# Iterative Enhancement: A Practical Technique for Software Development

VICTOR R. BASILI AND ALBERT J. TURNER

*Abstract*—**This paper recommends the "iterative enhancement" technique as a practical means of using a top-down, stepwise refinement approach to software development. This technique begins with a simple initial implementation of a properly chosen (skeletal) subproject which is followed by the gradual enhancement of successive implementations in order to build the full implementation. The development and quantitative analysis of a production compiler for the language SIMPL-T is used to demonstrate that the application of iterative enhancement to software development is practical and efficient, encourages the generation of an easily modifiable product, and facilitates reliability.**

*Index Terms*—**Iterative enhancement, SIMPL, software analysis, software development, software evaluation measures, top-down design.**

## INTRODUCTION

SEVERAL techniques have been suggested as aids for producing reliable software that can be easily updated to meet changing needs [1]–[4]. These include the use of a top-down modular design, a careful design before coding, modular well-structured components, and a minimal number of implementors. Although it is generally agreed that the basic guideline is the use of a top-down modular approach using "stepwise refinement" [5], this technique is often not easy to apply in practice when the project is of reasonable size. Building a system using a well-modularized, top-down approach requires that the problem and its solution be well understood. Even if the implementors have previously undertaken a similar project, it is still difficult to achieve a good design for a new system on the first try. Furthermore, design flaws often do not show up until the implementation is well underway so that correcting the problems can require major effort.

One practical approach to this problem is to start with a simple initial implementation of a subset of the problem and iteratively enhance existing versions until the full system is implemented. At each step of the process, not only extensions but also design modifications can be made. In fact, each step can make use of stepwise refinement in a more effective way as the system becomes better understood through the iterative process. As these iterations converge to the full solution, fewer and fewer modifications need be made. "Iterative enhancement" represents a practical means of applying stepwise refinement.

This paper discusses the heuristic iterative enhancement algorithm and its application to the implementation of a fully instrumented production compiler for the programming language SIMPL-T [6]. The SIMPL-T project represents a successful practical experience in using the approach in conjunction with several of the standard informal techniques to develop a highly reliable and easily modifiable product in a relatively short amount of time.

The next section of this paper contains a discussion of the basic iterative enhancement method, independent of a specific application. The following section discusses the application of the method as used in the development of the compiler for SIMPL-T, and includes some initial results from a quantitative analysis of the SIMPL-T project.

## OVERVIEW OF THE METHOD

The first step in the application of the iterative enhancement technique to a software development project consists of a simple initial implementation of a skeletal subproblem of the project. This skeletal implementation acts as an initial guess in the process of developing a final implementation which meets the complete set of project specifications. A *project control list* is created that contains all the tasks that need to be performed in order to achieve the desired final implementation. At any given point in the process, the project control list acts as a measure of the "distance" between the current and final implementations.

In the remaining steps of the technique the current implementation is iteratively enhanced until the final implementation is achieved. Each iterative step consists of selecting and removing the next task from the list, designing the implementation for the selected task (the *design phase*), coding and debugging the implementation of the task (the *implementation phase*), performing an analysis of the existing partial implementation developed at this step of the iteration (the *analysis phase*), and updating the project control list as a result of this analysis. The process is iterated until the project control list is empty, i.e., until a final implementation is developed that meets the project specifications.

Although the details of the algorithm vary with the particular problem class and implementation environment,

Manuscript received August 5, 1975. This work was supported in part by the Office of Naval Research under Grant N00014-67-A-0239-0021 (NR-044-431) to the Computer Science Center of the University of Maryland, and in part by the Computer Science Center of the University of Maryland.

V. R. Basili is with the Department of Computer Science, University of Maryland, College Park, Md. 20742.

A. J. Turner is with the Department of Mathematical Sciences, Clemson University, Clemson, S. C.

a set of guidelines can be given to further specify the various steps in the process. The development of the first step, the skeletal initial implementation, may be achieved by defining the implementation of a skeletal subset of the problem. A skeletal subset is one that contains a good sampling of the key aspects of the problem, that is simple enough to understand and implement easily, and whose implementation would make a usable and useful product available to the user. This subset should be devoid of special case analysis and should impose whatever restrictions might be necessary to facilitate its implementation without seriously affecting its usability. The implementation itself should be simple and straightforward in overall design and straightforward and modular at lower levels of design and coding so that it can be modified easily in the iterations leading to the final implementation.

The project control list guides the iterative process by keeping track of all the work that needs to be done in order to achieve the final implementation. The tasks on the list include the redesign or recoding of components in which flaws have been discovered, the design and implementation of features and facilities that are missing from the current implementation, and the solution of unsolved problems. The sequence of lists corresponding to the sequence of partial implementations is a valuable component of the historical documentation of the project.

Each entry in the project control list is a task to be performed in one step of the iterative process. It is important that each task be conceptually simple enough to be completely understood in order to minimize the chance of error in the design and implementation phases of the process.

A major component of the iterative process is the analysis phase that is performed on each successive implementation. The project control list is constantly being revised as a result of this analysis. This is how redesign and recoding work their way into the control list. Specific topics for analysis include such items as the structure, modularity, modifiability, usability, reliability and efficiency of the current implementation as well as an assessment of the achievement of the goals of the project. One approach to a careful analysis is the use of an appropriate set of guidelines as follows.

1) Any difficulty in design, coding, or debugging a modification should signal the need for redesign or recoding of existing components.
2) Modifications should fit easily into isolated and easy-to-find modules. If not, then some redesign is needed.
3) Modifications to tables should be especially easy to make. If any table modification is not quickly and easily done, then a redesign is indicated.
4) Modifications should become easier to make as the iterations progress. If not, then there is a basic problem such as a design flow or a proliferation of "patches."
5) "Patches" should normally be allowed to exist for only one or two iterations. Patches should be allowed, however, in order to avoid redesigning during an implementation phase.
6) The existing implementation should be analyzed frequently to determine how well it measures up to the project goals.
7) Program analysis facilities should be used whenever available to aid in the analysis of the partial implementations.
8) User reaction should always be solicited and analyzed for indications of deficiencies in the existing implementation.

Certain aspects of the iteration process are dependent on the local environment in which the work is being performed, rather than on the specific project. Although the techniques used in the design and implementation phases of each iteration step should basically be top-down stepwise refinement techniques, the specifics can vary depending on such factors as installation standards and the number of people involved. Much has been written elsewhere about such techniques, and they will not be discussed further here. The procedures used in the analysis phase for each partial implementation are dependent upon such local factors as the program analysis facilities available, the programming languages used, and the availability of user feedback. Thus, to some extent the efficient use of the iterative enhancement technique must be tailored to the implementation environment.

In summary, iterative enhancement is a heuristic algorithm that begins with the implementation of a subproblem and proceeds with the iterative modification of existing implementations based on a set of informal guidelines in order to achieve the desired full implementation. Variants of this technique have undoubtedly been used in many applications. However, iterative enhancement is different from the iterative techniques often discussed in the literature, in which the entire problem is initially implemented and the existing implementations are iteratively refined or reorganized [2] to achieve a good final design and implementation.

## APPLICATION OF THE METHOD TO COMPILER DEVELOPMENT

Compiler development falls into a class of problems that can be called *input directed*. Such problems have well-defined inputs that determine the processing to be performed. The application of the iterative enhancement method to compiler development will be discussed in this section. In order to be more specific, it is assumed that the syntax of the language $L$ to be compiled is defined by a context free grammar $G$.

Since a compiler is input directed, the skeletal compiler to be initially implemented can be specified by choosing a skeletal language, $L_0$, for $L$. The language $L_0$ may be slightly modified sublanguage of $L$ with a grammar $G_0$ that is essentially a subgrammar of $G$.

In choosing $L_0$, a small number of features of $L$ are

chosen as a basis. For example, this basis might include one data type, three or four statement types, one parameter mechanism, a few operators, and other features needed to give $L_0$ the overall general flavor of $L$. The language derived from this basis can then be modified for ease of implementation and improved usability to obtain $L_0$.

The remainder of this section describes the use of iterative enhancement in an actual compiler implementation.

### *A Case Study: the SIMPL-T Project*

The iterative enhancement method was used at the University of Maryland in the implementation of a compiler for the procedure-oriented algorithmic language SIMPL-T [6] on a Univac 1108. The SIMPL-T project is discussed in this section, beginning with a brief illustration of the scope of the project.

*Overview:* SIMPL-T is designed to be the base language for a family of programming languages [7]. Some of its features are as follows.

1) A program consists of a set of separately compiled modules.
2) Each module consists of a set of global variables and a set of procedures and functions.
3) The statement types are assignment, if-then-else, while, case, call, exit, and return.
4) The data types are integer, character, and character string.
5) There are extensive sets of operators and intrinsics for data manipulation.
6) There is a one-dimensional array of any data type.
7) Procedures and functions may optionally be recursive.
8) Scalar arguments may be passed by reference or by value; arrays are passed by reference.
9) Procedures and functions may not have internal procedures or functions; neither procedures nor functions may be passed as parameters.
10) There is no block structure (but there are compound statements).
11) Procedures, functions, and data may be shared by separately compiled modules.

Characterizing the overall design of the language, its syntax and semantics are relatively conservative, consistent and uncluttered. There are a minimal number of language constructs, and they are all rather basic. A stack is adequate for the runtime environment. These design features contributed to a reasonably well-defined language design which permitted the development of a reasonably well-understood compiler design.

The following are characteristics and facilities of the SIMPL-T compiler:

1) It is programmed in SIMPL-T and is designed to be transportable by rewriting the code generation modules [8].
2) It generates very good object code on the 1108. (In the only extensive test [9], the code produced was better than that generated by the Univac optimizing Fortran compiler.)
3) Good diagnostics are provided at both compile and runtimes.
4) An attribute and cross-reference listing is available.
5) There are traces available for line numbers, calls and returns, and variable values.
6) Subscript and case range checking are available.
7) There are facilities for obtaining statistics both at compile time and after a program execution.
8) Execution timing for procedures, functions, and separately compiled modules is available.

In summary, the compiler is a production compiler that generates efficient object code, provides good diagnostics, and has a variety of testing, debugging, and program analysis facilities. The compiler itself consists of about 6400 SIMPL-T statements, and the library consists of about 3500 (assembly language) instructions. (The statement count does not include declarations, comments, or spacing. The compiler consists of 17 000 lines of code.)

*The Initial Implementation:* The skeletal language implemented initially in the SIMPL-T project was essentially the language SIMPL-X [10]. Some of the restrictions (with respect to SIMPL-T) imposed for the initial implementation were:

1) There was only one data type (integer).
2) Only call by value was allowed for scalar parameters.
3) All procedures and functions were recursive.
4) Only the first 12 characters of an identifier name were used.
5) Case numbers were restricted to the range 0–99.
6) Both operands of a logical operator (.AND.,.OR.) were always evaluated.

Since the compiler was to be self-compiling, some character handling facility was needed. This was provided by an extension that allowed character data to be packed in an integer variable just as in Fortran.

Restrictions were also made on compiler facilities for the initial implementation. Only a source listing and reasonable diagnostics were provided, leaving the debugging and analysis facilities for later enhancements.

The design of the initial skeletal implementation was a rather straightforward attempt to provide a basis for future enhancements. This allowed the initial implementation to be completed rather quickly so that the enhancement process could get underway. It is instructive to note that while most of the higher level design of the compiler proved to be valid throughout the implementation, most of the lower level design and code was redone during the enhancement process. This illustrates the difficulty in doing a good complete project design initially, especially in light of the fact that the initial implementation was an honest attempt to achieve a good basis upon which to build later extensions.

The importance of using a simple approach in the initial

implementation was illustrated by the experience with the initial SIMPL-X code generation module. Although it was not intended to generate really good code, far too much effort was expended in an attempt to generate moderately good code. As a result, most of the initial debugging effort was spent on the code generator (which was later almost completely rewritten anyhow). A simple straightforward approach would have allowed the project to get underway much faster and with much less effort.

A final comment on the skeletal implementation is that it is clear in retrospect that had the compiler not been self-compiling it would have been better to use an even more restricted subset of SIMPL-T. This was not considered at the time because programming the compiler in the initial subset would have been more difficult.

The design and implementation phases of each iteration were performed using a basic top-down approach. Every attempt was made to ensure a high level of clarity and logical construction.

It is worth noting that the SIMPL-T language itself was also being iteratively enhanced in parallel with the compiler development. As experience was gained by using the language to program the compiler, new features were added and old features were modified on the basis of this experience. Thus user experience played a major role not only in the implementation of the software project (i.e., the compiler) but also in the specification of the project (i.e., the language design).

*The Analysis Phase:* The analysis performed at the end of each iterative step was basically centered around the guidelines given above in the overview of the method. Some of the specific techniques used are briefly discussed below.

Since the intermediate compilers were mostly self-compiling, a large amount of user experience was available from the project itself. This user experience together with the valuable test case provided by the compiler for itself represent two of the advantages of self-compilers.

A second source of user experience in the SIMPL-T project was derived from student use in the classroom. Since classroom projects are not generally ongoing, there was normally no inconvenience to students in releasing the intermediate versions of the compiler as they were completed. These two sources of user experience are examples of how the details of applying iterative enhancement can be tailored to the resources available in the implementation environment.

Testing the intermediate compilers was done by the usual method of using test data. Again the self-compiling feature of the compiler was valuable since the compiler was often its own best test program. The bug farm and bug contest techniques [11] were also used and some of the results are given below.

Timing analyses of the compiler were first done using the University of Maryland Program Instrumentation Package (PIP). PIP provides timing information based on a partition of core and is thus more suitable for assembly language programs than for programs written in higher level languages. However the information obtained from PIP was of some value in locating bottlenecks, especially in the library routines.

When the timing and statistics facilities for object programs were added to the compiler, new tools for analysis of the compiler itself became available. The timing facility has been used to improve the execution speed through the elimination of bottlenecks, and the statistics facilities have been used to obtain information such as the frequency of hashing collisions. Future plans call for further use of the timing information to help improve compiler performance. The statistical facilities were also used to obtain the quantitative analysis discussed at the end of this section.

*Project Summary:* The SIMPL-T project was completed during a 16 calendar month period. Since other activities took place in parallel with the implementation effort, it is difficult to accurately estimate the total effort, but a fairly accurate effort for the language and compiler design, implementation, and maintenance (excluding the bootstrap and library implementations) is 10 man-months. Counting only the code in the final compiler, this time requirement represents an average output of almost 30 statements (75 lines) of debugged code per man-day. It is felt that the use of iterative enhancement was a major contributing factor in this achievement.

Experience has thus far indicated that the compiler is reasonably easy to modify. Two fairly large modifications have been made by people not previously participating in the compiler implementation. One of these efforts involved the addition of a macro facility and in the other, single and double precision reals were added [9]. Both efforts were accomplished relatively easily even though there was little documentation other than the compiler source listing.

Finally, the reliability of the compiler has been quite satisfactory. During the two and one-half month duration of the bug contest a total of 18 bugs were found, many of which were quite minor. (All bugs regardless of severity were counted.) Of course, several additional bugs had been found before the contest and some have been found since, but overall their number has been small. As could be predicted, most of the bugs occurred in the least well understood components: error recovery and code generation.

*Project Analysis:* In an attempt to justify that the heuristic iterative enhancement algorithm gives quantitative results, an extensive analysis of four of the intermediate compilers plus the final compiler was performed. As of this writing (June 1975) the analysis is only in the early stages, but some of the preliminary statistics computed are given in Table I. The interpretation of some of these statistics has not been completed, but they have been included as a matter of interest.

The compilers referenced in Table I are

1) One of the early SIMPL-X compilers (SIMPL-X 2.0).

2) The SIMPL-X compiler after a major revision to correct some structural defects (SIMPL-X 3.1).
3) The first SIMPL-T compiler, written in SIMPL-X (SIMPL-X 4.0).
4) Compiler (3), rewritten in SIMPL-T (SIMPL-T 1.0).
5) The current SIMPL-T compiler at the time of the analysis (SIMPL-T 1.6).

The statistics were computed by using the existing statistical facilities of the SIMPL-T compiler, and by adding some new facilities.

An explanation of the statistics given is as follows.

1) Statements are counted as defined by the syntax. A compound statement such as a WHILE statement counts as one statement plus one for each statement in its statement list.
2) A separately compiled module is a collection of globals, procedures, and functions that is compiled independently of other separately compiled modules and combined with the other modules for execution.
3) A token is a syntactic entity such as a keyword, identifier, or operator.
4) Globals were only counted if they were ever modified. That is, named constants and constant tables were not counted.
5) A data binding occurs when a procedure or function $P$ modifies a global $X$ and procedure or function $Q$ accesses (uses the value of) $X$. This causes a binding $(P,X,Q)$. It is also possible to have the (different) binding $(Q,X,P)$; however $(P,X,P)$ is not counted. The counting procedure was modified so that if $P$ and $Q$ execute only in separate passes and the execution of $P$ precedes that of $Q$, then $(P,X,Q)$ is counted but $(Q,X,P)$ is not counted.

The reasons for choosing these statistics were based on intuition and a desire to investigate quantitatively the data and control structure characteristics of the sequence of compilers.

It is interesting to note that the statistics indicate a trend towards improvement in the compiler with respect to many generally accepted theories of good programming principles, even though the redesign and recoding efforts that caused this trend were done only on the basis of the informal guidelines of the iterative enhancement algorithm. As the project progressed, the trend was toward more procedures and functions with fewer statements, more independently compiled segments, less nesting of statements, and a decrease in the use of global variables. These improvements occurred even though the changes were being made primarily to correct difficulties that were encountered in incorporating modifications during the iterative enhancement process.

The meaning of many of the trends indicated in Table I is clear. For example, due to the difficulties encountered in working with larger units of code, the number of procedures and functions and the number of separately compiled modules increased much more than did the number of statements. Similarly, the decrease in nesting level corresponds to the increase in the number of procedures and functions.

One of the harder to explain sequences of statistics is the average number of tokens per statement. The probable cause for the large jump between compilers 1) and 2) is the relaxation of several Fortran-like restrictions imposed for the initial bootstrap. The more interesting jump between compilers 3), written in SIMPL-X, and 4), written in SIMPL-T, seems to suggest that writing in a more powerful language (SIMPL-T) may also affect the writing style used by a programmer. That is, with more powerful operators more operators are used per statement.

The statistics for globals, locals, and parameters indicate a clear trend away from the use of globals and toward increased usage of locals and parameters. The large drop in the number of globals accessible to the average procedure or function between compilers 3) and 4) and compilers 4) and 5) corresponds to the increase in the number of separately compiled modules for 4) and 5). Splitting one separately compiled module into several modules decreases the number of accessible globals because the globals are also divided among the modules and are usually not made accessible between modules.

The notion of data binding is more complex than the notions considered above and the data binding statistics require more effort to interpret. Note, for example, that if the number of procedures and functions doubles, then the data binding count would most likely more than double due to the interactions between the new and old procedures and functions. Similarly, splitting a separately compiled module into several modules would tend to decrease the number of possible bindings due to the decrease in the number of accessible globals.

In light of these considerations, the data binding counts in Table I seem reasonable except for the decrease in actual bindings from compiler 4) to compiler 5). A more detailed investigation of this decrease revealed that it was primarily due to the elimination of the improper usage of a set of global variables in the code generation component of the compiler. The sharing of these variables by two logically independent sets of procedure had caused several problems in modifying the code generator, and the data accessing was restructured in an attempt to eliminate these problems.

Finally, the percentage of possible data bindings that actually occurred can be interpreted as an indication of how much variables that are declared globally are really used as globals. (If every procedure and function both modified and accessed all its accessible globals, then the percentage would be 100.) As with the other measures, ideal values (in an absolute sense) are not clear, but the trend toward higher values that is shown in Table I is the desired result.

## CONCLUSION

Two major goals for the development of a software product are that it be reasonably modifiable and reliable.

TABLE I
MEASURES MADE ON FIVE DIFFERENT COMPILERS IN THE SIMPL-T PROJECT

| | (1) | (2) | (3) | (4) | (5) |
|---|---|---|---|---|---|
| Number of Statements | 3404 | 4217 | 5181 | 5847 | 6350 |
| Number of Procedures and Functions | 89 | 189 | 213 | 240 | 289 |
| Number of Separately Compiled Modules | 4 | 4 | 7 | 15 | 37 |
| Average Number of Statements per Proc/Func | 38.2 | 22.3 | 24.3 | 24.4 | 22.0 |
| Average Nesting Level | 3.4 | 2.9 | 2.9 | 2.9 | 2.8 |
| Average Number of Tokens per Statement | 5.7 | 6.3 | 6.6 | 7.2 | 7.3 |
| Number of Data Variables: | | | | | |
| Globals | 155 | 132 | 151 | 180 | 193 |
| Locals | 112 | 381 | 496 | 550 | 621 |
| Parameters | 35 | 184 | 215 | 257 | 388 |
| Average Number of Data Variables per Proc/Func: | | | | | |
| Globals | 1.7 | 0.7 | 0.7 | 0.8 | 0.7 |
| Locals | 1.3 | 2.0 | 2.3 | 2.3 | 2.1 |
| Parameters | 0.4 | 1.0 | 1.0 | 1.1 | 1.3 |
| Percentage of: | | | | | |
| Globals | 51.3 | 18.9 | 17.5 | 18.2 | 16.1 |
| Locals | 37.1 | 54.7 | 57.5 | 55.7 | 51.7 |
| Parameters | 11.6 | 26.4 | 24.9 | 26.0 | 32.3 |
| Average Number of Globals Accessible by a Proc/Func | 52.0 | 52.2 | 57.4 | 33.9 | 22.3 |
| Number of Actual Data Bindings | 2610 | 6662 | 8759 | 12006 | 10442 |
| Number of Possible Data Bindings | 243780 | 814950 | 1337692 | 497339 | 342727 |
| Percentage of Possible Bindings that Occurred | 1.1 | 0.8 | 0.7 | 2.4 | 3.0 |

This paper recommends the iterative enhancement technique as a methodology for software development that for many projects facilitates the achievement of these goals and provides a practical means of using a top-down stepwise refinement approach.

The technique involves the development of a software product through a sequence of successive design and implementation steps, beginning with an initial "guess" design and implementation of a skeletal subproblem. Each step of the iterative process consists of either a simple, well-understood extension, or a design or implementation modification motivated by a better understanding of the problem obtained through the development process.

It is difficult to make a nonsubjective qualitative judgment about the success of a software technique. However the preliminary statistics from an analysis of the SIMPL-T project do indicate some desirable quantitative results. These statistics suggest that the informal guidelines of the heuristic iterative enhancement algorithm encourage the development of a software product that satisfies a number of generally accepted evaluation criteria.

The measure of accomplishment for the SIMPL-T project was based upon relative improvement with respect to a set of measures. A question remains as to what are absolute measures that indicate acceptable algorithm termination criteria. More work on several different projects and studies of the implications of these measures are needed to help determine some quantitative characteristics of good software.

A need also exists for developing a formal basis for software evaluation measures. An analytical basis for evaluation would not only increase the understanding of the meaning of the measures but should also shed some light on appropriate absolute values that indicate the achievement of good characteristics.

The implementation and analysis of the SIMPL-T system have demonstrated that not only is the iterative enhancement technique an effective means of applying a modular, top-down approach to software implementation, but it is also a practical and efficient approach as witnessed by the time and effort figures for the project. The development of a final product which is easily modified is a by-product of the iterative way in which the product is developed. This can be partially substantiated by the ease with which present extensions and modifications can be made to the system. A reliable product is facilitated since understanding of the overall system and its components is aided by the iterative process in which the design and code are examined and reevaluated as enhancements are made.

## REFERENCES

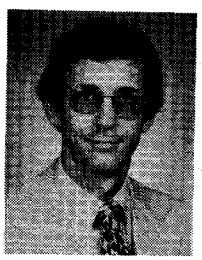
[1] H. D. Mills, "On the development of large, reliable programs," *Rec. 1973 IEEE Symp. Comp. Software Reliability*, Apr. 1973, pp. 155–159.

[2] ——, "Techniques for the specification and design of complex programs," in *Proc. 3rd Texas Conf. Computing Systems*, Univ. Texas, Austin, Nov. 1974, pp. 8.1.1–8.1.4.
[3] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, *Structured Programming*. London: Academic, 1972.
[4] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Commun. Ass. Comput. Mach.*, vol. 15, pp. 1053–1062, Dec. 1972.
[5] N. Wirth, "Program development by stepwise refinement," *Commun. Ass. Comput. Mach.*, vol. 14, pp. 221–227, Apr. 1971.
[6] V. R. Basili and A. J. Turner, "SIMPL-T: a structured programming language," Univ. of Maryland, Comp. Sci. Ctr., CN-14, Jan. 1974.
[7] V. R. Basili, "The SIMPL family of programming languages and compilers," Univ. of Maryland, Comp. Sci. Ctr., TR-305, June 1974.
[8] V. R. Basili and A. J. Turner, "A transportable extendable compiler," in *Software—Practice and Experience*, vol. 5, 1975, pp. 269–278.
[9] J. McHugh and V. R. Basili, "SIMPL-R and its application to large, sparse matrix problems," Univ. of Maryland, Comp. Sci. Ctr., TR-310, July 1974.
[10] V. R. Basili, "SIMPL-X, a language for writing structured programs," Univ. Maryland, Comp. Sci. Ctr., TR-223, Jan. 1973.
[11] M. Rain, "Two unusual methods for debugging system software," in *Software—Practice and Experience*, vol. 3, pp. 61–63, 1973.

**Victor R. Basili** was born in New York, N.Y., on April 13, 1940. He recieved the B.S. degree in mathematics from Fordham College, New York, N.Y., the M.S. degree in mathematics from Syracuse University, Syracuse, N.Y., and the Ph.D. degree in computer science from the University of Texas, Austin, in 1961, 1963, and 1970, respectively.

From 1963 to 1967 he was with the Department of Mathematics and Computer Science, Providence College, as an Instructor, and for the latter two years, as Assistant Professor. From 1970 to 1975 he was Assistant Professor, and is currently Associate Professor, in the Department of Computer Science, University of Maryland, College Park. He is a consultant with the Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, Hampton, Va., the Naval Research Laboratory, Washington, D.C., and the Naval Surface Weapons Center, Dahlgren Laboratory, Dahlgren, Va. He has been involved in the design and development of the graph algorithmic language, GRAAL, the SIMPL family of programming languages and compilers, and the SL/1 language for the CDC Star computer. His special fields of interest include design, implementation, modeling, and analysis of programming languages and software methodology.

Dr. Basili is a member of the Association for Computing Machinery, the IEEE Computer Society, and the American Association of University Professors.



**Albert J. Turner** received the B.S. and M.S. degrees in mathematics from the Georgia Institute of Technology, Atlanta, and is currently a candidate for the Ph.D. degree in computer science from the University of Maryland, College Park.

Software development efforts in which he has had a major role include the implementation of an administrative data processing system at West Georgia College, and the development of the SIMPL family of programming languages and compilers at the University of Maryland. He is currently a faculty member in the Department of Mathematical Sciences, Clemson University, Clemson, S.C. His major interests are the design, modeling, and implementation of programming languages, and the design and implementation of computer software.