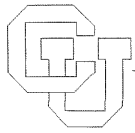
A Software Lifecycle Methodology and Tool Support

Leon J. Osterweil

CU-CS-154-79



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

A SOFTWARE LIFECYCLE METHODOLOGY AND TOOL SUPPORT

Leon J. Osterweil
Department of Computer Science
University of Colorado at Boulder
Boulder, Colorado 80309

CU-CS-154-79 April, 1979

INTERIM TECHNICAL REPORT
U.S. ARMY RESEARCH OFFICE
CONTRACT NO. DAAG29-78-G-0046

THE FINDINGS IN THIS REPORT ARE NOT TO BE CONSTRUED AS AN OFFICIAL DEPARTMENT OF THE ARMY POSITION, UNLESS SO DESIGNATED BY OTHER AUTHORIZED DOCUMENTS.

We acknowledge U.S. Army Research support under contract no. DAAG29-78-G-0046 and National Science Foundation support under grant no. MCS77-02194.

ABSTRACT

This paper describes a system of techniques and tools for aiding in the development and maintenance of software. Improved verification techniques are applied throughout the entire process and management visibility is greatly enhanced. The paper discusses the critical need for improving upon past and present methodology. It presents a proposal for a new production methodology, a verification methodology, and the system architecture for a family of support tools.

INTRODUCTION

There has been growing interest recently in the problem of producing high-quality software at reasonable cost [1-6]. The cost of producing programs has been observed to range up to and sometimes beyond $200 per line [7]. In spite of these costs, embarrassing and occasionally disastrous errors and shortcomings have been found in such code. People actively involved in software development have become all too accustomed to a variety of problems, including

cost/schedule overruns,
poor visibility into development status,
unreliability,
maintenance difficulties,
inconclusive verification, and
inadequate or nonexistent documentation

These problems have received a lot of attention during the last few years, and the quest for improved, modern software practices has been generally a search for ways to eliminate or at least alleviate these problems wherever possible.

As a consequence of intense multidisciplinary investigation of these and related problems, some basic findings have emerged [8-11]. The key findings are that software is an intangible product and that it is critically important that its production be carefully managed. Unfortunately software management is currently more of an art than an exact science. The reasons are not hard to find. First, software is not tangible; hence much management science does not apply directly. Second, there are few if any basic software development principles and disciplines.

In view of the growing magnitude of U.S. software activities (currently estimated at $10-20 billion per year [12,13]), it is not surprising that considerable effort is being spent on discovering workable software development and management principles. An important step in this direction is the realization that software production is an activity that properly takes place in phases, and that it should be managed as such. The phased approach to software development is now widely accepted as a basis for improving project cost effectiveness through improved

visibility and control.

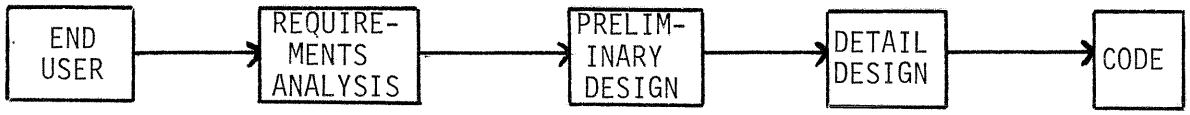


Figure 1. Phased approach to software development.

Figure 1 illustrates typical names and the usual ordering of some of these phases. Generally, the first phase, requirements analysis, should result in the production of a requirements document specifying the end user's needs and wishes for the software. The next phase, preliminary design, should be the identification and analysis of the functional capabilities needed to achieve the requirements. The next phase, detail design, should be the derivation and definition of specific data aggregates and algorithmic modules capable of effecting these functional capabilities. The final step, coding, is then the process of implementing these specifications as computer source code.

Many discussions of the phased approach also include documentation, testing and maintenance as sequential phases of software production. It is our opinion that documentation and testing should not be considered phases, but rather pervasive activities throughout the development process. The next section explores this idea more fully. Maintenance also should not be considered a sequential phase, but rather an activity continuing throughout the useful life of the software.

Maintenance has become a catchall term for all activities occurring after the code is declared operational. In practice these activities are quite diverse, encompassing such things as (1) correcting coding errors, (2) repairing design flaws (and impacted code), and (3) upgrading of basic capabilities (resulting in redesign and recoding). It now becomes clear why 50-60% of total software lifecycle costs are ascribed to maintenance [14,15]. Figure 2 illustrates this notion of maintenance as the iterative alteration and correction of requirements, design, and code.

We believe that the greatest benefits of this phased conceptualization of software development and maintenance will not be obtained until

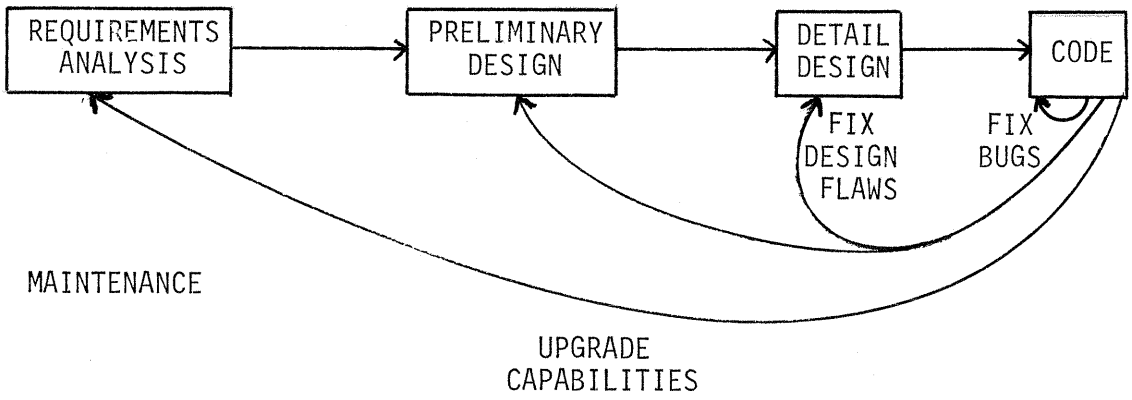


Figure 2. The maintenance process.

the conceptualization is supported by adequate tools and automation [16-18]. Specifically, what appear to be needed are tools and techniques to (1) facilitate the transition from one development phase to the next and (2) determine that the transitions have been made correctly. This paper proposes an integrated tool-supported methodology being designed to address effectively both of these objectives.

LIFE CYCLE VERIFICATION

Careful management throughout the life cycle is critical to the success of any software project. This careful management must be based upon adequate visibility into the development (and hence maintenance) process. The phased approach dictates that milestones be inserted into the process as monitoring points. By itself, however, it does not specify how this monitoring is to be done. Clearly the tantalizingly intangible nature of the evolving software product is the problem.

Thus the driving philosophy behind our approach is that project related products and information be made as visible and tangible as possible. It is important to observe that such things as reports, summaries, and analyses must be considered key project information. Indeed, such information may be more useful in improving project visibility and manageability than more obvious and mundane items, such as listings and design diagrams. For this reason, much emphasis is placed upon techniques for producing useful reports, summaries, and analyses at all phases of the development (and hence, maintenance) cycle.

Ideally, these reports, summaries, and analyses will be automatically drawn from rigorous representations of requirements, design, and code. Thus important emphasis is also placed upon rigor, formality, and machine readability of all project source materials.

If this is done, then thorough, objective, complete reports on project status can be easily and automatically generated at critical points in the life cycle. These reports would represent both the status of the project and inferences drawn from source materials. Verification of the soundness of efforts in a given development phase is then obtainable by a comparison of the inferences drawn from one phase to status summaries and inferences drawn from the previous phase.

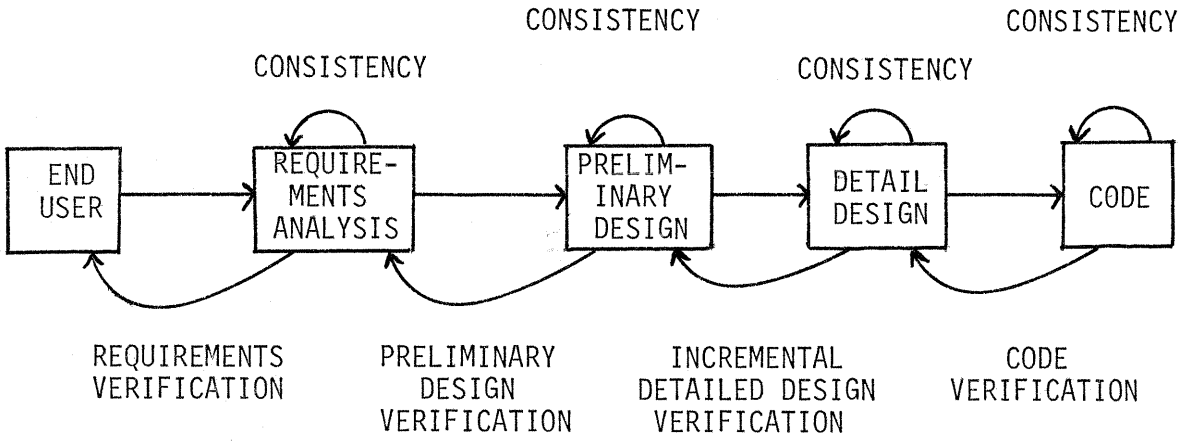


Figure 3. Lifecycle verification.

Figure 3 illustrates this idea and embodies the important principle that verification and testing are activities that must occur during every phase of the development and maintenance cycles. These incremental verification steps are exactly what are needed to assure that the software product is developing satisfactorily. Their purpose is to provide management (and project personnel) with the perceptions and insight needed to prevent drift, poor coordination, and misdirection.

Thus, for example, in Figure 3 we see that a verification of requirements back to the end user is dictated. This would be supported by the creation of reports based upon the requirements as specified. The

reports would give the results of consistency cross-checks and analyses of the interplay among requirements. Clearly this is most effectively done if the requirements are represented in a rigorous, unambiguous, machine readable format. More on this is presented in a later section.

Figure 3 also shows a verification of preliminary design to requirements. This verification would be supported by reports on the consistency of data flows and interfaces within the design. More important, however, is that functional effects and characteristics of the designed system could be inferred from a rigorous, machine readable design representation. In comparing these inferred effects to the rigorous statement of required effects, a meaningful verification is obtained.

That verification could then serve as a basis for a management decision to proceed with the detailed design activity as planned. Here too it is possible to verify that the effect of the detailed design specification achieves the functional capabilities and performance characteristics promised by the preliminary design, provided that both are in rigorous, unambiguous, machine readable format and that analytic tools are available. In actuality we view design as a multistage hierarchical process with verification occurring at each incremental stage. This is described more fully in a later section of this paper.

Finally, Figure 3 shows a verification of the actual code to detail design. In this activity the actual code is automatically scrutinized by automated tools. Reports on the internal consistency and soundness of the code are produced. More important, however, is that inferences about the effect of the code can be drawn for comparison to detail design specifications. This verification is perhaps the most familiar because numerous tools of this type have been produced in recent years. Their potential effectiveness has not been fully achieved because they have not usually been coupled with the rigorous design specifications needed for thorough verification. Nevertheless, these early tools and techniques are extremely important to us. They serve as models of the tools and techniques needed for verification of the earlier phases of the software life cycle. Further, the weaknesses of these early efforts serve to underscore the importance of rigor and machine readability at

all phases of the life cycle as the basis for verification, visibility, and hence manageability [19-21].

INTEGRATED VERIFICATION METHODOLOGY

In this section an overview of a verification methodology is presented. This verification methodology has been evolved previously for application to source code [16,20]. We have observed that it seems applicable, however, to any rigorous algorithmic or combinatorial expression of a problem or its solution. In this section the methodology itself is sketched; its applicability to the various life cycle phases is shown later.

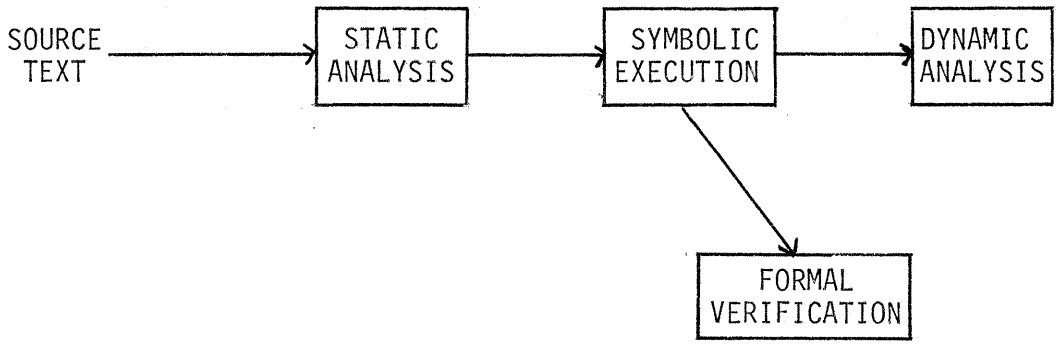


Figure 4. Integrated verification methodology.

Figure 4 shows the juxtaposition of the four major techniques used as components of the integrated verification methodology. As shown in Figure 4, incoming source representations (code, design representations, or requirements representations) are first scanned by a static analyzer. Static analyzers are capable of examining algorithmic representations for inconsistencies and certain errors without requiring actual or simulated execution. Systems such as the DAVE[20-22] static analysis system have proven to be useful in this way.

DAVE is capable of inferring the nature of data flows both within and between modules of FORTRAN programs. The reports of these inference scans are useful documentation, providing visibility to project personnel and management. Further, instances of inconsistent data flow can be detected and reported as errors. DAVE can also demonstrate the absence of certain data flow errors such as uninitialized variable references and

mismatched subprogram invocation lists. This also is useful management information.

Examination of the nature of DAVE's analysis shows that the analysis is actually performed on a graph representation of the source program. Hence DAVE's basic analytic capabilities seem equally applicable to graph and algorithmic representations such as those available during design and requirements. This seems to be a common characteristic of most static analysis techniques. Hence static analysis of data flow and algorithmic consistency is the logical first step in providing visibility and verification at each phase of the software life cycle.

Dynamic analysis lies at the other end of the methodology pictured in Figure 4. In dynamic analysis explicit inputs to an algorithmic process specification are used to explore the actual functioning of the process. This provides a different kind of visibility and enables different verification. Whereas static analysis was able to ferret general descriptions of data flows out of a general representation, dynamic analysis is able precisely to identify improper handling of specific input scenarios. With dynamic analysis the exact effect of a specified scenario can be determined. This is invariably the most important kind of visibility to project management and to a customer. Verification can be derived from this visibility by comparing observed execution effects to precise statements of intent.

Perhaps the most significant work in this area is the PET system [23,24]. The PET system and a prototype PL/1 Automated Verification System are designed to monitor, respectively, executing FORTRAN and PL/1 programs for adherence to specified statements of intent. The statements of intent are to be created by the designers, developers, and testers, and may employ the full power of the First Order Predicate Calculus either locally or globally within the program. Hence if the statements of intent embody a program's detailed design, these systems are capable of verifying the correct implementation of functions to handle expected input scenarios.

Careful consideration of this technique shows that this dynamic analysis capability is applicable to any algorithmic specification that has flow of control and contains representations of functional

transformations for all modules. Hence dynamic verification is seen to be an extremely important capability applicable to the verification of designs and code. The approach can also be applied to simulated processes used to model early requirements and analyze their interactions.

Dynamic analysis techniques provide definitive visibility and verification for specific input data sets and scenarios, but general visibility can be difficult and expensive to achieve. Static analysis is capable of wide scope, but is less capable of specifics and details. In an important sense the two techniques are nicely complementary, but an important middle ground needs to be more fully addressed.

This middle-ground capability, specific detailed visibility and verification for classes of algorithmic scenarios, is supplied by a relatively new technique known as symbolic execution. Experiments in symbolic execution of source code have been carried out by Clarke [25], Howden [26-28] and King [29]. These results have shown that this technique is capable of providing precise visibility into the functional effect of specific paths and classes of paths through a source program. Clarke and King have also shown that automatic constraint solving and theorem proving techniques can be coupled with symbolic execution to achieve verification. Their work shows that source code can sometimes be shown to adhere to specific statements of intent not unlike those employed by the PET system.

Work to date on symbolic execution shows that this technique is applicable to source code but, is probably better applied to designs and requirements specifications. Symbolic execution is capable of portraying functional effect to whatever level of detail is specified by the input source text. The experiments on code indicate that too much detail is present in code. This results in excessively long and cumbersome expressions of effect and proves to be an obstacle to visibility, rather than an aid. Further, the excessive detail complicates automated verification. Higher-level designs and requirements are inherently freer of detail and thus are typically more amenable to symbolic execution.

All of this is excellent justification for placing symbolic execution methodologically in between static analysis and dynamic analysis,

as shown in Figure 4.

This placement is further supported by observing that essentially the same statements of intent have been used as the basis for verification using both symbolic execution and dynamic analysis. Symbolic execution, however, has been shown to be effective only in some cases. This suggests that when verification is desired, symbolic execution should be attempted first because stronger, more general results are possible. Dynamic analysis might then be employed to verify specific cases for which symbolic execution verification attempts failed.

Experiments have shown that for actual source code, dynamic analysis is likely to be the more successful verification technique. In dealing with designs, however, it appears that the loss of detail will make symbolic execution more effective and dynamic analysis less effective. This shift in effectiveness should become more pronounced at higher levels of design. Finally in verifying requirements, it appears that symbolic execution shows much promise. It is important to note that methodologically this implies that much important definitive verification will be achievable solely with symbolic execution early in the program development cycle. Accordingly, detailed verification emphasis should shift gradually to dynamic analysis as the coding phase is approached and begun.

Formal verification is the final technique contained in the integrated verification methodology. Formal verification is best viewed as the logical outgrowth of symbolic execution (although historically the reverse has been true). In formal verification the complete definitive functional effect of an algorithmic specification is determined and compared to the complete definitive statement of the program's intent. The determination of effect is made by symbolically executing every algorithmic path. This is the sense in which formal verification can be viewed as an outgrowth of symbolic execution, while the distinction between the two is based upon thoroughness and completeness.

Thus, as was the case for symbolic execution, the expected effectiveness and practicality of formal verification is expected to be greatest for higher-level designs. Formal verification of actual code is not expected to be effective at all because of the inundating effect of

excessive detail. Interestingly, experience has shown this to be the case. Most researchers advocate the application of formal verification to high-level algorithmic outlines [30-32]. Formal verifications of actual code, on the other hand, exhibit graphically the numbing effect of the detail contained in code [30,31]. Thus, formal verification is incorporated into our proposed methodology as an option that is expected to be most effectively exercisable only at the higher levels of requirements and design.

ARCHITECTURE OF A PROPOSED IMPLEMENTATION

The preceding section has described a methodology capable of providing for visibility and verification at each phase of the software development cycle. It was shown earlier that these are critical capabilities needed in order to manage software development. It was shown, moreover, that visibility and verification are equally as necessary in the management of a successful software maintenance activity.

In this section we present the general outlines of an architecture for a system capable of supporting the development and maintenance of software. The architecture dictates an integrated visibility and verification functional capability applicable at all stages of the development and maintenance cycles. Thus it supplies the informational basis for effective project management. It also incorporates editing, graphics, and file management capabilities necessary for conveniently accessing data and implementing decisions.

The heart of the proposed system is a data base containing all of the information needed for making and implementing management decisions about a given program. Thus the data base is to contain source code, object code, documentation, support libraries, and project utilities. In this respect it helps fulfill the librarian functions of the chief programmer team concept [33].

In addition, the requirements and design specifications for the program must also reside in the data base. This reflects the philosophy that a program is much more than code that executes on a computer. A program is a systematic orderly plan for solving a problem. As such it

must contain a clear expression of the nature of the problem as well as the solution to the problem. Hence the program requirements and all available levels of design are integral components of the program and must reside in the program data base.

If all of these essential program components are placed in a centrally accessible data base, project personnel and management then have access to all of those materials without which they cannot effectively do their jobs. The architecture dictates user interface and internal structuring to facilitate this access as well as to restrict alteration of critical components. In this way the data base management system serves as an extension and implementation of the project configuration management scheme. The data base also reflects our stress on management through visibility and verification, as it contains the reports and analyses produced by the components of the verification methodology. The data base management system must be designed to facilitate management access to these reports because the reports most effectively convey the project status.

The components of the integrated verification methodology might be viewed as part of the data base management system. They must be capable of being invoked to produce analytical reports on appropriate source text within the data base. They must then leave their reports within the data base also. By simplifying and placing the execution of analytic capabilities at the disposal of project management, a means is provided for gaining visibility when it is needed and in a variety of powerful ways. This provides a strong basis for decision making. As already noted, moreover, such data base manipulation capabilities provide a means for implementing certain decisions (e.g., reject or incorporate modules, retest or elaboratively continue testing other modules). Here too we see that the data base management system implements many important configuration management and control functions.

Figure 5 is a diagram of the architecture as just described. It is important to note that the verification processes pictured here are exactly those shown in Figure 3. Figure 5 shows, however, that all are supported by the same core of analytic capabilities represented

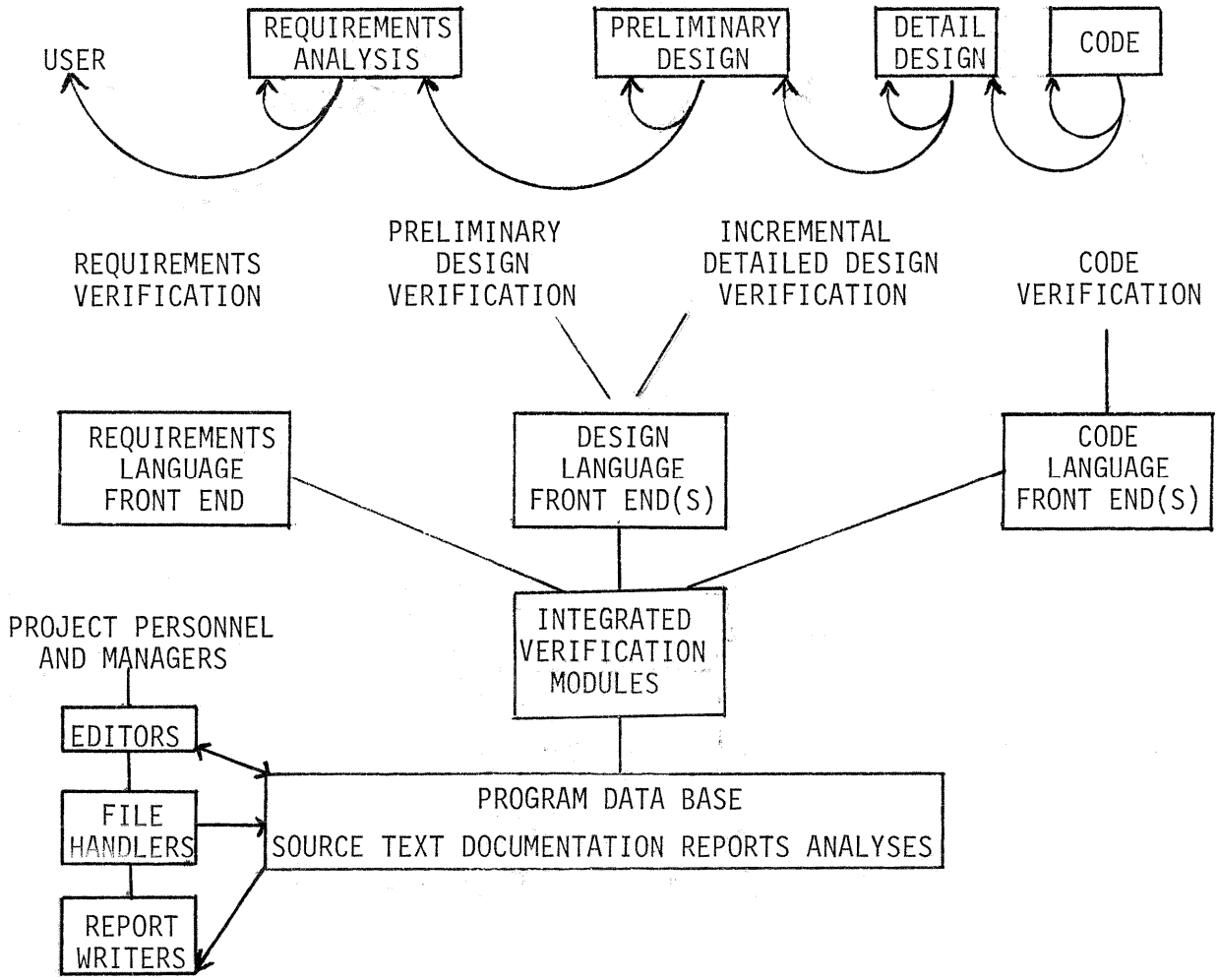


Figure 5. System architecture.

in Figure 4. This is possible only if the requirements and design are captured and stored in rigorous unambiguous formats; if so, then syntax analyzers for each format (as well as for all code languages) could be created. These would then be used as front ends to produce standard representations for analysis by the modules of the integrated verification methodology.

Rigorous requirements and design notations are currently receiving considerable attention [34-37]; thus these assumptions seem quite justified. Interestingly enough, early experience with them indicate that the discipline of representing requirements and design rigorously is highly beneficial in itself [24,37]. The philosophy of compelling this is thus deemed an advantage rather than an obstacle.

ACKNOWLEDGMENTS

The ideas presented here were stimulated and shaped by conversations with Ted Biggerstaff, Lori Clarke, John Darringer, Lloyd Fosdick, Ed Foudriat, Bob Glass, Linda Hammond, Bill Howden, Dave Kasik, Sharon Lamb, Vern Leck, H. F. Lee, Larry Peters, Bill Riddle, Dick Robinson, Bill Rzepka, Ed Senn, Mark Smith, Terry Straeter, Dick Taylor, Armand Vito, and Roger Weber.

REFERENCES

1. J. R. Brown, Getting Better Software Cheaper and Quicker, in Practical Strategies for Developing Large Software Systems, Addison-Wesley, Reading, Massachusetts, 1975, pp. 131-154, 19.
2. Proceedings 1975 International Conference on Reliable Software, IEEE Cat. No. 75 CHO 940-7CSR, Los Angeles, 1975.
3. Proceedings Second International Conference on Software Engineering, IEEE Cat. No. 76CH 1125-4C, San Francisco, 1976.
4. Proceedings Third International Conference on Software Engineering, Atlanta, 1978.
5. IEEE Transactions on Software Engineering, SE Series.
6. J. R. Brown, A. J. DeSalvio, D. E. Heine, and J. G. Purdy, Automated Software Quality Assurance, in Program Test Methods (W. C. Hetzel,ed.) Prentice-Hall, Englewood Cliffs, N. J., 1973, pp. 181-203.
7. B. W. Boehm, The High Cost of Software, in Practical Strategies for Developing Large Software Systems (E. Horowitz, ed.), Addison-Wesley, Reading, Massachusetts, 1975.
8. J. R. Brown, Improving Quality and Reducing Cost of Aeronautical Systems Software Through Use of Tools, in Proceedings of Air Force Aeronautical Systems Software Workshop, April 1974.
9. R. D. Williams, Managing the Development of Reliable Software, in Proceedings of the International Conference on Reliable Software. April 1975, pp. 3-8.
10. R. K. E. Black, Effects of Modern Programming Practices on Software Development Costs, in Proceedings of Fall Compcon 77, September 1977, pp. 250-253.
11. J. R. Brown, Programming Practices for Increased Software Quality, in Software Quality Management, Petrocelli Books, New York City, 1978.
12. W. E. Carlson, Software Research in the Department of Defense, Proceedings Second International Conference on Software Engineering. IEEE Cat. No. 76 CH 1125-4c, pp. 379-383.
13. B. W. Boehm, Software and Its Impact: A Quantitative Assessment, Datamation, May 1973, pp. 48-59.

14. D. S. Alberts, The Economics of Software Quality Assurance, AFIPS Conference Proceeding 45, 433-442 (1976).

15. J. S. Gansler, The DOD Defense Systems Software Management Program--Current Status, Software Management Conference Proceedings, Winter 1977-1978 Series AIAA-DPMA, pp. 5-11.

16. L. J. Osterweil, A Proposal for an Integrated Testing System for Computer Programs, University of Colorado Department of Computer Science Technical Report No. CU-CS-093-76, August 1976.

17. J. R. Brown and R. H. Hoffman, Automating Software Developement: A Survey of Techniques and Automated Tools, TRW-SS-72-03, May 1972.

18. D. J. Reifer, Automated Aids for Reliable Software, Proceedings of the International Conference on Reliable Software, April 1975, pp. 131-142.

19. J. R. Brown, Why Tools?, Proceedings of Computer Science and Statistics: Eighth Annual Symposium on the Interface, February 1975, pp. 310-312.

20. L. J. Osterweil and L. D. Fosdick, Some Experience with DAVE--A FORTRAN Program Analyzer, AFIPS Conference Proceedings 45, 909-916(1976).

21. L. J. Osterweil, A Methodology for Testing Computer Programs, AIAA Conference on Computers in Aerospace, Los Angeles, November 1977, pp. 52-62.

22. L. J. Osterweil and L. D. Fosdick, DAVE--A Validation Error Detection and Documentation System for FORTRAN Programs, Software Practice and Experience 6, 473-486 (September 1976).

23. L. G. Stucki and G. L. Foshee, New Assertion Concepts for Self-Metric Software Validation, Proceedings 1975 International Conference on Reliable Software. IEEE Cat. No. 75-CH09 40-7CSR, 1975, pp. 59-71

24. L. G. Stucki, The Use of Dynamic Assertions to Improve Software Quality, Ph.D. Dissertation, School of Engineering, University of California at Los Angeles, June 1976.

25. L. A. Clarke, A System to Generate Test Data and Symbolically Execute Programs, IEEE Transactions on Software Engineering SE-2, 215-222 (September1976).

26. W. E. Howden, Experiments with a Symbolic Evaluation System, AFIPS Conference Proceedings 45, 899-908 (1976).

27. W. E. Howden, DISSECT--A Symbolic Evaluation and Program Testing System, IEEE Transactions on Software Engineering SE-4, 70-73 (January 1978).

28. W. E. Howden and L. G. Stucki, Final Report Methodology for the Effective Test Case Selection, Phase II. McDonnell Douglas Technical Report MDC G5800, April 1975.

29. J. C. King, Symbolic Execution and Program Testing, CACM 19, 385-394 (July 1976).

30. D. I. Good, R. L. London, and W. W. Bledsoe, An Interactive Program Verification System, 1975 International Conference on Reliable Software. IEEE Cat. No. 75-CH0940-7CSR, 1975, pp. 482-492.

31. B. Elspas, K. N. Levitt, R. J. Waldinger, and A. Waksman, An Assessment of Techniques for Proving Program Correctness ACM Computing Surveys, 4, 97-147 (June 1972).

32. R. L. London, A View of Program Verification, 1975 International Conference on Reliable Software, IEEE Cat. No. 75-CH0940-7CSR, 1975, pp. 534-545.

33. F. T. Baker, Chief Programmer Team Management of Production Programming, IBM Systems Journal 11, 56-73 (1972).

34. M. W. Alford, A Requirements Engineering Methodology for Real-Time Processing Requirements, IEEE Transactions on Software Engineering SE-3, 60-69 (January 1977).

35. D. T. Ross and K. E. Schoman, Jr., Structured Analysis for Requirements Definition, IEEE Transactions on Software Engineering SE-3, 6-15 (January 1977).

36. S. A. Stephens and L. L. Tripp, A Requirements Expression and Validation Tool, Proceedings Third International Conference on Software Engineering, Atlanta, May 1978.

37. J. R. Brown, Functional Programming Final Technical Report, TRW Technical Report No. 29580-6001-RU-00, July 1977.

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: CU-CS-154-79

2. GOVT ACCESSION NO.:

3. RECIPIENT'S CATALOG NUMBER:

4. TITLE (and Subtitle): "A Software Lifecycle Methodology and Tool Support"

5. TYPE OF REPORT & PERIOD COVERED:

6. PERFORMING ORG. REPORT NUMBER:

7. AUTHOR(s): Leon J. Osterweil

8. CONTRACT OR GRANT NUMBER(s): DAAG29-78-G-0046
MCS77-02194 (NSF)

9. PERFORMING ORGANIZATION NAME AND ADDRESS:
Dept. of Computer Science
University of Colorado at Boulder
Boulder, Colorado 80309

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS:

11. CONTROLLING OFFICE NAME AND ADDRESS:
U. S. Army Research Office
Post Office Box 12211
Research Triangle Park, NC 27709

12. REPORT DATE: April, 1979

13. NUMBER OF PAGES: 15

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office):

15. SECURITY CLASS. (of this report): Unclassified

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE: NA

16. DISTRIBUTION STATEMENT (of this Report):

Approved for public release; distribution unlimited.

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

NA

18. SUPPLEMENTARY NOTES:

The findings in this report are not to be construed as an official Department of the Army position, unless so designated by other authorized documents.

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

Testing; Verification, Data Flow Analysis; Software Tools

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

This paper describes a system of techniques and tools for aiding in the development and maintenance of software. Improved verification techniques are applied throughout the entire process and management visibility is greatly enhanced. The paper discusses the critical need for improving upon past and present methodology. It presents a proposal for a new production methodology, a verification methodology, and the system architecture for a family of support tools.

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

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)