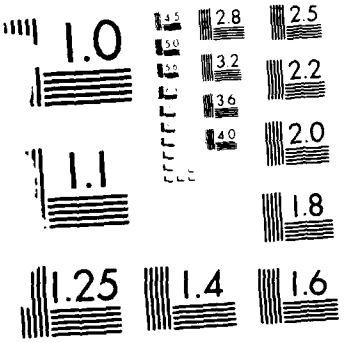
AD-A165 323 DEFINING METRICS FOR ADA SOFTWARE DEVELOPMENT PROJECTS 1/1
(U) MARYLAND UNIV COLLEGE PARK S B SHEPPARD ET AL
MAR 83 N00014-82-K-0225

UNCLASSIFIED F/G 9/2 NL



Proceedings / Computer Science and Statistics
Symposium on the Interface; Houston, Texas, March 1983

DEFINING METRICS FOR ADA SOFTWARE DEVELOPMENT PROJECTS

Sylvia B. Sheppard, John W. Bailey & Elizabeth Kruesi

General Electric Company
Arlington, Virginia

The Ada language is not just another programming language; Ada will be installed with a predefined environment that will provide programmers with a standard set of tools to perform their jobs. In addition to tools requisite for any environment (i. e., a compiler, an editor, etc.), tools may be provided for data collection. These data will provide feedback to the project's managers and programmers during the software development process and will predict the operational characteristics of the system under development. In the past, instances of data collection efforts on software projects have been erratic. However, with the commonality that will now occur for Ada environments, there is an opportunity to introduce systematic methods and procedures for measurement on a wide scale. Thus there is a need for the selection of metrics that are relevant and useful in an Ada environment.

Different metrics are useful for different purposes and at different times in the life cycle of a software development project. For example, project managers need measures to predict resources and schedules, to track costs and to locate potential problem areas. These needs differ from those of a customer or end user who would like some measure of the reliability of the system and proof of the thoroughness of the testing that has been done. Still different are the needs of a programmer who would benefit from feedback about the complexity of a just-compiled module. The problem of defining a unified set of metrics is not simple and straightforward. We must consider the needs of a variety of people who come into contact with a system over a long period of time.

Seven researchers from the University of Maryland* and from General Electric have joined in an eighteen-month collaborative effort to study this problem. Part of the effort has involved collecting measurements from an on-going Ada software development project. Our general approach has been to collect a great deal of information, to try to evaluate the different types of measures, and to select those that appear most useful. This paper will describe the software project selected, the data collection effort and the candidate metrics that are being considered.

*Members of the University of Maryland team are: Victor Basili, John Gannon, Elizabeth Katz, and Marvin Zelkowitz.

We chose a realistically large and complex software development project for study. It involved re-implementation of a portion of a working ground support system for a communication satellite. The original system was developed by General Electric and consisted of approximately 100,000 lines of FORTRAN and assembly code. A subset of the original system was selected for redesign and implementation in Ada. This subset is executable apart from the larger system. It includes functions to receive an operator's inputs, to perform complex computations and to display output graphically. Also included are several concurrent processes to monitor and display telemetry data from the satellite.

The project began in February 1982 with a month of formal training in Ada. Following this, the lead programmer and back-up programmer produced a requirements document describing the subsystem. The design was developed in an Ada-like program design language. Coding began in August and was completed in November, 1982. Approximately 10,000 lines of Ada source code, including comments, were produced. Some compilation has been done using the NYU Ada/Ed interpreter. Testing will be done as soon as a full Ada compiler is available.

Our approach to measurement for this project has been systematic and thorough. We began by defining goals for our data-collection effort. The goals fell naturally into three categories: those relating to software development projects in general, those relating to the use of Ada as a design and implementation language, and those relating to metrics for the APSE, the Ada Programming Support Environment. Following that, a number of specific questions related to each goal were developed. The goals and the questions relevant to each are listed in the Appendix.

Three methods of data collection were chosen to answer the questions associated with the goals. First, eight different data collection forms were adapted from those developed by Victor Basili and Marvin Zelkowitz for the NASA Software Engineering Laboratory.(1) The forms were designed to be completed by the members of the development team. The data collected on the forms provides a complete record of activities during the development process. The forms focus on three types of data: effort, changes and errors. These data will be discussed in detail below.

Second, an on-line procedure was developed for recording all versions of the design and the code for later analysis by a YACC-generated processor. The processor provides a static view of the system by counting such data as the types of Ada features used in each module, the data exchanged between modules, etc.

Third, the members of the programming team were interviewed after training and at the end of each major phase of the development cycle to determine what new concepts they had learned, what were their attitudes about Ada, and how those attitudes were changing.

In order to assure the quality of the data collected, the research team was diligent about cross-checking the data for accuracy. The results of this activity were surprising to researchers who had previously been involved in laboratory experiments. Techniques for controlled experimentation are well developed, and on-line programming experiments can be used to collect extremely precise time and error data. However, in this field study, we found a very different situation. Athough the programming team understood the necessity for recording the information precisely, there were wide variations in estimates. For example, one programmer recorded a time of one hour for a design walkthrough and another recorded a time of three hours for the same walkthrough, although each person attended the whole session. An inquiry revealed that the walkthrough on the design had been completed in an hour, but the group had remained together for an additional two hours in order to discuss methodology. Thus it was necessary to provide constant monitoring of the data collection effort at the site of the programming activity. In fact, all team meetings were attended by at least one member of the research team.

Collecting data with an on-line system would have relieved some of the problems we found with manual data collection procedures. Incomplete or inconsistent data values would have been more readily apparent. Further, on-line entry of data might have seemed more palatable to the programming team. The extensive data collection effort done here interfered to some degree with their progress of the project: filling out the forms required time and effort. On-line data collection might have been less apparent to the programming team, even though the same data would have been obtained. Finally, such a system would eliminate the need for later entry of the data into the data base. In our selection of metrics, we are considering the ease with which the data for the metrics can be collected and the degree to which it can be obtained unobtrusively.

A large number of candidate metrics have been defined and investigated. All of those that have a possibility of being useful are being evaluated. Area C, as shown in the Appendix, shows questions specifically related to defining metrics for the APSE. Some of these metrics may be useful in any software development environment, regardless of the design or implementation language used; others may be useful only for the Ada environment. Some metrics may provide enough information to make other measures superfluous. Two or more may reflect the same information, but the data for one may be more difficult to collect. Some metrics may not contribute useful information at all. Thus we are comparing the metrics in order to discard those that are not needed and to select a group that will be truly useful.

Data collection for this project has centered largely on two kinds of metrics, those dealing with the software development process itself and those dealing with the evolving product. Data collection for the process metrics include: a) the effort expended, b) the changes to the design or code, and c) the errors that occur.

For effort data, we collected the number of hours spent in various activities during the life cycle. We also recorded the names of the modules and the types of activity (e. g., design review or code reading) on which the time was expended. Because the human effort required to complete a project generally accounts for its largest single cost, effort data can be used to predict costs of future projects with similar parameters (e. g., size and application). Effort metrics are also useful for measuring productivity and for assessing the impact of new tools and techniques on productivity.

Every software development will undergo changes to previous documents (whether anticipated or not). Some development techniques, such as iterative enhancement, specifically assume that continual evolution is desirable. Others assume there will be a single pass through each phase with any further changes being carefully controlled. In either case, information about changes provides important insights into the project. Data collected about a change to our system included: the reason for the change, the time spent making the change, additional documents examined during the change, and changes to other documents because of the change. These data help determine the costs of various types of changes and assist in predicting the modifiability of the system. Change data are also useful for evaluating whether the development methodology is successful for the environment. For example, large numbers of major changes might indicate that a different development methodology should be used. In such a case, an incremental development approach might be beneficial in establishing the desirable features of the system. (2)

Data about errors included a description of the error, the activities used to detect the error, the time at which the error entered the system, and an evaluation of whether the error was related to the Ada programming language. These data provide information about the quality of

the product. We can also use error data to improve our methodology. If we know which types of errors are more difficult to correct, we can put an emphasis on trying to locate and correct those types of errors during walkthroughs or other reviews of the product. For errors related to the use of Ada, error data can be used to determine where emphasis is needed in future training courses.

Product metrics are the other major type of metric we have been examining. They deal with the characteristics of the software itself and can be divided into two categories: static and dynamic. Static metrics can be collected from the software at any point in its development. Dynamic metrics are run-time measures. There are a myriad of measurements that can be collected for both categories; only a few representative ones are discussed below.

Static metrics are useful for determining the complexity and quality of the code. Collecting such metrics at several phases of the life cycle provides a view of the way in which the system changes across the phases. Static metrics can be subdivided into three types: size, control and data metrics. Size metrics are indicators of the volume of a product and the amount of work performed. They correlate well with effort and are used for estimating costs, comparing products and measuring productivity.(3) Lines of code are frequently used to measure size. However, there are alternate ways to count lines of code. Comment lines may be included or omitted, depending on the use of the metric. If we are measuring productivity, we probably want to include both code and comment lines. However, the self-documenting features of the Ada syntax may produce a smaller ratio of comment lines to code lines than is usually considered good programming practice in other languages. Further difficulties may arise when computing the size of a system which incorporates previously written components or is a modification of a previously developed system. Should productivity measures include only new development work or should they reflect all of the code that is delivered? Certain features of the Ada language (i. e., packages, generics) will make possible large libraries of reusable components that can be incorporated into a new system much the way in which circuits are now constructed with off-the-shelf chips. This new trend in software design, as well as the self-documenting capabilities of the syntax, will necessitate fresh approaches to size measures which are meaningful for Ada.

Control flow metrics measure the complexity of a product. McCabe's v(G) is a count of the number of basic control path segments in a computer program.(4) This value depends on the number of decision nodes and the branches emanating from those nodes. V(G) was originally developed as part of a strategy for testing software. However, it has been suggested that v(G) is also useful as a measure of psychological complexity.(5) The theory is that a program with many decisions is psychologically complex: the more decisions, the more difficult a program is to understand and modify. One exception to this is a case statement where there is one path for each of several similar choices. McCabe suggests limiting v(G) to 10 for any module except where v(G) is inflated by a large case statement.

The current method of counting v(G) may not be sufficient for the Ada language. Ada's explicit structures for exception handling alter the control flow normally found in structured languages. Additional paths can be explicitly generated to handle situations that would cause faulty execution in other languages. For example, in some other languages, an attempt to read past an end-of-file marker would produce an error message and would terminate the job stream. In Ada it is possible to specify the means to handle this possibility and to continue processing. This exception handling, however, alters the normal flow of control of the program, and it is necessary to consider alternatives for calculating v(G). One method would ignore the potential occurrence of exceptions. A second method would account for all possible paths of execution as the result of the occurrence of an exception. The first alternative alters the premise that all basic paths through the module are being counted and tested. The second alternative is theoretically more appealing, but it would greatly increase the number of paths and would thus make testing very difficult.

Data metrics analyze the organization of the data structures within and between modules in an attemot to measure the ease with which they can be understood and modified. Data metrics of interest include the number of average live variables per statement, the percentage of global variables, and the number of programmer-defined types. Some data metrics, such as the information flow metrics of Henry and Kafura, focus on the interconnections between system components.(6, 7) Myers has indicated that interfaces are important because many serious, hard-to-find errors in systems result from a lack of understanding of module interdependencies and from changes made to global data areas.(8)

More work is also needed in the area of data metrics for Ada. Is the density of data flow across modules as proposed by Henry and Kafura a useful metric for the Ada language? The Ada block structure, visibility rules and packaging features all affect the way in which data is apportioned and the degree to which data is or is not visible at various locations in the code. Thus there is a need to define how to measure the number of global and local data structures and how to measure the quality with which the data structures are encapsulated in packages. These measures should be helpful in

determining whether the encapsulation makes the best use of Ada's features for producing maintainable, reuseable software.

Dynamic metrics provide another method for measuring the software product. Dynamic measures can be divided into execution and test coverage metrics. Execution metrics are useful for tuning a system to make it run efficiently. Execution metrics include the number of times each statement is executed and the CPU time used.

Test coverage metrics help determine the degree and quality of the testing that has been done. McCabe's metric has already been discussed. Other candidate test coverage metrics include the number of statements executed and the number of branches executed. Because Ada has mechanisms for concurrent processing, metrics are needed to measure tasking features and usage. Further, concurrent processing presents opportunities for both starvation and deadlock; methods are needed for predicting the possibility of these occurrences during the testing phase of the life cycle.

The analysis of the data from this project and the subsequent selection of a suggested set of metrics for an Ada environment is currently underway. The systematic approach to the goal-driven data collection effort will provide a beginning methodology for subsequent efforts for monitoring software projects. The metrics selected will be useful for all of the differing needs of those associated with an Ada system throughout the life cycle. If we can automate and insert a common set of metrics into the APSE, data collection may become an integral part of software methodology, and comparisons across Ada projects of all types will become feasible. Further, quantitative indices of the progress of a project will be available for managers, procurement officers, designers and others with a need for such information.

APPENDIX

Area A: Generic goals for any software development project

Goal A1: Characterize the effort in the project.
1) How was the effort distributed over the phases of the project?
2) How was the effort for the project distributed over time?
3) How was the effort distributed across different functions in the software?
4) How are the error distributions similar to or different from other comparable software developments?

Goal A2: Characterize the changes.
1) How are the changes to the system distributed over the software development cycle?
2) How is the time for handling a change distributed? How long does it take to design and implement the change?
3) Are certain features of Ada or certain types of errors associated with particular programmers? Why?
4) Do certain programmers have problems with certain aspects of the language?
5) Do programmers want to use features available in other languages that are not available in Ada?
6) Are some features of the language overused, used incorrectly, or used inappropriately in the programmers' enthusiasm to use what they have learned?
7) Do people with no previous high level language experience have more or fewer problems with Ada than people with high level language experience?

Area B: Goals relating to Ada as a design and implementation language

Goal B1: Characterize the errors made.
1) How were the errors found? (e.g., design review, inspection of output, etc.)
2) What were the non-Ada causes of the errors? (e.g., requirements misinterpreted, mistake in computation, etc.)
3) What features of Ada are commonly involved in errors?
4) Are there features of Ada that cause problems when they are used together?
5) Are errors attributed to confusion with another language? to a lack of understanding of Ada? to a lack of experience with a feature?
6) Are the errors made when using Ada as a design language different than those made when coding?
7) Where was the information found that was needed to correct the error? (e.g., Ada Reference Manual, another programmer, etc.)
8) Is the error characteristic of the feature or of the particular application it involved?

Goal B2: Determine whether certain aspects of Ada are difficult to use for certain applications.
1) Are there certain aspects of Ada that do not apply to this type of project?
2) Are there techniques usually used for this type of application that are difficult to implement in Ada?

Goal B3: Determine which aspects of Ada contribute positively to the design and programming environment.
1) Are errors easy to find? to correct?
2) Is there a large amount of parallel development once the interfaces are defined?
3) How effective is Ada in reducing interface errors? producing software that is easy to change? reducing the development effort, especially in realtime problems?

Goal B4: Determine which combinations of Ada's features are naturally used together.
1) How fully is the language used?

2) Are there certain features of Ada that are avoided because they are difficult to learn? difficult to use? poorly implemented? error prone?

Goal B5: Determine the effect of using Ada as a PDL.

1) Does Ada PDL allow sufficient abstraction at the early stages of design?
2) Is the language really being used as a design language?
3) Does the use of Ada PDL cause a preoccupation with syntax during the design stage?
4) What is the expansion of Ada PDL to code?
5) Does Ada PDL guide the design of the project or are portions of the system primarily other language programs written in Ada syntax?
6) Is there an adequate combination of features of Ada for use as a PDL?
7) Are the most expensive errors found while using a particular set of features of Ada as a PDL?
8) Are errors uncovered at the design stage that ordinarily would have been uncovered during coding because of the use of Ada PDL?
9) What percentage of the interface errors are uncovered during the design stage?

Goal B6: Characterize the programmers and associate their background with their use of Ada.

1) What are the programmers' opinions of Ada before they begin this project? during? afterward?
2) What is each programmer's background with other languages?
3) Is there a relationship between how far into development the change was needed and how much effort was spent on the change? How many sections it affected?
4) What kind of changes were made? (e.g., error correction, planned enhancement, etc.)
5) How many components are involved in the typical change?
6) How many changes are caused by a previous change?
7) How was the need for change determined?
8) How many and what kind of interface changes need to be made?

Area C: Goals Relating to Metrics for the APSE

Goal C1: Select a set of static (size, control and data) metrics for the APSE

1) Are there differences in the implications of various counting measures? Are some measures more useful than others?
2) Do certain program measures provide enough information to make other measures superfluous?
3) Which static metrics can be applied throughout the design and code phases. Which cannot?
4) Which static metrics help predict run-time behavior (e.g., reliability, etc.)?
5) Which static metrics can be measured most easily?

Goal C1.1: Develop a set of size metrics for the APSE

1) What size metrics best predict effort?
2) What serves as a useful size metric (e.g., lines of code, modules) in Ada?
3) What constitutes a statement in Ada?
4) How should an executable statement be defined in Ada?
5) What features of Ada should be grouped when counting the number of times certain features are used?
6) How useful is Halstead's software science approach with Ada?

Goal C1.2: Develop a set of control metrics for the APSE

1) How can tasking and exceptions be integrated into the control metrics?
2) How useful is McCabe's cyclomatic complexity measure? How does the cyclomatic complexity compare with the essential complexity?
3) How useful are measures of nesting complexity and depth?

Goal C1.3: Develop a set of data metrics for the APSE

1) How can the complexity of data structures be measured?
2) What influences the number of programmer defined types?
3) How does the use of Ada influence the number of inputs to and outputs from a module?
4) How does Ada influence the use of global data?
5) How does the use of modules affect the treatment of data within a program?
6) How should the span of a variable be measured? Is there a use for the span information?
7) What do the data bindings suggest about the structure of the system?
8) Does the density of the data flow across modules provide useful feedback about the structure of the system? i.e., are information flow metrics (Henry & Kafura) useful?

Goal C2: Select a set of dynamic (test coverage and execution) metrics for the APSE

Goal C2.1: Develop a set of test coverage metrics for the APSE

1) Do any of the following measures of test coverage lead to a useful strategy for testing: number of statements executed? number of decisions executed, or number of independent paths executed?
2) Can these measures be extended to provide test coverage for concurrent processing or will new measures need to be developed? Are there measures to detect starvation, potential deadlocks, etc.?
3) Are there other features of Ada (e.g., exception handling) that require new measures for test coverage? What are those measures?

Goal C2.2: Develop a set of execution metrics for the APSE

1) What are useful execution metrics?
2) What additional information do execution statistics provide beyond what can be gained from a static view of the system?
3) Are there measures of execution complexity?
4) Are certain Ada features or combinations of features expanded into very fast or very slow code?

Goal C3: Develop a subjective evaluation system for evaluating some program and design features that are not easily or practically measured in other ways

1) Can a diverse set of experts (Ada, applications, and methodology experts) accurately evaluate the subjective aspects of the project?
2) How well do the results of these evaluations correlate with results from objective measures?
3) How well do these evaluations correlate with the opinions of the development team?
4) Can we conclude anything from the subjective results?

Acknowledgements

The categorization of the metrics used on this project was developed by Victor Basili and other members of the University of Maryland team. The software development methodology employed benefited from their direction. The authors are grateful to Victor Basili, John Gannon, Elizabeth Katz and Marvin Zelkowitz for their help. This research program is monitored by the Office of Naval Research (ONR) under contract #N00014-82-K-0225 to the University of Maryland with funding from ONR and the Ada Joint Program Office. The views expressed in this paper, however, are not necessarily those of the Office of Naval Research, the Ada Joint Program Office or the Department of Defense.

REFERENCES

[1] Basili, V. R. and Zelkowitz, M. V., Analyzing medium-scale software development, in Proceedings of the 3rd International Conference on Software Engineering (1978) 116-123.
[2] Basili, V. R., Changes and errors as measures of software development, in Basili, V. R. (ed.), Models and Metrics for Software Management and Engineering, Computer Society Press (1980) 62-64.
[3] Basili, V. R., Product metrics, in Basili, V. R. (ed.), Ibid., 214-217.
[4] McCabe, T., A complexity measure, IEEE Transactions on Software Engineering, 2 (1976) 308-320.
[5] Curtis, B., Sheppard, S. B., and Milliman, P. Third time charm: stronger prediction of programmer performance by software complexity metrics, in Proceedings of the 4th International Conference on Software Engineering (1979) 356-360.
[6] Henry, S. and Kafura, D., Software structure metrics based on information flow, IEEE Transactions on Software Engineering 5 (1981) 510-518.
[7] Kafura, D. and Henry, S., Software quality metrics based on interconnectivity, The Journal of Systems and Software 2 (1981) 121-133.
[8] Myers, G. J., Software Reliability: Principles and Practice (Wiley-Interscience, New York, 1976).

DTIC

FILMED

4-86

END