LOC} - \text{Blanks})$ <p>If a procedure does not meet this percentage, it will be listed in the <code>Exception</code> report. The default value is 60.</p>
COMMENT_SYMBOL	<p>Comment Symbol. This parameter indicates that if this symbol is found, the rest of the line is a comment. The default value is set to 1, which will treat the rest of the line as a comment; a value of 0 will not treat it as a comment.</p> <p>This parameter is available for FORTRAN only.</p>
CONDCOMPILE	<p>Conditional Compilation. This parameter determines how compiler directives should be analyzed. A value of 0 means that conditional compilation is not enabled and, hence, all code contained in <code>#ifdefs</code>, <code>#if</code>, <code>#else</code>, etc. is analyzed. A value of 1 means that conditional compilation is enabled and only if something has been defined will the <code>#ifdef</code> code portion be analyzed; otherwise the <code>#else/#ifndef</code> code portion is analyzed.</p> <p>The default is set to 1. This option is available for "C" and "C++" only.</p>
COUNTCR	<p>Count Carriage Returns. This parameter indicates whether executable carriage returns should be counted as an operator (value 1) or not counted (value 0). Counting carriage returns as an operator will affect the counts for <code>n1</code> and <code>N1</code>. This option is provided so that the FORTRAN analyzer is consistent with the other <i>METRICTM</i> analyzers that count a statement ender (such as a semicolon, <code>;</code>, in languages such as "C", "C++", and Ada) as an operator. The default is 1.</p> <p>This parameter is available for FORTRAN only.</p>
COUNTINC	<p>Count Include Lines. The parameter determines whether or not to count include lines. A value of 0 (the default) means include lines will not be counted; a value of 1 will count include lines.</p> <p>This parameter is available for FORTRAN only.</p>

DLINES	<p>Debug Lines. This parameter indicates whether debug lines (lines with a 'D' in column 1) are counted as executable statements (a value of 1) or treated as comment lines (a value of 0). The default is to treat them as comment lines. The default value is 0.</p> <p>For FORTRAN only.</p>
EO	<p>Programmer Error Rate for each Procedure (for "C" and Ada), Function (for "C++"), or Subprogram (for Fortran). This is the value for predicting the number of programming errors (E^{\wedge}) a program should have. The default is set to 3200.</p>
FREEFORMAT	<p>Free Format Line. This parameter identifies the format that lines in the source code take on. A value of 1 means that column restrictions of older FORTRAN code is not followed. A value of 0 means that columns are used and a line of code does not exceed 72 columns. The default value is 1.</p> <p>This parameter is available for FORTRAN only.</p>
GOTOS	<p>Maximum Gotos. This parameter indicates the maximum number of GOTOS that a subprogram can have before an exception is generated. The default value is 5.</p> <p>This parameter is available for FORTRAN only.</p>
HEADER_SUMMARY	<p>Header Summary. This parameter indicates whether a Summary report should be generated in the <i>x.rpt</i> file even if no executable code was encountered. Since no operators and operands would have been found, the information produced would be lines of code, comment lines, and blank lines. A value of 1 generates the header summary and a value of 0 does not. The default is 0.</p> <p>This parameter is available for "C" and "C++" only.</p>

INCLUDE_DIR	<p>Included Directories. This parameter indicates the directories to look for include files if they could not be found given the information in the file being analyzed. Any number of INCLUDE_DIR entries may appear and each one will be searched in turn until either the included file is found or no more places are listed to look. The format for these entries is:</p> <pre>INCLUDE_DIR /dir1/dir2 INCLUDE_DIR /dir1/subdir2</pre> <p>where the path that is entered is the path leading to the directory in which to search for the file.</p> <p>This parameter is available for “C”, “C++” and Fortran.</p>
INLINE_COMMENTS	<p>Count In-line Comments. This parameter indicates whether comments appearing on a line not by itself should be included in the comment count. A value of 1 counts inline comments; a value of 0 does not. If inline comments are counted, then the sum of the comments, blanks, and executable statements may exceed the Lines of Code (LOC) count at both the procedure by procedure level and the Summary level. You do not want to count in-line comments if you are trying to derive non-comment source statements. 1 is the default value for “C”, “C++” and Ada; 0 is the default value for FORTRAN.</p>
LOC	<p>Lines of Code in a Procedure (for “C” and Ada), Function (for “C++”), or Subprogram (for Fortran). This default threshold value is set to 62 lines of code for each procedure, function, or subprogram.</p> <p>You can also edit this parameter from the GUI for Xmetric with the Configuration window (available with the Options pull-down menu).</p>
MAXPKG ^N	<p>Package Maximum ^N. This parameter indicates what the minimum and maximum estimated length of a package should be. Any time a package does not meet or exceeds the standard, it will be listed in the Package Exception report. A value of zero will not generate an exception; only non-zero entries will cause an exception to be listed. The default is 0.</p> <p>This parameter is available for Ada only.</p>

MAXPKGSEMI	<p>Package Maximum Statements. This parameter indicates the minimum and maximum number of executable statements that a package should contain. Any time that a package does not meet or exceeds the standard, it will be listed in the package exception report. A value of zero will not generate an exception; only non-zero entries will cause an exception to be listed. The default is 0.</p> <p>This parameter is available for Ada only.</p>
MAXSTDN [^]	<p>Procedure Maximum N[^]. This parameter indicates the minimum and maximum estimated length that a procedure should have. Any time a procedure does not meet or exceeds the standard, it will be listed in the procedure Exception report. A value of zero will not generate an exception; only non-zero entries will cause an exception to be listed. The default is 0.</p> <p>This parameter is available for Ada only.</p>
MINPKG [^]	<p>Package Minimum N[^]. This parameter indicates what the minimum and maximum estimated length of a package should be. Any time a package does not meet or exceeds the standard, it will be listed in the Package Exception report. A value of zero will not generate an exception; only non-zero entries will cause an exception to be listed. The default is 0.</p> <p>This parameter is available for Ada only.</p>
MINPKGSEMI	<p>Package Minimum Statements. This parameter indicates the minimum and maximum number of executable statements that a package should contain. Any time that a package does not meet or exceeds the standard, it will be listed in the package exception report. A value of zero will not generate an exception; only non-zero entries will cause an exception to be listed. The default is 0.</p> <p>This parameter is available for Ada only.</p>

MINPSEMI	<p>Procedure Minimum Statements. This parameter indicates the minimum number of executable statements that a procedure should have. Any time a procedure does not meet the standard, it will be listed in the procedure <code>Exception</code> report. A value of zero will not generate an exception; only non-zero entries will cause an exception to be listed. The default is 0.</p> <p>This parameter is available for Ada only.</p>				
MINSTDN [^]	<p>Procedure Minimum N^{\wedge}. This parameter indicates the minimum and maximum estimated length that a procedure should have. Any time a procedure does not meet or exceeds the standard, it will be listed in the procedure <code>Exception</code> report. A value of zero will not generate an exception; only non-zero entries will cause an exception to be listed. The default is 0.</p> <p>This parameter is available for Ada only.</p>				
NESTED	<p>Nested Comments. Indicates whether your compiler allows nested comments or not. An entry of 0 means nested comments are not supported, and an entry of 1 means nested comments are supported. The default is set to 0.</p> <p>This parameter is available for “C” and “C++” only.</p>				
NONEXE	<p>Nonexecutable Word File. This file contains a sorted lists of all reserved words that are nonexecutable. This parameter is available for “C” and “C++” only.</p> <table><tr><td>“C”</td><td><code>cnonexe.tab</code></td></tr><tr><td>“C++”</td><td><code>cppnonexe.tab</code></td></tr></table> <p>Please refer to the appendices for further information on the above files.</p>	“C”	<code>cnonexe.tab</code>	“C++”	<code>cppnonexe.tab</code>
“C”	<code>cnonexe.tab</code>				
“C++”	<code>cppnonexe.tab</code>				
NONEXCT	<p>Reserved Word Count. It specifies the number of entries in the nonexecutable word file (<code>NONEXCT</code>).</p> <p>This parameter is available for “C” and “C++” only. The default for “C” is 25 and the default for “C++” is 40.</p>				

`OUTSIDE_COMMENT` Count Outside Comments. This parameter indicates whether comments that occur before the start of a procedure should be counted as part of the procedures comments.

A value of 1 counts outside comments; a value of 0 does not. Note that comment lines at the start of a file will be associated with the first procedure encountered, giving it a higher comment count than maybe it should have.

Also when counting outside comments, the procedure's lines of code count will be incremented by the number of outside comments. You can then calculate the value of Non-Comment Source Statements by subtracting out blanks and comments from the procedures lines of code. 0 is the default value for FORTRAN; 1 is the default value for "C", "C++" and Ada.

`PAGE_BREAK` Print Page Breaks. This parameter indicates whether a new page in the report should be started for each file in the set of files being analyzed. If a large number of files are being analyzed and each file only contains a few subprograms, you will get a lot of wasted paper when the reports are printed with each file on a new page. A value of 1 prints page breaks (default value), a value of 0 does not. The default is set to 1.

This parameter is available for FORTRAN only.

`PAGE_HEADINGS` Print Page Headings. This parameter indicates whether a new report heading is printed for each new file analyzed when pages are not skipped.

If `PRINT_HEADINGS` is set to 1, when a new file is being analyzed, two lines in the report file are skipped, a new report heading is printed, then all subprograms associated with the file are printed in the report. If `PRINT_HEADINGS` is set to 0 then all subprograms from all files are listed in the report as if they did not come from different files. The default is set to 1.

This parameter is available for FORTRAN only.

PAGELength	Report Page Length. This parameter indicates how many lines should be printed in a report before breaking for a new page. The default is 58 lines.
PKG_COUNT_ALL_LINES	<p>Count Package Lines. This parameter indicates whether or not to include all task and procedure lines of code as part of the package's lines of code.</p> <p>A value of 1 includes all task and procedure lines of code as part of the package's lines of code in the procedure by procedure Complexity Report. A value of 0 (the default) treats only the code not contained in a task/procedure as part of the package's lines of code for the Procedure by Procedure Complexity Report.</p> <p>In the case of a value of 0, the total of lines of code in the procedure by procedure report for tasks, procedures, and packages will equal the value in the Package Intermediate Summary report Lines of Code (LOC) entry.</p> <p>In the case of a value of 1, the lines of code entry for the package in the procedure by procedure report will be the same as the lines of code entry in the Package Intermediate Summary report.</p> <p>The default is 0. This parameter is available for Ada only.</p>
PKGLOC	<p>Package Lines of Code. This parameter indicates the maximum lines of code a package should contain. Any time a package exceeds this standard it will be listed in the package exception report. A value of zero will not generate an exception; only non-zero values will cause an exception to be listed. The default is 62.</p> <p>This parameter is available for Ada only.</p>

PKGSPAN	<p>Package Span of Reference. This parameter indicates the maximum span of reference between variables that a package should contain. Any time a package exceeds the standard, it will be listed in the <code>Package Exception</code> report. A value of zero will not generate an exception; only non-zero entries will cause an exception to be listed. The default is 0.</p> <p>This parameter is available for Ada only.</p>
PKGVG	<p>Package Summary Level. This parameter indicates whether an <code>Intermediate Summary</code> complexity report should be generated for packages. The values entered here may be either 0 or 1. A value of 0 inhibits the breaks from being printed. A value of 1 prints the report. Note that only the outermost package will have a report printed. Packages nested within other packages will not have an intermediate report printed. The value entered here may be overridden with the use of the command line parameter <code>-p0</code> or <code>-p1</code>. The default is set to 1.</p> <p>This parameter is available for Ada only.</p>
PKGVG2	<p>Package Cyclomatic Complexity. This parameter indicates what the maximum cyclomatic complexity for a package should be. Any time a package exceeds the standard, it will be listed in the <code>Package Exception</code> report. A value of zero will not generate an exception; only non-zero entries will cause an exception to be listed. The default is 0.</p> <p>This parameter is available for Ada only.</p>
PKGVG2	<p>Package Extended Cyclomatic Complexity. This parameter indicates what the maximum cyclomatic complexity for a package should be. Any time a package exceeds the standard, it will be listed in the <code>Package Exception</code> report. A value of zero will not generate an exception; only non-zero entries will cause an exception to be listed. The default is 0.</p> <p>This parameter is available for Ada only.</p>

PRINTER	Report Width. The determines the width of the procedure by procedure complexity report. The values entered here may be 80 or 132. Reports with 132 columns allow larger field sizes as well as additional metrics. The value entered here may be overridden with the use of the command line parameter <code>-80</code> or <code>-132</code> . The default is set to 132 columns.
PRINTEXP	Exception Report. This parameter indicates whether you want the <code>Exception</code> report printed. A value of 1 means print the <code>Exception</code> report, a value of 0 means do not print it. The command line option <code>-ne</code> will override the default value of 1.
PRINTGENERIC	Print Generic Report. This parameter indicates whether or not the generic instantiated information is printed in the <code>Generic</code> report. A value of 1 prints the generic information; a value of 0 does not. The default is 1. This parameter is available for Ada only.
PRINTPKGEXP	Print Package Exception. This parameter indicates whether or not an <code>Package Exception</code> report <code>filename.pex</code> should be produced. A value of 1 means to print the report. A value of 0 means do not print the report. Note that if the <code>Package Summary</code> level option is turned off, no <code>Package Exception</code> report will be generated. The command line option <code>-pen</code> overrides this entry. The default is 1. This parameter is available for Ada only.
PRINTPRAGMA	Print Pragma Report. This parameter indicates whether or not the interface pragma information is printed in the <code>Generic</code> report. A value of 1 (the default) prints the pragma information; a value of 0 does not. This parameter is available for Ada only.
PROTECTEDMEMS	Protect Members/Class. The default is 0. This parameter applies only to “C++”.
PUBLICMEMS	Public Members/Class. The default is 0. This parameter applies only to “C++”.
PRIVATEMEMS	Private Members/Class. The default is 0. This parameter applies only to “C++”.

RESCT	<p>Reserved Word Count. For “C” and “C++”, it specifies the number of entries in the reserved word file (RESFILE) and the nonexecutable word file (NON-EXCT). For Ada and Fortran, it specifies the number of entries in the reserved word file. You must specify this value if more entries exist than the default values.</p> <p>“C”, “C++” and Ada have a default word count of 100. Fortran’s default is 330.</p>
RESFILE	<p>Reserved Word File. This file contains a sorted list of all operators and executable reserved words.</p> <p>“C” <code>cresword.tab</code></p> <p>“C++” <code>cppresword.tab</code></p> <p>Ada <code>adaresword.tab</code></p> <p>Fortran <code>forreswo.tab</code></p> <p>Please refer to the appendices for further information on the above files.</p>
SEMI	<p>Executable Semi-colons in a Procedure (“C” and Ada), or Function (for “C++”). Specifies the default threshold value of executable statements in a procedure or function. The default is set to 50 semi-colons. Not available for FORTRAN.</p> <p>You can also edit this parameter from the GUI for Xmetric with the Configuration window (available with the Options pull-down menu).</p>
SPAN	<p>Average Span of Reference of variables in a Procedure (“C” and Ada), Function (for “C++”), or Subprogram (for Fortran). The default value is set to 10.</p>
SPEED	<p>Programmer Speed for each Procedure (for “C” and Ada), Function (for “C++”), or Subprogram (for Fortran). This is the default threshold values for predicting development time (T^{\wedge}). The default is set to 18.</p>
SPAN	<p>Average Span of Reference of variables in a Procedure (“C” and Ada), Function (for “C++”), or Subprogram (for Fortran). The default threshold is set to 10.</p>

STATEMENT	Executable Statements. Specifies the default threshold value of executable statements in a subprogram. The default is set to 50 semi-colons. Available only for FORTRAN.
STDEXPLICIT	Explicit In-line Functions. The default is 0. This parameter applies only to “C++”.
STDFRIEND	Friend Functions/Class. The default is 0. This parameter applies only to “C++”.
STDFRIENDCLS	Friend Classes/Class. The default is 0. This parameter applies only to “C++”.
STDINLINE	In-line Members/Class. The default is 0. This parameter applies only to “C++”.
STDMEMBERS	Members Per Class. The default is 0. This parameter applies only to “C++”.
STDVIRTUAL	Virtual Members/Class. The default is 0. This parameter applies only to “C++”.
SUMMARYONLY	Summary Only. This option indicates whether you want only the summary of the <code>Complexity</code> report (*.rpt) printed. A value of 1 means print summary report only, a value of 0 means also produce the procedure by procedure report. The command line option <code>-s</code> will override this option.
UNIQUE_VARIABLES	Count Unique Variables. This parameter indicates how the average variable name length is computed. A value of 1 calculates it based on unique variable names and a value of 0 calculates it based on the number of occurrences of each variable. The default value is 1.
VARIABLE_LENGTH	Average Variable Name Length. This parameter indicates what the average variable name length should be for each procedure. If a procedure does not meet this average, it will be listed in the <code>Exception</code> report. The default value is 8.

VG	<p>Cyclomatic Complexity number for a Procedure, Function, or Subprogram. Specifies the cyclomatic complexity number. The default threshold value is set to 10.</p> <p>You can also edit this parameter from the GUI for Xmetric with the Configuration window (available with the Options pull-down menu).</p>
VG+	<p>Extended Cyclomatic Complexity number for a Procedure, Function, or Subprogram. Specifies the extended cyclomatic complexity number. The default is set to 15.</p> <p>You can also edit this parameter from the GUI for Xmetric with the Configuration window (available with the Options pull-down menu).</p>
VOLUME	<p>Volume. This parameter indicates what the maximum volume of a procedure should be. Anytime a procedure exceeds this standard, it will be listed in the exception report. The default value is 3200.</p>
WARNINGS	<p>Print Warnings. This parameter indicates whether warning messages should be printed in the error file, <i>filename.err</i> (value 1) or not printed (value 0). The most common warning will be that a file does not contain any executable code. The default is 1.</p>

“C” Notes

*METRIC*TM for “C” consists of several files, the following of which will be explained in this chapter:

- `cmetric`
- `cresword.tab`
- `cnonexe.tab`

cmetric

This file is the actual analyzer. It uses the remaining files in determining if a word is an **operator** or an **operand** or if it is a nonexecutable word. A complete description on how to use `cmetric` is given in the *COMMAND LINE ACTIVATION* chapter (See CHAPTER 7 - Command Line Activation” on page 145.).

cresword.tab

This file contains a sorted list of all operators and reserved words for standard “C”. For a more detailed explanation of the contents of this file, see Section A.1.

If you are using an extension to standard “C” and need to add/remove items to/from the list, you may do so in one of two ways:

- Edit the file, `cresword.tab`, and add those entries not appearing in the list and remove the entries that should not be in the list. If the number of entries exceeds 100, you must make modifications to the configuration file `.uxmetriccfg`. Refer to the correct section for details on how to increase the number of allowable entries (See Section 7.5 - “Configuration File Processing” on page 156.).

- Create a new reserved word file containing the operators and executable reserved words for your version of “C”. If this option is chosen, you must use the configuration file indicating that a different reserved word file is being used. See the correct section for details on how to do this (See Section 7.5 - “Configuration File Processing” on page 156.).

Before you modify the list, be sure to read the section that describes the process, and understand the usage of some of the items (See Section 5.1.2 - “Software Development” on page 116.).

cnonexe.tab

This file contains a sorted list of all reserved words that are nonexecutable as defined in ANSI standard “C” (for example, `int`, `char`, and `struct`). For a more complete description of the contents of this file, refer to Section A.2. When you analyze a program using *METRICTM*, statements beginning with one of these words are skipped, and therefore do not affect the operator or operand count.

If you are using an extension to standard “C” and need to add/remove items to/from the list you may do so in one of the following ways:

- Edit the file `cnonexe.tab` and add those words not appearing in the list or remove those words from the list that you wish to be counted.
- Create a new nonexecutable word file containing the words not to be counted. If this option is chosen, you must use the configuration file, indicating that a different nonexecutable word file is being used. See the correct section for information on how to do this (See Section 7.5 - “Configuration File Processing” on page 156.).

Note that if there are more than 25 entries in the nonexecutable word file, you must make a modification to the file `.uxmetriccfg`. See the correct section for details on how to do this (See Section 7.5 - “Configuration File Processing” on page 156.).

A.1 Description of the Reports

This section describes the reports created by *METRIC*TM: the **Complexity** report, the **Summary** report, the **Exception** report, and the **Error** report.

The Complexity Report

The **Complexity** report by Procedure, *filename.rpt* includes the following fields:

- Procedure Name
- Unique Operators (n1)
- Unique Operands (n2)
- Total Operators (N1)
- Total Operands (N2)
- Length (N)
- Predicted Length (N[^])
- Purity Ratio – estimated length divided by length (P/R)
- Volume (V)
- Effort (E)
- Cyclomatic Complexity (VG1)
- Extended Cyclomatic Complexity (VG2)
- Lines of Code (LOC)
- Number of Comment Lines (CMT)
- Number of Blank Lines (BLK)
- Number of Executable Semi-Colons (< i >)
- Average Maximum Span of Reference of Variables (SP)
- Variable Name Length (VL)

Please refer to the correct chapter for an in-depth discussion of the various fields. (See CHAPTER 3 - System Introduction” on page 39.)

The screenshot shows a terminal window titled 'xmetric Ver 1.3/C [xcalc.c]'. The menu bar includes 'File', 'Options', 'Report', 'Charts', and 'Help'. The main content is a 'Complexity Report by Procedure for: /home/16/stu.2.6/product/demos/advisor/metric.demo/xcalc.c'. The report is a table with columns: Procedure, n1, n2, N1, N2, N, N*, P/R, V, E, VG1, VG2, and LOC. The data is as follows:

Procedure	n1	n2	N1	N2	N	N*	P/R	V	E	VG1	VG2	LOC
parse_double	8	6	11	6	17	40	2.32	65	259	1	1	11
open_the_display	21	15	57	26	83	151	1.82	429	7810	6	7	26
main	108	211	1404	840	2244	2359	1.05	18664	4012377	83	113	413
Syntax	6	18	67	34	101	91	0.90	463	2624	1	1	21
XCalcError	6	7	12	7	19	35	1.85	70	211	1	1	6
SetupTICalc	28	53	174	131	305	438	1.44	1934	66912	20	20	40
SetupHPCalc	28	56	175	133	308	460	1.49	1969	65464	23	23	40
DrawDisplay	24	48	191	132	323	378	1.17	1993	65765	12	12	36
DrawKey	22	22	49	37	86	196	2.28	470	8686	3	5	18
InvertKey	9	9	17	13	30	57	1.90	125	813	1	1	9
LetGoKey	28	21	100	52	152	227	1.49	853	29586	8	10	68
digit	29	26	114	66	180	263	1.46	1041	38304	9	15	32
bkspf	18	6	33	14	47	91	1.93	215	4525	4	5	12
decf	13	11	32	15	47	86	1.83	215	1910	4	4	16
eef	15	12	36	19	55	102	1.85	262	3106	5	5	16
clearf	13	12	29	16	45	91	2.03	209	1811	2	3	14
negf	18	14	60	31	91	128	1.41	455	9068	6	6	26
twoop	33	23	162	67	229	271	1.18	1330	63921	13	13	62
twof	28	22	87	42	129	233	1.80	728	19459	10	10	28
entrf	13	12	31	19	50	91	1.82	232	2390	4	4	16
equf	29	22	104	46	150	239	1.59	851	25797	10	10	42

FIGURE 68 Complexity Report for "C"

The Summary Report

The **Summary** report (part of *filename.rpt*) consists of **Complexity** report fields, with the exception of average maximum span of reference of variables and average variable name length. In addition it includes the values:

- Estimated errors (B^{\wedge})
- Estimated development time (T^{\wedge})
- Average cyclomatic complexity
- Average extended cyclomatic complexity
- Number of procedures

Please refer to the correct chapter for an explanation of the **Summary** report's fields (See CHAPTER 4 - System Operation" on page 71.).

The command line also allows you create an **Intermediate Summary** report whenever there is a change in directory (*-bn*). This report may be helpful if different subsystems of an application being analyzed are placed in different directories and you want a **Summary** report for each subsystem.

You can also create an **Intermediate Summary** report whenever a change in the first *n* characters of the filename occurs (*-bn*). This report may be helpful if the files comprising the different subsystems of an application all have the same prefix and you want a **Summary** report for each subsystem.

See the correct chapter for command line activation (See CHAPTER 7 - Command Line Activation” on page 145.).

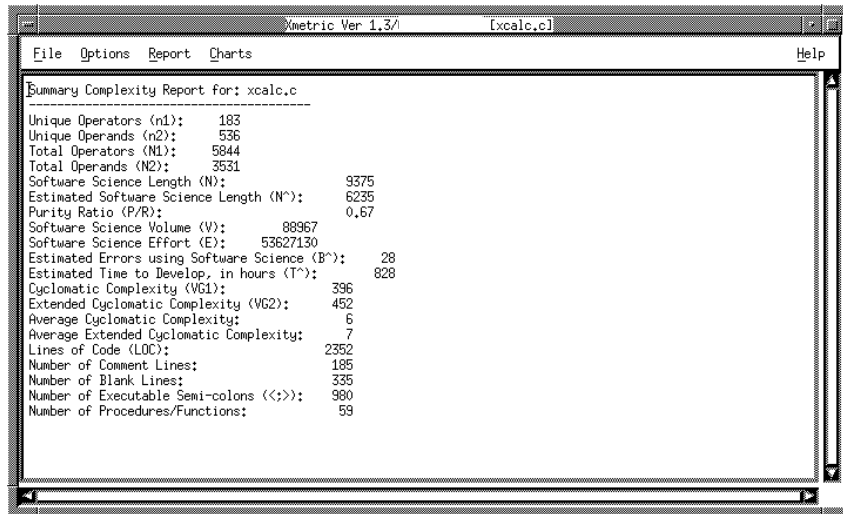
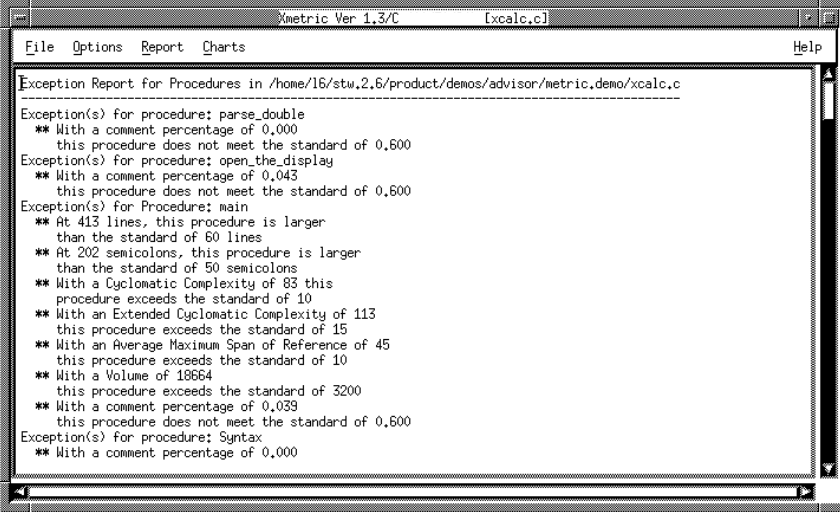


FIGURE 69 Summary Report for “C”

The Exception Report

The **Exception** report is the third report created by *METRIC*TM. Each procedure in the source files which exceeds a set of predefined complexity maximums is included in this report. This report uses either the defined standards specified in the configuration file or the GUI **Configuration Options** window to determine what the maximum complexities are (See Section 4.5.6 - "Setting Report Threshold Values" on page 95.).



```
Exception Report for Procedures in /home/16/stw.2.6/product/demos/advisor/metric.demo/xcalc.c
-----
Exception(s) for procedure: parse_double
** With a comment percentage of 0,000
   this procedure does not meet the standard of 0,600
Exception(s) for procedure: open_the_display
** With a comment percentage of 0,043
   this procedure does not meet the standard of 0,600
Exception(s) for Procedure: main
** At 413 lines, this procedure is larger
   than the standard of 60 lines
** At 202 semicolons, this procedure is larger
   than the standard of 50 semicolons
** With a Cyclomatic Complexity of 83 this
   procedure exceeds the standard of 10
** With an Extended Cyclomatic Complexity of 113
   this procedure exceeds the standard of 15
** With an Average Maximum Span of Reference of 45
   this procedure exceeds the standard of 10
** With a Volume of 18664
   this procedure exceeds the standard of 3200
** With a comment percentage of 0,039
   this procedure does not meet the standard of 0,600
Exception(s) for procedure: Syntax
** With a comment percentage of 0,000
```

FIGURE 70 Exception Report for "C"

The Error Report

The **Error** report will only be created if there was an error encountered during processing and analysis. Possible errors are listed below:

- **message:** <<< Unable to open input file <filename> >>>
- **cause:** The file specified to be analyzed could not be found or opened successfully
- **remedy:** Check the spelling of the filename. Make sure the extension (if any) is correct. Make sure the path name is correct.

- **message:** <<< Unable to open reserved word file <filename> >>>
- **cause:** The reserved word file could not be found.
- **remedy:** Check the spelling of the filename. Make sure the extension (if any) is correct. Make sure the pathname is correct.

- **message:** <<< Number of operators exceeds limit >>>
- **cause:** When modifying the reserved word file to include additional words, you exceeded the default limit, or the reserved word limit is too small.
- **remedy:** Add or modify the configuration file parameter RESCT.

- **message:**
<<< Unable to open nonexecutable word file <filename> >>>
- **cause:** The reserved word file could not be found.
- **remedy:** Check the spelling of the filename. Make sure the extension (if any) is correct. Make sure the path name is correct.

- **message:**
<<< Number of nonexecutable words exceeds limit >>>
- **cause:** The number of nonexecutable words in the nonexecutable word file exceeds the default limit or the limit is too small.
- **remedy:** Add or modify the configuration file parameter NON-EXCT to accommodate the number of entries. Typedefs are stored internally as nonexecutable words and therefore require a larger NONEXCT.

- **message:** <<< Heap overflow - reorganize source program >>>
- **cause:** Your file to be analyzed contains too many unique operands or operators. You have run out of memory to continue processing.
- **remedy:** Remove RAM resident routines or do not analyze as many source files at one time.

- **message:**<<< Improper pathname or no files found >>>
- **cause:** The file names entered could not be opened.
- **remedy:** Check the spelling of the file names and make sure the directory path is correct.

- **message:** <<< Unable to create reserved word dynamic array >>>
- **cause:** Not enough memory is available to load the reserved word lists.
- **remedy:** Make sure that the entry RESCT is not overly large.

- **message:** <<< Unexpected end of file - analysis aborting >>>
- **cause:** The end of file has been reached before the end of the procedure.
- **remedy:** Make sure your source file compiles. If it does, contact **Software Research**.

- **message:** <<<Unable to open include file <filename> >>>
- **cause:** The include file could not be found given the specification in the source code.
- **remedy:** Make sure the include file exists. If it exists in a different directory than specified in the source code, make INCLUDE_DIR entries in the configuration file.

A.2 Counting Rules

This chapter describes the rules *METRIC*TM uses in analyzing “C” programs. It addresses issues associated with the reserved word and nonexecutable word files used in the analysis, miscellaneous counting rules of operators and operands, cyclomatic complexity, executable semicolons, variable name length, span of reference, and line of code counting rules. Description of the Halstead Software Science entries can be found in the correct chapter (See CHAPTER 4 - System Operation” on page 71.).

Reserved Word/Nonexecutable Word File

The reserved word file, `cresword.tab`, contains a list of “C” operators. To see what this file is comprised of, simply print out the file. The nonexecutable word file, `cnonexe.tab`, contains a list of nonexecutable words for ANSI “C”. A few of the items in `cresword.tab` are not part of standard “C” and are explained below.

- In “C”, a parenthesis has three uses: it can change the default ordering of arithmetic operations, it follows a procedure call, and it follows a control statement. To distinguish these uses, three different parentheses have been defined in the file: “(” indicates an arithmetic paren, “(c” indicates a paren following a control statement, and “(p” indicates a paren following a procedure call. Each of these are a different use of a parenthesis and, therefore, each is a different operator.
- In “C”, the asterisk, *, has two uses: as a multiplication sign and as a pointer. To distinguish these uses, two asterisks have been defined: “*” indicates multiplication, and “*p” indicates a pointer. Since these are different meanings, each is counted as a different operator.
- In “C”, the ampersand, &, has two uses: as a unary AND operator and as an address operator. To distinguish these uses, two ampersands have been defined: “&” indicating unary AND, and “&p” indicating the address operator. Since these are different meanings, each is counted as a different operator.
- Certain items in the list are not counted. These are the items that must be paired with another and consist of:), },], while when associated with do and “:” when associated with “?”. Also not counted are the single quote, ', and double quote, ". These signals the start of a string and is counted as part of the string.

Any statement preceded by one of the words in the nonexecutable word file is considered nonexecutable and, hence, ignored with the following exception:

- A nonexecutable word can precede a procedure name and still allow the procedure to be recognized. For example:

```
long roundit(x)
double x;
{
.
.
.
}
```

will see that `roundit(x)` is a procedure, skip over `double x`, and begin analyzing the code contained within `{...}`.

- When a nonexecutable word appears after a parenthesis, then a cast operation is being performed. In this case, the entire string between parentheses is considered to be an operand. For example in the following,

```
y = (float) x;
return((struct tnode *)t);
```

`float` and `struct tnode *` are considered to be operands. The parenthesis, in this case is counted as an arithmetic parenthesis.

Miscellaneous Operator/Operand Rules

The following are some miscellaneous rules used when counting operators and operands

- Procedure calls are counted as operators.
- All operands are considered to be global in the Summary report. Therefore, local variables of the same name defined in different procedures are counted as multiple occurrences of the same variable for purposes of the Summary report.
- C is case sensitive. Therefore, `x` and `X` are two different operands.
- When using `gotos`, the `goto` label is treated as an operator and the occurrence of the label in the code is an operand. The colon following the label is an operator.

Cyclomatic Complexity

The following control structures increment the cyclomatic complexity count: `if`, `while` (unassociated with `do`), `do`, `for`, `?:`, and `case`. Occurrences of `else` do not increment the count. The extended cyclomatic complexity is incremented for each of these operators plus `&&` and `||`.

Span of Reference

The span of reference counts the maximum number of lines between references to each variable in a procedure (either use or assignment). The average of all the maximum references is then computed. This average for each procedure is listed in the **Complexity** report.

Executable Semi-colons

Executable semicolons begin counting with the first executable line of code. Hence, all declarations are not included. `for` loops will contribute two executable semicolons.

Average Variable Name Length

The average variable name length provides a 'readability index' based on variable naming conventions. The value can be calculated in two ways, based on the configuration file entry `UNIQUE_VARIABLES`. If `UNIQUE_VARIABLES` is set to 1, then the length of all unique variable names (used at least once) is divided by the number of unique variables. If `UNIQUE_VARIABLES` is set to 0, then a weighted calculation is done. With this method, for all variables, the length of the variable name is multiplied by the number of times the variable is used and this sum is divided by the total count of variable usage. Note that this is not the same as dividing by n^2 (in the first case) or N^2 (in the second case). n^2 and N^2 include numeric constants and literal strings whereas these calculations are concerned only with variables.

Lines of Code

Lines of code for a procedure include all lines of code from the procedure heading to the last statement of the procedure, `}`. This includes comments, blanks and continued lines. In addition, if `OUTSIDE_COMMENTS` are being counted, then the line of code count for the procedure will be incremented by the number of outside comments encountered.

Comments

The number of comments in a procedure is the count of all comments on a line by themselves encountered within the body of the procedure. If a single comment spans multiple lines, the number of lines that it spans is added to the comment count. If `INLINE_COMMENTS` is 1, then comments on the same line as an executable statement will also increment the count. If `OUTSIDE_COMMENTS` is 1, then comments outside of the scope of any procedure immediately before the procedure are added to the count.

A.3 Creating a Shell Script File

If there are a group of programs to be analyzed, where a separate report is to be produced for each program, it may be convenient to analyze them all at once through the use of a shell script. To create a shell script, create a file, *filename*, and enter *METRIC*TM command lines as though they were being entered at the system prompt. Make sure that for the filename you choose, there does not currently exist a command of the same name. You may use the command line mode with or without display to analyze the files.

For example, suppose we create a shell script `shtest` that contains the following:

```
#!/bin/sh -f
cmetric cmetric
cmetric filematch tool
```

“`cmetric`” must appear as the first word on the line since it is the program to be executed. The source file(s) to be analyzed and any command line parameters follow. Be sure to include pathnames for those files to be analyzed that do not reside in the current directory.

You can run the shell script file in either of the following ways:

- At the system prompt, give the filename as an argument to the `sh` command. For example:

```
sh shtest
```

- Use the `chmod` command to make the file executable and then use the name of the shell script like any other command. For example:

```
chmod 755 shtest
```

Each source file in the shell script will be processed in turn. If a file is not found in the shell script, processing will continue with the next file. An error file will be created for each file in which an error occurred.

A.3.1 Conditional Compilation Directives

*METRIC*TM for “C” will support the following conditional compilation directives:

- #define
- #undef
- #ifdef
- #else
- #elif
- #if sym
- #if defined(sym)
- #ifndef
- #endif

A symbol can be defined using a combination of either of the following methods:

- On the command line, specify any number of `-csym` options where *sym* is the symbol to be defined. If every symbol is to be defined, specify `-call`. However, if `-call` is used, the `#undef sym` will not work and all symbols will remain defined through completion of analysis.
- In the source files being analyzed, if a statement such as

```
#define sym
```

appears, then *sym* will be defined through the completion of analysis of all the files (or until a `#undef sym` statement occurs).

If there are a large number of symbols to be defined whose definitions do not occur in the source files to be analyzed, it may not be feasible to list them all on the command line. In this case you can list them in a separate file. This file must be specified as the first file to be analyzed so that all the definitions can be picked up. For example, suppose the file, `hdrfile.h`, consists of the following:

```
#define VAX
#define unix
#define DEMO
```

then `cmetric` should be invoked with the command:

```
cmetric hdrfile.h *.c
```

Note that the file `hdrfile.h` does not contain any executable code. Therefore, an error file, `hdrfile.err`, will be created with the message:

```
File hdrfile.h does not contain any executable code
```

This is just a warning message and does not affect the analysis process.

Some processing notes when using conditional compilation directives:

- If the configuration file parameter `CONDCOMPILER` is 1 and no `-csym` options are specified on the command line and no `#define` statements are found, then all `#else` portions of `#ifdef` (or `#if`), all `#ifndef`, `#if !sym`, and `#if !defined(sym)` portions of the code will be analyzed.
- If the configuration file parameter `CONDCOMPILER` is 0 and no `-csym` options on the command line appear, then none of the `#define` statements will be picked up and all compiler directives will be ignored and every line of code will be analyzed.
- If the configuration file parameter `CONDCOMPILER` is 0 and one or more `-csym` options are specified, then the configuration file parameter is overridden. Also, any `#define` statements in the code will be picked up.

A.3.2 Comments About METRIC

If you analyze your programs before they are completed and you have empty procedures such as the following:

```
procA()  
{  
}
```

You will get spurious results for that procedure in the **Complexity** report. The above example does not contain any operands and many of the metrics are not defined for programs without operators or operands.

Throughout this manual, it was said the *METRICTM* can be used to analyze any compilable “C” program. There is an exception to this. Some programmers find it convenient to write “C” code as if it were Pascal by doing the following:

```
#define BEGIN {  
#define END }  
  
.  
.  
.  
main()  
BEGIN  
.  
.  
.  
END
```

Even though the above code compiles, *METRICTM* cannot analyze this type of code correctly. *METRICTM* looks for “{” to determine when executable code starts and “}” to signal the end. Analyzing code written in this manner will cause much, if not all, of the code to be treated as nonexecutable, and thus ignored. Calculations will be made but they will not be correct.

If any of your code is written in this manner, you can run it through a preprocessor commonly included with most C compilers that expands all #define statements and then run *METRICTM* on the expanded code. Redefinitions of other special words (e.g.: if) can also cause inaccuracies in the reports. For this reason, it is highly recommended that code to be analyzed be run through a preprocessor first if it contains any of this type of #define statements.

“C++” Notes

*METRIC*TM for “C++” consists of several files, the following of which will be explained in this chapter:

- `cppmetric`
- `cppresword.tab`
- `cppnonexe.tab`

cppmetric

This file is the actual analyzer. It uses the remaining files in determining if a word is an *operator* or an *operand* or if it is a nonexecutable word. A complete description on how to use `cppmetric` is given in the `COMMAND LINE ACTIVATION` chapter.

cppresword.tab

This file contains a sorted list of all operators and reserved words for standard “C++”. See Section B.2 for a more detailed explanation of the contents of this file .

If you are using an extension to standard “C++” and need to add/remove items to/from the list, you may do so in one of two ways:

- Edit the file, `cppresword.tab`, and add those entries not appearing in the list and remove the entries that should not be in the list. If the number of entries exceeds 100, you must make modifications to the configuration file `.uxmetriccfg`. Refer to the correct section for details on how to increase the number of allowable entries (See Section 7.5 - “Configuration File Processing” on page 156.).
- Create a new reserved word file containing the operators and executable reserved words for your version of “C++”. If this

option is chosen, you must use the configuration file indicating that a different reserved word file is being used. See the correct section for details on how to do this (See Section 7.5 - “Configuration File Processing” on page 156.).

Before you modify the list, be sure to read Section the correct section and understand the usage of some of the items (See Section 5.1.2 - “Software Development” on page 116.).

cppnonexe.tab

This file contains a sorted list of all reserved words that are nonexecutable as defined in ANSI standard “C++”. For example, `int`, `char`, and `struct`. For a more complete description of the contents of this file, refer to Section B.2. When you analyze a program using *METRICTM*, statements beginning with one of these words are skipped, and therefore do not affect the operator or operand count.

If you are using an extension to standard “C++” and need to add/remove items to/from the list you may do so in one of the following ways:

- Edit the file `cppnonexe.tab` and add those words not appearing in the list or remove those words from the list that you wish to be counted.
- Create a new nonexecutable word file containing the words not to be counted. If this option is chosen, you must use the configuration file, indicating that a different nonexecutable word file is being used. See Section B.2 for information on how to do this.

Note that if there are more than 25 entries in the nonexecutable word file, you must make a modification to the file `.uxmetriccfg`. See Section B.2 for details on how to do this.

B.1 Description of the Reports

This section describes the reports created by *METRIC*TM: the **Complexity** report, the **Summary** report, the **Exception** report, the **Error** report, the **C++ Class Report**, the **Class Summary** report, the **Class Hierarchy** report, and the **Class Exception** report.

The Complexity Report

The **Complexity** report by Procedure, *filename.rpt*, includes the following fields:

- procedure name
- in-line functions (FT)
- unique operators (N1)
- unique operands (N2)
- total operators (N1)
- total operands (N2)
- length (N)
- predicted length (N[^])
- purity ratio <em estimated length divided by length (P/R)
- volume (V)
- effort (E)
- cyclomatic complexity (VG1)
- extended cyclomatic complexity (VG2)
- lines of code (LOC)
- blank lines of code (BLK)
- comment lines of code (CMT)
- executable semi-colons (< ; >)
- average maximum span of reference of variables (SP)
- variable name length (VL)

Note: All fields, except FT, are discussed in a prior chapter (See CHAPTER 5 - Helpful Hints” on page 115.). FT identifies in-line functions. You can tell if a function is a an in-line function, if a function is listed as *function | classname* and PV is listed in the FT column.

APPENDIX B:

Complexity Report by Function for: /home/16/stw.2.6/product/demos/advisor/metric.demo/ex.cpp

Function	FT	n1	n2	N1	N2	N	N'	P/R	V	E	VG1	VG2	L
List::put_elem()	1	8	1	12	13	24	1.85		41	31	1	1	
List::put_elem()	2	8	2	12	14	26	1.86		47	70	1	1	
List::get_elem()	1	8	1	12	13	24	1.85		41	31	1	1	
List::print()	1	7	1	10	11	20	1.79		33	24	1	1	
Stack::push()	3	8	3	11	14	29	2.05		48	100	1	1	
Stack::pop()	2	6	2	9	11	18	1.59		33	50	1	1	
Stack::print()	2	9	2	15	17	31	1.80		59	98	1	1	
main()	2	7	2	11	13	22	1.67		41	65	1	1	
ATest()	3	9	4	15	19	33	1.75		68	170	1	1	

FIGURE 71 Complexity Report for "C++"

The Summary Report

The **Summary** report consists of **Complexity** report fields, with the exception of average maximum span of reference of variables (SP) and average variable name length (VL). In addition it includes the values:

- Estimated errors (B^{\wedge})
- Estimated development time (T^{\wedge})
- Average cyclomatic complexity
- Average extended cyclomatic complexity
- Number of procedures

Please refer to the correct chapter for an explanation of the **Summary** report's fields (See CHAPTER 5 - Helpful Hints" on page 115.).

The command line also allows you create an **Intermediate Summary** report whenever there is a change in directory ($-bn$). This report may be helpful if different subsystems of an application being analyzed are placed in different directories and you want a **Summary** report for each subsystem.

You can also create an **Intermediate Summary** report whenever a change in the first n characters of the filename occurs ($-bn$). This report may be helpful if the files comprising the different subsystems of an application all have the same prefix and you want a **Summary** report for each subsystem.

Please see the correct chapter for command line activation (See CHAPTER 7 - Command Line Activation" on page 145.).

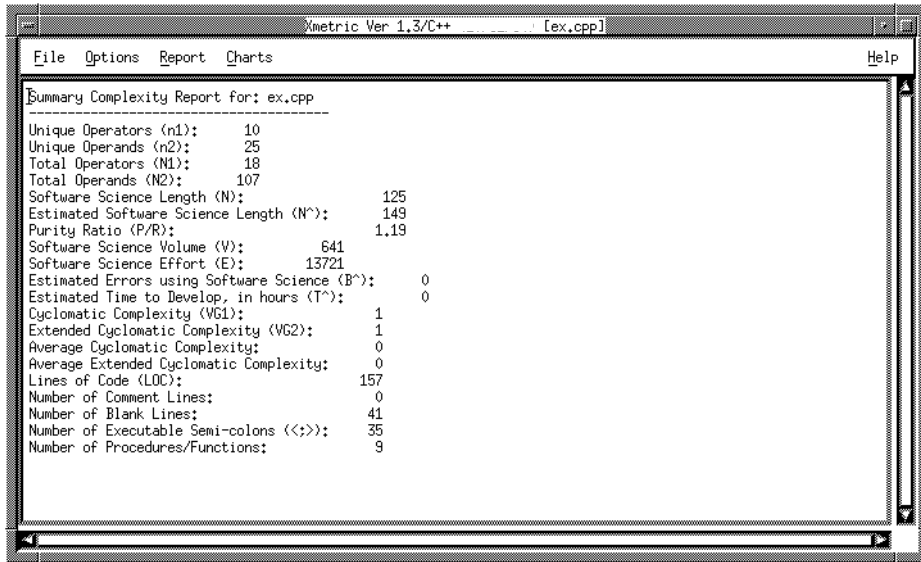
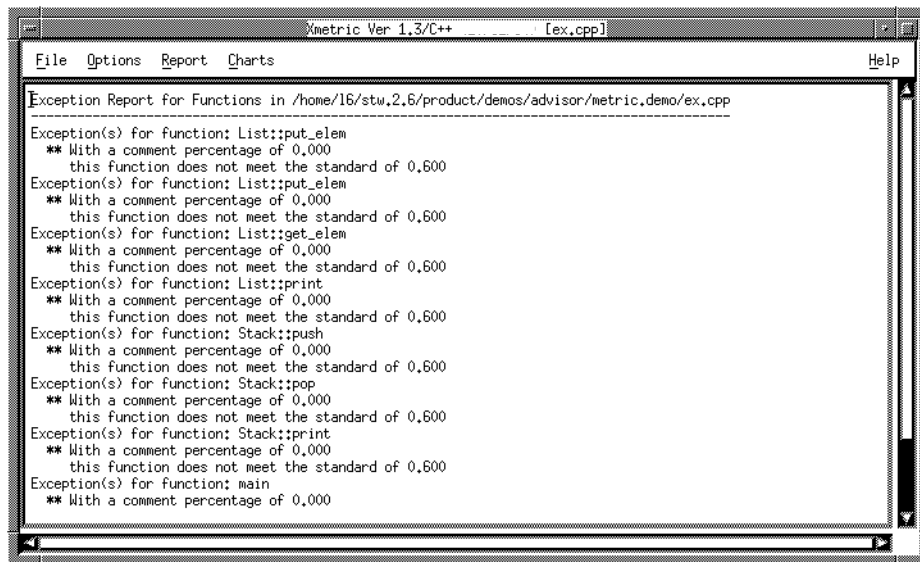


FIGURE 72 Summary Report for "C++"

The Exception Report

The **Exception** report is the next report created by *METRIC*TM. Each procedure in the source files which exceeds a set of predefined complexity maximums is included in this report. This report uses either the defined standards specified in the configuration file or the GUI's **Configuration Options** window (See Section 4.5.6 - "Setting Report Threshold Values" on page 95.) to determine what the maximum complexities are.



The screenshot shows a window titled "Xmetric Ver 1.3/C++" with a menu bar containing "File", "Options", "Report", "Charts", and "Help". The main text area displays the following report content:

```
-----
Exception Report for Functions in /home/16/stw.2.6/product/demos/advisor/metric,demo/ex.cpp
-----
Exception(s) for function: List::put_elem
  ** With a comment percentage of 0,000
  this function does not meet the standard of 0,600
Exception(s) for function: List::put_elem
  ** With a comment percentage of 0,000
  this function does not meet the standard of 0,600
Exception(s) for function: List::get_elem
  ** With a comment percentage of 0,000
  this function does not meet the standard of 0,600
Exception(s) for function: List::print
  ** With a comment percentage of 0,000
  this function does not meet the standard of 0,600
Exception(s) for function: Stack::push
  ** With a comment percentage of 0,000
  this function does not meet the standard of 0,600
Exception(s) for function: Stack::pop
  ** With a comment percentage of 0,000
  this function does not meet the standard of 0,600
Exception(s) for function: Stack::print
  ** With a comment percentage of 0,000
  this function does not meet the standard of 0,600
Exception(s) for function: main
  ** With a comment percentage of 0,000
```

FIGURE 73 Exception Report for "C++"

The Error Report

The **Error** report will only be created if there was an error encountered during processing and analysis. Possible errors are listed below:

- **message:** <<< Unable to open input file <filename> >>>
- **cause:** The file specified to be analyzed could not be found or opened successfully.
- **remedy:** Check the spelling of the filename. Make sure the extension (if any) is correct. Make sure the path name is correct.

- **message:** <<< Unable to open reserved word file <filename> >>>
- **cause:** The reserved word file could not be found.
- **remedy:** Check the spelling of the filename. Make sure the extension (if any) is correct. Make sure the pathname is correct.

- **message:** <<< Number of operators exceeds limit >>>
- **cause:** When modifying the reserved word file to include additional words, you exceeded the default limit, or the reserved word limit is too small.
- **remedy:** Add or modify the configuration file parameter RESCT.

- **message:** <<< Unable to open nonexecutable word file <filename> >>>
- **cause:** The reserved word file could not be found.
- **remedy:** Check the spelling of the filename. Make sure the extension (if any) is correct. Make sure the path name is correct.

- **message:** <<< Number of nonexecutable words exceeds limit >>>
- **cause:** The number of nonexecutable words in the nonexecutable word file exceeds the default limit or the limit is too small.
- **remedy:** Add or modify the configuration file parameter NONEXCT to accommodate the number of entries. Typedefs are stored internally as nonexecutable words and therefore require a larger NONEXCT.

- **message:** <<< Heap overflow - reorganize source program >>>

- **cause:** Your file to be analyzed contains too many unique operands or operators. You have run out of memory to continue processing.
- **remedy:** Remove RAM resident routines or do not analyze as many source files at one time.

- **message:** <<< Improper pathname or no files found >>>
- **cause:** The file names entered could not be opened.
- **remedy:** Check the spelling of the file names and make sure the directory path is correct.

- **message:** <<< Unable to create reserved word dynamic array >>>
- **cause:** Not enough memory is available to load the reserved word lists.
- **remedy:** Make sure that the entry RESCT is not overly large.

- **message:** <<< Unexpected end of file - analysis aborting >>>
- **cause:** The end of file has been reached before the end of the procedure.
- **remedy:** Make sure your source file compiles. If it does, contact **Software Research**.

- **message:** Unable to open include file <filename>
- **cause:** The include file could not be found given the specification in the source code.
- **remedy:** Make sure the include file exists. If it exists in a different directory than specified in the source code, make INCLUDE_DIR entries in the configuration file.

C++ Class Report

The **C++ Class Report**, *filename.cls*, specifies the number of members in each class, the type of members in each class, the number of in-line and virtual functions in each class, as well as how many friend functions and friend classes are associated with each class.

It contains the following fields:

- Class being analyzed.
- Its base class (`Baseclass`).
- Public functions and variable members (`Public Var/Fct`).
- Private functions and variable members (`Private Var/Fct`).
- Private functions and variable members (`Private Var/Fct`).
- Total functions and variable members (`Total Membs`).
- Count of In-Line Function Members (`Inline`).
- Count of Virtual Function Members (`Virt`).
- Count of Friend Classes and functions (`Friend Cls/Fct`).

Class Summary

“C++” also produces a summary listing the averages of the different members listed in the **C++ Class Report**, a part of *filename.cls*.

It contains the following fields:

- Number of Classes.
- Average Number of Members per Class.
- Average Number of Private Members per Class.
- Average Number of Public Members per Class.
- Average Number of Protected Members per Class.
- Average Number of In-line Members per Class.
- Number of Explicit In-Line Functions.
- Average Number of Friend Functions per Class.
- Average Number of Friend Classes per Class.

Class Hierarchy

The **Class Hierarchy** report, *filename.cht*, lists all the base classes and their derived classes for all files analyzed. It can help you easily identify what classes are derived from other classes.

Class Exception Report

The **Class Exception** report, *filename.cex*, contains a listing of all the classes which exceed predefined standards. This report is very similar to the **Exception** report. These standards or defaults are listed in the configuration file. (See Section 7.5 - "Configuration File Processing" on page 156.)

B.2 Counting Rules

This chapter describes the rules *METRICTM* uses in analyzing “C++” programs. It addresses issues associated with the reserved word and nonexecutable word files used in the analysis, miscellaneous counting rules of operators and operands, cyclomatic complexity, executable semicolons, variable name length, span of reference, and line of code counting rules. Description of the Halstead Software Science entries can be found in the another section (See Section 5.1.2 - “Software Development” on page 116.).

Reserved Word/Nonexecutable Word Files

The reserved word file, `cppresword.tab`, contains a list of “C++” operators. To see what this file is comprised of, simply print out the file. The nonexecutable word file, `cppnonexe.tab`, contains a list of nonexecutable words for ANSI C++. A few of the items in `cppresword.tab` are not part of standard “C++” and are explained below.

- In “C++”, a parenthesis has three uses: it can change the default ordering of arithmetic operations, it follows a procedure call, and it follows a control statement. To distinguish these uses, three different parentheses have been defined in the file: “(” indicates an arithmetic paren, “(c” indicates a paren following a control statement, and “(p” indicates a paren following a procedure call. Each of these are a different use of a parenthesis and, therefore, each is a different operator.
- In “C++”, the asterisk, *, has two uses: as a multiplication sign and as a pointer. To distinguish these uses, two asterisks have been defined: “*” indicates multiplication, and “*p” indicates a pointer. Since these are different meanings, each is counted as a different operator.
- In “C++”, the ampersand, &, has two uses: as a unary AND operator and as an address operator. To distinguish these uses, two ampersands have been defined: “&” indicating unary AND, and “&p” indicating the address operator. Since these are different meanings, each is counted as a different operator.
- Certain items in the list are not counted. These are the items that must be paired with another and consist of:), },], while when associated with do and “:” when associated with “?”. Also not counted are the single quote, ', and double quote, ". These signals the start of a string and is counted as part of the string.

Any statement preceded by one of the words in the nonexecutable word file is considered nonexecutable and, hence, ignored with the following exception:

- A nonexecutable word can precede a procedure name and still allow the procedure to be recognized. For example:

```
long roundit(x)
    double x;
    {
        .
        .
        .
    }
```

will see that `roundit(x)` is a procedure, skip over `double x`, and begin analyzing the code contained within `{...}`.

- When a nonexecutable word appears after a parenthesis, then a cast operation is being performed. In this case, the entire string between parentheses is considered to be an operand. For example in the following,

```
y = (float) x;
return((struct tnode *)t);
```

`float` and `struct tnode *` are considered to be operands. The parenthesis, in this case is counted as an arithmetic parenthesis.

Miscellaneous Operator/Operand Rules

The following are some miscellaneous rules used when counting operators and operands:

- Procedure calls are counted as operators.
- All operands are considered to be global in the **Summary** report. Therefore, local variables of the same name defined in different procedures are counted as multiple occurrences of the same variable for purposes of the **Summary** report.
- C is case sensitive. Therefore, `x` and `X` are two different operands.
- When using `gotos`, the `goto` label is treated as an operator and the occurrence of the label in the code is an operand. The colon following the label is an operator.

Cyclomatic Complexity

The following control structures increment the cyclomatic complexity count: `if`, `while` (unassociated with `do`), `do`, `for`, `?:`, and `case`. Occurrences of `else` do not increment the count. The extended cyclomatic complexity is incremented for each of these operators plus `&&` and `||`.

Span of Reference

The span of reference counts the maximum number of lines between references to each variable in a procedure (either use or assignment). The average of all the maximum references is then computed. This average for each procedure is listed in the **Complexity** report.

Executable Semi-colons

Executable semicolons begin counting with the first executable line of code. Hence, not all declarations are included. `for` loops will contribute two executable semicolons.

Average Variable Name Length

The average variable name length provides a 'readability index' based on variable naming conventions. The value can be calculated in two ways, based on the configuration file entry `UNIQUE_VARIABLES`. If `UNIQUE_VARIABLES` is set to 1, then the length of all unique variable names (used at least once) is divided by the number of unique variables. If `UNIQUE_VARIABLES` is set to 0, then a weighted calculation is done. With this method, for all variables, the length of the variable name is multiplied by the number of times the variable is used and this sum is divided by the total count of variable usage. Note that this is not the same as dividing by n^2 (in the first case) or N^2 (in the second case). n^2 and N^2 include numeric constants and literal strings whereas these calculations are concerned only with variables.

Lines of Code

Lines of code for a procedure include all lines of code from the procedure heading to the last statement of the procedure, `}`. This includes comments, blanks and continued lines. In addition, if `OUTSIDE_COMMENTS` are being counted, then the line of code count for the procedure will be incremented by the number of outside comments encountered.

Comments

The number of comments in a procedure is the count of all comments on a line by themselves encountered within the body of the procedure. If a single comment spans multiple lines, the number of lines that it spans is added to the comment count. If `INLINE_COMMENTS` is 1, then comments on the same line as an executable statement will also increment the count. If `OUTSIDE_COMMENTS` is 1, then comments outside of the scope of any procedure immediately before the procedure are added to the count.

B.3 Creating a Shell Script File

If there are a group of programs to be analyzed, where a separate report is to be produced for each program, it may be convenient to analyze them all at once through the use of a shell script. To create a shell script, create a file, *filename*, and enter *METRICTM* command lines as though they were being entered at the system prompt. Make sure that for the filename you choose, there does not currently exist a command of the same name. You may use the command line mode with or without display to analyze the files.

For example, suppose we create a shell script `shtest` that contains the following:

```
#!/bin/sh -f^M
cppmetric cppmetric
cppmetric filematch tool
```

“cppmetric” must appear as the first word on the line since it is the program to be executed. The source file(s) to be analyzed and any command line parameters follow. Be sure to include pathnames for those files to be analyzed that do not reside in the current directory.

You can run the shell script file in either of the following ways:

- At the system prompt, give the filename as an argument to the `sh` command. For example:

```
sh shtest
```

- Use the `chmod` command to make the file executable and then use the name of the shell script like any other command. For example:

```
chmod 755 shtest
shtest
```

- Each source file in the shell script will be processed in turn. If a file is not found in the shell script, processing will continue with the next file. An error file will be created for each file in which an error occurred.

B.3.1 Conditional Compilation Directives

METRICTM for “C++” will support the following conditional compilation directives:

- #define
- #undef
- #ifdef
- #else
- #elif
- #if sym
- #if defined(sym)
- #ifndef
- #endif

A symbol can be defined using a combination of either of the following methods:

- On the command line, specify any number of `-csym` options where `sym` is the symbol to be defined. If every symbol is to be defined, specify `-call`. However, if `-call` is used, the `#undef sym` will not work and all symbols will remain defined through completion of analysis.
- In the source files being analyzed, if a statement such as

```
#define sym
```

appears, then `sym` will be defined through the completion of analysis of all the files (or until a `#undef sym` statement occurs).

If there are a large number of symbols to be defined whose definitions do not occur in the source files to be analyzed, it may not be feasible to list them all on the command line. In this case you can list them in a separate file. This file must be specified as the first file to be analyzed so that all the definitions can be picked up. For example, suppose the file, `hdrfile.h`, consists of the following:

```
#define VAX
#define unix
#define DEMO
```

then `cppmetric` should be invoked with the command:

```
cppmetric hdrfile.h *.c
```

Note that the file `hdrfile.h` does not contain any executable code. Therefore, an error file, `hdrfile.err`, will be created with the message:

File `hdrfile.h` does not contain any executable code

This is just a warning message and does not affect the analysis process.

The following are some processing notes when using conditional compilation directives:

- If the configuration file parameter `CONDCOMPILER` is 1 and no `-csym` options are specified on the command line and no `#define` statements are found, then all `#else` portions of `#ifdef` (or `#if`), all `#ifndef`, `#if !sym`, and `#if !defined(sym)` portions of the code will be analyzed.
- If the configuration file parameter `CONDCOMPILER` is 0 and no `-csym` options on the command line appear, then none of the `#define` statements will be picked up and all compiler directives will be ignored and every line of code will be analyzed.
- If the configuration file parameter `CONDCOMPILER` is 0 and one or more `-csym` options are specified, then the configuration file parameter is overridden. Also, any `#define` statements in the code will be picked up.

B.3.2 Comments About METRIC

If you analyze your programs before they are completed and you have empty procedures such as the following:

```
procA()  
{  
}
```

You will get spurious results for that procedure in the **Complexity** report. The above example does not contain any operands and many of the metrics are not defined for programs without operators or operands.

If your code contains `typedef`, `class`, `struct`, or `enum` definitions in the header files, make sure these files are analyzed first. `cppmetric` needs to see these definitions so that it can correctly distinguish when executable code begins. Otherwise, the operator and operand count will be inaccurate, and the analysis may not proceed correctly. If you get the error message:

```
Number of nonexecutable words exceed limit
```

when analyzing your files, you have more than 40 `typedefs`, `classes`, `structs`, and/or `rnums`. Simply modify the configuration file parameter `NONEXCT` to accommodate the number of declarations. `cppmetric` allows 40 words above what is specified in the `NONEXCT` parameter for these declarations.

Throughout this manual, it was said the *METRICTM* can be used to analyze any compilable “C++” program. There is an exception to this. Some programmers find it convenient to write “C++” code as if it were Pascal by doing the following:

```
#define BEGIN {  
#define END }  
  
.  
.  
.  
main()  
BEGIN  
  
.  
.  
.  
END
```

Even though the above code compiles, *METRICTM* cannot analyze this type of code correctly. *METRICTM* looks for “{” to determine when executable code starts and “}” to signal the end. Analyzing code written in this manner will cause much, if not all, of the code to be treated as nonex-

ecutable, and thus ignored. Calculations will be made but they will not be correct.

If any of your code is written in this manner, you can run it through a preprocessor commonly included with most “C++” compilers that expands all `#define` statements and then run *METRICTM* on the expanded code. Redefinitions of other special words (e.g.: `if`) can also cause inaccuracies in the reports. For this reason, it is highly recommended that code to be analyzed be run through a preprocessor first if it contains any of this type of `#define` statements.

Ada Notes

*METRIC*TM for Ada consists of several files, two of which will be explained in this chapter:

- *adametric*
- *adaresword.tab*.

adametric

This file is the actual analyzer. It uses the remaining files in determining if a word is an *operator* or an *operand* or if it is a nonexecutable word. A complete description on how to use **adametric** is given in the **COMMAND LINE ACTIVATION** chapter (See CHAPTER 7 - "Command Line Activation" on page 145.).

adaresword.tab

This file contains a sorted list of all operators and reserved words for standard Ada. For a more detailed explanation of the contents of this file, see Section C.2.

If you are using an extension to standard Ada and need to add/remove items to/from the list, you may do so in one of two ways:

- Edit the file, *adaresword.tab*, adding those entries not appearing in the list and removing the entries that should not be in the list.
- Create a new reserved word file containing the operators and executable reserved words for your version of Ada. If this option is chosen, you must use the configuration file indicating that a different reserved word file is being used. See the correct section for details on how to do this (See Section 7.5 - "Configuration File Processing" on page 156.).

C.1 Description of the Reports

This section describes the reports created by *METRICTM*: the **Complexity** report, the **Summary** report, the **Exception** report, the **Error** report, the **Generic** report, the **Package Exception** report, and the **Package Intermediates** report.

The Complexity Report

The **Complexity** report by Procedure, *filename.rpt*, includes the following fields:

- Procedure Name
- Unique Operators (n1)
- Unique Operands (n2)
- Total Operators (N1)
- Total Operands (N2)
- Length (N)
- Predicted Length (N[^])
- Purity Ratio – estimated length divided by length (P/R)
- Volume (V)
- Effort (E)
- Cyclomatic Complexity (VG1)
- Extended Cyclomatic Complexity (VG2)
- Lines of Code (LOC)
- Number of Comment Lines (CMT)
- Number of Blank Lines (BLK)
- Number of Executable Semi-Colons (< i >)
- Average Maximum Span of Reference of Variables (SP)
- Variable Name Length (VL)

Please refer to the correct chapter for an in-depth discussion of the various fields (See CHAPTER 3 - System Introduction” on page 39).

Note in the **Complexity** report, a '^' in front of the procedure name indicates that it is really a task name. An '*' in front of the procedure name indicates that it is a package.

If a file analyzed does not contain any executable code, it will be listed in with the name *emptyfile*. Also, all values will be zero except lines of code (LOC) which indicates the total lines of code in the file.

Procedure	n1	n2	N1	N2	N	N^	P/R	V	E	VG1	VG2	LOC
PUSH	5	4	8	6	14	20	1,40	44	166	1	1	8
POP	5	4	8	6	14	20	1,40	44	166	1	1	8
*G2STACK	1	1	1	1	2	1	0,50	1	1	1	1	16
*INSTANCE	1	1	1	1	2	1	0,50	1	1	1	1	16
CREATE_A_FRAME	6	4	8	4	12	24	1,96	40	120	1	1	8
CREATE_A_FRAME	6	4	8	4	12	24	1,96	40	120	1	1	8
WRITE_INTO_A_FRAME	6	4	8	4	12	24	1,96	40	120	1	1	8
CLEAR_A_FRAME	6	3	7	3	10	20	2,03	32	95	1	1	8
^SCREEN_MANAGER	17	12	52	26	78	113	1,44	379	6978	1	5	27
*FRAME	1	1	1	1	2	1	0,50	1	1	1	1	62

FIGURE 74 Complexity Report for Ada

The Summary Report

The **Summary** report, *filename.rpt* consists of **Complexity** report fields, with the exception of average maximum span of reference of variables and average variable name length. In addition it includes the values:

- Estimated errors (B^{\wedge})
- Estimated development time (T^{\wedge})
- Average cyclomatic complexity
- Average extended cyclomatic complexity
- Number of procedures/functions
- Number of nonexecutable semi-colons
- Number of Tasks
- Number of Packages

Please refer to the correct chapter for an explanation of the **Summary** report's fields (See CHAPTER 7 - Command Line Activation" on page 145.).

The command line also allows you create an **Intermediate Summary** report whenever there is a change in directory ($-bn$). This report may be helpful if different subsystems of an application being analyzed are

placed in different directories and you want a **Summary** report for each subsystem.

You can also create an **Intermediate Summary** report whenever a change in the first *n* characters of the filename occurs (*-bn*). This report may be helpful if the files comprising the different subsystems of an application all have the same prefix and you want a `\f6Summary\f1` report for each subsystem.

See the correct chapter for command line activation (See CHAPTER 7 - Command Line Activation” on page 145.).

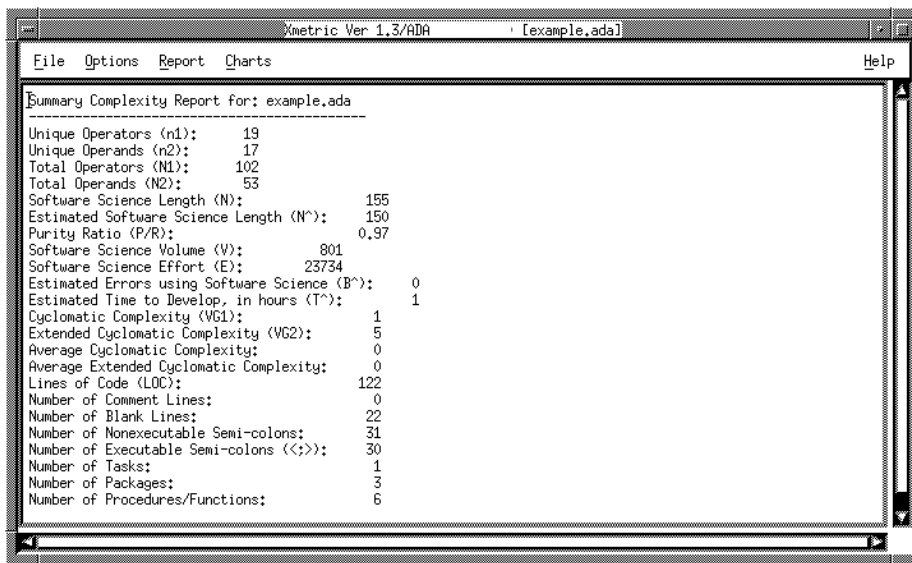


FIGURE 75 Summary Report for Ada

The Exception Report

The **Exception** report, *filename.exp* is the third report created by METRIC™. Each procedure in the source files which exceeds a set of pre-defined complexity maximums is included in this report. This report uses either the defined standards specified in the configuration file or the standards set in the GUI's **Configuration Options** window to determine what the maximum complexities are.

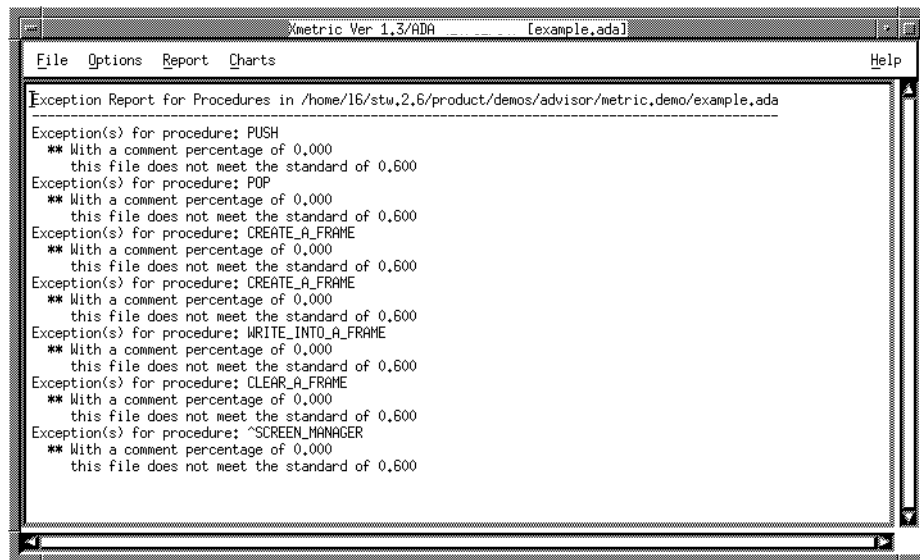


FIGURE 76 Exception Report for Ada

The Error Report

The **Error** report, *filename.err* will only be created if there was an error encountered during processing and analysis. Possible errors are listed next:

- **message:** <<< Unable to open input file <filename> >>>
- **cause:** The file specified to be analyzed could not be found or opened successfully.
- **remedy:** Check the spelling of the filename. Make sure the extension (if any) is correct. Make sure the path name is correct.
- **message:** <<< Unable to open reserved word file <filename> >>>

- **cause:** The reserved word file could not be found.
- **remedy:** Check the spelling of the filename. Make sure the extension (if any) is correct. Make sure the pathname is correct.

- **message:** <<< Number of operators exceeds limit >>>
- **cause:** When modifying the reserved word file to include additional words, you exceeded the default limit, or the reserved word limit is too small.
- **remedy:** Add or modify the configuration file parameter RESCT.

- **message:** <<< Heap overflow - reorganize source program >>>
- **cause:** Your file to be analyzed contains too many unique operands or operators. You have run out of memory to continue processing.
- **remedy:** Remove RAM resident routines or do not analyze as many source files at one time.

- **message:** <<< Improper pathname or no files found >>>
- **cause:** The file names entered could not be opened.
- **remedy:** Check the spelling of the file names and make sure the directory path is correct.
- **message:** <<< Unable to create reserved word dynamic array >>>
- **cause:** Not enough memory is available to load the reserved word lists.
- **remedy:** Make sure that the entry RESCT is not overly large.
- **message:** <<< Unexpected end of file - analysis aborting >>>
- **cause:** The end of file has been reached before the end of the procedure.
- **remedy:** Make sure your source file compiles. If it does, contact **Software Research**.

The Generic Report

The **Generic**, or generic instantiation report (*filename.gen*), lists, for each generic, the generic name, the number of times it is instantiated and the names of the routines it is instantiated by. Below is sample report:

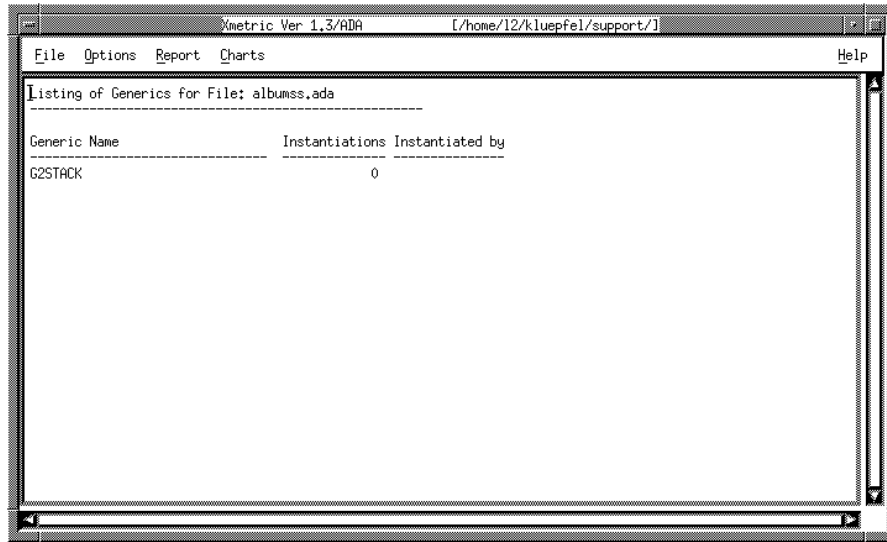


FIGURE 77 Generic Report

The Package Exception Report

Each package in the source files analyzed which do not meet or exceed a set of predefined complexity standards is included in this report, *file-name.pex*. This report uses the standards specified in the *METRICTM* configuration file to determine what the complexities should be. Here is list of some of the messages that will appear when a standard is not met.

- With *x* lines of code, this package exceeds the standard of *n* lines of code.
- With *x* executable statements, this package does not meet the standard of *n* executable statements.
- With *x* executable statements, this package exceeds the standard of *n* executable statements.
- With a cyclomatic complexity of *x*, this package exceeds the standard of *n*.
- With an extended cyclomatic complexity of *x*, this package exceeds the standard of *n*.
- With an average span of reference of *x*, this package exceeds the standard of *n*.
- With an estimated length of *x* this package does not meet the standard of *n*.
- With an estimated length of *x* this package exceeds the standard of *n*.

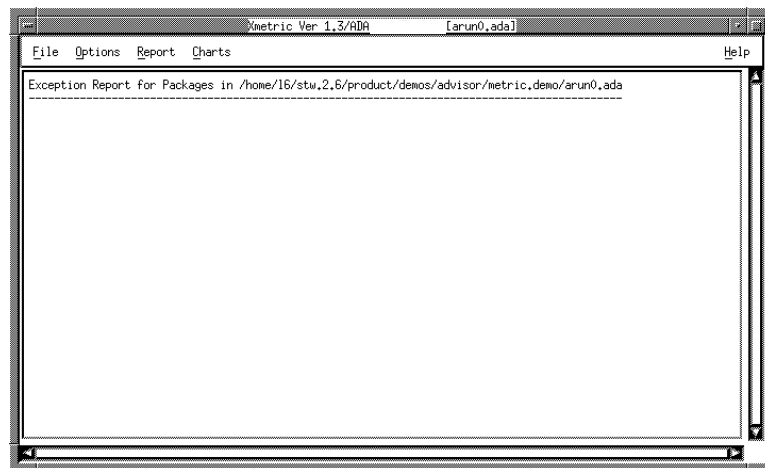
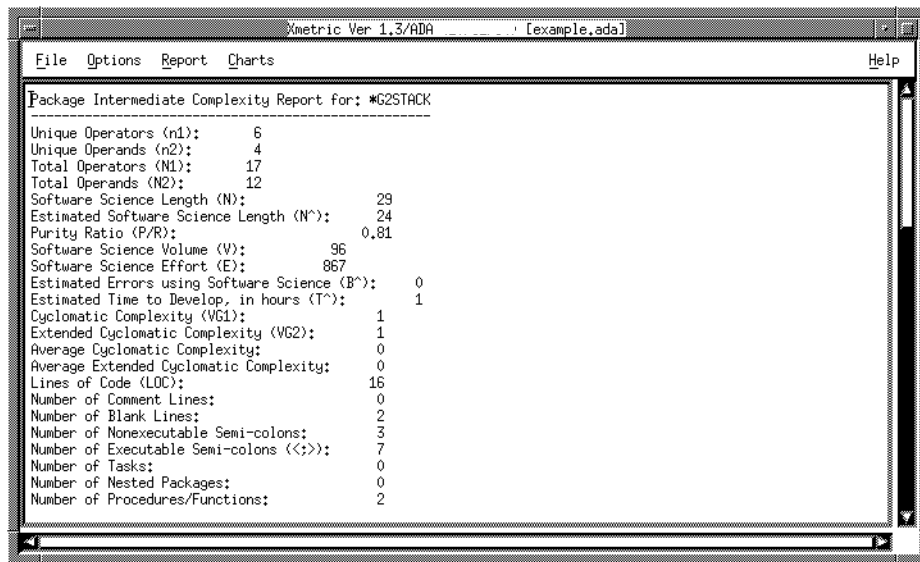


FIGURE 78 Package Exceptions Report

The Package Intermediates Report

The **Package Intermediates Report** is an *Intermediate Summary*, `file-name.rpt`, report at the package level. Package breaks are only performed at the outermost level – nested package will not have a break. This report has the same complexity fields as the **Summary** report, with the addition of the **Number of Nested Packages**.



```
Xmetric Ver 1.3/ADA [example.ada]
File Options Report Charts Help
-----
Package Intermediate Complexity Report for: #G2STACK
Unique Operators (n1):      6
Unique Operands (n2):     4
Total Operators (N1):     17
Total Operands (N2):     12
Software Science Length (N):      29
Estimated Software Science Length (N^):  24
Purity Ratio (P/R):          0.81
Software Science Volume (V):      96
Software Science Effort (E):     867
Estimated Errors using Software Science (B^):  0
Estimated Time to Develop. in hours (T^):  1
Cyclomatic Complexity (VG1):      1
Extended Cyclomatic Complexity (VG2):  1
Average Cyclomatic Complexity:    0
Average Extended Cyclomatic Complexity:  0
Lines of Code (LOC):           16
Number of Comment Lines:        0
Number of Blank Lines:         2
Number of Nonexecutable Semi-colons:  3
Number of Executable Semi-colons (<>):  7
Number of Tasks:               0
Number of Nested Packages:      0
Number of Procedures/Functions:  2
```

FIGURE 79 Package Intermediates Report

C.2 Counting Rules

This chapter describes the rules *METRIC*TM uses in analyzing Ada programs. It addresses issues associated with the reserved word and nonexecutable word files used in the analysis, miscellaneous counting rules of operators and operands, cyclomatic complexity, executable semicolons, variable name length, span of reference, and line of code counting rules. Description of the Halstead Software Science entries can be found in the correct chapter (See Section 5.1.2 - “Software Development” on page 116.).

Reserved Word File

The reserved word file, `adaresword.tab`, contains a list of Ada operators. To see what this file is comprised of, simply print out the file. A few of the items in `adaresword.tab` are not part of standard Ada and are explained below.

- In Ada, a parenthesis has three uses: it can change the default ordering of arithmetic operations, it follows a procedure call, and it follows an array. To distinguish these uses, three different parentheses have been defined in the file: “(” indicates an arithmetic paren, “(s” indicates a paren following an array name, and “(p” indicates a paren following a procedure call. Each of these are a different use of a parenthesis and, therefore, each is a different operator.
- Certain items in the list are not counted. These are the items that must be paired with another and consist of: END), and IN, when associated with FOR and LOOP, when associated with FOR or WHILE. Also not counted are the single quote, ‘, and double quote, “. These signal the start of a string and are counted as part of the string.

Miscellaneous Operator/Operand Rules

The following are some miscellaneous rules used when counting operators and operands:

- Procedure calls are counted as operators. All procedure and function names are stored so that they can be distinguished from array names (which are operands). It is important that all separately compiled units be included in the list of files to be analyzed if references to procedures/functions in them are made from other files. Otherwise, the operator and operand count will be inaccurate.

- All operands are considered to be global in the **Summary** report. Therefore, local variables of the same name defined in different procedures are counted as multiple occurrences of the same variable for purposes of the **Summary** report.
- When using `gotos`, the `goto` label is treated as an operator and the occurrence of the label in the code is an operand. The colon following the label is an operator.
- When using `DECLARE` statements, all variable declarations are ignored. Also, if procedures/functions are declared, the heading is skipped, and the code portion of the procedure/functions is counted as part of the outer procedure/function. It will not be listed in the `Complexity` report nor will the count of procedures be incremented.
- When a `'` is encountered that is not enclosing a single character and it is not an instantiation of a variable (ie, it is an attribute), then `'attr` is an operator.
- When an `ACCEPT` statement is encountered with parameters in its heading, all parameters are operands, and all `:typedef`'s are ignored.
- When using the `NEW` operator, the new `var1` is treated as one operator.

Cyclomatic Complexity

The following control structures increment the cyclomatic complexity count: `if`, `elseif`, `while`, `loop` (unassociated with `for` and `while`), `for`, and `when`. Occurrences of `else` do not increment the count. The extended cyclomatic complexity is incremented for each of these operators plus `&&` and `||`.

Span of Reference

The span of reference counts the maximum number of lines between references to each variable in a procedure (either use or assignment). The average of all the maximum references is then computed. This average for each procedure is listed in the **Complexity** report.

Executable Semi-colons

Executable semicolons begin counting with the first executable line of code. Hence, all declarations are not included.

Average Variable Name Length

The average variable name length provides a 'readability index' based on variable naming conventions. The value can be calculated in two ways, based on the configuration file entry `UNIQUE_VARIABLES`. If `UNIQUE_VARIABLES` is set to 1, then the length of all unique variable names (used at least once) is divided by the number of unique variables. If `UNIQUE_VARIABLES` is set to 0, then a weighted calculation is done.

With this method, for all variables, the length of the variable name is multiplied by the number of times the variable is used and this sum is divided by the total count of variable usage. Note that this is not the same as dividing by n_2 (in the first case) or N_2 (in the second case). n_2 and N_2 include numeric constants and literal strings whereas these calculations are concerned only with variables.

Lines of Code

Lines of code for a procedure include all lines of code from the function heading to the last statement of the function, `}`. This includes comments, blanks and continued lines. In addition, if `OUTSIDE_COMMENTS` are being counted, then the line of code count for the procedure will be incremented by the number of outside comments encountered.

Comments

The number of comments in a procedure is the count of all comments on a line by themselves encountered within the body of the procedure. If a single comment spans multiple lines, the number of lines that it spans is added to the comment count. If `INLINE_COMMENTS` is 1, then comments on the same line as an executable statement will also increment the count. If `OUTSIDE_COMMENTS` is 1, then comments outside of the scope of any procedure immediately before the procedure are added to the count.

C.3 Creating a Shell Script File

If there are a group of programs to be analyzed, where a separate report is to be produced for each program, it may be convenient to analyze them all at once through the use of a shell script. To create a shell script, create a file, *filename*, and enter *METRIC*TM command lines as though they were being entered at the system prompt. Make sure that for the filename you choose, there does not currently exist a command of the same name. You may use the command line mode with or without display to analyze the files.

For example, suppose we create a shell script `shtest` that contains the following:

```
#!/bin/sh -f
adametric adacode
adametric filematch tool
```

“adametric” must appear as the first word on the line since it is the program to be executed. The source file(s) to be analyzed and any command line parameters follow. Be sure to include pathnames for those files to be analyzed that do not reside in the current directory.

You can run the shell script file in either of the following ways:

- At the system prompt, give the filename as an argument to the `sh` command. For example:

```
sh shtest
```

- Use the `chmod` command to make the file executable and then use the name of the shell script like any other command. For example:

```
chmod 755 shtest
shtest
```

- Each source file in the shell script will be processed in turn. If a file is not found in the shell script, processing will continue with the next file. An error file will be created for each file in which an error occurred.

C.3.1 Comments About METRIC

If you analyze your programs before they are completed and you have empty procedures such as the following:

```
procedure procA is
begin
end;
```

you will get spurious results for that procedure in the procedure list. The above example does not contain any operands and many of the metrics are not defined for programs without operators or operands.

If a package does not contain any executable code of its own, it will be listed in the **Complexity** report with values of 1 in all the fields except LOC. The lines of code figure represents all the lines of code in the package body. An example is a package of the form:

```
package body pkgname is
  procedure A is
  begin
    .
    .
    .
  end A;
  procedure B is
  begin
    .
    .
    .
  end B;
end pkgname;
```

Note that a package body will always have at least one operator - the semicolon following the 'end'. In the **Complexity** report, package names are prefixed with an asterisk, '*', and task names are prefixed with '^'.

If a file does not contain any executable code (ie, it contains only declarations), in the **Complexity** report it will be listed as a procedure called `emptyfile`. Furthermore, all values will be zero except for lines of code (LOC) which will reflect the total lines of code in the file.

FORTRAN Notes

*METRIC*TM for FORTRAN consists of several files, the following of which will be explained in this chapter:

- `fmetric`
- `forreswo.tab`

fmetric

This file is the actual analyzer. It uses the remaining files in determining if a word is an *operator* or an *operand* or if it is a nonexecutable word. A complete description on how to use `fmetric` is given in another chapter (See CHAPTER 7 - "Command Line Activation" on page 145.).

forreswo.tab

This file contains a sorted list of all operators and reserved words for standard FORTRAN preceded by a number. The meaning of the numbers is as follows:

- | | |
|---|---|
| 0 | Ignore the word for it must always occur with another word. |
| 1 | The word is a nonexecutable word and does not increment any count. |
| 2 | The word is an operator and therefore increments the operator count. |
| 3 | The word is a control word. It increments the operator count and cyclomatic complexity. |
| 4 | The word increments the operator count, cyclomatic complexity and extended cyclomatic complexity. |

If you need to modify the list to change the counting rules or add/remove items to/from the list, you may do so in one of two ways:

1. Edit the file, `forreswo.tab`, and modify those entries that are to be changed. If the number of entries exceeds 330, you must make modifications to the configuration file `.uxmetriccfg`. Refer to the correct section for details on how to increase the number of allowable entries (See Section 7.5 - “Configuration File Processing” on page 156.).
2. Create a new reserved word file containing the counting rules and operators you wish to use. If this option is chosen, you must use the configuration file indicating that a different reserved word file is being used. See the correct section for details on how to do this (See Section 4.4.2 - “Writing Reports to a File” on page 82.).

Before you modify the list, be sure to read the section that describes this activity (See Section 4.4.2 - “Writing Reports to a File” on page 82.).

D.1 Description of the Reports

This section describes the reports created by *METRIC*TM: the **Complexity** report, the **Summary** report, the **Exception** report, and the **Error** report.

The Complexity Report

The **Complexity** report by Subprogram, *filename.rpt* includes the following fields:

- Subprogram Name
- Procedure Name
- Unique Operators (n1)
- Unique Operands (n2)
- Total Operators (N1)
- Total Operands (N2)
- Length (N)
- Predicted Length (N[^])
- Purity Ratio – estimated length divided by length (P/R)
- Volume (V)
- Effort (E)
- Cyclomatic Complexity (VG1)
- Extended Cyclomatic Complexity (VG2)
- Lines of Code (LOC)
- Number of Comment Lines (CMT)
- Number of Blank Lines (BLK)
- Executable Carriage Returns (<CR>)
- Average Maximum Span of Reference of Variables (SP)
- Variable Name Length (VL)

Please refer to the correct chapter for an in-depth discussion of the various fields (See CHAPTER 3 - "System Introduction" on page 39).

APPENDIX D:

Complexity Report by Subprogram for: /home/16/stw,2.6/product/demos/advisor/metric,demo/example.f

Procedure	n1	n2	N1	N2	N	N [^]	P/R	V	E	VG1	VG2	LOC
MAIN	7	1	10	3	13	20	1.51	39	410	1	1	5
example.f\$MAIN	33	17	147	51	198	236	1.19	1117	55315	9	9	56
example.f\$MAIN	15	11	44	22	66	97	1.46	310	4653	6	6	16
example.f\$MAIN	3	1	4	1	5	5	0.95	10	15	1	1	1
example.f\$MAIN	3	1	4	1	5	5	0.95	10	15	1	1	1
example.f\$MAIN	3	1	4	1	5	5	0.95	10	15	1	1	1
example.f\$MAIN	3	1	4	1	5	5	0.95	10	15	1	1	1
example.f\$MAIN	3	1	4	1	5	5	0.95	10	15	1	1	1
ITREE	3	1	4	1	5	5	0.95	10	15	1	1	1
example.f\$MAIN	3	1	4	1	5	5	0.95	10	15	1	1	1
example.f\$MAIN	3	1	4	1	5	5	0.95	10	15	1	1	1
example.f\$MAIN	3	1	4	1	5	5	0.95	10	15	1	1	1
example.f\$MAIN	3	1	4	1	5	5	0.95	10	15	1	1	1
example.f\$MAIN	3	1	4	1	5	5	0.95	10	15	1	1	1

FIGURE 80 Complexity Report for FORTRAN

The Summary Report

The **Summary** report, *filename.rpt* consists of **Complexity** report fields, with the exception of average maximum span of reference of variables and average variable name length. In addition it includes the values:

- Estimated errors (B^{\wedge})
- Estimated development time (T^{\wedge})
- Average cyclomatic complexity
- Average extended cyclomatic complexity
- Number of subroutines/functions

Please refer to the correct section for an explanation of the **Summary** report's fields (See Section 2.1.7 - "STEP 7: Viewing a Summary Report" on page 22.). The command line also allows you create an **Intermediate Summary** report whenever there is a change in directory (*-bn*). This report may be helpful if different subsystems of an application being analyzed are placed in different directories and you want a **Summary** report for each subsystem.

You can also create an **Intermediate Summary** report whenever a change in the first *n* characters of the filename occurs (*-bn*). This report may be helpful if the files comprising the different subsystems of an application all have the same prefix and you want a **Summary** report for each subsystem.

See the chapter that describes command line activation (See CHAPTER 7 - "Command Line Activation" on page 145.).

APPENDIX D:

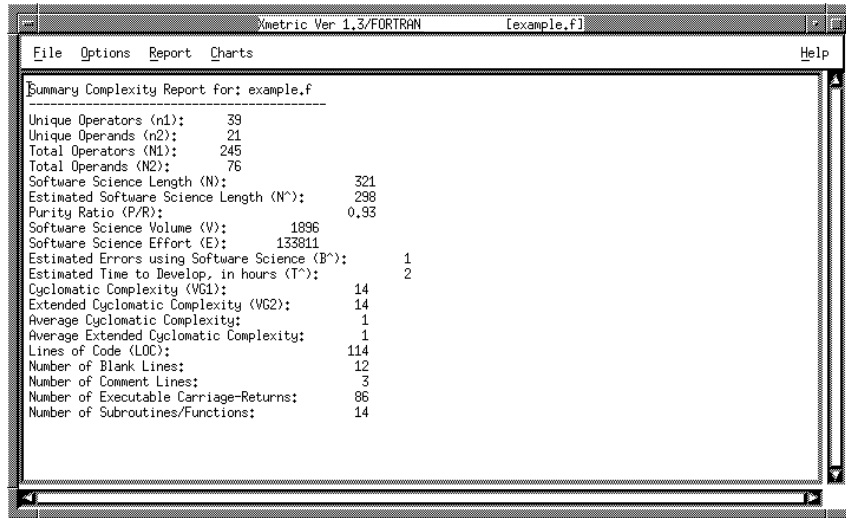
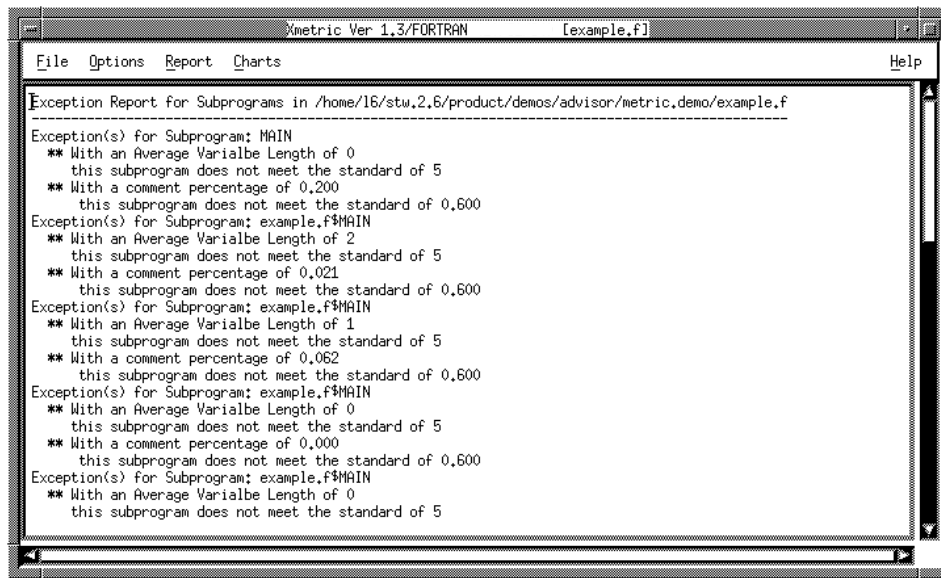


FIGURE 81 Summary Report for FORTRAN

The Exception Report

The **Exception** report is the third report created by *METRIC*TM. Each procedure in the source files which exceeds a set of predefined complexity maximums is included in this report. This report uses either the defined standards specified in the configuration file or the GUI's **Configuration Options** window to determine what the maximum complexities are.



The screenshot shows a window titled "Xmetric Ver 1.3/FORTRAN" with a file name "[example.f]". The menu bar includes "File", "Options", "Report", "Charts", and "Help". The main text area displays the following report content:

```
Exception Report for Subprograms in /home/16/stw,2,6/product/demos/advisor/metric,demo/example,f
-----
Exception(s) for Subprogram: MAIN
** With an Average Variable Length of 0
   this subprogram does not meet the standard of 5
** With a comment percentage of 0,200
   this subprogram does not meet the standard of 0,600
Exception(s) for Subprogram: example.f$MAIN
** With an Average Variable Length of 2
   this subprogram does not meet the standard of 5
** With a comment percentage of 0,021
   this subprogram does not meet the standard of 0,600
Exception(s) for Subprogram: example.f$MAIN
** With an Average Variable Length of 1
   this subprogram does not meet the standard of 5
** With a comment percentage of 0,062
   this subprogram does not meet the standard of 0,600
Exception(s) for Subprogram: example.f$MAIN
** With an Average Variable Length of 0
   this subprogram does not meet the standard of 5
** With a comment percentage of 0,000
   this subprogram does not meet the standard of 0,600
Exception(s) for Subprogram: example.f$MAIN
** With an Average Variable Length of 0
   this subprogram does not meet the standard of 5
```

FIGURE 82 Exception Report for FORTRAN

The Error Report

The **Error** report will only be created if there was an error encountered during processing and analysis. Possible errors are listed below:

- **message:** <<< Unable to open input file <filename> >>>
- **cause:** The file specified to be analyzed could not be found or opened successfully.
- **remedy:** Check the spelling of the filename. Make sure the extension (if any) is correct. Make sure the path name is correct.

- **message:** <<< Unable to open reserved word file <filename> >>>
- **cause:** The reserved word file could not be found.
- **remedy:** Check the spelling of the filename. Make sure the extension (if any) is correct. Make sure the pathname is correct.

- **message:** <<< Number of operators exceeds limit >>>
- **cause:** When modifying the reserved word file to include additional words, you exceeded the default limit, or the reserved word limit is too small.
- **remedy:** Add or modify the configuration file parameter RESCT.

- **message:** <<< Heap overflow - reorganize source program >>>
- **cause:** Your file to be analyzed contains too many unique operands or operators. You have run out of memory to continue processing.
- **remedy:** Remove RAM resident routines or do not analyze as many source files at one time.

- **message:** <<< Improper pathname or no files found >>>
- **cause:** The file names entered could not be opened.
- **remedy:** Check the spelling of the file names and make sure the directory path is correct.

- **message:** <<< Unable to create reserved word dynamic array >>>
- **cause:** Not enough memory is available to load the reserved word lists.
- **remedy:** Make sure that the entry RESCT is not overly large.

- **message:** <<< Unexpected end of file - analysis aborting >>>
- **cause:** The end of file has been reached before the end of the procedure.
- **remedy:** Make sure your source file compiles. If it does, contact **Software Research**.

- **message:** <<<Unable to open include file <filename> >>>
- **cause:** The include file could not be found given the specification in the source code;
- **remedy:** Make sure the include file exists. If it exists in a different directory than specified in the source code, make `INCLUDE_DIR` entries in the configuration file.

D.2 Counting Rules

This section describes the rules *METRICTM* uses in analyzing FORTRAN programs. It addresses issues associated with the reserved word and non-executable word files used in the analysis, miscellaneous counting rules of operators and operands, cyclomatic complexity, executable semicolons, variable name length, span of reference, and line of code counting rules. Description of the Halstead Software Science entries can be found in the chapter that describes that topic (See CHAPTER 3 - "System Introduction" on page 39.).

Reserved Word File

The reserved word file, `forreswo.tab`, contains a list of FORTRAN operators. To see what this file is comprised of, simply print out the file. A few of the items in `forreswo.tab` are not part of standard FORTRAN and are explained below.

- In FORTRAN, a parenthesis has four uses: it can change the default ordering of arithmetic operations, it follows a subprogram call, it follows a control statement, and it follows an array name. To distinguish these uses, four different parentheses have been defined in the file: “(” indicates an arithmetic paren, “(c” indicates a paren following a control statement, “(s” indicates a paren following an array name, and “(p” indicates a paren following a subprogram call. Each of these are a different use of a parenthesis and, therefore, each is a different operator.
- In FORTRAN, the asterisk, *, has two uses: as a multiplication sign and to represent a default device unit. To distinguish these uses, two asterisks have been defined: “*” indicates multiplication, and “*w” indicates a default device. Since these are different meanings, each is counted as a different operator.
- In FORTRAN, END can be used to signal the end of a subprogram or used in a file access statement. Therefore, two ENDS have been defined: “END” is for the end of a subprogram/program, and “END” is for the end used in a file access command.
- Certain items in the list are not counted. These are the items that must be paired with another and consist of: END and) and TO when associated with ASSIGN. Also not counted are the single quote, '. This signals the start of a string and is counted as part of the string.

Any statement preceded by one of the '1' words in the reserved word file is considered nonexecutable.

A nonexecutable word can precede a subprogram name and still allow the subprogram to be recognized. For example:

```
INTEGER FUNCTION ROUNDIT(X)
.
.
.
END
```

will see that `ROUNDIT(X)` is a subprogram and will analyze the code up to the `END`.

Miscellaneous Operator and Operand Rules

The following are some miscellaneous rules used when counting operators and operands.

- Subprogram calls are counted as operators.
- All operands are considered to be global in the **Summary** report. Therefore, local variables of the same name defined in different subprograms are counted as multiple occurrences of the same variable for purposes of the **Summary** report.
- When using `gotos`, the `goto` label is treated as an operator and the occurrence of the label in the code is an operand.
- When using `do` loops, the `DO` label is treated as an operator and the occurrence of the label in the code is an operand.
- `FORMAT` is generally considered to be a nonexecutable word but to be consistent with the other *METRICTM* analyzers, it is counted as an operator. The label preceding the `FORMAT` is an operand and the entire format specification from the opening paren to the closing paren is one operand.
- `ENTRY` statements are nonexecutable and, hence ignored. However, their appearance in the source code is indicated in the **Exception** report.

Cyclomatic Complexity

The following control structures increment the cyclomatic complexity count: `IF`, `DO`, and `GOTO` when used in a computed `GOTO` or an assigned `GOTO`. Occurrences of `else` do not increment the count. The extended cyclomatic complexity is incremented for each of these operators plus `.AND.`, and `.OR..`

Span of Reference

The span of reference counts the maximum number of lines between references to each variable in a subprogram (either use or assignment). The average of all the maximum references is then computed. This average for each subprogram is listed in the **Complexity** report.

Executable Carriage Returns

Executable carriage returns begin counting with the first executable line of code. Hence, all declarations are not included. Commented lines are also not included. Executable lines that are continued on the next line are not counted.

Average Variable Name Length

The average variable name length provides a 'readability index' based on variable naming conventions. The value can be calculated in two ways, based on the configuration file entry `UNIQUE_VARIABLES`. If `UNIQUE_VARIABLES` is set to 1, then the length of all unique variable names (used at least once) is divided by the number of unique variables. If `UNIQUE_VARIABLES` is set to 0 then a weighted calculation is done.

With this method, for all variables, the length of the variable name is multiplied by the number of times the variable is used and this sum is divided by the total count of variable usage. Note that this is not the same as dividing by n_2 (in the first case) or N_2 (in the second case). n_2 and N_2 include numeric constants and literal strings whereas these calculations are concerned only with variables.

Lines of Code

Lines of code for a subprogram include all lines of code from the subprogram heading to the last statement of the subprogram (`END`). This includes comments, blanks and continued lines. In addition, if `OUTSIDE_COMMENTS` are being counted, then the line of code count for the subprogram will be incremented by the number of outside comments encountered.

Comments

The number of comments in a subprogram is the count of all comments on a line by themselves encountered within the body of the subprogram.

In addition, if `D LINES` is 0 then all lines preceded with `D` are comment lines. If `INLINE_COMMENTS` is 1, then comments on the same line as an executable statement will also increment the count. If `OUTSIDE_COMMENTS` is 1, then comments outside of the scope of any subprogram immediately before the subprogram are added to the count.

D.3 Creating a Shell Script File

If there are a group of programs to be analyzed, where a separate report is to be produced for each program, it may be convenient to analyze them all at once through the use of a shell script. To create a shell script, create a file, *filename*, and enter *METRICTM* command lines as though they were being entered at the system prompt. Make sure that for the filename you choose, there does not currently exist a command of the same name. You may use the command line mode with or without display to analyze the files.

For example, suppose we create a shell script `shtest` that contains the following:

```
#!/bin/sh -f
fmetric forfile
fmetric forfile2 forfile3
```

“fmetric” must appear as the first word on the line since it is the program to be executed. The source file(s) to be analyzed and any command line parameters follow. Be sure to include pathnames for those files to be analyzed that do not reside in the current directory.

You can run the shell script file in either of the following ways:

- At the system prompt, give the filename as an argument to the `sh` command. For example:

```
sh shtest
```

- Use the `chmod` command to make the file executable and then use the name of the shell script like any other command. For example:

```
chmod 755 shtest
shtest
```

Each source file in the shell script will be processed in turn. If a file is not found in the shell script, processing will continue with the next file. An error file will be created for each file in which an error occurred.

D.4 Comments About METRIC

If you analyze your programs before they are complete and you have empty procedures such as the following:

```
SUBROUTINE ABC  
END
```

you will get spurious results for that subprogram in the subprogram list. The above example does not contain any operands and many of the metrics are not defined for programs without operators or operands.

APPENDIX D:

Index

A

adamic 207
adamic command 147
allocating testing resources 67, 122
ampersand 180, 198
analyzing the Complexity report 16
Apply button 96, 101, 104, 113
apportioning duties to programmers 123
Ascend button 89
asterisk 180, 198, 230
average Cyclomatic Complexity 57
average Extended Cyclomatic Complexity 57

C

C++ Class option 137
C++ Class Report 196
changing the configuration file 116
chapter organization xiv
Charts cascading menu 101, 113, 132
Charts Pull-Down Menu 140, 141
Charts pull-down menu 99, 103, 113
Class Exception report 197
Class Exceptions option 138
Class Hierarchy option 138
Class Hierarchy report 197
Class Summary option 137
Class Summary report 196
Close button 96, 101, 104, 113
cmetric command 147
Columns option 96, 131
Comment Percent option 96, 132
Comments 182, 200, 232
completing a session 34
complex modules 20
Complexity option 86, 137
Complexity Report 173

Complexity report 86, 189, 208, 223
Complexity Report fields 41
Complexity report, looking at 86
complexity, ordering procedures 89
conditional compilation directives 184, 203
configuraiton option, MINIPKGN 160
configuration file 115
configuration file processing 156
configuration option, ADVLENGTH 156
configuration option, ANALYZEINCLUDE 156
configuration option, CLASSCHART 156
configuration option, CLASSEXCEPTION 156
configuration option, CLASSREPORT 157
configuration option, COLSTART 157
configuration option, COMMENT_PERCENT 157
configuration option, COMMENT_SYMBOL 157
configuration option, CONDCOMPILE 158
configuration option, COUNINC 158
configuration option, COUNTCR 158
configuration option, DLINES 158
configuration option, EO 159
configuration option, FREEFORMAT 159
configuration option, GOTOS 159
configuration option, HEADER_SUMMARY 159
configuration option, INCLUDE_DIR 160
configuration option, INLINE_COMMENTS 160
configuration option, LOC 160
configuration option, MAXPKGKN 160
configuration option, MAXPKGSEMI 161
configuration option, MAXSTDN 161
configuration option, MINIPGSEMI 161
configuration option, MINPKGSEMI 162
configuration option, MINSTDN^ 162
configuration option, NESTED 162
configuration option, NONEXCT 162
configuration option, NONEXE 162
configuration option, OUTSIDE_COMMENT 162
configuration option, PAGE_BREAK 163

INDEX

configuration option, PAGE_HEADINGS **163**
configuration option, PAGELength **163**
configuration option,
 PKG_COUNT_ALL_LINES **163**
configuration option, PKGLOC **164**
configuration option, PKGSPAN **164**
configuration option, PKGVG **165**
configuration option, PKGVG2 **165**
configuration option, PKSUMMARYLEVEL **165**
configuration option, PRINTER **166**
configuration option, PRINTEXP **166**
configuration option, PRINTGENERIC **166**
configuration option, PRINTPKGEXP **166**
configuration option, PRINTPRAGMA **166**
configuration option, PRIVATEMEMS **166**
configuration option, PROTECTEDMEMS **166**
configuration option, PUBLICMEMS **166**
configuration option, RESCT **166**
configuration option, RESFILE **167**
configuration option, SEMI **167**
configuration option, SPAN **167**
configuration option, SPEED **167**
configuration option, STATEMENT **167**
configuration option, STDEXPLICIT **168**
configuration option, STDFRIENDCLS **168**
configuration option, STDINLINE **168**
configuration option, STDMEMBERS **168**
configuration option, STDVIRTUAL **168**
configuration option, SUMMARYONLY **168**
configuration option, UNIQUE_VARIABLES **168**
configuration option, VARIABLE_LENGTH **168**
configuration option, VG **168**
configuration option, VG+ **168**
configuration option, VOLUME **168**
configuration option, WARNINGS **168**
Configuration Options window **95, 96, 131**
configurationoption, STDFRIEND **168**
Control Flow metrics **87, 93**
control flow metrics **3**
counting lines of code **1**
counting rules **180, 198, 216, 230**
cppmetric **187, 207**
cppmetric command **147**
Cyclomatic Complexity **50, 181, 199, 217, 231**
Cyclomatic Complexity measures **7, 16**
Cyclomatic Complexity option **96, 132**

D

Data Objects metrics **87**
Defines option **96, 131**
Descend button **89**
developing metrics **3**

development time **119**
Directories selection window **72**
directory, demos **11**
display area **127**

E

edge **50**
Effort **49**
END **230**
entropy **123**
environment variable, METRICCFG **156**
error messages **178, 194, 211, 228**
Error Report **178**
Error report **26, 94, 194, 211, 228**
Error report, analyzing **26**
Error report, looking at **94**
Errors option **94, 137**
estimated errors **58**
estimated time to develop **58, 60**
Exception Report **177**
Exception report **24, 61, 93, 193, 211, 227**
Exception report, analyzing **24**
Exception report, looking at **93**
Exceptions option **93, 137**
Executable Carriage Returns **231**
Executable Semi-Colons **181**
Executable Semi-colons **199, 217**
Exit option **74, 103, 107, 130**
exiting METRIC **114**
expecting too much **69**
Extended Cyclomatic Complexity **52**
Extended Cyclomatic Complexity option **96, 132**

F

field, B^ **58**
field, Baseclass **196**
field, E **49**
field, Friend Cls/Fct **196**
field, FT **189**
field, Inline **196**
field, N^ **46**
field, N1 **43**
field, n1 **43**
field, N2 **43**
field, n2 **43**
field, P/R **46**
field, Private Var/Fct **196**
field, Public Var/Fct **196**
field, T^ **60**
field, Total Membs **196**
field, V **48**

field, VG1 **50**
 field, VG2 **52**
 field, Virt **196**
 File pull-down menu **83, 84, 85, 103, 107, 127**
 file selection dialog boxes **72**
 file, .adaresword **207**
 file, .uxmetriccfg **61**
 file, .Xmetric.I.def **99**
 file, cnonexe.tab **172, 173**
 file, cppnonexe.tab **188**
 file, cppresword.tab **187, 207**
 file, filename.cex **82, 197**
 file, filename.cht **82, 197**
 file, filename.cls **82, 196**
 file, filename.err **82, 178, 194, 211, 228**
 file, filename.exp **82, 177, 193, 211, 227**
 file, filename.gen **82, 212**
 file, filename.pex **82, 214**
 file, filename.rpt **82, 152, 173, 175, 189, 191, 208, 209, 215, 223, 225**
 file, forreswo.tab **221**
 file, METRICtm **11**
 file, sr.c **11**
 file, xcalc.c **11**
 file, Xmetric.I.def **28**
 file, Xmetric.II.def **30**
 file, Xmetric.III.def **32**
 Files selection window **72**
 Filter entry box **72**
 flow of control **50**
 fmetric **221**
 font
 italics **xv**
 italix **xv**
 font, bold face **xv**
 font, courier **xv**

G

Generic option **137**
 Generic report **212**

H

Help Button **144**
 help dialog frame **74**
 Help option **130, 138**

I

identifying complex modules **1**
 Intermediate Summary Report **175**
 Intermediate Summary report **191**

invocation window **11**
 invoking from STW **79**
 invoking METRIC **78**

K

Kiviat charts **28, 99**
 Kiviat charts, using **28, 99**
 Kiviat diagram, creating your own **113**
 Kiviat diagrams **98**

L

langmetric -132 switch **150**
 langmetric -80 switch **150**
 langmetric -bd switch **148**
 langmetric -bn switch **148**
 langmetric -cen switch **149**
 langmetric -chtd switch **149**
 langmetric -crn switch **149**
 langmetric -csym switch **149**
 langmetric -gn switch **149**
 langmetric -i switch **149**
 langmetric -n switch **149**
 langmetric -pen switch **149**
 langmetric -pn switch **150**
 langmetric -prn switch **150**
 langmetric -sn switch **150**
 Language option **81, 136**
 Length Equation **46**
 Lines of Code **182, 200, 232**
 Lines of Code option **96, 132**
 Load Multiple Files option **85, 129**
 Load Single File file selection **127**
 Load Single File option **84, 127**

M

main system features **8**
 Main window **78, 125**
 maintenance problem **1**
 managing maintenance **67**
 manual organization **xiv**
 message dialog boxes **75**
 metric accuracy, assessing **4**
 metric validity, determining **4**
 metrics as a feedback tool **116**
 metrics in estimation **119**
 metrics in software development **116**
 metrics in software maintenance **123**
 metrics in software testing **121**
 metrics in the review process **118**
 metrics to control entropy **123**

INDEX

metrics, why 1
METRICtm process 7
Multiple File selection 129
multiple file selection dialog box 85
multiple files, analyzing 18
multiple files, loading 18
multiple users 115

N

name patterns 147
no reports generated, error 115
node 50
nonexecutable reserved words 188
nonexecutable word file 180, 198
number of total operands 43
number of total operators 43
number of unique operands 43
number of unique operators 43

O

operand rules 181, 199, 216, 231
operator 187, 207
operator rules 181, 199, 216, 231
Option pull-down window 81
Options pull-down menu 95, 101, 104, 107, 113, 131
Order Complexity Option 89
Order Complexity option 137
OSF/Motif style GUI 72

P

Package Exception report 214
Package Exceptions option 137
Package Intermediates option 137
Package Intermediates report 215
parenthesis 180, 198, 216, 230
Potential Volume 49
predicates 52
Predicted Length 46
processing a source code file 80
producing less complex code 64
Program Level 49
pull-down menus 76
Purity Ratio 46

R

Report option 95, 96, 131
Report pull-down menu 86, 91, 93, 94

Report Pull-Down Window 137
Report pull-down window 91
reports, accuracy 63
reserved word file 180, 198, 216, 230, 231
reserved words 187

S

saving reports 82
scroll bars 72
Search option 74
Select Language window 81, 136
selecting a keysave file, general purpose 72
selecting a language window 81
selecting a source code file 14, 18, 84
selecting multiple source code files 85
Selection entry box 72
Semicolons option 96, 132
Set Report Files Basename option 83, 129
Set Report Files Basename window 83, 129
setting up 11, 115
shell script file, creating 183, 202
shell script, creating 234
single file selection dialog box 84
Size metrics 87, 93
size metrics 3
software complexity measures 7
software maintenance 123
software metrics 1
software metrics in development 64
software science counting rules 44
Software Science measures 16
Software Science metrics 87, 93
Sort Report By 89
Sort Report By window 137
source files, where to put 115
Span of Reference 181, 199, 217, 231
special text xv
SR variable 115
static code analyzer 40
statistical techniques 6
Stroud Number 60
STW 79
STW/MAN 79
Summary Only option 91, 137
Summary Report 175
Summary report 22, 91, 191, 209, 225
Summary report fields 55
Summary report, analyzing 22
Summary report, looking at 91
System Operation 71

T

text
 "double quotation marks" **xv**
 boldface xv
 italics xv
 special xv
text, **boldface xv**
text, **courier xv**
text, **italix xv**
threshold values, violations **116**
Type I Configuration window **101, 133**
Type I Kiviat chart **28**
Type I option **133, 140**
Type II Configuration window **104, 134**
Type II Kiviat chart **30, 103**
Type II option **103, 104, 134, 141**
Type III Configuration window **135**
Type III option **107, 113, 135, 142**
Type User option **113**

U

User **142**
using a file selection dialog box **73**
using message dialog boxes **75**
using METRIC **40**
using pull-down menus **76**
using the help frame **74**

V

Variable Name Length **182, 200, 217, 231**
Volume **48**

X

Xkiviat -nvalue switch **154**
Xkiviat -q switch **154**
Xkiviat -s x y switch **154**
Xkiviat -v switch **154**
Xmetric command **78**
Xmetric utility **146**
Xmetric-LAda switch **146**
Xmetric-LC switch **146**
Xmetric-LC++ switch **146**
Xmetric-LF77 switch **146**
xterm window **11**

