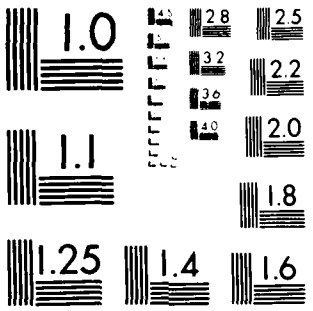
AD-A136 037 RESEARCH IN PROGRAMMING LANGUAGES AND SOFTWARE ENGINEERING(U) MARYLAND UNIV COLLEGE PARK DEPT OF COMPUTER SCIENCE V R BASILI ET AL. 07 MAR 83 1/1

UNCLASSIFIED AFOSR-TR-83-1150 F49620-80-C-0001 F/G 9/2 NL

END
DATE FILMED
1 - 84
DTIC



UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

(12)

AD-A136 037

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFOSR-TR- 83-1150	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) RESEARCH IN PROGRAMMING LANGUAGES AND SOFTWARE ENGINEERING		5. TYPE OF REPORT & PERIOD COVERED FINAL , 1 JAN 82-31 DEC 82 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) V.R. Basili, J.D. Gannon, R.G. Hamlet, N. Roussopoulos, M.D. Weiser, R.T. Yeh and M.V. Zelkowitz		8. CONTRACT OR GRANT NUMBER(s) F49620-80-C-0001
9. PERFORMING ORGANIZATION NAME AND ADDRESS Department of Computer Science University of Maryland College Park MD 20742		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS PE61102F; 2304/A2
11. CONTROLLING OFFICE NAME AND ADDRESS Mathematical & Information Sciences Directorate Air Force Office of Scientific Research/NM Bolling AFB DC 20332		12. REPORT DATE May 7, 1983 13. NUMBER OF PAGES 8
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

DTIC ELECTE DEC 20 1983 S B

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This paper describes work in progress in the following topics: (1) Program Metrics; (2) Program Testing - Experimental Investigations, Step-wise Testing, Testing of Concurrent Specifications, and Testing-theory Critique; (3) Theoretical Issues in Software Engineering; (4) Debugging with Slices; (5) PLACES; (6) Programming Environments; (7) Concurrent, Distributed Systems; and (8) Graphical Design and Documentation. The paper also lists other papers and articles arising from this research effort.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

DTIC FILE COPY

83 12 181

FINAL SCIENTIFIC REPORT

RESEARCH IN PROGRAMMING LANGUAGES AND SOFTWARE ENGINEERING

AFOSR F49620-80-C-0001

V.R. Basili, J.D. Gannon, R.G. Hamlet, N. Roussopoulos,
M.D. Weiser, R.T. Yeh, and M.V. Zelkowitz

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

1. Program Metrics

Building on the work described in [Basili and Reiter 81] which was awarded the IEEE Transactions on Software Engineering Best Paper Award, we developed a family of structural complexity metrics which considered such factors as size, nesting level, and type of control structure. The number of program changes made during development were counted on a per segment (procedure or function) basis. The relationships between program changes and some members of the structural complexity family were investigated. The investigation provides further evidence that a disciplined team approach aids program development [Basili and Hutchens 1983].

We have also used multiple regression analysis to determine if other types of metrics could be used in conjunction with the syntactic metrics to achieve a better model. The data metrics considered included average live variables per statement, number of input/output parameters, number of data bindings, and number of unique operands. We found that the addition of data metrics made minimal improvements in the statistical model. The syntactic metric is the single most important factor, followed by several of the team category variables.

We have investigated the use of cluster analysis in conjunction with data metrics as a means of determining the modularization of systems with respect to data hiding. The small size of the projects studied makes analysis of modularization difficult

to interpret, but we noticed some interesting phenomena. In particular, programs which use data hiding techniques have a clustering which is clearly different from those that do not.

We have also used this technique on production software. Preliminary results of this study were reported at a workshop on Software Performance Evaluation [Basili and Hutchens 1982]. These studies show that the clustering for at least one type of production software gives a reasonable modularization of the system. The modularization is similar to the functional modularization used by the developers and described in the system documentation.

An algorithm for decomposing programs into their prime subcomponents and a method for using this decomposition as the basis of a program metric is described in [Gannon et al. 83]. A tool implementing the algorithm was developed and used to analyze a set of commercial FORTRAN programs. The data collected suggested that programs in an older version of the software contain more complex subcomponents than those in the newer version.

2. Program Testing

2.1. Experimental Investigations

A controlled experiment was performed to compare the techniques of functional testing, structural testing, and code/reading inspection. Thirty subjects, from the upper level "Software Design and Development" course at the University of Maryland, applied each of the three different methods to three different programs with "seeded" errors (in a Latin Square experimental design). The results are being used to contrast the methodologies with regard to mean number of errors found, cost-effectiveness (in terms of number of errors found per unit of effort), classes of errors characteristicly uncovered and errors that were observable but not reported. The examination extends previous work [Myers 78, Hwang 81] by incorporating different testing techniques and a greater number of programs, while adding statistical significance to the conclusions.

Two of the above testing strategies are integrated in the "Clean-Room" software development approach [Dyer 82], and a second investigation was executed to analyze and evaluate this methodology. Motivated by the desire to produce more reliable software, the approach completely separates the development process from the testing process. The developers apply the techniques of code/reading inspections and structured walk-throughs to source code before submitting it for on-line testing. An independent group then functionally tests the delivery from a statistically selected set of test data. This iterative development process was utilized in ten three-person team projects. The operational failures of the projects are being analyzed with respect to severity level, product function and delivery

increment for use in Mean-Time-Between-Failure and other reliability models.

A case study [McMullin and Gannon 83] was performed in which we specified, implemented, and validated a record-oriented text editor similar to one discussed in [Kernighan and Plauger 81] using the DAISTS system. Algebraic axioms served as the specification notation; and the implementation was tested with a compiler-based system that uses the axioms to test implementations with a finite collection of test cases. Formal specifications were sometimes difficult to produce, but helped reveal errors during unit testing. Thorough exercising of the implementations by the specifications resulted in only two errors persisting until integration.

We have begun distributing DAISTS; the system is installed on the University of Illinois and Melbourne VAX systems, and has been provided to the Universities of Iowa and Sydney. It was used in courses at Melbourne as the basis for some student projects. A description of the implementation appeared in [McMullin et al. 82].

2.2. Step-wise Testing

The problem of testing software that is divided into modules, particularly when these components are intended to be processed concurrently, is a difficult and important one. There are two difficulties: the complete execution patterns are so complex that the test results cannot be understood (particularly in the early, buggy stages); it would be advantageous to test partially completed software. A process of iterated, incremental testing is suggested to handle both difficulties [Hamlet 82/16].

2.3. Testing of Concurrent Specifications

The difficulty of testing concurrent software lies in controlling the nondeterminism of the possible interleaving of parallel execution histories. A simple finite-state model is useful in describing the difficulties encountered. Two specification languages now under development (PAISLey, Bell Laboratories; Gist, USC/ISI) are "executable," but do not pay much attention to the question of multiple sequencing possibilities. These languages are described from the testing viewpoint, and a critique presented of the software development methodology based on each. Although not intended to specify concurrent programs, the PROLOG language has some unusual and revealing test properties, which illuminate problems in the more practical languages [Hamlet 82/13].

2.4. Testing-theory Critique

A paper summarizes and evaluates the three themes in testing theory today, which arise from logic (proving-based theory),

practice (tool-based theory), and probability (random testing theory) [Hamlet 82/15].

3. Theoretical Issues in Software Engineering

As a part of a conference in Brisbane, Queensland, Dr. Hamlet was invited to prepare a series of lectures on current software-engineering work of interest to theoreticians. The primary topics are the formalization of specifications, and the foundations of testing theory [Hamlet 82/8].

4. Debugging with Slices

When debugging, it is common for a programmer to know that the value of some variable at a given line is wrong. The portion of a program containing only and all information relevant to a variable's value at a given line is called a slice. Informally, a slice may be defined for a statement n and a set of variables V, to be those statements relevant to the computation of V just before the execution of statement n.

We have developed an interactive tool on the VAX 11/780 that computes slices of programs. Our tool runs on a modified version of the csh called wsh (window shell).

The window shell is a modified version of the Berkeley Unix csh that allows a user to create multiple virtual terminals on his terminal screen. Each virtual terminal appears as a bordered window that can be used to invoke programs within the window. The user can create new windows, connect the keyboard to any window, destroy windows, move a window to a new location, hide (make invisible), unhide and uncover a window. A window may cover (partially or completely) another overlapping window on the screen. A program running in a window may also create windows and do any window function the user could do.

We have developed an interactive debugging tool based on slicing and data flow information. The user is presented with a menu of operations to select from. He may elect to display the program and highlight lines where a given variable is assigned, referenced or appears; slice on a variable at some statement and highlight the slice; or create a file containing just the statements of a slice. He may also see a summary of data flow information about the program or invoke other programs to run in a window. The tool is friendly to the user and can explain any of its functions.

5. PLACES

Research on the design and evaluation of data abstraction features continued with the evaluation of those features in the PL/I system - PLACES, previously developed under this grant. An evaluation of abstract data types was performed by two

programming classes at the University of Maryland - one using the features and the other using standard PL/I. The results of this study were previously reported to AFOSR [Zelkowitz 82b] and are summarized here:

(1) The cost of implementing abstract data types within a language that includes structures or records and pointer variables (e.g., PL/I) is relatively low and cost effective.

(2) There was some run-time overhead in using abstractions. Programmers, using good design techniques developed more modules than in the standard PL/I group. However, many of the modules were of a trivial (e.g., one line of code) type, and inline expansion of these procedures eliminated most of the increased overhead.

(3) Programmers seemed to have little trouble adapting to the data abstraction discipline. Using two measures of program complexity - length and cyclomatic complexity - programs using abstractions were less complex.

In summary, abstract data types seem effective in practice, and if taught correctly, should be effective in producing good programs in a language like Ada.

6. Programming Environments

With the advent of the megabyte workstation with multimegabyte disk for a relatively low cost of $10,000 to $20,000, the means to develop programs will change in the future. Powerful single-user workstations can be used to build and develop programs. An initial design of an integrated environment was started during this year [Zelkowitz 82a].

The basic idea is to use a structured editor that knows the details of the individual language. Such an editor, BABLE, is now being designed. Some of the important issues that are being considered are:

(1) The editor will work with an externally defined grammar. Thus several source languages can be processed. Currently Pascal is the target language, but PL/I and Ada are planned later.

(2) A major issue is the management of the user's view into the program. Typically a user has from 24 to 60 lines of screen in which to manage many thousands of source lines. Effective man/machine interaction is important for increasing programmer productivity.

(3) The integration of all phases of the life cycle is important for managing development. The use of a program design language is being developed, and its integration into code

production is being studied.

7. Concurrent, Distributed Systems

A software specification technique suitable for concurrent, distributed systems has been developed. The technique combines an abstract, nonprocedural specification language with a formal proof system for a programming language. The complete specification of a program is a set of hierarchically structured module specifications. Module external specifications are abstract. Module internal specifications are descriptions of hidden implementations, either in terms of submodules or actual code. Defined verification procedures establish that the external specification of a module is an accurate characterization of its internal implementation. [Reed and Yeh 83]

Several concurrent programming languages have been studied to determine their suitability to real-time programming. It has been determined that a completely new programming language is not necessary to meet these requirements. We are now in the process of selecting a subset of Ada with a few modifications borrowed from Modula-2. This language will be incorporated into a set of abstraction-based design and debugging tools. These tools will primarily be concerned with the problem of meeting timing deadlines.

8. Graphical Design and Documentation

We have implemented an interactive system for graphical design and documentation [Roussopoulos and Kelly]. The system is supported by a database that has very general graphical representations of hierarchical nature and annotations for documentation purposes. Through a friendly user interface, the user specifies the systems requirements, he annotates them and proceeds to the design. The design is specified and annotated through the same interface. During these interactions with the user, a database is constructed which contains the requirements and design specifications along with their documentation.

The graphical representations and their underlying logical constructs facilitate both the definition of the functional and operational characteristics of the software as they are perceived by the end user (requirement specifications) and the design properties of the software as specified by the analysts. The system supports the logical constructions required for most structural, control flow and data flow modeling primitives such as closed objects of various shapes and sizes, connectors, and annotations. Closed objects can be opened up and further specified and/or examined at a more detailed level. The database is designed in such a way that it can support this hierarchical decomposition and a zoom-in/out accessing facility.

The system is running on a VAX 11/780 under UNIX. Interaction with the system is currently available only through Tektronix 4020 series terminals, but drawing functions are performed in 4010 mode. Hard copies are generated by a Tektronix photostatic device.

9. References

[Basili and Hutchens 1982]
V.R. Basili and D.H. Hutchens, Clustering as a Method of Analyzing System Structure, 5th Minnowbrook Workshop on Software Performance Evaluation, July 20-23, 1982.

[Basili and Hutchens 1983]
V.R. Basili and D.H. Hutchens, Analyzing a Syntactic Family of Complexity Metrics, IEEE Transactions on Software Engineering, (to appear).

[Basili and Reiter 1981]
V.R. Basili and R.W. Reiter, Jr. A Controlled Experiment Quantitatively Comparing Software Development Approaches. IEEE Transactions on Software Engineering, 7, 3, May 1981, pp. 299-319.

[Dyer 82]
M. Dyer, "Cleanroom Software Development Method," IBM Federal Systems Division, Bethesda, MD, presentation given October 14, 1982.

[Gannon et al. 83]
J.D. Gannon, M.S. Hecht, and R.J. Herbold. Prime program decomposition, 16th Hawaii International Conference on Systems Sciences, (January 1983), 25-29.

[Hamlet 82/8]
R.G. Hamlet. Theoretical issues in software engineering, TR 82/8 Department of Computer Science, University of Melbourne, Parkville, 1982.

[Hamlet 82/13]
R.G. Hamlet. Testing of concurrent programs and partial specifications, TR 82/13 Department of Computer Science, University of Melbourne, Parkville, 1982. (Also position paper for a panel session at Hawaii International Conference on System Sciences, Honolulu, January, 1983.)

[Hamlet 82/15]
R.G. Hamlet. Three approaches to program testing theory, TR 82/15 Department of Computer Science, University of Melbourne, Parkville, 1982.

[Hamlet 82/16]
R.G. Hamlet. Step-wise debugging, TR 82/16 Department of

Computer Science, University of Melbourne, Parkville, 1982. (Also to appear in ACM SIGSOFT High-level Debugging Symposium, March, 1983.)

[Hwang 81]
S-S. V. Hwang, "An Empirical Study in Functional Testing, Structural Testing, and Code Reading/Inspection*", Scholarly Paper 362, Dept of Computer Science, University of Maryland, College Park, MD 20742 (December 1981).

[Kernighan and Plauger 81]
B.W. Kernighan and P.J. Plauger. Software Tools in Pascal. Addison-Wesley Publishing Company, Reading, MA, (1981).

[McMullin et al. 82]
P.R. McMullin, J.D. Gannon, and M.D. Weiser. Implementing a compiler-based test tool, Software - Practice and Experience 12, (1982), 971-979.

[McMullin and Gannon 83]
P.R. McMullin and J.D. Gannon. Combining testing with formal specifications: A case study, IEEE Trans. Soft. Eng., (to appear). (A version of this paper was presented at the IEEE Workshop on Effectiveness of Testing and Proving Methods, Avalon, California, May, 1982.)

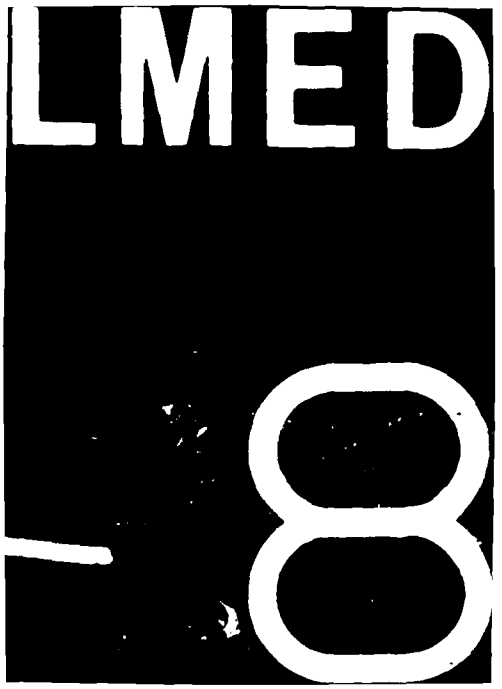
[Myers 78]
G. J. Myers, "A Controlled Experiment in Program Testing and Code Walkthroughs/Inspections," Communications of the ACM, pp. 760-768 (September 1978).

[Roussopoulos and Kelley 83]
N. Roussopoulos and S. Kelley. A Relational Database to Support Graphical Design and Documentation, ACM SIGMOD Conference on Databases, San Jose, CA., (May 1983), (to appear).

[Reed and Yeh 83]
Joylyn Reed and Raymond T. Yeh. Specification of Concurrent, Distributed Software, Proceedings IEEE Workshop on Computer Systems Organization, New Orleans, La., 1983, (to appear).

[Zelkowitz 82a]
M.V. Zelkowitz, et. al., An editor for rapid prototypes, ACM SIGSOFT Second Software Engineering Symposium: Rapid Prototyping, Columbia MD, April, 1982.

[Zelkowitz 82a]
M.V. Zelkowitz, What has happened to data abstractions? - experience with one implementation, Sixth International Conference on Software Engineering (Poster Session), Tokyo Japan, September, 1982.


LMED
8