METRICS OF INTEREST IN AN ADA DEVELOPMENT*

Victor R. Basili
Elizabeth E. Katz

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

Abstract

The emergence of Ada provides the opportunity and necessity for measurement, analysis, and experimentation in software development. Over the past several months, we have been studying a software project developed in Ada. One of the goals of the study is to identify metrics which are useful for evaluating and predicting the complexity, quality, and cost of Ada programs. This paper defines a set of metrics for use with software development in Ada. The metrics are gathered into six categories: effort, changes, dimension, language use, data use, and execution. They are described further using formula generators, distributions, and formulas. Examples of each metric, as well as specific uses, are also included. Finally, our continuing research in this area is described.

Introduction

Over the past several months, the University of Maryland and General Electric have been collaborating on the study of a software development project written in Ada. The purpose of the study is to gain insight into how a typical Ada development might proceed so that we might make recommendations about training, methodology, data collection, and metrics to the future Ada users community.

The project we are monitoring involves the redesign and implementation in Ada of a portion of a satellite ground control system that was originally written in FORTRAN. One of the goals of the project is to investigate how programmers with varied backgrounds react to Ada; therefore, four team members, each with a different background, were chosen to develop the software. The chief programmer is experienced with the application area but has used only FORTRAN and assembly language. The backup programmer, also familiar with the application area, has used FORTRAN, assembly language, COBOL, PL/I, LISP, ALGOL, and SNOBOL. The third programmer has a recent B.S. in computer science and is fluent in Pascal and other block-structured languages. The librarian has no previous experience except for a brief exposure to FORTRAN. None of the team members had any prior exposure to Ada.

The goal of the training program was to give the programmers the type of instruction in Ada that might be provided in industry. The training program lasted one month. It began with twenty-one videotaped lectures by Ichbiah, Firth, and Barnes. Although the tapes themselves contain fifteen hours of material, the team members spent four days watching them, asking questions, and discussing points of general interest. The videotapes were followed by an on-site seminar taught by George W. Cherry of Language Automation Associates. The six days of seminar were spread over four weeks to allow the team members to practice writing, compiling, and executing Ada programs, to review their class notes and other reference materials, and to work together on a 500-line group project.

In addition to the Ada training, the team members were given a half-day class on methodology taught by Victor R. Basili of the University of Maryland. This class was designed to incorporate software engineering techniques into the environment and included such topics as chief-programmer teams, design and code walkthroughs, program librarians, and structured programming.

After the training was completed, the team used an Ada-like PDL to design the system from slightly revised specifications of the existing system. They were not allowed to look at the existing system so as to not bias their design. The design and coding phases of the project occurred between April and December 1982 with most of the Ada processing done using the Ada/Ed interpreter.

Case Study Organization

This case study is driven by a set of goals that were developed to answer the question of why are we gathering the data. By setting out with a set of goals, we know why each piece of data is collected and how each piece is supposed to help us evaluate the project. This approach should also assure us that we are gathering all of the data that we need. The goals also provide an organization for the data collection process. The approach consists of six steps: 1) the development and categorization of a set of goals, 2) the development of a set of questions of interest or hypotheses based upon those goals that attempt to quantify the abstractions of

*This work was supported in part by an Office of Naval Research Grant #N00014-92-K-0024 to the University of Maryland.

Ada is a trademark of the Department of Defense (Ada Joint Program Office)

the goals, 3) the development of metrics and data distributions that answer the questions, 4) the development of forms and other mechanisms for collecting the data, 5) the actual data collection process, and 6) the validation and analysis of the data. A detailed discussion of this approach to data collection can be found in [Basili & Weiss 82].

The primary goal of this endeavor, to study an Ada project in order to make recommendations, was broken down into more specific goals to guide the study. These goals are divided into four major categories: changes and effort goals, Ada and PDL goals, data collection goals, and metric goals. Each goal within a category is associated with a series of questions whose answers might help meet that goal. These questions cannot include every question one might ask, but they are meant to be representative of the questions of greatest interest. A list of data sources is listed after each question to guide the data collection process. This data might be collected from forms, static analysis programs, evaluations by various people, or other sources.

One of the goals of the project is to generate a set of metrics for the APSE. The goals provide the purpose, but the metrics, based on the goals, provide the precision to analyze the data. We plan to define the metrics and then collect the data to show that this approach is worthwhile. The descriptions of the metrics and their definitions are given later in this paper.

Note that the metric goals are quite different from the other categories of goals. They interact and are orthogonal to the other goals. Therefore, although this paper concerns the metrics in particular, the complete list of goals is given in Appendix I. Also note that this set of goals is by no means static. It changes as new aspects of the project are discovered. However, the cross-references are kept up-to-date, and in general, the changes are additions and revisions for clarity, rather than deletions.

The source of much of the effort and change data is a set of forms that were developed to gather information about the software development process. Most of these forms were adapted from the NASA Software Engineering Laboratory [SEL 82]. They include a change report form, component status report (which records effort per component), review and walkthrough reports, and extensive error description reports. Most of these forms are completed by the programming team and validated by the monitoring team before their entry into an automated database. The questions of the goals and the information gathered from the forms are cross-referenced for traceability. The set of metrics described later in this paper are also cross-referenced with the goals and data. The forms, however, are not the only method of data collection. In addition, we will use some other tools to examine the code in detail, to perform data flow analysis, and to experiment with slicing [Weiser 81].

The Metrics Definition Process

The metrics, as described above, provide the precision for the data analysis while the goals provide the purpose. Just as the goals are not static, the metrics list changes as we learn more about what we are studying. This list will not cover all metrics that people might find necessary, but it is meant, at this stage, to be one in a series of iterations which will eventually become the recommended list.

It seems reasonable to pattern the structure of the metrics on the structure of the goals, but this structure does not provide the precision needed. Therefore, a tree structure is developed from the general concepts underlying the measures into the details of particular measures, how they might be collected, and most importantly, how they might be used.

At the first level of this metric tree, the measures are placed in six categories: effort, changes, dimension, language use, data use, and execution. Each node at this level is labeled with a general name followed by an explanation of what it is and why it is important. These categories encompass a particular concept that may be measured in more than one way.

The second level contains one or more of three elaboration schemes. The first of these is a formula generator. A formula generator is a parametrized formula which can be used to represent a variety of similar formulas. This generator allows a large number of measures of similar form to be defined implicitly by a general formula. If this elaboration is used, examples of its use will be given at the third level. The second form of elaboration is by distribution. A general measure might be described by a variety of distributions which capture various aspects of the data under study. Distributions encompass all objects being studied but separate them into a variety of disjoint categories. Distributions may be closely related to the third form of elaboration, the formula. Formulas are the most standard of the elaborations used in this list. A particular measure may be defined in many different ways by different people. Each formula is a description of one of those ways. Each formula may be a tree which provides further elaboration if one level is not sufficient. A formula may be related to a distribution in that only a particular aspect of the distribution is examined. In the list below, effort is an example of the use of a formula generator, changes is an example of the use of distributions, and size is an example of the use of formulas.

After the elaboration, examples will usually be given. Then, each measure is accompanied by a series of uses for which the measure might be important. Each use is accompanied by a reference to the literature, where applicable. These references may be used as an indication of how the measure has been used in other studies. Further information on other measures may be found in [Basili 80]. Suggestions are given for how the data might be collected for the measure, and finally, the measure is cross-referenced to the goals. In this way, the list of

measures can be viewed on a variety of individual levels without losing the perspective of the entire metrics scheme.

The format for the list entries is given below.

name of general measure
  explanation: A brief statement about the importance, use, or problems with this measure.
  elaboration: A formula generator, distribution, and/or formula is described in detail.
  derived measures and/or specific examples:
  uses/references
    i) first use,[reference]
    ii) second use,[reference]
  how collected:
  goals: Which goals does this measure help attain?

Several abbreviations are used to represent the various forms and methods used to gather data. They are:

AN: analysis of the programs
CRF: change request form
CSF: component summary form
CSR: component status report
EDF: error description form
IDC: individual document change report
P: programming team members
Q: subjective questionnaire
REV: review and walkthrough form
RSF: resource summary Form

EFFORT
  EXPLANATION: Effort can be categorized by the phase of the project, it can be associated with a particular activity within the project, and it can be associated with a particular part of the program.
  ELABORATION: One method of representing the numerous effort measures is through a formula generator. eff is similar to a procedure which takes four parameters: time units, personnel units, activity units, and program units. It represents the effort in time units for personnel to perform the activity for the program. Another way to view the representation is as a sum as in the equation below
  eff(time, personnel, activity, program) =
    SUM over activity(program) by personnel in time units. One might think of time, personnel, activity, and program as enumerated types and of eff as a procedure that computes various combinations of the members of the types. The types are defined as:
    time: minutes, hours, weeks, months, years
    personnel: programmer, librarian, support staff, reviewers, all
    activity: requirements, design, coding, testing, maintenance, training, reviewing, making changes, meetings, forms, methodology, librarian, other, all
    program: module, subsystem, system
  The all-encompassing personnel designation is "all." A particular programmer can be specified by "programmer(name)," and all programmers is written "programmer(all)." Sometimes a particular activity, such as a review, involves more than one person. If these people can be described as a class, such as "reviewers(exec)," they can be used as a personnel designation. The all-encompassing activity is "all." When some activities overlap, the overlap only counts once in "all." The all-encompassing program is "system." When module or a subsystem is a parameter, it is written "mod(module name)" or "sub(subsystem name)." If just the type name is given, one of the members of the type should be used. This is an important shorthand for time units in particular.
  EXAMPLES:
    total programmer effort in months =
      eff(time: months, personnel: programmer(all), activity: all, program: system)
    programmer effort for design of the system =
      eff(time: time, personnel: programmer(all), activity: design, program: system)
    effort to review exec module =
      eff(time: time, personnel: reviewers(exec), activity: review, program: mod(exec))
    % total programmer effort spent in design =
      eff(time: time, personnel: programmer(all), activity: design, program: system)
     / eff(time: time, personnel: programmer(all), activity: all, program: system)
  USES
    general references for the measure: [Walston & Felix 77], [Putnam 78], [Belady & Lehman 76]
    If examining by phase activity,
    i) determining where time was spent
    ii) pinpointing milestones
    If examining by programmer activity, such as training, methodology, or making changes,
    iii) determining the importance of that activity
    iv) pinpointing which activities might need improvement
    v) comparison with other projects
    vi) prediction of future effort
    vii) estimate cost
  HOW COLLECTED: CSR, CRF, IDC, EDF, RSF.
  GOALS:I.1, I.2, I.6, II.6

CHANGES
  EXPLANATION: Changes can be described by a variety of distributions and their associated formulas. Changes are studied to determine where improvements may be made in the methodology so that fewer changes occur and to determine which types of changes are the most costly so that they can be avoided. The study of the change data can also point out abnormalities in the project.
  ELABORATION: Each of these distributions has a number of categories which describe the change in a particular manner. A graph of the distribution would show how many changes were of each category such that the total graphed would equal the total number of changes. A distribution is referred to as a type, and a category within the type is called a mem. There are two similar notations for changes and errors. The notation chg(type, mem, program, activity) is used to represent the number of mem changes using the distribution type made to the program during activity. For example, the number of planned enhancements made to exec throughout the project would be written as chg(purpose, planned, mod(exec), all). The number of clerical errors made in exec throughout the project would be written as err(kind, clerical, mod(exec), all).

The change types and their members are:

purpose of change: error correction, planned enhancement, implementation of requirements change, improvement of clarity, maintainability, or documentation, insertion or deletion of debug code, optimization of time, space, or accuracy, and other. [SEL 82]

how need for change determined: during a walkthrough, planned, requirements changed, while making another change, or testing. [SEL 82]

The error types and their members are:

kind of error: requirements misinterpreted, functional specification incorrect or misinterpreted, design error, misunderstanding of external environment (not language), error in use of PDL, error in use of programming language, error in use of compiler, clerical error, other.

abstract error categories: initialization, control structure, interface, data, computation. [Basili & Perricone 82]

omission or commission: Errors of commission are present as a result of an incorrect executable statement. Errors of omission result from forgetting to include some entity within a module. [Basili & Perricone 82]

how the error was discovered: design reading, design walkthrough, code reading, code walkthrough, talks with other programmers, reading documentation, system error messages, project specific error messages, trace, dump, inspection of output, pre-acceptance test runs, acceptance testing, Ada's run time checking, other.

Program and activity are defined as in effort. As a notation for the data as a distribution, DIST is similar to mathematical summation. Instead of adding the terms, however, a set of ordered pairs is created. Each pair consists of the index and the value of the term evaluated at that index. The first example below shows the use of DIST.

EXAMPLES

changes distributed by purpose:
DIST for all i in purpose chg(type: purpose, mem: i, program: system, activity: all)

number of clarity changes in module exec:
chg(type: purpose, mem: improve clarity, program: mod(exec), activity: all)

number of error corrections during design
chg(type: purpose, mem: error, program: system, activity: design)

number of changes found in walkthroughs:
chg(type: need, mem: walkthroughs, program: system, activity: all)

number of requirements changes during testing:
chg(type: purpose, mem: requirements change, program: all, activity: testing)

number of errors discovered by design reading:
err(type: discovered, mem: design reading, program: system, activity: all)

number of errors occurring during testing:
SUM all i in kind of err(type: kind, mem: i, program: system, activity: testing)

number of errors of commission in module exec:
err(type: com/om, mem: commission, program: mod(exec), activity: all)

distribution of errors in exec by kind of error:
DIST for all i in kind of err(type: kind, mem: i, program: mod(exec), activity: all)

USES

i) to predict the effect of future changes [Belady & Lehman 76]

ii) to estimate the reliability of the program [Weiss 81], [Endres 75]

iii) to determine which methods of error detection are most useful

iv) to determine where further instruction may be needed

HOW COLLECTED: CRF

GOALS: I.2, I.3, I.4, I.5, I.6, II.3, II.6

DIMENSIONS

EXPLANATION: Dimensions include all measures of how big the project will be or is. It may be used to predict effort both from the staffing and duration standpoints. It may also be used as a measure of complexity. One other use of this measure is to define the object over which other measures will be made, e.g. errors per lines of source.

ELABORATION: The definitions below are varied; therefore, a brief description is given before each of them. Note that the Halstead measures under the "Use of Ada" heading are also measures of dimension.

lines: A line is a series of characters terminated by a carriage return. Several types of lines are defined as follows:

lines of source: A line of source is a line in the delivered product.

comment lines: A comment line is a series of characters between a "--" and a carriage return.

PDL lines: A PDL line is a line, within a PDL module, which is neither blank nor starts with a "--".

Ada lines: A Ada line is a line, within a Ada module, which is neither blank nor starts with a "--".

statements: A statement is an entity of the language which serves a particular purpose. A statement may extend over several lines, and more than one statement may occur on a line. Several types of statements are defined as follows:

executable statements: An executable statement is a simple or compound statement which would cause an action or holds the place for an action which takes place at run time. If a computation must be done at run time in a declaration, the declaration should be counted as an executable statement. An initialization option is executable even though the compiler may be able to perform the initialization at compile time. A null statement is executable. Extra words, such as "end loop," are not counted as separate statements.

declaration statements: A declaration statement is a statement which describes the type or representation of some program entity such as a type, a variable, or a procedure.

statement nesting Statement nesting is a measure of how deeply nested the statements of the program are nested. In Ada, the following statements may be nested within one another: accept, block, case, if, loop, and select. Unnested statements have a nesting level of one. If a nestable statement has level n, then statements nested within it have level n+1. Two aspects of this measure are of particular interest.

average statement nesting: This measure is the sum of the statement nesting at each statement divided by the total number of statements.
maximum statement nesting: This measure is the maximum value of statement nesting within the program unit.

program units Program units can be a module, a subsystem, or the entire program. Entire program and module will probably be used most often.
A module in Ada is either a subprogram, a package, or a task. In other languages, other program entities may be defined as modules.
A subsystem is a set of related modules within the program. If subsystem is used in a measure, it should be defined precisely before use.
Entire program is the union of all the modules or of all the subsystems.

USES
i) characterize the project
ii) measure how much memory the program will need [Halstead 77]
iii) comparison with other projects
iv) as a predictor of effort [Walston & Felix 77], [Freburger & Basili 79]
v) as a predictor of cost [Belady & Lehman 76], [Freburger & Basili 79], [Basili & Reiter 79]
vi) as a measure of complexity [Halstead 77], [McCabe 76], [Basili & Reiter 79]
vii) as the common denominator for measurements of number of errors, changes, and such per lines of source

HOW COLLECTED: "wc" on the Vax, instrumented language processor, by hand
GOALS: II.4, IV.3, IV.2.

THE USE OF ADA
EXPLANATION: Because Ada is a new language which will be used extensively, one would like to know how it will be used in order to better train one's programmers. Few people are using Ada today, and no one is sure of how it will be used in the future. Some people have claimed that Ada has too many features and that no one will use all of them. Others claim that all of the features of Ada are necessary. Still others claim that programmers with different backgrounds will use different subsets of Ada's features. The following subset of metrics applies to Ada specifically. These measures might show how Ada is being used, and where training might focus to obtain the best results.
ELABORATION: The measures of the use of Ada can be organized into three groups: measures of the use of Ada features, various distributions of the use of Ada constructs, and Halstead measures. Each of these groups is described below with an elaboration and examples.

THE USE OF ADA FEATURES
elaboration: A feature of Ada is a semantic entity which captures a specific concept of the language. Examples of Ada features are tasking, exception handling, and overloading. Unfortunately, the use of some of these features cannot be measured. The following measures are examples of how one might quantify the use of some of Ada's features.

tasking
number of tasks served by this task
number of potential customers
number of active customers: Both the average and maximum measures could be studied.
number of statements between entry and rendezvous: This is a measure of the amount of shared code.
number of tasks working in parallel: This is a measure of the amount of concurrency in the program.
exception handling
number of unique exception handlers
number of programmer-defined exceptions
total number of exception handlers
average number of exception handlers per program unit

CONSTRUCT DISTRIBUTIONS
elaboration: Just as changes are described by distributions, this measure uses a variety of distributions to describe how each programmer and the team as a whole uses Ada. In addition, one of the distributions is a distribution of the others and describes which constructs were involved in errors. Here, a construct is a module, statement, or other syntactic entity. Some features, as defined above, might also be considered constructs. Constructs may also be grouped together for other distributions. The notation for the distributions is Ada(type, construct, programmer) where type is one of the distributions, construct is one or all of the constructs in the distribution, and programmer is the one or all of the programmers.
Some types and their elements are:
module: subprogram, package, task
statement: declaration, executable
declaration: exception, number, object, package, pragma, renaming, subprogram, subtype, task, type
executable: abort, accept, assign, block, case, code, delay, exit, goto, if, loop, null, pragma, proc-call, raise, return, select
error: other types are the elements and can be further specified.
examples:
distribution of executable statements:
DIST for all i in executable Ada(type: executable, construct: i, programmer: all)
distribution of errors in declarations:
DIST for all i in declaration Ada(type: error, construct: i, programmer: all)
distribution of module by type:
DIST for all i in module Ada(type: module, construct: i, programmer: all)

HALSTEAD MEASURES
elaboration: Halstead's software science measures are derived from the counts of operators and operands as defined below. Only the executable statements of the program are analyzed for these measures. See the definition of executable statement in "lines of code." Note that these definitions are specific to Ada because the Halstead measures deal with the detailed use of the language. An in-depth description of each of these measures is given in [Halstead 77].

operator: An operator is a syntactic element which acts upon a syntactic object. The verbiage of control statements is reduced to a single operator. Grouping operators, such as parentheses, a counted as a unit. Attributes are considered operators; for example, "a'range" would be the operand "a" and the operator "'range". A complete list of the operators for Ada is given in Appendix II.

operand: An operand is a data object which is manipulated by operators. For example, "name.a" is two operands with operator ".". For counts of unique operands, each subpart of a data structure will be counted as a unique entity.

derived measures

n1: number of unique operators

n2: number of unique operands

N1: total usage of all operators

N2: total usage of all operands

Vocabulary: n = n1 + n2

Length: N = N1 + N2

Expected length:

$N' = n1 \log2 n1 + n2 \log2 n2$

Volume, a measure of program size,

$V = N \log2 n$

Potential volume, the most succinct form of the program where n2* is the number of different I/O parameters,

$V^* = (2 + n2^*) \log2 (2 + n2^*)$

Program level: $L = V^*/V$

USES

i) to aid in future training

ii) to determine where follow up training is needed

HOW COLLECTED: instrumented language processor, EDF

GOALS: I.4, II, IV.2

DATA USE

EXPLANATION: A program can be described by its use of data as well as by its use of the language. By measuring the use of data within the program, one can determine the complexity of that program and, therefore, the effort required to develop or understand it. Here, complexity refers to the extent of the use of global variables, the extent of coupling between program units, and the amount of information that must be remembered while examining the program.

ELABORATION: Data usage can be measured in many ways, but these measures fall into three categories. Each category presents a different view of the data. Each of these views is elaborated below with examples of precise measures. However, the precise measures are very detailed; therefore, a reference to the literature is given for each of them.

UNIT-DATA

The first view is by measures which quantify the use by a program unit of the data available to it. One needs to examine the global and local data as well as the unit itself to make these measures.

example:

unit-global usage pair: A unit-global usage pair (p,r) is an instance of a global variable "r" being used by a unit "p." Precise measures of interest here are possible pairs and actual pairs. [Basili & Turner 75]

UNIT-UNIT VIA DATA

The second view of the data is by measures which quantify the communication between program units. To determine these measures, one must examine the various program units and the data they share.

example:

data bindings: A unit-global-unit data binding is a triple (p,r,q) which represents the occurrence of the following three conditions: 1) unit p modifies global variable r, 2) variable r is accessed by unit q, and 3) p and q are distinct units. Precise measures are possible data bindings and actual data bindings. [Stevens, Myers, & Constantine 74], [Basili & Turner 75], [Yau & Collofello 80], [Basili & Hutchens 80], [Henry & Kafura 81]

LOCAL

The following measures quantify the use of the data within a program unit. One need only examine the unit itself to determine these measures.

example

span: The number of statements between two consecutive textual references to the same identifier is called the span of the variable. [Elshoff 76]

average live variables per statement: A live variable is one that has a value which will be used later in the measured unit. For example, if x is assigned the value 10 which is used 5 lines later, x is live across those 5 lines. If its value is no longer needed, it will no longer be live. Records will be a problem here, i.e. parts may be live while other parts are dead. We might want to count 1) only the lowest level items as well as 2) all records with some part live as live. [Dunsmore & Gannon 80]

USES:

Each of the references listed above give uses for their measures

i) to estimate the use of global in the program

ii) to estimate the information exchanged in the program.

iii) to estimate the effort required to understand the program.

HOW COLLECTED: data flow analyzer, slicer, other language processors

GOALS: IV.4

EXECUTION

EXPLANATION: The execution of the program has been measured since programming began. Although the time to execute and the space used to execute are still important, other aspects of the execution are also of concern to the software developer. One aspect is that of testing. How can one be sure that all faults have been removed from the code? What types of testing will reveal the most faults with the least effort? These questions are difficult to answer, but the measures below quantify some of the characteristics of the executing program.

ELABORATION: The measures below are divided into two categories: time/space and structured testing. They are described briefly before the detail.

time/space: Time and space measures quantify how efficiently a program performs its task. These measures are highly variable depending on the type of program and any restrictions on the environment. Time can be measured in milliseconds, hours, or days depending upon one's interests and program. Size can be measured by looking at stack space, static data space, dynamic data space, program space, and others items of interest for one's needs.

structured testing: Structural testing methods select test data which cause specific statements, branches, paths, or features to be executed. Statements and features are defined above. Branches are refinements of statements, where a control statement has one or more branches emanating from it. Paths are combinations of the branches. In general, a program with a loop has an infinite number of paths through that loop. Measures of particular interest here are:

coverage: If every construct is executed by at least one test in the set, the program is covered.

counts: Each time that a construct is executed by a test in the set, the count for that construct is incremented by one.

USES:

i) to locate faults in the program [Howden 81]

ii) to determine whether the program can perform the required task within the bounds of its environment and requirements

ii) to determine whether further testing is necessary [Howden 81]

HOW COLLECTED: instrumented code, support environment

GOALS: IV

Conclusions

Currently, we are analyzing the data gathered for this project. Some of the measures described above can be obtained easily from the data. Others may not be made until an instrumented Ada compiler is available. These are not the only measures that can be taken. However, the list above is representative of what we are collecting. The list of metrics is growing, and suggestions for further measures should be addressed to the authors.

Acknowledgement

The authors wish to thank John Bailey, John Gannon, Elizabeth Kreusi, Sylvia Sheppard, and Marvin Zelkowitz for their work with this study and their comments on this paper.

Appendix I

GOALS FOR THE ADA PROJECT

The purpose of the goals is to direct the study of the Ada project. The goals are divided into five sections: Changes and Resources, Ada and PDL/Ada, Data Collection, and Metrics. In the complete list, each goal has a series of questions following it which will aid in the attainment of that goal. Each question has a letter or series of letters following it which indicates where the information to answer that question will be found. Space considerations limit the size of this appendix. Complete copies of the list of goals may be obtained from the authors.

I. Changes and Resources

I.1: Characterize the effort in the project.
I.2: Characterize the changes.
I.3: Characterize the errors.
I.4: Characterize Ada errors.
I.5: Characterize the other errors.
I.6: Characterize the non-error changes.

II. Ada and PDL/Ada

II.1: Evaluate the effect of using an Ada-like PDL with respect to the goals of a PDL.
II.2: Determine which subsets of Ada features are used naturally.
II.3: Determine the effect of using an Ada-like PDL when Ada is the language of implementation.
II.4: Determine how Ada works for this application.
II.5: Characterize the programmers and associate their backgrounds with their use of Ada.
II.6: Determine whether there are aspects of Ada that contribute positively to the design and programming environment.

III. Data Collection

III.1: Evaluate the data collection and validation process.

IV. Metrics

IV.1: Select a set of static metrics for the APSE.
IV.2: Develop a set of size metrics for the APSE.
IV.3: Develop a set of control metrics for the APSE.
IV.4: Develop a set of data metrics for the APSE.
IV.5: Select a set of dynamic metrics for the APSE.
IV.6: Develop a set of test coverage metrics for the APSE.
IV.7: Develop a set of execution statistics for the APSE.
IV.8: Select a set of software development process metrics for the APSE.
IV.9: Determine the effectiveness of the predictive power of certain measures during development.
IV.10: Develop a subjective evaluation system for evaluation of program and design characteristics that are not practically or easily measured in other ways.
IV.11: Provide a data base for future Ada projects to be used to predict some properties of those projects.

Appendix II

THE HALSTEAD OPERATORS FOR ADA

The following list contains all of the unique operators we are counting for the Halstead measures. A full definition of each operator is omitted for space considerations.

abort, accept, access, all, ampersand, and, andthen, array, assign, assign-init, at, bar, bar-exception, begin-token, block, body, box, case, case-variant, char, colon, colon-mode, colon-named, colon-renames, comma, constant, dash, dash-unary, declare, delay, delta, digits, do, dot, dot-dot, dot-entryname, dot-select-comp , else, else-in-select, elsif, end, entry, equal, exception, exception-decl, exit, for, for-use, function, function-generic, function-spec, generic, goto, greater, greater-eq, id, if, in, in-out, in-set, is, label, left, left-label, less, less-eq, limited, loop, mod, new, new-generic-package, not, not-eq, not-in, null-token, number, of, or, or-in-select, orelse, others, others-exception, out, package, package-body, package-renames, param-generic, paren-entryname, paren-expr, paren-funccall, paren-generic, paren-index, paren-list, paren-range, paren-subunit, pdl-brace, plus, plus-unary, pragma, private, procedure, procedure-generic, procedure-spec, quote, quote-attribute, quote-qualified, raise, range, record, rem, renames, return, reverse, right, right-label, select, semi, separate, slash, star, star-star, string, subtype, task, task-body, task-is, task-renames, terminate, then, type, type-generic, type-incomplete, use, when, when-in-case, when-in-exception, when-in-exit, when-in-select, when-in-variant, while, with, with-generic, xor, and yields

References

[Basili 80] V.R. Basili, Tutorial on Models and Metrics for Software Management and Engineering, IEEE Computer Society Press, 1980.

[Basili & Hutchens 80] V.R. Basili and D.H. Hutchens, "A Study of a Family of Structural Complexity Metrics," Proc. ACM-NBS Nineteenth Annual Technical Symposium: Pathways to System Integrity, Gaithersburg, MD. June 1980, pp. 13-15.

[Basili & Perricone 82] V.R. Basili and B.T. Perricone, "Software Errors and Complexity: An Empirical Investigation," University of Maryland Technical Report TR-1195, 1982.

[Basili & Reiter 79] V.R. Basili and R.W. Reiter, "An Investigation of Human Factors in Software Development," Computer, Dec. 1979, pp. 21-38.

[Basili & Turner 75] V.R. Basili and A.J. Turner, "Iterative Enhancement: A Practical Technique for Software Development," IEEE Trans. Software Eng., Vol. SE-1, No. 4, Dec. 1975, pp. 390-396.

[Basili & Weiss 82] V.R. Basili and D.M. Weiss, "A Methodology for Collecting Valid Software Engineering Data," University of Maryland Technical Report TR-1235, Dec. 1982.

[Belady & Lehman 76] L.A. Belady and M.M. Lehman, "A Model of Large Program Development," IBM Systems Journal, Vol. 15, No. 3, 1976, pp. 225-252.

[Dunsmore & Gannon 80] H.E. Dunsmore & J.D. Gannon, "Analysis of the Effects of Programming Factors on Programming Effort," The Journal of Systems and Software Vol. 1, 1980, pp. 265-273.

[Elshoff 76] J.L. Elshoff, "An Analysis of some Commercial PL/1 Programs," IEEE Trans. Software Eng., Vol. SE-2, No. 2, 1976, pp. 113-120.

[Endres 75] A. Endres, "An Analysis of Errors and Their Causes in Systems Programs," IEEE Trans. Software Eng., Vol. SE-1 , No. 2 , June 1975, pp. 140-149.

[Freburger & Basili 79] "The Software Engineering Lab: Relationship Equations," University of Maryland Technical Report TR-764, May 1979.

[Halstead 77] M. Halstead, Elements of Software Science, Elsevier Computer Science Library, 1977.

[Henry & Kafura 81] S. Henry and D. Kafura, "Software Quality Metrics Based on Interconnectivity," Journal of Systems and Software, Vol. 2, No. 2, 1981, pp. 121-131.

[Howden 81] W.E. Howden, "A Survey of Dynamic Analysis Methods," in Tutorial on Software Testing and Validation Techniques, 2nd ed., ed. E. Miller and W. Howden, IEEE Computer Society Press, 1981.

[McCabe 76] T.J. McCabe, "A Complexity Measure," IEEE Trans. Software Eng., Vol. SE-2, No. 4, 1976, pp. 308-320.

[Putnam 78] L. Putnam, "A General Empirical Solution to the Macro Software Sizing and Estimating Problem," IEEE Trans. Software Eng., Vol. SE-4, No. 4, 1978, pp. 345-361.

[SEL 82] Software Engineering Laboratory, SEL-81-104, The Software Engineering Laboratory, NASA Goddard Space Flight Center, February 1982.

[Stevens, Myers, & Constantine 74] W.P. Stevens, G.J. Myers and L.L. Constantine, "Structural Design," IBM Systems Journal, Vol. 13, No. 2, 1974, pp. 115-139.

[Walston & Felix 77] C. Walston and C. Felix, "A Method of Programming Measurement and Estimation," IBM Systems Journal, Vol. 16, No. 1, 1977, pp. 54-73.

[Weiser 81] M.D. Weiser "Program Slicing," Proc. Fifth International Conference on Software Engineering, San Diego, California, March 9-12, 1981, pp. 439-449.

[Yau & Collofello 80] S.S. Yau and J.S. Collofello, "Some Stability Measures for Software Maintenance", IEEE Trans. Software Eng., Vol. SE-6, No. 6, 1980, pp 545-552.