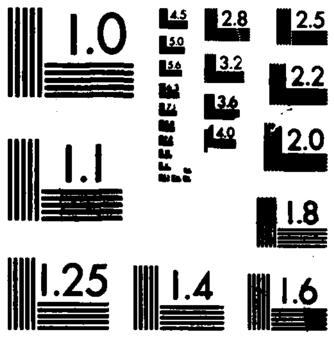
AD-A183 060 PROGRAMMING LANGUAGES AND SOFTWARE ENGINEERING(U) 1/1
MARYLAND UNIV COLLEGE PARK DEPT OF COMPUTER SCIENCE
V R BASILI ET AL. 30 APR 87 AFOSR-TR-87-0930
UNCLASSIFIED F49620-85-K-0008 F/G 12/5 NL


1.0
4.5
5.0
5.6
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION	1b. RESTRICTIVE MARKINGS
AD-A183 060	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
	5. MONITORING ORGANIZATION REPORT NUMBER(S) AFOSR-TR- 87-0930

6a. NAME OF PERFORMING ORGANIZATION University of Maryland	6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION AFOSR/NM
6c. ADDRESS (City, State, and ZIP Code) College Park, Md. 20742		7b. ADDRESS (City, State, and ZIP Code) AFOSR Bldg 410 Bolling AFB DC 20332-6448
8a. NAME OF FUNDING/SPONSORING ORGANIZATION AFOSR	8b. OFFICE SYMBOL (If applicable) NM	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F49620-85-K-0008
8c. ADDRESS (City, State, and ZIP Code) Bldg 410 Bolling AFB DC 20332-6448		10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.
61102F	2304	A3	

11. TITLE (Include Security Classification)
Programming Languages And Software Engineering

12. PERSONAL AUTHOR(S)
Victor R. Basili, John D. Gannon, Marvin V. Zelkowitz

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Final	FROM 1/1/85 TO 2/28/87	4/30/87	

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	

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

This effort dealt with seveal issues critical to the improvement of software engineering techniques. A notion of abstraction for use in formal program specification was developed. A systematic way of building large programs by integrating reusable components was studied. An empirical study comparing the effects of three code reading techniques of three PASCAL programs seeded with different sets of faults was executed. Finally, in the area of advanced programming environments, improvements were made in the effectiveness of language-based environments, in the efficiency of language-based editors through the use of nonlocal productions, and in automatic inference of user data types. More than thirty published references and papers resulted from this effort.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☐ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION

22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code)	22c. OFFICE SYMBOL
Capt. John Thomas	(202) 767-5026	NM

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

AFOSR-TR- 87-0930

TECHNICAL SUMMARY

AFOSR GRANT F49620-85-K-0008
January 1, 1985 - February 28, 1987

Principal Investigators:

Victor R. Basili

John D. Gannon

Marvin V. Zelkowitz

Date: April 30, 1987

This report represents the technical summary of AFOSR grant F49620-85-K-0008 to the Department of Computer Science of the University of Maryland for the period January 1, 1985 until February 28, 1987. Copies of all relevant papers have been previously forwarded to AFOSR as they appeared. As a large multi-person grant, some of this represents work that has been completed, and some of it represents congoing research just begun.

1. Data Abstraction

Because large-scale software development is a struggle against internal program complexity, the modules into which programs are divided play a central role in software engineering. Modules that encapsulate complex data types are perhaps the most important sequential programming-language idea to emerge since the design of ALGOL 60. Such a module serves two purposes. First, in its abstraction role, it allows the programmer to ignore the details of operations (procedural abstraction) and value representations (data abstraction) in favor of a concise description of their meaning. Second, encapsulation is a protection mechanism isolating changes in one module from the rest of a program. The first role helps people to think about what they are doing; the second allows program changes to be reliably made with limited effort.

The essence of data-abstraction is captured by a diagram showing the relationship between a *concrete* world, the objects manipulated directly by a conventional programming language, and an *abstract* world, objects that the programmer chooses to think about instead of the more detailed program objects. Within each world, the items of interest are operations mapping objects to (possibly different) objects. The two worlds are connected by a *representation* function that maps from concrete to abstract.

A data-abstraction theory must define *correctness,* intuitively the property that the programmed concrete operations do properly mirror the abstract maps in our minds. A theory also defines a *proof method,* a means of establishing the correctness of any particular module.

In the data-abstraction diagram:

$$\begin{tikzcd}
\{\text{abstract states}\} \arrow[r, "m"] & \{\text{abstract states}\} \\
\{\text{concrete states}\} \arrow[u, "A"] \arrow[r, "\boxed{P}"] & \{\text{concrete states}\} \arrow[u, "A"']
\end{tikzcd}$$

the abstract function is m, the representation mapping is A, and the concrete operation is the meaning of some procedure P, written [P]. We say that the diagram *commutes* if and only if beginning in the lower left corner and passing in both possible directions gives the same result whenever the abstract path is defined; that is, $A \ o \ m \subseteq \boxed{P} \ o \ A$. In the view that the abstract function is a specification, a commuting diagram corresponds to a correct implementation with "don't care cases: when the abstract function m is undefined, the program function [P] may take any value.

We have developed a formal semantics of modules and its corresponding proof theory, and have been integrating our ideas into projects designed to evaluate the efficacy of these ideas. In addition, we have investigated the tradeoffs in representing abstract objects directly or indirectly. The former method leads to better accessing code at the cost of more re–compilations if changes are made to an object's concrete representation, while the latter method mimimizes re–compilations at the expense of access time [GANN87a].

2. Module Interconnection Issues

The ability to interconnect modules and reuse older modules is becoming of increasing importance as software grows larger and more complex. Two studies have begun on addressing this important topic.

2.1. Program Specifications

During the past year, we have formalized a notion of abstraction for use in formal program specification. We formalized abstractions as functions called "abstraction functions." Our abstraction functions are useful for specifying properties about objects which can be viewed in many abstract ways. We developed two example specifications using abstraction functions and characterized the domains for which abstraction functions are generally useful. We found that abstraction functions are generally useful in the specification of user interfaces. All of this work is aimed at the goal of making formal specification and verification a practical part of software development. We hope that our abstraction functions will make it easier to formally specify programs that have complex user interfaces. We also made an effort to develop our specifications and specification techniques so that they will be manipulable by automated verification systems. This, we believe, is very important to the eventual practical use of formal specification and verification.

2.2. Software Construction with Reusable Modules

We are doing research on the systematic way of building large programs by integrating reusable components. The main goal of the research is to use the general components without modification but with adaptation. In order to show the effective way of constructing programs we planned to build a prototype system that supports building process. To do this we follow several steps:

1. Classify the general purpose components.
2. Decide specification method and structure for the reusable components.
3. Decide the organization of components library.
4. Find the way of searching the components with partial specification.
5. Search the interconnection problems and the consistency problem that may occur during the integration of separately built components.
6. Build components for some problem domain.
7. Do experimentation.

Last year we performed an extensive survey of current research efforts for reusability in general and components reuse; we also searched for possible direction. Then we decided a research plan to build a prototype system that supports software system builder (i.e., designer) in integrating the reusable components into a large program. [JOO87] The prototype system will consist of 1) a library of reusable modules, 2) tools to support the creation, cataloging and searching the reusable modules, and 3) connection supporting tools with adaptation and consistency checking facility.

3. Code Reading Studies

The two code reading studies, the "How_Reading_Effects–Cleanroom" and the "How_Specification_Effects_Reading" studies, were done during the Fall of 1986. The first study tries to compare the effect of code reading to the effect of code reading followed up by functional testing. The second study tries to analyze the effect of code reading based on stepwise abstraction depending on whether the code reader does the stepwise abstraction with or without the original specification.

An empirical study comparing the effect of three code reading techniques (code reading with specification, code reading without specification, and code reading with specification followed by functional testing) on three PASCAL programs (an abstract data type, a text formatter, and a database maintainer) seeded with different sets of faults (only hard faults which is intended to simulate the situation after system test, hard+medium faults which is intended to simulate the situation after unit test, or hard+medium+easy faults which is intended to simulate the situation after compilation) was executed with thirty–two subjects at the University of Maryland. The length of the programs (in LOC) was 140 (abstract data type), 220 (text formatter), and 500 (database maintainer). The number of faults was (2: hard, 4: hard+medium, 7:hard+medium+easy) for the abstract data type, (3, 6, 9) for the text formatter, and (4, 7, 12) for the database maintainer. The study consisted of three sessions. Each subject (students of a software engineering course) used worked with each code reading technique, each program, and each fault profile once. For example, if a subject applied code reading technique "1" to program "2" with fault profile "3" in session 1, he/she might have applied code reading technique "2" to program "3" with fault profile "1" in session 2, and code reading technique "3" to program "1" with fault profile "2" in session 3. The statistical approach chosen was a fractional factorial design. This formal statistical approach enables the distinction of differences in the impact of the three code reading techniques on different programs and fault profiles, while allowing for variation in the ability of the individuals testing or in the programs being tested.

The **results of the "How_Reading_Effects_Cleanroom" study** can be summarized as follows:

1) Only for one program (the abstract data type) code reading followed by functional testing detected a (statistically significantly) higher number of faults compared to just code reading.

2) Code reading was significantly more effective in terms of fault detection rate (number of faults detected per hour).

3) Functional testing applied after code reading was much more effective than indicated by earlier studies [BASI85e]. Almost all faults missed by code reading were detected by functional testing.

The results do not give a strong indication that code reading followed by functional testing produces better results (in terms of number of faults detected) than just code reading. However, independent of the quality of code reading, functional testing applied after code reading seems to be much more effective than applied in isolation. Therefore, it might be concluded that code reading is a very good preparation for code reading.

The **results of the "How_Specification_Effects_Reading" study** can be summarized as follows:

1) Code reading without specification resulted in more faults being detected than code reading with specification.

2) Code reading without specification is about two times more expensive than code reading with specification.

Overall, it can be concluded that reading without the specification is more effective and expensive.

In addition, a couple of additional observations should be mentioned:

1) The effectiveness of (all types of) code reading is better in the case of few faults (just hard faults) and in the case of a (relatively) high number of faults (hard+medium+easy faults). A medium number of faults (hard+medium faults) generally resulted in the lowest effectiveness. One possible explanation is that human beings expect a certain level of faults (x faults per LoC); whenever, they find significantly less or more faults they get into some kind of exception mode and try harder. To a certain extent this result is supported by weaker results of the same type from an earlier study. In the "When_To_Read" study [ROMB86] no significant difference has been detected between the different fault profiles. One of the reasons might be that we included too many syntax faults in this previous study. However, when we looked at the results from the "When_To_Read" study again, a similar pattern (although not significant from a statistical point of view) was recognized: code reading applied to a small or high number of faults resulted in detecting a higher percentage of faults than applied to a medium number of faults.

2) The effectiveness of code reading by stepwise abstraction depends heavily on experience. This result is supported by similar results from all previous code reading experiments.

In conclusion: Results from these two studies support many results from prior code reading studies [BASI85e], [ROMB86]. In addition, code reading seems to be most effective after compilation or after system test. There is no strong empirical support for recommending code reading being followed up by functional testing; however, functional testing might be more effective if preceded by code reading as a preparation technique. Finally, code reading by stepwise abstraction can be significantly more effective if the abstraction process is performed without knowledge of the original specification.

4. Environment Models

Advanced programming environments will play an important role in achieving significant increases in programmer productivity; consequently, research in programming environments is of major importance. Under this grant we have made progress in several areas: increasing the effectiveness of language based environments, improving the efficiency of language-based editors through use of nonlocal productions, programming environments that automatically infer user data types, and semantics-based programming environments. We summarize below work carried out in each of these areas under the AFOSR grant.

4.1. Environment Development

During this grant, development of the SUPPORT environment matured. The interaction among the various windows and facilities - program generator, parser, internal editor, interpreter - became stable. A major task during 1985 was the transporting of the system to an IBM PC. This project demonstrated the use of environments on machines as small as the PC. Some of the facilities had to be redesigned to take into account the architectural features of the Intel 8088 family of microprocessors [ZELK85c]. In spite of the small machine size, SUPPORT runs efficiently on a 256K PC and can process a 600 line program. In a 640K machine, programs up to 3,000 lines can be processed. This shows that such small machines can be used effectively as single user workstations.

Beginning January of 1986 the system was used in the introductory programming class of computer science majors for their use in building Pascal programs. The effectiveness of this system in a real user environment was measured [ZELK87b]. Syntax editing is most accepted by inexperienced users, and the issues of navigation and code modification are paramount issues for acceptance.

Based upon this early evaluation, work began on specifying a hierarchy of editing features where syntax editing can provide an effective improvement of productivity. A six level hierarchy was proposed consisting of text editing, text inclusion, syntax editing, macro processing, knowledge processing and data flow analysis. Each level provides more power to the system, and a full implementation of all six levels provides a powerful specifications language model for text generation [ZELK87a]. Work on this area is continuing.

Along with the programming environment, a diagnostic run time system – Drs. – has been built for program testing. A basic design of this system has been the use of higher level Pascal syntax as a command language. Drs. allows users to investigate the runtime data stack, execute Pascal statements from the command level, trace variables and to display the source program in various ways. This work resulted in a M.S. thesis for B. Kowalchack, and a paper on the system was published [KOWA87].

Work began on a natural language extension to the SUPPORT system. The idea behind this extension is to use artificial intelligence techniques in the design phase of program development. A user is expected to write a design comment, as in:

:Sort A in ascending order using a bubble sort;

and the system will respond by producing the Pascal code that implements this comment. It is expected that such techniques can greatly improve the efficiency of expert programmers. An early paper on the topic was produced and more work is expected [ZELK85b].

Analysis of program structure was furthered by the Ph.D. research of William Bail [BAIL85]. He developed a mathematical model of program structure based upon the prime program decomposition of the resulting flowgraph. This model differs from previous models in that it considers both the syntactic structure of the program as well as data structure dependencies, and thus makes it possibly better amenable for analyzing object oriented designs. A technical report was written outlining the basic model. Further work will continue in 1986 refining this model with respect to real programs.

4.2. Efficiency issues in advanced programming environments

The first research endeavor supported has been the study of nonlocal productions as a means to achieving high responsiveness and efficiency in language-based editors. Results from this research have been encouraging; a paper is being readied for submission to the ACM Transactions on Principles of Programming Languages based on the work. The goal is to make possible the automatic generation of highly efficient programming environments. In this approach, one describes the syntax and semantics of a programming language in a high-level, declarative notation, and from this description a programming environment is automatically generated. A problem of this approach has been that the resulting environments were not efficient enough to provide satisfactory speed of response to users. The approach of nonlocal productions has proved to be a convenient extension to the description language that results in significant improvements in system response time to users. Nonlocal productions have been defined using a technique call graph projection. Efficient incremental updating of semantic information can be achieved by a generator-time analysis of the possible dependencies of attributes in the description of the programming language.

In several language-based editors, designers have opted for nonlocal information flow as a part of incremental static semantic analysis. A user's program is typically represented as a parse tree decorated with static semantic information, and the tree is threaded with links that permit information to propagate in a single step from one place in the tree to another place that may be far from the first. Nonlocal information flow appears to be necessary to achieve reasonable interactive responsiveness in such systems. However, most of the approaches have been somewhat heuristic or ad hoc. Of the more rigorous approaches, there have been a variety of restrictions that authors have found necessary to impose. None of the approaches is wholly satisfactory for all applications; the approach presented here is a generalization and reformulation of the priority-relation technique first discussed in [JOHN85]. Our research is based on a new definition of nonlocal attribute grammars based on graph projections. The new definition provides the twin advantages of being natural for writers of attribute grammars (the notation is almost identical to the standard notations for attribute grammars) and offering a clean mathematical basis for studying incremental semantics in the presence of nonlocal dependencies. A new general incremental update algorithm was developed and analyzed as a graph coloring problem. This technique provides the basis for an optimal priority-based incremental update scheme. This work represents the culmination of work by the authors over the past few years to combine the efficiency of the nonlocal approach with the advantages of rigor found in Reps's approach to incremental semantics.

A graph coloring approach provides the basis for a very general optimal priority-based incremental update scheme. Nonlocal productions have been used by several authors, and priority-based evaluation has been adopted by at least two groups: Zadeck, and the present authors. In neither case, however, was an optimal algorithm for incremental updating presented and proved. The technique of nonlocal productions has proved successful as an engineering solution to provide efficient

LBE's, but optimality results have remained elusive. The work described here contrasts with that of Reps, in which only upward remote references are permitted and operations proportional to parse tree depth are required, and that of Hoover in which dynamic, evaluation–time measures are employed.

Our priority calculation algorithm for projection–based nonlocal attribute grammars is patterned on Knuth's characteristic graph computation algorithm. Just as a characteristic graph is the smallest of a potentially infinite set of graphs that all share a particular structural property, we define a notion of rendezvous ancestor graphs that are associated with grammar symbols. Assume that in some parse tree there is an attribute that has $N >= 2$ distinct paths to some other attribute. If a subtree of a node is replaced by the appropriate rendezvous ancestor graph for that node, there still will be N distinct paths between the two attributes. This path information is precisely what is needed for computation of priorities. Just as graph projection is used on the compound dependency graph of an attribute grammar to contain nonlocal dependencies, projection can be employed in deriving and using rendezvous ancestor graphs. It was the pleasing symmetry of this construction that convinced us of the appropriateness of a projection–based approach to nonlocal productions.

4.3. Automated type inference in programming environments

A language–based editor is a full–screen interactive text editor that has programmed into it knowledge of a particular programming language. As a programmer enters or modifies his program, the editor provides constant feedback about the state of syntactic and static semantic correctness of the program. Thus, functions normally reserved to a compiler are integrated into the text editor. Until recently language–based editors have assumed a fairly passive role in the process of consistency checking; declarative information provided by the user was compared to uses. For instance, if an identifier is declared to be an array variable, the editor checks whether it is used correctly. In [JOHN86] it was demonstrated that a language–based editor can assume a much more active role. That work reported on a system that was implemented. MOE is a language–based editor that performs active type inference as the user enters his program. Instead of relying on the user to provide type information, MOE makes determinations as to the types of all constructs in the user's program. If inconsistent uses are noted, they are highlighted to indicate the presence of errors. As a simple example, if the same identifier is used in a context that requires a boolean value and in another context that requires an integer value then a type conflict is noted. The presence of a rich type structure in the language makes the incremental type inference problem nontrivial. MOE attempts by counting usage frequencies to infer the intended type of each component of the user's program. If a conflict is detected, error highlighting is made more noticable for the minority assertion if there is one. The presumption is that if an identifier is used frequently as, say, an integer and only once as a boolean then the error is probably the latter usage and the boolean use will be highlighted at a higher intensity than the integer usages.

MOE has successfully demonstrated the feasibility of incorporating an active style of automated type inference into language-based text editors.

4.4. A semantics-based programming environment

Most current programming environments focus primarily on issues of syntax and static or compile-time semantics. A new project was initiated to explore the possibility of creating a programming environment around a denotational definition of the semantics of a programming language. Program testing and maintenance are of critical importance in the software life-cycle, and and environment that is designed to support this phase of the life-cycle based on a semantic understanding of the programming language is an interesting prospect. Initial results based on theoretical investigations and an experimental implementation have been promising [JOHN87].

Denotational definitions have had a major impact on programming languages. Understanding of mathematical foundations of programming languages has made for better engineered languages. We hypothesize that program development and testing can similarly benefit from a rigorous mathematical basis. The new research environment will thus provide information with which to assess our hypothesis.

The environment is based on GL, a language designed to support interactive experimentation with denotational semantics of programming languages. GL is an expressional language that might best be described as an implementation of lambda calculus augmented with several useful basic data types including l-values.

A unique aspect of the GL environment is that it presents a visible, user-accessible implementation of the continuation semantics of GL. The user is expected to understand a denotational definition of GL, and to interact with the system in terms of that definition. In particular, if a computation is temporarily halted the expression continuation extant at that point can be interactively captured and later applied to other values and stores. The implementation of this feature is via a pair of routines called *setjmpup* and *longjmpup* that provide what might be called a partial continuation facility. A partial continuation is a function over stores or store/value pairs that represents execution of a partially executed program from its current state to some later state possibly before its halt state. The semantics of partial continuations is interesting, and GL contains continuations and partial continuations as first-class objects.

The GL environment is fairly complete; it has an experimental polymorphic type inference mechanism that supports self-application and reports likely sources of user error in a robust manner, and it has a flexible breakpoint and trace facility that permits program execution to be observed and controlled at a variety of levels of granularity.

The environment is designed around a continuation semantics for GL, and the user of the environment interacts with the computer in terms of this denotational definition. At any point when execution of a program has been stopped, the user can interactively obtain a continuation for the computation in progress and bind it to an identifier in the programming environment. The captured continuation can then be applied to a variety of arguments in an exploratory manner. The use of continuations as first-class objects in a programming language is extremely useful; the language Scheme exemplifies this principle.

The normal mode of use is for users to capture continuations in their programs at various interesting points and apply them in a flexible, interactive way to a variety of arguments and stores. Since GL permits functions to be used as fully general first-class objects, applications of captured continuations can be imbedded in arbitrary GL program fragments. A user could thus easily write a small program to apply a given continuation to all odd integers between 100 and 1000, and algorithmically determine if his program behaves as desired under those circumstances.

Efficient implementation of an interactive continuation-based environment required a novel approach to the internal design of the execution component of the system. The GL environment provides a Read-Eval-Print loop that permits recursive invocations of itself. Thus the runtime activation record stack consists of an activation of Read followed by some number of invocations of Eval, another activation of Read followed by more invocations of Eval, etc. From the point of view of implementation a continuation is simply a sequence of Eval activation records between two successive activations of Read.

As mentioned above we implementated two procedures that implement what might be called partial continuations. The routines are called *setjmpup* and *longjmpup* to suggest an analogy with the Unix system calls *setjmp* and *longjmp*. *Setjmpup* captures a continuation from after its point of call to the completion of one of its ancestors in the called-by relation. This is accomplished by capturing some number of activation records off the top of the runtime stack. *Longjmpup* evaluates such a continuation by concatenating a vector of activation records onto the runtime stack. This mechanism provides an efficient way to implement dynamically obtainable continuations; if the user assigns the continuation of a function he is executing to a variable, the object assigned to the variable is the stack of Eval frames down to the previous Read frame. This technique has proved to be feasible within a relatively low-level implementation language (Pascal) and to be sufficiently efficient.

5. Published References

(1) [BAIL85] Bail, W.G. and M.V. Zelkowitz. Program complexity using hierarchical abstract computers, Computer Science Technical Report TR-1593, University of Maryland, (December, 1985).

(2) [BASI85a] Basili, V.R. and D.H. Hutchens. System Structure Analysis: Clustering with Data Bindings, *IEEE Transactions on Software Engineering,* 11, 8, (August 1985), pp 749-757.

(3) [BASI85b] Basili, V.R., and R.W. Selby. Data Collection and Analysis in Software Research and Management, NATO Advanced Study Institute on The Challenge of Advanced Computing Technology to System Design Methods, (August, 1985).

(4) [BASI85c] Basili, V.R., R.W. Selby, and F.T. Baker. Cleanroom Software Development: An Empirical Evaluation. University of Maryland Technical Report TR-1415, (February 1985).

(5) [BASI85d] Basili V. R. and R. W. Selby, Calculation and use of an environment's characteristic software metric set, ACM/IEEE 8th International Conference on Software Engineering, London Eng, (August, 1985) 386-390.

(6) [BASI85e] Basili V. R. and R. W. Selby, Jr., *Comparing the Effectiveness of Software testing Strategies,* Dept. Com. Sci., Univ. Maryland, College Park, Tech. Rep. TR-1501, May 1985.

(7) [BASI86a] Basili V. R., R. W. Selby, Jr., and D. Hutchens, Experimentation in Software Engineering, *IEEE Transactions on Software Engineering, (July 1986).*

(8) [BASI86b] Basili V. R. and R. W. Selby, Jr., Four Applications of Software Data Collection and Analysis Methodology, in *Software System Desing* (J. Skwirzynski, editor), Springer-Verlag Lecture Notes in Computer Science, Vol. F22, 1986.

(9) [BASI87] Basili V. R. and D. Rombach, Tailoring the Software Process to Project Goals and Environments, 9th International Conference on Software Engineering, Monterey, CA, (March 1987).

(10) [DAY85] Day, J.D., and J.D. Gannon, A test oracle based on formal specifications, Proceedings of SOFTFAIR II, (December 1985), 126-130.

(11) [DUNL85] Dunlop D. D. and V. R. Basili, Generalizing specifications for uniformly implemented loops, *ACM Transactions on Programming Languages and Systems* 7, 1 (January, 1985) 137-158.

(12) [GANN85] Gannon, J.D., R.G. Hamlet, and H.D. Mills. Functional semantics of modules, *Formal Methods and Software Development,* Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT), volume 2, Lecture Notes in Computer Science, 186, Springer-Verlag, (March 1985), 42-59.

(13) [GANN86] J.D. Gannon, Testing tools using formal specifications and coverage metrics, Proceedings of Software-Testsysteme, Bremen, Germany, (June 1986), 5-11.

(14) [GANN87a] J.D. Gannon and M.V. Zelkowitz. Two implementation models of abstract data types, *Journal of Computer Languages* 12, 1 (January, 1987), *(to appear).*

(15) [GANN87b] J.D. Gannon, R.G. Hamlet, and H.D. Mills, Theory of modules, *IEEE Trans. Soft. Eng. (January, 1987), (to appear).*

(16) [JOHN85] Johnson, G. F. and C. N. Fischer, A Meta-Language and System for Nonlocal Incremental Attribute Evaluation in Language-Based Editors, Proc. of the Twelfth ACM Symposium on Principles of Programming Languages, (January, 1985), 141-151.

(17) [JOHN86] Johnson, G. F. and J. A. Walz, A Maximum Flow Approach to Anomaly Isolation in Unification-based Incremental Type Inference, Proc. of the Thirteenth ACM Symposium on Principles of Programming Languages, (January, 1986), 44–57.

(18) [JOHN87] Johnson, G. F., GL – A Denotational Testbed with Contintuations and Partial Continuations as First-class Objects, Proc. of the ACM Conference on Interpreters, (June 1987).

(19) [JOO87] Joo B. G., Software Development Using Reusable Components", Proposal for Ph.D. Dissertation Research(Draft), March 1987.

(20) [KOWA87] Kowalchack B. and M. V. Zelkowitz, Drs.: A language oriented diagnostic runtime system, *2nd Conference on the Role of Language in Problem Solving,* Elsevier Science Publishers (1987) 377–389.

(21) [LIN85] Lin, K.-J., and J.D. Gannon. Atomic remote procedure call, *IEEE Trans. Soft. Eng.* 11, 10, (October 1985), 1126–1135.

(22) [MILL87a] H.D. Mills, V.R. Basili, J.D. Gannon, and R.G. Hamlet. Teaching principles of computer programming. Proceedings ACM 15th Annual Computer Science Conference, St. Louis, (February 1987).

(23) [MILL87b] H.D. Mills, V.R. Basili, J.D. Gannon, and R.G. Hamlet. A first course in computer science: mathematical principles for software engineering. Proceedings of the SEI Conference on Software Engineering Education, Pittsburgh, (April 1987), (to appear).

(24) [ROMB86] Rombach H. D., V. R. Basili, and R. W. Selby, Jr., *The Role of Code Reading in the Software Life Cycle,* Proc. of the Ninth Minnowbrook Workshop on Software Performance Evaluation, Blue Mountain Lake, New York, August 5–8, 1986.

(25) [WEIS85] Weiser, M.D., J.D. Gannon, and P.R. McMullin, Comparison of structural test coverage metrics, *IEEE Software* 2, 2, (March 1985), 80–85.

(26) [ZELK85a] Zelkowitz M. V. et al., The still unnamed production programming oriented research tool (SUPPORT) environment, IBM AEP Conference, Alexandria VA, (June, 1985), 97–112.

(27) [ZELK85b] Zelkowitz M. V. et al., The SUPPORT Pascal programming environment, 8th Minnowbrook Workshop, Blue Mountain Lake NY, (July, 1985).

(28) [ZELK85c] Zelkowitz M.V., et al. The engineering of environments on small machines, IEEE International Conference on Computer Workstations, San Jose, CA, (November, 1985), 61–69.

(29) [ZELK85d] Zelkowitz M.V., B. Kowalchack, and P. Forcheri. A knowledge-based design facility, Computer Science Technical Report TR-1594, University of Maryland, (December, 1985).

(30) [ZELK86] Zelkowitz M. V. and B. Kowalchack, A knowledge based design facility for a syntax sensitive editor, 9th Minnowbrook Workshop on Software

Performance Evaluation, Blue Mt. Lake, NY (August, 1986).

(31) [ZELK87a] Zelkowitz M. V., An editor for program design, IEEE Compcon, San Francisco CA (February, 1987) 242–246.

(32) [ZELK87b] Zelkowitz M. V., B. Kowalchack, D. Itkin, L. Herman, A SUPPORT tool for teaching computer programming, Software Engineering Institute Conference on Software Engineering Education, Pittsburgh, PA (May, 1987) *(to appear)*

END
9-87
DTIC