A STRUCTURE COVERAGE TOOL FOR ADA™ SOFTWARE SYSTEMS

Liqun Wu, Victor R. Basili, and Karl Reed[a]

Department of Computer Science
University of Maryland
College Park, MD 20742

ABSTRACT

Coverage metrics have traditionally been used to evaluate the effectiveness of procedures for testing software systems. In practice, however, the metrics are heavily influenced by the characteristics of traditional programming languages such as Fortran and Pascal. Languages such as Ada differ from traditional languages to such an extent that it is necessary to develop new metrics.

This paper proposes a number of coverage measures for Ada features such as packages, generic units, and tasks, and discusses their interpretation in relation to the traditional coverage metrics. It also propose a mechanism for collecting these coverage measures. In addition, it suggests that coverage metrics may also be interpreted as indicators of dynamic system performance.

INTRODUCTION

The last few years have seen an increased emphasis upon the development of techniques for assessing and controlling the quality of software products. Research in the U.S.[6, 5, 2, 13, 4], and practice in the Japanese computer manufacturers' software factories[17, 26, 12, 23] have recognized the importance of measures of program structure on one hand, and techniques for quality assurance on the other as fundamental aspects of software quality assurance.

The vast bulk of the work to date distinguishes between measures of program and system structure, which can be obtained by automatic analysis of source code, and dynamic measures of program quality which develop confidence in program quality by testing the system[22] would be McCabe's cyclomatic number for the former[15] and test coverage for the latter[18, 14].

There have been few reports of efforts to integrate both static and dynamic measures to allow an overall assessment of software quality.

The TAME[b] project at the University of Maryland's Department of Computer Science intends to integrate tools for obtaining both static and dynamic measures of program quality into a single environment. This will allow the quality of a software product to be monitored at all stages of its evolution and allow judgements to be made upon the basis of various values obtained. Although TAME is overwhelmingly language independent, its first application will be for monitoring systems written in Ada.

Most of the useful code-based metrics measure the static structure of the source because there is a serious shortage of measures of dynamic structure, with Conte et. al.[10] citing only a simple liveness measure.

Considerable experience has been gained in using these two classes of metrics to study systems implemented in traditional programming languages such as FORTRAN, COBOL, Pascal, and PL/I, and many of the quantities measured reflect their characteristics.

The availability of languages such as Ada and PROLOG, and their use in applications systems, necessitates the development of new measures, since these languages differ significantly from those currently under study. Structural coverage metrics will need to be tailored to allow for the impact of these new features upon programming practice.

---

[a] The third author is on leave from the Department of Computing at the Royal Melbourne Institute of Technology, Melbourne, Australia.

Ada is a trademark of the U.S. Department of Defense - Ada Joint Program Office.

Support for this research provided by NASA Grant NSG-5123 to the University of Maryland.

[b] See[6] and[7] for a complete description.

The results of a joint study between the University of Maryland and General Electric [1] show that a number of Ada features such as packages, generic units, exceptions, and tasks, appear to be misused, and to be a source of program faults. Applying existing measures to Ada programs without explicitly recognizing its unique[20] characteristics may lead to a totally misleading picture of a systems structure and quality.

Traditional measures of test coverage include[22]

a) the percentage of source code instructions executed,

b) the percentage of partial paths traversed[c],

c) percentage of predicate outcomes exercised,

d) the percentage of procedure or function calls made,

e) the percentage of procedures executed,

The first three of these measures are essentially statement level measures and reveal little about the system structure. On the other hand, the last two measure the procedural level and provide an indication of system structure.

Measures relating to procedure usage will need to be reassessed because of the impact of Ada's generic units and facilities for partitioning systems. In addition, new measures should be defined for tasks and exception handling.

Measures of the dynamic behavior of a large system consisting of a number of program-units cannot be obtained easily by examining its source code. They may, however, be deduced from an analysis of the system's behavior during execution[3] and interpretation of statistics normally associated with testing, such as various coverage measures, and for altering the strategies used for planning tests[18]

The objective of test coverage measures is to allow some estimate to be made of the extent to which a set of tests is likely to have revealed errors in the test subject[16, 25, 22] generation, and the interpretation of coverage metrics, will depend upon whether a series of unit tests, a subsystem test, or a complete system test are being planned.

In what follows, we propose a number of measures of static and dynamic structure for systems implemented in Ada. We discuss their interpretation in relation to traditional test coverage measures as a first step in the development of a more complete method of dynamic structure measurement through testing. These are, in general, re-interpretations of existing measures.

Many of the measures are not specific to Ada and can be applied to any language supporting multiple entry points,(e.g. Fortran, PL/I, Assembler), internally and externally callable procedures (e.g. Cobol, PL/I Assembler), generic declarative structures, or tasking (e.g. Module).

[c] A partial path is the shortest section of code connecting two decision statements which does not contain any other decision statement.

ADA AND COVERAGE MEASURES

There are four important features of Ada which impact the design of coverage measures which will be discussed further in this paper. These are:

a) generic units

b) packages

c) exceptions, and

d) tasking.

The first two of these deal with the procedural level of system structure, while the last two support non-deterministic system behavior. A complete set of Ada coverage measures must include the statement level metrics mentioned earlier. Table I shows the relationship of the various measures.

Generic Unit

Generic units may be instantiated into a new unit by a declaration either overloading an existing unit name or creating a new one. Instantiations may apply a generic unit to different types. It is not possible, therefore, to assume that a particular generic package has been properly tested until all its instantiations have been tested. Steps must also be taken to ensure that all references to individual instantiations are correctly counted. At least one currently available path analysis tool fails to make these distinctions

Ada's provision for block statements, executable units containing declarations[11] , raises the possibility of a particular instantiation of a generic procedure not actually being elaborated with the result that type checking may not be complete[d]. We will need a coverage measure for generic unit elaboration for this reason.

Packages and Libraries

The potential use of Ada library units and packages as devices for arbitrarily partitioning a system also presents a problem[8] In this case, the public entities in the package or library constitute a potential set of entities that can be referenced from any part of the project. It will therefore be necessary to decide which entities should be included in coverage counts and the method to be used in categorizing them.

[d] We note that block statements also require special treatment for similar reasons, however, we have limited our discussion to those features which illustrate our point.

A particular package may contain units which can be referenced from within the package as well. This provides another basis for estimating coverage[e].

Tasking and Exception Handling

Ada's tasking and exception handling mechanisms add a new dimension to path coverage since it may be desirable to verify that paths which traverse more than one program unit are exercised and to verify that various interactions actually occur. Each of these aspects of Ada must be accounted for in any proposed measures and explicitly considered in their interpretation.

The issues discussed here apply to other languages which support similar features.

COVERAGE METRICS FOR ADA

We focus on the impact of Ada's facilities for partioning systems and on the impact of generic units because these are the features which necessitate re-evaluating the approach to coverage measures. Any complete set of measures for Ada would, of course, include specific examples for packages and procedures of both the generic and non-generic variety. In that sense, some of our definitions are themselves generic.

Procedure Interaction and partioning

We wish to distinguish between the static structure and dynamic behavior of a system which may consist of Ada library units, packages and a program. We may then obtain an indication of system complexity based upon this difference.

Let us consider a system which references "n" different units. Let us also suppose that there are a total of "m" references to the "n" units. The ratio n/m would provide an indication of the static complexity of such a system. Two systems may have identical (n,m) pairs; however, they may have different interactive complexity since one may activate most of its references to the m units, while the other may not.

The total number of references executed by a set of tests which represent a typical input would also provide an indication of absolute dynamic complexity. The ratio n/m corresponds to a procedure call coverage metric.

We will base our discussion on procedures since these are among the most significant program units from a system structure point of view. A system written in Ada will appear as a collection of packages and subprogram units, each of which may have multiple procedures.

We can distinguish several different types of procedure reference, depending on the location of a reference and its target....

a) Intra package reference ... the reference in the current package is to a procedure in that package,

b) Extra package reference ... the reference in the current package is to a procedure in some other package,

c) Inter package reference ... the reference is directed to a procedure in this package from some program unit outside the package,

d) Combined inwards package references ... the sum of a) and c), and

e) Combined outwards package references ... the sum of a) and b).

These can be repeated for the program reference ... giving

f) Intra program reference ... the reference is to a procedure in the program, and

g) Extra program reference ... the reference is to a procedure outside the program.

Dynamic Interaction Measures

It is possible to define a wide range of these metrics which measure the amount of interaction between specific packages; however, we restrict ourselves to the following simple examples to illustrate the principle[f].

A) The number of extra referenced procedures in a package called compared to the number of procedures in a package

B) The number of combined inwards procedures referenced procedures in a package called compared to the number of combined inwards package references, and

C) The number of extra procedure references in a package executed compared to the number of extra procedure references.

Measure A cannot be interpreted without a knowledge of the the number of combined inwards referenced procedures for the package concerned and the context of the test. It will convey no information in addition to a static count of the referenced procedures in a package if the package is in a system test. However in the case of a unit test, it will show that the components of the package are inadequately tested.

---

[e] Other languages share this property.

[f] We could consider interaction between a given package and each of its partners, and some subset of its partners, for example. We could also consider all possible combinations of these against the procedure reference categories presented above.

Measure B will show the extent to which a particular package's inwards calls are being utilized by a particular test. It will, as the procedure coverage asymptotes, provide an indication of the total dynamic complexity of the package's inwards communications.

It is also interesting to consider the implications of this measure for a single test case since it provides an indication of the dynamic complexity of the path traced by the individual case. This may be useful in judging the difficulty in debugging faults found during the test.

A similar interpretation can be applied to to measure C.

GENERICS AND COVERAGE MEASURES

A generic unit is a template from which particular instances of a general unit can be obtained. A declaration provides a program unit name and an optional generic_actual_part[g] specifying the name and the types to used by this particular version of of the package. The elaboration of the declaration of an instance of a generic unit creates a new version which is distinct from other versions generated from the same generic unit. This is true even if instances have been instantiated with the same generic_actual_part.

Instantiated units are indistinguishable from ordinary units. All relevant coverage measures should therefore be collected for both types of units. For example, procedure call coverage measures should clearly be collected in both cases.

In addition, elaborating an instantiation effectively completes the process of type declaration and can produce errors. Coverage measures are necessary for this case also. Any knowledge of the instantiation can be obtained only by an examination of the instantiating statement and the generic unit, and any data about its behavior can only be collected by monitoring the original generic package or the point of instantiation.

The mechanism used to obtain coverage measures for procedures cannot readily be used for generic units. We shall discuss an appropriate method in the section on implementation[h].

---

[g] See[11] page 12.8.

[h] The Ada instantiation and generic package mechanism automates the process of type translation which might otherwise be achieved explicitly by a procedure which existed solely for that purpose. Monitoring procedure usage would, of course, be simpler in such a case, but the semantics of the linkage between a procedure pair might not be apparent. Ada makes the link explicit but complicates the measurement process.

Coverage Measures For Generics

We will discuss an number of measures for generic units, focusing on instantiation since it is this feature that makes the generic unit special.

I. The first dynamic measure for generic unit should be:

- **Elaboration Coverage for a Generic Unit**

** the number of instantiations elaborated for a generic unit divided by the total number of actual instantiations for this generic unit.

It is necessary because type checking can not be said to be complete until a declaration is elaborated, as we have already said.

This measure is essentially a procedure coverage metric, but it may also be used to provide an indication of the extent and nature of actual reuse of a particular generic unit, as distinct from that which was intended.

An equivalent static measure would be the number of instantiating statements for a given generic unit.

- **Total Elaboration Coverage for Generic Units**

If the average number of instantiations actually elaborated for all generic packages is large, it could mean that this system is making effective use of its subcomponents.

- **Generic Unit Instance Coverage**

From a test confidence point of view, exercising each instance of a generic unit is necessary since errors occurring in one instance may not appear in another. A test set which exercises a large proportion of generic instances without producing errors will raise user confidence significantly. We recommend two measures in this case, one for individual generics, and one for a complete system.

Individual instantiations may have different combinations of data types and operators as parameters as represented by their individual generic_actual_part. This is a major reason for insisting that each instance of a generic unit be exercised. Ada's overloading of operators can lead to a situation where a previously tested program unit may be used with new data types which are not valid for its semantics. However, the instantiation will not be invalidated if there exist operators for data types supplied. This is a sufficiently subtle problem to warrant special attention.

It is therefore important to collect coverage measures for data type usage which provide a perspective of how data types are exercised dynamically. We need to know the the structure of each unique generic_actual_part associated with each generic procedure, since this specifies a semantically unique instantiation.

• Unique Instantiation

** an instantiation of a particular executable generic unit is said to be unique if the general_actual_part specified differs in terms of actual type usage from that used in any other instantiation of that generic unit.

Based on this, we get the following coverage measure:

• Unique Instantiation Execution Coverage

** the number of unique instantiations being executed for a executable generic package divided by total number of unique instantiations declared for that generic package.

A high value of this measure means that the unique instantiations for the generic have been extensively tested. The tester may have increased confidence in the unit if no errors were detected.

It may also be useful to exercise a generic package with generic_actual_parts applying it to a variety of data types. This would require some prior knowledge of the problem domain for which the unit was intended. A testing philosophy of this type could lead to the certification of a generic package for a set of data type combinations.

The method of collecting this information will be discussed in the section on implementation.

Parameter Utilization

A generic procedure may have been created by extending the application of some existing procedure to a new type set. It will be necessary, therefore, to monitor the internal statement coverage, during subsequent testing, to ensure that the behavior of all operators are verified. One particular possibility is that the type of only one of a generic procedure's parameters may be altered. In this case, a tester will be interested in ensuring that all statements affected by that parameter are exercised.

We can use the method described by Rapp and Weyuker[19] to obtain the set of "all-use-paths" for a particular parameter[h]. Designing tests which ensure that the set of "all-use-paths" for a particular parameter were covered would ensure that all statements in which the parameter appears are exercised. This will ensure that any new uses of type conversions and operators are actually tested.

The following metric would indicate effectiveness of parameter use coverage test.

Parameter Usage Coverage

* the number of statements in which a parameter appears which are executed divided by the total number of those statements.

Tests aimed at collecting this metric would ensure that the impact of parameter type changes was evaluated.

A similar static measure could be proposed which would show the the extent to which a package was likely to be influenced by a particular parameter.

• Parameter Usage Factor

** the number of statements in which a parameter appears compared to the total number of statements.

TASK AND EXCEPTION COVERAGE

A complete system written in Ada may contain independent tasks. A full test of such a system may be the only satisfactory mechanism of evaluating task interaction, since task unit tests would validate the function of the task and not its user. A similar argument can be applied to exceptions; a unit containing exceptions can not be considered tested until it has raised all its exceptions. Coverage measures are therefore required for tasks and exceptions. In the case of the former, we would wish to know...

a) whether every task was activated,

b) whether every entry to a given task is used.

Coverage metrics can be defined readily for these cases.

An identical set of measures would be needed for exceptions.

Task Execution Sequence

While not a coverage metric, we advocate recording the task execution sequence. This will simplify the detection of errors due to poor synchronization strategies and allow the construction of tests designed to force particular task ordering.

IMPLEMENTATION

The statistics necessary for computing the coverage measures that we have proposed can be collected by the use of automatically inserted probes[i] written in Ada is under construction as part of the TAME project[7] environment.

---

[h] See also Weiser et. al.[25] and Weiser[24]

[i] See Brown and Hoffman[9] and Stucki[21, 22] for examples from other languages.

Structure test coverage execution requires two components: an instrumented program and an execution monitor. Probes are inserted into the source code to obtain the instrumented program. The execution monitor initializes the probes, records when they are executed, and reports on its results.

We use a generic probe package to implement the execution monitor. Elizabeth Katz first used this approach in a prototype coverage system, but that approach has been extended. As Fig. 1 shows, this package has two generic parameters and declares two procedures. Size (a parameter given to the generic package initiate_record) indicates how many probes have been added to the code. The increment procedure is called with a probe number in this range whenever the associated probe in the instrumented program is executed. A separate probe number is allocated for each unique object under observation. The report procedure reports on the results whenever it is called.

We will now consider how the nature of these probes varies with the various items being monitored and discuss their insertion in the program to be instrumented.

TABLE I - RELATIONSHIP BETWEEN TRADITIONAL AND ADA COVERAGE MEASURES

LANGUAGE AND	MEASURE	
SYSTEM FEATURE	TRADITIONAL	ADA
Statement Level	Statement, Path, Predicate Coverage	Statement, Path, Predicate Coverage
Procedural Level	Call and Procedure Coverage	Call and Procedure Coverage
Generic	——	Instance and Elaboration Coverage
System Partioning	——	Package, Inter and Intra Package Coverage
Task and Exception Coverage	——	Task and Exception Coverage Task Sequence

nstrumenting The Program

An instrument consists of two components: the robe and its monitor. The probe is inserted in the ode, and the probe monitor collects the data.

robe Monitor

Probe monitors will be implemented as generic package. They consist of variables which record values for some coverage measures whenever the procedure increment is called and compute the necessary statistic when the procedure report is called at the end of each unit's execution.

Inserting a probe into a generic unit would cause multiple instances of that unit to call a single probe monitor. Some difficulty would then be experienced in deciding which instance of the package had actually called the monitor.

```
generic
  instance_no : natural;
  c_table : record_type;
package probe_monitor is
  procedure increment(probe_number: 1..size);
  procedure report;
end probe_monitor;
```

Fig. 1. Structure of probe monitor

Using generic packages for probe monitors solves that problem since a new version of the monitor can be created with every instantiation of the the unit being monitored.

This is implemented by inserting an instantiation statement for the probe monitor inside the specification part of every generic package. Whenever a generic package is instantiated, a new instance of the probe monitor is generated. The variables in one instance will give coverage measures for one instance of a generic package. Therefore, we can easily distinguish the coverage measures of a generic package for different instances. The structure of the instrumented specification part for a generic package is shown in Fig. 2.a. The instantiation for such a generic package is given in Fig. 2.b.

There are two parameters passed to the probe monitor. The first is an instance number identifying the particular instantiation of the generic unit, and the second is a table represented by Fig. 3. The instance number is not used for non-generic units.

The table has two components. The first one gives the unit name (e.g. name of generic package), and the second one is an array which holds details of each probe active for that generic unit[1]. The details include the probe's usage, location, count, value, etc. Therefore, every instance of the probe monitor must be aware of the meaning of each of its probes and the total number in use.

The tables themselves are instances of the generic table shown in Fig. 4. Property_list is a record whose structure is determined by the number and nature of the properties of each probe as described in the bottom

[1] The non-generic units a treated as a single unit.

```
generic
 ...
 c_no: natural;
package XX is
 ...
 xx_monitor is new probe_monitor
  (instance_no=>c_no; c_table=>xx_table);
 ...
end XX;
```

a. The structure of instrumented specification part for a generic package

```
yy is new xx(...; c_no => xx_no);
```

b. The instantiation statement for the generic package

Fig. 2

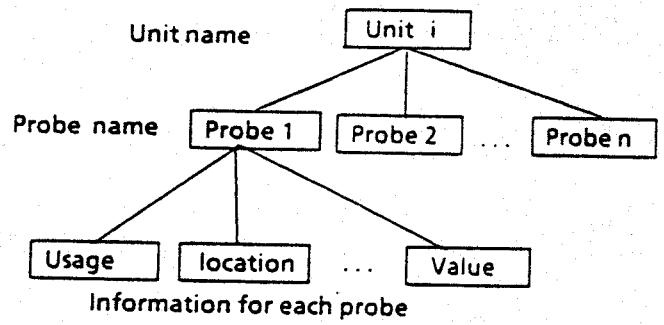

Fig. 3

row of Fig. 3. The generic table has the form shown in Fig. 4.

All instantiation statements for this generic table will be inserted at the beginning of the main program, since the number of probes needed for each program unit can be determined during compile time. One instance of the probe monitor can therefore collect the coverage measures for all non-generic program units or one instance of a generic unit.

Task Monitor

The task monitor is implemented by a task with one entry which receives the signals from tasks and records the sequence of task execution. Each task may have several entries which may be invoked by other program units (including tasks) and may also have statements which invoke entries in other tasks. Since the tasks can be executed concurrently, the invoking sequence may be very complicated. The task monitor records the invoking sequence for each system. The structure of the task monitor is given in Fig. 5.

The specification part of the task monitor will be inserted before the main program; therefore, the monitor can run concurrently with the main program.

The entry has one parameter for receiving signals. Whenever an entry in a task is invoked, the task monitor is invoked by a statement inserted at the beginning of the original task entry. This statement passes the name of this entry to the monitor.

```
generic
 unit_name : string;
 size      : natural;
package initiate_record is

 type table_type is record
       information : property_list;
       serial_no: natural;
 end record;

 type table is array [1..size] of table_type;

 type record_type is record
       name : string := unit_name;
       name_no_table : table;
 end record;

end initiate_record;
```

Fig. 4 Generic Table

```
task task_monitor is
 entry signal(name_of_the_calling_procedure)
end task_monitor;

task body task_monitor is
 ...
begin
 accept signal(name) do
      ...
 end signal;

 -- put the name in a list
      ...
end task_monitor;
```

Fig. 5 Structure of Task Monitor

Probes

The probes for recording non-task activity consist of a call to the increment procedure in the appropriate probe monitor and code to ensure that the call is made only once. As already discussed, the probes are inserted before or after the components to be measured and may contain code which determines whether the event being monitored actually occurred.

A structure coverage tool which collects a full range of Ada related measures is currently being constructed as part of the TAME project. The tool will implement the instrumenting concepts described in this paper and will be used to explore the impact of Ada on testing strategies.

CONCLUSION

We have discussed the impact that a language such as Ada has upon the traditional measures used for evaluating test effectiveness. In particular, we have drawn attention to the impact of Ada's system partitioning facilities, and its generic capabilities, and suggested new measures which recognize them explicitly.

We have proposed that some individual language features be counted explicitly during testing. The particular features differ significantly from those traditionally measured. Previously, ensuring that all statements and components were executed would be a reasonable goal. Applying those same criteria to Ada programs without explicitly considering these new features might lead to unjustified confidence in the results of some testing processes. A failure to distinguish between particular instantiations of generic units is a case in point.

We have also suggested that coverage measures may also be useful in determining a system's dynamic characteristics and as an indicator of its complexity.

Our future work will include a further investigation of these concepts.

ACKNOWLEDGEMENTS

The authors wish to acknowledge discussions with number of TAME project members and faculty in the Department of Computer Science at the University of Maryland. John Gannon and Mark Weiser acted as sounding boards and sources of information, and Elizabeth Katz and Dieter Rombach, assisted in clarifying a number of aspects of Ada's semantics.

We are particularly indebted to Ms. Katz who read several early drafts and assisted in constructing some of the test cases. Her assistance was invaluable and enabled the paper to be completed on schedule.

We thank them for their help, however, we must accept the responsibility for the opinions expressed herein, which are our own unless otherwise stated.

REFERENCES

1. V. R. Basili, E. E. Katz, N. M. Panililo-Yap, C. Loggia Ramsey and, "Examining the Modularity of Ada Programs," *IEEE Computer* **Vol. 18 No. 9** pp. 53-65 (Sep. 1985).
2. V.R. Basili and E.E. Katz, "METRICS OF INTEREST IN ADA DEVELOPMENT," *IEE-CS WORKSHOP ON SOFTWARE ENGINEERING TECHNOLOGY TRANSFER*, pp. 22-29 IEE COMPUTER SOCIETY PRESS, (1983).
3. V. R. Basili and J. Ramsey, "Structural Coverage of Functional Testing," *Computer Science Technical Report Series* **TR-1442"** Department of Computer Science. University of Maryland, (Sept. 1984).
4. V.R. Basili and D.M. Weiss, "A Methodology for Collecting Valid Software Enginering Data," *IEEE Trans. on Software Eng.* **Vol. SE-10 No. 6** pp. 728-738 (Nov. 1984).
5. V.R. Basili, "Quantatative Evaluation of Software Methodology," *Proc. First Pan Pacific Computer Conference*, Australian Computer Society, (Sep. 1985).
6. V. R. Basili and H. D. Rombach, "Tailoring the Software Process to Project Goals and Environments," *TR-1728*, Department of Computer Science, University of Maryland, (Nov. 1986). To appear in the Proc. of the Ninth International Conference on Software Engineering, Monterey,USA March-April 1987
7. V. R. Basili and H. D. Rombach, "TAME: TAILORING AN ADA MEASUREMENT ENVIRONMENT," *Proc. of the Joint Ada Conference*, (March 16-19 1987).
8. V. R. Basili and E. E. Katz, "Examining the Modularity of Ada Programs," *Proc. of the Joint Ada Conference*, (March 16-19, 1987).
9. J. R. Brown and R. H. Hoffman, "Evaluating the Effectiveness of Software Verification- Practical Experience With an Automated Tool," *FJCC AFIPS Conf. Proc.* **Vol. 41, Part I** pp. 181-190 (1972).
10. S.D. Conte H.E. Dunsmore and V.Y. Shen, *SOFTWARE ENGINERING METRICS AND MODELS.* The Benjamin/Cummings Publishing Company, Inc, Menlo Park, Cal. 94025 (1986).

11. US DoD. "REFERENCE MANUAL FOR THE Ada PROGRAMMING LANGUAGE," *ANSI/MIL-STD-1815-1983*, United States Department Of Defence, (Feb 17 1983).

12. K. Fujino. "Software Development for Computers and Communications at NEC," *IEEE Computer* **Vol.17 No. 11** pp. 57-67 (Nov. 1984).

13. W. Harrison K. Magel R. Kluczny and A. DeKock, "Aying Software Complexity metrics to Program Maintenance," *IEEE Computer* **Vol. 15 No. 9** pp. 65-79 IEEE Computer Society, (Sep. 1982).

14. W. E. Howden, "A Survey of Dynamic Analysis Methods," *IEEE Tutorial : Software Testing amd Validation Techniques*, pp. 209-231 IEEE Computer Society Press

15. T. McCabe, "A Complexity Measure," *IEEE Trans. Software Eng.* **Vol. SE-2** pp. 308-320 (Dec. 1976).

16. E. Miller, "Introduction to Software Testing Technology," *In Tutorial: Software Testing & Validation (E. Miller and W. E. Howden, eds.)* **2nd Edition** pp. 4-16 IEEE Computer Society, (1981).

17. Y. Mizuno, "Software Quality Improvement," *IEEE Computer* **Vol. 16 No. 3** pp. 66-72 (Mar. 1983).

18. J. Ramsey and V. R. Basili, "ANALYZING THE TEST PROCESS USING STRUCTURAL COVERAGE," *Proc. 8th International Conference on Software Engineering*, pp. 306-311 (August 28-30, 1985).

9. S. Rapp. and E. J. Weyuker, *Computer Science Department Technical Report* **Report No. 23** Department of Computer Science Courant Institute of Mathematical Sciences New York University, (Dec. 1981).

20. Jean E. Sammet, "Why Ada Is Not Just Another Programming Language," *Communication of the ACM* **vol. 29 , no. 8** p. 722 (Aug. 1986).

21. L. G. Stucki, "A Prototype Automatic Testing Tool," *FJCC AFIPS Conf. Proc.* **Vol. 41, Part II** pp. 829-836 (1972).

22. L. G. Stucki, "NEW DIRECTIONS IN AUTOMATED TOOLS FOR IMPROVING SOFTWARE QUALITY," *Current Trends in Programming Methodolgy, Vol.II R. T. Yeh (ed.)*, pp. 80-111 Prentice-Hall, Inc., (1977). 2nd ed. IEEE Computer Society Press"" also in "TUTORIAL: Software Testing & Validation" 2nd ed. IEEE Computer Society Press

23. D. Tajima and T. Matsubara, "The Computer Software Industry In Japan," *IEEE Computer* **Vol. 14 No. 5** pp. 89-96 (May 1981).

24. M. Weiser, "Program Slicing," *IEEE Trans. of Software Engineering* **Vol. SE-10, No. 4** pp. 352-357 (Jul. 1984).

25. M. D. Weiser J. D. Gannon P.R. McMullin, "Comparison of Structural Test Coverage Metrics," *IEEE Software* **Vol. 2 , No. 2** pp. 80-85 (Mar. 1985).

26. M. Zelkowitz, R. Yeh, R. Hamlet, J. Gannon and V. R. Basili, "Software Engineering Practices in the U.S. and Japan," *IEEE Computer* **Vol. 17 No. 6** pp. 57-66 (Jun. 1985).