

PROCEEDINGS
OF
ELEVENTH ANNUAL SOFTWARE ENGINEERING WORKSHOP

Organized by:
Software Engineering Laboratory
GSFC

December 3, 1986

GODDARD SPACE FLIGHT CENTER
Greenbelt, Maryland

Page intentionally left blank

FOREWORD

The Software Engineering Laboratory (SEL) is an organization sponsored by the National Aeronautics and Space Administration Goddard Space Flight Center (NASA/GSFC) and created for the purpose of investigating the effectiveness of software engineering technologies when applied to the development of applications software. The SEL was created in 1977 and has three primary organizational members:

NASA/GSFC (Systems Development and Analysis Branch)

The University of Maryland (Computer Sciences Department)

Computer Sciences Corporation (Flight Systems Operation)

The goals of the SEL are (1) to understand the software development process in the GSFC environment; (2) to measure the effect of various methodologies, tools, and models in this process; and (3) to identify and then to apply successful development practices. The activities, findings, and recommendations of the SEL are recorded in the Software Engineering Laboratory Series, a continuing series of reports that includes this document.

Single copies of this document can be obtained from

NASA Scientific and Technical Installation Facility
P.O. Box 8757
B.W.I. Airport, Md. 21240

Page intentionally left blank

AGENDA

ELEVENTH ANNUAL SOFTWARE ENGINEERING WORKSHOP
NASA/GODDARD SPACE FLIGHT CENTER
BUILDING 8 AUDITORIUM
DECEMBER 3, 1986

- 8:00 a.m. Registration--'Sign-In'
Coffee, Donuts
- 8:45 a.m. INTRODUCTORY REMARKS
Frank E. McGarry
(NASA/GSFC)
- 9:00 a.m. Session No. 1
Topic: Research in Software
Engineering Laboratory (SEL)
- "Determining Software Productivity
 Leverage Factors" Frank E. McGarry
 (NASA/GSFC)
- "Studies of Software Methods and
 Environments" Vic Basili/Marv Zelkowitz
 (University of Maryland)
- "Designing with ADA for Satellite
 Simulation" Bill Agresti (CSC)
- 10:30 a.m. BREAK
- 11:00 a.m. Session No. 2
Topic: Empirical Studies of
Software Technology
- Discussant: Jerry Page (CSC)
- "Studying Software Engineering Documen-
 tation From a Cognitive Perspective" Elliot Soloway (YALE)
- "Empirical Research on the Design Process" Vincent Shen (MCC)
- "A Quantitative Analysis of the Impact of
 Modern Software Engineering Techniques" John Gaffney (IBM)
- 12:30 p.m. LUNCH
- 1:30 p.m. Session No. 3
Topic: Software Environments
- Discussant: Keiji Tasaki
(NASA/GSFC)

AGENDA (Continued)

	“The Mothra Software Testing Environment”	Eugene Spafford (Georgia Tech)
	“A Value-Chain Analysis of Software Productivity Components”	Barry Boehm (TRW)
	“The Open Architecture of the IDE Tool Environment”	Tony Wasserman (IDE, Inc.)
3:00 p.m.	BREAK	
3:30 p.m.	Session No. 4	Topic: Software Testing
		Discussant: Ed Seidewitz (NASA/GSFC)
	“Data Diversity--A New Approach to Fault-Tolerant Software”	John Knight (University of Virginia)
	“An Empirical Study of Error Detection Using Self-Test”	Nancy Leveson (University of California, Irvine)
	“An Experimental Comparison of Ada and FORTRAN Program Reliability”	Amrit Goel (Syracuse University)
5:00 p.m.	ADJOURN	

SUMMARY OF THE ELEVENTH ANNUAL SOFTWARE
ENGINEERING WORKSHOP

Prepared by

Leon Jordan

COMPUTER SCIENCES CORPORATION

January 1987

SUMMARY OF THE ELEVENTH ANNUAL SOFTWARE
ENGINEERING WORKSHOP

The Eleventh Annual Software Engineering Workshop was held on December 3, 1986, at the National Aeronautics and Space Administration (NASA)/Goddard Space Flight Center (GSFC) in Greenbelt, Maryland. This annual meeting is held to report and discuss experiences in the measurement, utilization, and evaluation of software methods, models, and tools. The workshop was organized by the Software Engineering Laboratory (SEL), whose members represent NASA/GSFC, the University of Maryland, and Computer Sciences Corporation (CSC). The workshop was conducted in four sessions:

- Research in the SEL
- Empirical Studies of Software Technology
- Software Environments
- Software Testing

Twelve papers were presented, and the audience actively participated in all discussions through general commentary, questions, and interaction with the speakers. Over 360 persons representing 59 private corporations, 9 universities, 18 agencies of the Federal Government, and 8 NASA centers attended the workshop.

SESSION 1 - RESEARCH IN THE SEL

Session 1 was moderated by Mr. Frank McGarry of GSFC. McGarry presented a high-level summary of SEL studies and results and a profile of areas of future efforts for improving the quality of software production (Determining Software Productivity Leverage Points in the SEL). He used a technological index that measures the level of application of disciplined approaches and discussed leverage points at different stages of the software production process.

The leverage points identified are increasing development efficiency, decreasing required rework, and continuing the delivery of reliable software. McGarry discussed the SEL approach to software quality improvement by listing targets and mechanisms for each leverage point. For increasing efficiency, there were improving management and testing effectiveness and effort distribution using structured techniques, testing approaches, code reading, and automated tools. For decreasing rework, the major targets were in the area of design and interface errors using prototyping, testing approaches, structured techniques, and independent verification and validation (IV & V). To sustain the high quality of delivered systems, measures, models, and IV & V are applied to minimize the error rate of delivered software and decrease system complexity.

McGarry showed the effect of the approach on quality in terms of the improvement in development effort on recent projects compared to earlier projects. Rework decreased from 45 to 25 percent of development effort; documentation decreased from 45 to 35 percent; coding decreased from 20 to 15 percent; and code reuse increased from 15 to 25 percent.

McGarry also discussed efforts to encourage tool use by means of a software development environment under development. Future efforts in the SEL will focus on cutting the size of developed systems using software reuse tools, Ada, and library languages. Prototyping and test aids will be used to decrease rework. Environments, formal training, and expert systems for management support will be used to improve process efficiency.

Drs. Victor Basili and Marvin Zelkowitz described some tools and environments used for software development (Studies of Software Methods and Environments). Basili discussed expert systems by describing the study of Arrowsmith-P, used to aid

the software development project manager. The knowledge base contains historical information about previous projects in a homogeneous environment. From a baseline of normalized software metrics, abnormal values are determined and input to the system along with a set of possible explanations for the deviant values. Arrowsmith-P was used to explore such research issues as efficacy of inference mechanisms (rule-based deduction versus frame-based abduction), method of knowledge acquisition (top-down versus bottom-up), transportability, and feasibility.

In conclusion, Basili noted that software management has not evolved to a state where cause and effect are clearly understood or consistent, as shown by the evaluation of Arrowsmith-P. In comparing results with actual events, the rule-based system agreed with 1/4 to 1/2 of the correct interpretation. In comparing results with experts, experts agreed with each other 1/3 to 1/2 of the time, they agreed with themselves (top-down versus bottom-up) 1/4 to 1/2 of the time, and the system agreed with the experts 1/4 to 1/2 of the time. The bottom-up and rule-based deduction approaches performed better and seemed to perform as well as the experts.

Basili also described TAME, a project aimed at developing a set of methods and tools supporting a variety of metaprocess models that can be tailored to specific projects. The project will first develop a prototype supporting all kinds of measurement and evaluation activities and will then interface the TAME prototype to an existing software development environment. The last phase of the project will develop guidelines for the design of future environments.

He listed the TAME requirements in terms of purpose, potential users, and the user view of the system and presented the architecture as consisting of a PC-based user interface

level and mainframe-based evaluation, measurement, and repository levels. Ada dependency occurs at the measurement and repository levels. The first prototype, scheduled for completion by the fall of 1987, will implement a restricted subset of TAME requirements for the Micro VAX (VMS) and SUN-3 (UNIX) in Ada for the Ada-dependent levels and Pascal or C (SUN-3) for the Ada-independent levels.

Zelkowitz described an environment (SUPPORT) consisting of a set of automated design tools developed in Pascal at the University of Maryland. SUPPORT has multiple windowing capability, and he discussed its use in terms of an editing hierarchy. For potential application to NASA software projects, SUPPORT can be applied to document preparation in general and to code preparation by viewing code as a form of documentation. It can be used to generate templates for prologs. He also indicated that structure chart capability could be built into the system.

Dr. William Agresti of CSC presented a status update and design experiences on an experiment underway in the SEL to develop a system in parallel in FORTRAN and in Ada and to compare the two implementations (SEL Ada Experiment: Status and Design Experiences). The project, a flight dynamics simulator for the Gamma Ray Observatory, is expected to be 50 KSLOC (FORTRAN) at completion and is being done on a VAX-11/780 (FORTRAN) and a VAX-8600 (Ada). The Ada team is approximately one-half done with coding, and the FORTRAN team is three-quarters done with acceptance testing. Staff effort was projected by development phase to be 7.5 (FORTRAN) and 8.5 (Ada) staff-years (including 1.25 staff-years for Ada training).

In comparing designs in the two projects, Agresti first noted that the drivers for the FORTRAN project were the legacy of past designs and schedule constraints. The driver for the

Ada project was the desire to use new design methods to exploit Ada features. The Ada team used the Composite Specifications Model (reported in the Ninth SEL Workshop) to derive an object-oriented design. The design abstractions for the FORTRAN project were procedural; those for the Ada project were state machine and object oriented. Agresti compared the actual designs in terms of hierarchy (seniority relations) and operation. He also discussed other approaches to the problem of converting from a FORTRAN to an Ada environment and their consequences. Agresti said that committing to Ada from project start and resisting the FORTRAN legacy offer the best opportunities to cast requirements in a more language-neutral form.

In his conclusion, Agresti noted that project management issues were encountered during the design phase relating to differences in cost estimation (What is a module?), development products (Where are the structure charts?), milestones (When is design complete?), and the structure of reviews (What is presented at CDR?). In the area of staffing issues, he said that he had positive experience with programmers who had previous training in Pascal and design abstractions and exposure to several different programming languages. In the area of technical issues, he mentioned that it is important to allocate sufficient time to define Ada types, to consider the extent of using tasking and generics, and to assess the degree of package nesting versus library units. He also said that the Ada team produced a significantly different design from the FORTRAN team, that there are specific conditions that encourage Ada-oriented designs, and that Ada can influence every aspect of design, especially management expectations.

In the discussion that followed, Dr. Elliot Soloway commented that the FORTRAN language affects the design and that, in

the future, Ada will do the same. Agresti noted that Ada represents a richer language that is more descriptive of the problem domain and that this represents progress. Basili noted that the design is in the context of the language and that, if this influence were not present, it would be a problem. Dr. Barry Boehm indicated that Ada projects tend to attract bright people and asked how this was taken into account in the experiment. Agresti responded that there were differences with the application and level of experience and that these differences were part of the experiment data. He also noted that the Ada team training was in software engineering with emphasis on Ada, rather than just in Ada. McGarry added that the early training for FORTRAN projects was a 1-week course in structured techniques designed by the University of Maryland and that senior people reinforced that training throughout the development process.

SESSION 2 - EMPIRICAL STUDIES OF SOFTWARE TECHNOLOGY

Session 2 was moderated by Dr. Gerald Page of CSC. Dr. Elliot Soloway of Yale University reported on a 2-year study of software documentation using a cognitive approach to address questions about the content and format of documentation and when it should be available (Studying Software Engineering Documentation From a Cognitive Perspective). One approach of the study was to understand programmers and their expectations by direct observation rather than to try to understand software. Another approach was to start the study using small programs. In one part of the study, programmers were given a 250-line FORTRAN program of 14 subroutines and documentation that followed some generic guidelines. They were asked to make an enhancement and then were videotaped as they went about the task. In another phase of the study, new documentation was provided that presented information about nonlocal, causal interactions

(e.g., information about other downstream parts of the system that could generate "gotchas").

The most successful strategy employed by the subjects was a systematic or global strategy that started at the beginning of the program and documentation and traced the flow of the entire program using various forms of simulation. This strategy invariably led to the correct enhancement. However, this strategy cannot realistically be used for large programs because it is not worth the effort to understand a 100-KLOC program to generate a 15-line enhancement. Other subjects employed an "as-needed" strategy, where they jumped immediately to the enhancement area and backtracked when they thought they needed more information. Adopting this strategy led to mixed results. More often than not, these subjects did not come to understand nonlocal causal interactions and thus did not develop a correct enhancement. The conclusion of this phase of the study was that documentation must provide information about these nonlocal causal interactions.

Soloway said that the real issue is what information programmers need and, especially, when do they need it. To approach this question, he described a second study in which two programmers were given a 3-foot-high stack of documentation describing a 60-KSLOC program and were asked to make an enhancement. They each spent nearly 2 hours reading documentation, trying to understand what the system did in terms of the goals of the system and each of the various modules of the system. These goals were cited in the documentation but were interwoven with many low-level details that served to hide the main points and confuse the programmer. He observed that they needed this information first, and the first manual of the set of documentation was rewritten to provide it. When the experiment was rerun with the revised documentation, it took 20 minutes for the two subjects to

correctly identify the module where the change needed to be made.

Dr. Vincent Shen of MCC reported on a field study that was part of a larger research program to produce a design environment (Empirical Research on the Design Process). This study surveyed 19 large development projects, spending 1 or 2 days at each site, and taping 20- to 40-minute interviews to record their decision-making and communication process. Interviewees were system engineers, senior designers, project managers, division general managers, testing and quality assurance personnel, and customers. In the analysis of eight projects to date, attempts were made to identify key leverage points, technology transfer issues, and design problems for consideration of requirements for the design environment.

Shen identified the top five problems identified by the analysis as personnel (a project needs good people), communication and coordination, dealing with uncertainty and change, design representation and analysis, and technology transfer (it is difficult to get people to use new technology). In the area of personnel, the study revealed that an effective team is composed of an application specialist, a conceptualizer (to follow at a high level what is going on), a boundary spanner (to explain groups to each other), a gate keeper (to ensure that the team is not influenced by extraneous events or technology), a diagnostician, and a feature manager (who knows what currently needs to be done).

Problems in the area of communication and coordination are information overload, delay, and deprivation. Dealing with uncertainty and change is important to reduce wasted effort. Uncertainty generates "floating issues" (missing and conflicting information). Change generates new information

in the areas of goals, people, technology, policy, standards, and procedures.

The design representation and analysis process records the design status and rationale. This area relies on representation media (e.g., text, graphics, and prototypes) and on analytical and simulation models. Effective technology transfer is important to ensure the use of the best tools available. It is important to be aware of and to assess new technology and to counter cultural resistance.

In summary, Shen said that when conducting research in large, complex systems, engineering should focus on the needs in each of these five problem areas. These needs include support in role identification and communication breakdown detection; intelligent filters and active probes to support quick access to relevant information; rigorous methods to make the specification of floating issues and changes and their impacts explicit; mechanisms to compare the behavior of the design with the unclear customer needs; and experiments to test learning models of technology transfer.

In the discussion that followed, Boehm asked whether there were consistent patterns in how designers organized information. Shen responded that pictures were often used and that 80 percent wanted a word processing tool that integrated graphics.

Dr. John Gaffney of IBM reported on a technology index defined to quantify software quality and development productivity to determine the degree of using key software procedures and techniques instituted in the late 1970s and to determine how well the investment in these techniques has paid off (A Quantitative Analysis of the Impact of Modern Software Engineering Techniques). Gaffney noted that the data suggest that higher levels of productivity and lower

error density are associated with higher levels of the technology index.

This software development process technology index is a number between 1 and 100. It was evaluated for a variety of large projects totaling more than 3.8 MSLOC and signifies the degree of 13 process and 2 educational attributes.

These attributes are inspections, structured programming, structured design language, function model, state machine model, structured specification language, unit testing, development integration testing, function testing, systems testing, performance and limit testing, user testing, and management and nonmanagement technology education. Each attribute was scored by assigning a number from 0 to 16 indicating the degree of technical rigor and the extent of use. The index value was then taken to be the normalized sum of attribute scores.

Gaffney showed charts relating productivity and the index (correlation coefficient +0.697), latent error content and the index (correlation coefficient -0.582), and latent error content and productivity (correlation coefficient -0.672). He indicated that the index is useful in estimating the likely effects of changes or improvements in technology on productivity and quality, estimating the risk in applying some element of technology, and validating estimates of development costs. Empirically derived quantitative relationships between the index and productivity and latent error content can be used to validate estimates of these variables obtained by other methods. These same relationships may also be used for prediction to supplement the planning process.

In summary, Gaffney said that the index is good for estimation and planning and provides another "handle" on controlling software productivity and quality. In the discussion

that followed, he said that the method was applied to different types of projects and not just to similar types. In discussing how the measure captures a shift in technology (e.g., a shift from FORTRAN to Ada), he said that it captures the application of well-defined engineering processes such as design, code inspections, and testing.

SESSION 3 - SOFTWARE ENVIRONMENTS

Session 3 was moderated by Mr. Keiji Tasaki of GSFC.

Mr. Eugene Spafford of the Georgia Institute of Technology reported on a software testing environment (The Mothra Software Testing Environment) consisting of an integrated set of tools and interfaces that support the planning, definition, preparation, execution, analysis, and evaluation of tests of software systems. The Mothra system provides support from unit testing through system and acceptance testing. Persistent data are kept. The user interfaces provide a wide variety of information and information representation (e.g., graphics, windowing, animation, and data compression). No size constraints are imposed by the architecture on the size of the software system that can be tested in the environment; comparable functions can thus be applied across a familiar interface as the software being tested evolves in size and complexity. Development tools can be integrated into the testing process, allowing the use of user and system tools with which the tester is already familiar.

The environment provides three different aspects: views, thematic tools, and "shifting gears." The view aspect provides a way of managing large tests. Displays are used as the tester's view or window into a larger (virtual) test context. The thematic tools aspect relies heavily on mutation testing and uses different underlying sets of well-understood and natural activities that proceed in a specific sequence through several phases of testing. The shifting

gears aspect provides the ability to capitalize on testing at an appropriate level. Mothra will always spawn machine-intensive tasks and organize them for execution by a computer resource of appropriate power. In this way, a test that is justified technically will always be performed unless overriding economic limitations prevail.

Mothra is a subenvironment that runs on top of a host environment. Versions are operating on the VAX-11/780 and on ULTRIX on VAX workstations. Explicit operations also allow Mothra processes to spawn parallel and vectorized processes for execution by a Cyber-205. In concluding, he said that two versions have been implemented. An Ada capability has been designed, and performance studies have been started. In response to a question comparing mutation testing to other methods, he noted that mutation testing was developed about 1977 and that studies on large-scale COBOL and FORTRAN projects have shown that mutation testing represents a good testing method with few undetected errors.

Dr. Barry Boehm of TRW reported on the application of an analytical method to components of the software production process with the goal of improving software productivity (A Chain Value Analysis of Software Productivity Components). The method employs a value chain analysis that examines a canonical set of cost sources or value activities that represent the basic activities an organization can choose from to create added value for products.

Boehm presented a chain value analysis for the software development process. Operations consists of management (7 percent), quality assurance and configuration management (5 percent), requirements analysis (4 percent plus 1 percent rework), preliminary design (8 percent plus 3 percent rework), detailed design (10 percent plus 5 percent rework), code and unit test (8 percent plus 8 percent rework), and

integration and test (7 percent plus 13 percent rework). Infrastructure consumes 8 percent; human resource management, 3 percent (Boehm noted this is less than optimal given the labor-intensive nature of software development); and technology development, 3 percent (he noted that this is also less than optimal as an investment to improve productivity and quality). The breakdown of the operations activity indicates that the leading strategies for cost savings in software development involve making individual steps more efficient (by automated aids to software requirements analysis or testing), eliminating steps (by automatic programming or quality assurance), and eliminating rework (by early detection or rapid prototyping). He summarized these sources of savings in a software productivity tree to show how the various productivity options fit into an overall integrated productivity improvement strategy.

Boehm noted that the results of a study of rework costs using 1378 problem reports from two large projects indicated that 80 percent of the rework cost typically results from 20 percent of the problems. The implication here is that verification and validation should focus on identifying and eliminating specific high-risk problems rather than spreading early problem elimination effort uniformly across trivial and severe problems. This implies that a risk-driven approach to the life cycle, such as the spiral model, is preferable to a document-driven approach, such as the traditional waterfall model, and would be a way of focusing on the high-risk elements of development. In the ensuing discussion, he emphasized that risk driven versus document driven does not mean that documentation is absent and that there is no good way at present to estimate the cost of prototyping.

Mr. Anthony Wasserman of IUE, Inc., reported on an integrated set of tools based on an open, rather than a closed,

architecture (The Open Architecture of the IDE Tool Environment). The open architecture is characterized by the availability of information about all arguments and options for each tool (providing multilevel access to tools with separate invocations for different levels), all interfaces (file formats and data base schemas) so that other tools can build upon those specific interfaces, and common standards (e.g., ASCII, Pic, or Postscript). The open architecture allows user customization of the environment to support local options and preferences, allows the environment to be extended by developers and users with minimal effect on the existing data base, encourages the development of modest-sized software components rather than large monolithic tool systems, and provides multiple interfaces to the same functions to provide appropriate support to different classes of users. The tool set reported on is Software Through Pictures, a set of graphical editors supporting methods for software analysis and design, including structured systems analysis, structured design, entity-relationship modeling, and user-specified software engineering.

An effective way to present the user with a coherent view of the environment is to organize tools and files by logical level. The Unix system, for example, is organized in three levels: commands, files, and libraries. At least four levels should be present in an open architecture: the integrated environment level (starting point for the user and any global mechanisms needed by tools); tool levels (tools calling tools, libraries, and utilities represent multiple tool levels); data repository level (common repository of data for the integrated tools of the environment, including programs that manage the repository); and file interface level (containing text files used by or produced by tools

for use by other tools). Several examples of screens produced by Software Through Pictures transactions were presented. Wasserman showed how text editors could be used to modify templates (e.g., substituting an Ada template for a Pascal template).

In the discussion that followed, he indicated that the openness is derived from the Unix shell structure. He also stated that, even though the developer specifies the methodology, the user should be able to modify it by turning off specific tool features. In response to a question asking how this countered Fred Brooke's "no silver bullet" paper, he noted that the idea behind this tool set is modeling and communication and not the software itself.

SESSION 4 - SOFTWARE TESTING

Session 4 was moderated by Mr. Ed Seidewitz of GSFC. Dr. John Knight of the University of Virginia reported on a new approach to developing fault-tolerant software (Data Diversity - A New Approach to Fault Tolerant Software). The two best known techniques for developing fault-tolerant software are n-version programming and recovery blocks. Because both techniques rely on multiple implementations and, presumably, different designs, these are characterized by design diversity. A new approach relies on the observation that software often fails at a boundary point in the input space. Programs may work well for many input values, survive extensive testing, and then fail on an input case related to some boundary condition, usually associated with a transition in the processing algorithm. If, during testing, the special failure case is not generated exactly, the software usually works correctly. There is a strong implication that if software fails under certain execution conditions, it is very likely that a minor perturbation of those

execution conditions would allow the program to work. This new approach is characterized by data diversity.

Knight defined a technique for investigating the characteristics of failure regions in an input space that maps the two-dimensional cross-section of the multidimensional input space, indicating failure regions and boundary conditions. He then presented several examples obtained from 27 programs that had been subjected to one million test cases. He proposed a method in which data are reexpressed algorithmically. An algorithm is executed, and the output is tested. If the output is found unacceptable, the data are reexpressed by randomly choosing a point from a small circle with the original data point as center, and the algorithm is rerun. He then showed the relative performance for different circle radii with one, two, and three retries for several specific faults in the failure region.

In concluding, Knight noted that some programs rely on the relative placement of data. Data diversity is inexpensive, relying on a single implementation of the program, and minor costs are associated with reexpression and error detection. Empirical study showed that performance varied widely, that some faults were tolerated well and some faults not so well. He noted that data diversity is not universally applicable. Some data cannot be reexpressed, but in many instances, for example, noisy control systems and inaccurate sensors, data diversity should work well.

In the discussion that followed, Knight indicated that this method should be regarded as a "safety net." That is, after attempts have been made to eliminate design and other faults and there is a need to "get through this fault now" during operation, this method can be used to get the program out of the fault region.

Dr. Nancy Leveson of the University of California, Irvine, reported on an evaluation of the recovery block approach to testing in which self-tests (acceptance tests) are inserted in the program (An Empirical Study of Error Detection using Self-Test). She presented the preliminary results of an experiment that examined the relative effectiveness of self-test versus voting in detecting software errors. The experiment used programs developed for a previous n-version programming experiment that had been subjected to one million test cases so that nearly all faults were known.

Graduate students inserted self-tests in eight programs. Twenty faults were found, four by specification-based testing and eight by code reading. Six new faults were found that were not discovered in the one million test cases because intermediate results could be examined and not just the final output. This is probably also attributable to the fact that test case strategies could be employed that were more comprehensive than just examining one output. Leveson also noted that new faults were introduced by inserting the self-tests. This is expected any time code is added to a program.

In concluding, Leveson noted that large differences were observed in individual programmer ability. Other questions were raised, such as whether the original programmer would have performed better because of more extensive knowledge of the software. It appeared that the placement of checks was important and that specification-based checks alone were not as effective as using them with code-based checks. Procedures are needed to help formulate checks. No current fault-tolerant methods guarantee ultra-high software reliability. Current plans for future work include comparing fault-tolerant and fault-elimination methods, studying the efficiency of self-checks written by the original coder, and comparing clean-room and traditional development methods.

Dr. Amrit Goel of Syracuse University presented the results of a study to assess the comparative reliability of Ada and FORTRAN programs using the number of distinct errors found as the key reliability measure (An Experimental Comparison of Ada and FORTRAN Program Reliability). A comparison was also made between intermediate-level and experienced programmers. Two versions in each language were developed by intermediate-level programmers, and one version in Ada was developed by an experienced programmer. The number of errors found during compilation and unit testing were recorded. After removing errors found during compilation, each version was tested with 54 test cases developed by using a hybrid functional/structural testing methodology based on one Ada program. Errors detected by the 54 test cases were removed, one at a time, and the remaining test cases were run on the incrementally corrected programs.

In presenting the results, Goel noted that the plots of cumulative error symptoms versus test case number seemed to follow homogeneous Poisson processes. There were approximately 70 percent more errors in the FORTRAN programs than in the Ada programs (written by intermediate-level programmers). The Ada version written by the experienced programmer had half as many errors as those written by the intermediate-level programmers. Another set of 120 test cases was generated using the same testing philosophy as before, except that they were based on the structure and code of a FORTRAN version. Results were similar, with a few additional errors found. An additional 1000 random test cases were run that detected nine new errors in the five programs. In concluding, Goel indicated that the Ada programs seem to be much more reliable than FORTRAN and that, on the average, the Ada programs had 7 percent fewer errors.

In the discussion that followed, he noted that the programmers chose the program design methods, that the methods were not systematic, that they seemed more oriented to data flow, and that Ada features were not exploited. The differences in reliability seemed to be the result of the Ada compiler catching several typical FORTRAN errors that survived compilation. In response to a question asking how the results might differ by using Pascal, Goel said that a version was also written in Pascal, C, and APL. He conjectured that the figure would be 20 percent more errors in the Pascal program than in the Ada program (compared to 70 percent more for the FORTRAN program).

PANEL #1

RESEARCH IN THE SOFTWARE ENGINEERING LABORATORY
(SEL)

F. McGarry, NASA/GSFC

V. Basili, University of Maryland/M. Zelkowitz, University of Maryland

B. Agresti, Computer Sciences Corporation

Determining Software Productivity Leverage Factors in the SEL

Frank McGarry
Susan Voltz
Jon Valett

Abstract

For any organization responsible for the development of software systems, a typical ongoing goal is the attempt to improve the development process whereby the cost effectiveness of the software improves. To this end, numerous tools, methods, models, management techniques, development languages, etc. have been developed and are evolving. Yet to effectively utilize available techniques, a development organization must first determine what facets of the development process would be most conducive to change by determining what characteristics of the current development process are causing the most difficulty and thereby should be addressed with modified approaches. In other words, a development organization must determine what aspects of the current development process may provide the greatest leverage (currently consuming high amounts of resources) to a modified approach to the development process.

The Software Engineering Laboratory (SEL), in its attempt to assess the relative merits of various development techniques, has made efforts to quantitatively determine what characteristics of the development process, within one local environment, would provide the most potential for improving the overall development effort. By applying appropriate techniques to the identified areas, it is assumed that the most effective adjustment would be made.

To this end, the SEL has compared characteristics of the development process in early phases of the overall SEL study process (1978-1981) with the characteristics of more recent efforts (1984-1986) to determine:

1. What leverage points exist?
2. Has the application of selected techniques affected the profile of the development effort?
3. Has the development process improved?

This paper describes what leverage factors have been determined and how these factors have changed over 8 years.

1.0 Background

In 1977, the Software Engineering Laboratory (SEL) of NASA/GSFC began studying the characteristics of the software development process within the flight dynamics environment at Goddard. The SEL is organized as a partnership between NASA/GSFC, the University of MD, and Computer Sciences Corporation [1]. Each organization has been an integral part of the study effort from the inception of the SEL.

The approach taken to studying production software projects has included collecting detailed software development data from numerous projects as each of the projects utilized specific techniques in the software development effort.

1.1 Study Process (Chart 1)

The 3 step process to which the SEL approached the overall studies includes:

1. Determine the basic characteristics of the development process in the production environment (productivity, life cycle effort distributions, error rate, methods used, etc.). This step identifies the potential 'Productivity Leverage Factors'.
2. Apply modified development approaches to similar development projects (new tools, languages, methodologies, etc.) and extract detailed information on the development process.
3. Compare and assess the impact of applying the modified approach by observing changes to the measures of interest (e.g. productivity).

1.2 Projects Studied (Charts 2 and 3)

In carrying out the approach to both identifying the leverage factors and measuring the effect of available software development approaches, the SEL has utilized over 50 flight dynamics development projects at NASA/GSFC for detailed study. Detailed development data has been collected from each of these projects while numerous development methods, tools, etc. have been selectively applied to these projects. Several general results from the technology evaluation efforts were reported at the 10th Software Engineering Workshop in 1985 [2].

One approach that was utilized by the SEL in it's attempt to measure software techniques, was to define a 'Technology Index' [3] which was based on the level or extent to which known disciplined methods were applied to each project. This index characterizes the use of over 60 methods in developing the software and essentially is one organization's attempt at defining the level of discipline applied to the effort.

Based on 14 projects of similar size and complexity, one of the points the SEL reported was that it could not show a significant correlation between productivity and the 'technology index', but it did show a very high correlation between reliability (errors/SLOC) and technology index. These relationships were not trends over time, but only over level of applied technology. The conclusion is not that a more disciplined approach fails to improve productivity. The result merely states that for the SEL definition of 'technology index', no measurable improvement in productivity was apparent in the statistics from the sampled projects. This could possibly indicate:

1. The particular methods defined are not measurable to the degree required.
2. The technique was not applied correctly by the development team.
3. The technique studied was not appropriate for the environment used.

Another view of these projects was to look at the trends of measures of importance over time.

2.0 Leverage Factors, 1978-1981 (Chart 4)

By studying these projects, the SEL has additionally attempted to determine what specific elements of the software development process were affected over time by efforts made to improve key factors such as productivity and reliability. That is, if productivity was improving, where was the gain made (faster coders? less design required? less testing??) The SEL had put effort into developing profiles of early projects so that specific weaknesses could be identified and addressed. Although one of the goals of the SEL has been to measure the impact of specific software techniques, the end goal is to identify key leverage factors, then apply appropriate techniques to gain maximum improvement to the software as measured by some parameter of interest (e.g. productivity). Several interesting points of the development projects which were studied in the early time period (1978-1981) are noteworthy.

2.1 Distribution of Development Effort (Chart 5)

The detailed data from these earlier projects showed some statistics that were expected or not at all unusual while some of these figures have lead the SEL to concentrate more or less on certain aspects for improving software cost effectiveness. The distribution of manpower effort over the life cycle of development (beginning at requirements analysis and ending with completion of acceptance testing) has shown a fairly even distribution between design (27%), coding (25%) and testing (28%) with 'other' (which includes meetings, travel, training, etc.) accounting for about 20%. Of the 25% devoted to the coding phase approximately 15% was actually attributed to the writing or entering of code. This point has led the SEL to temper its efforts at concentrating on improving the 'coding' process since the 'leverage' gained from improvement here may be relatively limited.

A second point worth noting, is that when the SEL analyzed the effort attributed to changing and fixing software during the development process, (because of incorrect designs that had to be changed, or errors that were created during the design or code phase, or because of changing or misinterpreted requirements leading to changes in design or code) it found that approximately 40% of the total manpower spent during development was attributed to this 'rework' effort.

This effort data was based on 3 projects that had very good historical information during the earlier timeframe of 1978-1981. Data sources included:

- o SEL change/error reports
- o Specification change reports
- o Change histories of on- line source code

By studying in detail representative changes, it was found that the average change to code required approximately 1/4 of a persons day and the average change caused by a design or spec change required approximately 1/2 of a person day. Data for planned enhancements or planned changes were not included in the computation.

Several other potential leverage factors were identified by analyzing this early data:

1. Approximately 30% of the total development effort was spent on testing. If this process could be made more efficient, there was potential for improved cost effectiveness. Testing included unit tests, systems tests and acceptance tests. Although many people consider code reading to be a form of testing, this 30% value, did not include that particular effort.
2. In the 1978-1981 time frame, developers spent nearly 50% of their time documenting. This effort includes writing such documents as design descriptions, test plans, user's guides, system description and code commentary. The value was approximated by computing a page count of documents produced for each project studied and developing an average time per page by examining detailed records of the project development data and by observing and interviewing authors. The figures used included:
 - o 2 person hours per page of documentation.
 - o 3 person minutes per line of code commentary.

To the figures, the total 'tech publication' charges and secretary charges were added.

3. In looking at detailed history of the M & O phase of delivered software, it was found that the error rate was approximately 0.6 errors/1000 SLOC. This error rate has been deemed to be highly acceptable for this environment. For this reason, no concentrated or extended efforts would be put forth in attempting to improve the reliability of the delivered software. Therefore, the reliability of delivered software was not targeted as a major leverage factor in trying to improve the development process.

2.2 Early (1981) Approach to Quality Improvement (Chart 6)

Once the SEL studied in detail the software development profiles of projects developed in the 1978-1981 timeframe, three general goals for improving the software process were developed:

1. Increase efficiency of the development process: Two of the major manpower consumers had been identified as testing (30%) and documentation (40%). By identifying methods, tools and approaches that could possibly increase efficiency in these areas, as well as increase efficiency throughout all the development phases, there was potential for improving the cost effectiveness of the overall process.
2. Decrease Rework: Since its inception, the SEL has conducted numerous studies on the nature and cause of software errors (ref. 4, 5, 6) and changes during the development process. By isolating characteristics and general causes that may be major drivers for rework effort, the SEL has anticipated that appropriate methods and effective tools could address this major leverage factor. Past studies have shown that neither syntax errors (<3% of total errors) nor errors in software specification (<5%) were major problem areas, but software design errors and interface errors were major contributors to the errors in the flight dynamics systems. As a follow-up to the studies of software errors, the SEL has conducted additional experiments with several methods and approaches that could potentially address the types of errors created. These studies have included:
 - o Study of Software Verification Techniques [4]
 - o Characteristics of Software Prototyping [5]
 - o Assessment of an Independent V & V techniques [6]

3. Sustain High Quality of Delivered Systems: Although original studies showed that GSFC was not spending an appreciable effort in the maintenance/error fixing process for the delivered system, (therefore not a major leverage factor) it has been a goal, that by continually utilizing and improving the development process, the developed system should sustain the high level of reliability that had been measured originally. Therefore the reliability of delivered software would continue to be used as a measure of the effectiveness of modified approaches to software development; but improved reliability of delivered software was not defined as a major point of concern for the original SEL efforts.

3.0 Recent Trends in Productivity and Reliability (Chart 7)

In attempting to interpret recent trends of software development effectiveness, the SEL has studied two key factors, productivity (LOC/day) and reliability (errors/KLOC), over the past 7 or 8 years as efforts were put forth in experimenting and applying techniques directed at gaining leverage from key places in the development process. In determining the productivity and reliability trends, 6 projects of similar complexity and size, which were developed over the 8 year period (1978-1986), were studied. The data showed that productivity first decreased for several years then continually increased over the past 6 years.

The reliability of the developed systems has also continually improved, where the particular measure is computed as the number of errors per KLOC found from system testing through acceptance testing. This error rate is a strong indication of the amount of rework that would be necessary during the development process. The rate has decrease from 7 or 8 errors per 1000 SLOC to under 6 errors per 1000 SLOC. The error rate reported during maintenance also has shown a slight improvement.

In looking at the trends over the 8 year period, the SEL is still attempting to ascertain why the productivity of systems in the 1978-1980 time frame was extremely high as compared to systems built in the 1981-1983 time frame. Two explanations are being pursued as possible drivers:

1. High availability of experienced people. Between 1978 and 1980, six major flight dynamics general systems were completed. This unusually high number assured a large number of experienced people were available for the efforts and as one system was nearing completion, the wealth of experiences and expertise could be shared on newer efforts.

2. Strong Training in Development Methodologies. Early in the SEL study efforts, many of the project personnel were put through specific formal methodology training in preparation for the measurement of such technologies as structured programming. Additionally, senior staff members continually reinforced this training during the development process and worked closely with development personnel to assure that development methods were understood and were being utilized. Some SEL researchers feel in later years, the strong reinforcement of training had decreased, possibly leading to lower productivity.

3.1 Revised Leverage Factors, 1984-1986 (Chart 8)

Following the analysis of software development profiles of an early timeframe (1978-1982) and the identification of potential leverage factors for that era, the SEL then compared results to those of a more recent timeframe (1984-1986). The goal of this comparison was to determine how the identified leverage factors have changed over time and to identify new leverage factors for improving the software development process in the future. The results of this comparison identified changes in the distribution of development effort in three areas:

1. Rework: Effort attributed to changing and fixing software during the development process has decreased in recent years. Since this effort, however, remains a significant percentage of the total development effort (25%-35%), rework continues to be a key leverage factor in improving the software development process.
2. Documenting: Recent years have shown a decrease in effort related to writing software development documents and code commentary from over 45% in the earlier timeframe studied, to 35% of the total development effort in the recent timeframe. The decrease in documenting effort may be attributed to the production of fewer repetitive documents during the software development process or the higher user of support tools for developing documents. Through experience and the development of similar software systems over the past years, the understanding of what documents are essential to the development process has become more clear.
3. Coding: Effort involved in coding software has decreased slightly over the past few years. Coding effort remains a small percentage of the total development effort and may not be lowered much further. Therefore, improvement efforts in the area of software coding would not provide much leverage in improving the overall development process.

In addition to a decrease in rework, documenting, and coding efforts, the SEL discovered a recent increase in the percentage of code reused in the development process. This favorable increase may be attributed to an increased awareness of code reuse as a key leverage factor. However, given the homogeneity of the development environment studied by the SEL, code reuse is still relatively low. Therefore, code reuse remains an important leverage factor for improving the cost effectiveness of the software development process within this organization.

3.1.1 Software Development Environment (Charts 9 & 10)

An additional area in which leverage may be gained in improving the development process may include the use of software development environments. The SEL discovered that many of the tools available to the developers to aid in the software development process, were not being used as heavily as anticipated. Recent studies in the SEL, however, have shown that increased tool usage has a positive impact both on productivity and on the maintainability of development projects (ref. 7). Therefore, the SEL has begun to encourage the use of a development environment known as the Software Development Environment (SDE). SDE is a menu-driven, integrated set of development tools intended to aid developers in the IBM environment. SDE encourages the methodology deemed to be effective in its production environment and will hopefully increase the use of development tools.

3.1.2 Environment Experiment

As a first experiment with SDE, the SEL has compared effort and error data of two development projects, one developed under SDE and one not, to analyze the impact of this development environment. Preliminary results show that productivity and reliability of the software developed using SDE significantly improved over the system developed without SDE. In addition, a user-survey indicated that users of SDE, particularly newer developers, felt software quality improved by using SDE. These preliminary results indicate to the SEL that further work in environments could be an important leverage factor of the future.

3.2 Revised (1986) Approach to Quality Improvement (Chart 11)

In studying and comparing the recent software development profile to that of an earlier development era, the SEL has identified areas in which the development environment studied could most benefit from improvement efforts. The revised target leverage factors address the following goals:

1. Reduce the size of developed systems in order to increase the reuse of existing software. The approach to attaining the highest software reuse potential may include selecting and applying appropriate tools, methodologies, or languages that promote the reuse of software (e.g. Ada).
2. Decrease the effort attributed to rework in order to further decrease the amount of time spent changing and fixing software errors. Although the percentage of rework effort has declined recently in the studied environment, continued emphasis on this leverage factor could produce a gain in productivity and reliability. Possible approaches to meeting this particular goal may include improving design methodologies, prototyping, and utilizing testing aids.
3. Further increase the overall efficiency of the development process in order to decrease the dependency on people and experience in software development. In order to more easily learn from past development and management experience, however, development and management environments could be utilized, as well as expert systems that capture developers' or managers' expertise.

By concentrating on the revised leverage factors targeted by the SEL, an improvement in the cost and quality of developed software could be realized in the near future.

REFERENCES

1. Software Engineering Laboratory, SEL 81-104, The Software Engineering Laboratory, D. N. Card, F. E. McGarry, G. Page, et. al, February 1982.
2. McGarry, F., 'Studies and Experiments in the Software Engineering Lab (SEL)', Proceedings of the Tenth Annual Software Engineering Workshop, December 1985.
3. SEL 86-002, Measuring and Evaluating Software Technology, D. N. Card, F. E. McGarry, J. Valett, to be published.
4. SEL 85-001, Comparison of Software Verification Techniques, D. Card, R. Selby, F. E. McGarry, et. al, April 1985.
5. SEL 86-004, Collected Software Engineering Papers: Volume IV, November 1986.
6. SEL 81-110, Evaluation of an Independent Verification and Validation (IV&V) Methodology for Flight Dynamics, G. Page, F. E. McGarry, and D. N. Card, June 1985.
7. McGarry, F.; Valett, J.; and Hall, D., 'Measuring the Impact of Computer Resource Quality on the Software Development Process and Product', Proceedings of the Hawaiian International Conference on Systems Sciences, January 1985.

THE VIEWGRAPH MATERIALS
FOR THE
FRANK McGARRY PRESENTATION FOLLOW



ELEVENTH ANNUAL

**SOFTWARE ENGINEERING
WORKSHOP**

DECEMBER 3, 1986

F. McGarry
NASA/GSFC
11 of 22

**DETERMINING SOFTWARE PRODUCTIVITY
LEVERAGE FACTORS IN THE SEL**

SEL EFFORTS TO DEFINE LEVERAGE FACTORS

SOFTWARE EXPERIMENTS IN PRODUCTION ENVIRONMENT: NASA APPLICATIONS

- DEVELOP PROFILE OF ENVIRONMENT
 - EXTRACT DETAILED DEVELOPMENT DATA
 - DETERMINE POTENTIAL LEVERAGE FACTORS
- EXPERIMENT WITH PROPOSED TECHNOLOGIES
 - APPLY VARIOUS TECHNOLOGIES (METHODS, MODELS, AND TOOLS) TO APPLICATIONS PROGRAMS
 - EXTRACT DETAILED DEVELOPMENT DATA
- MEASURE IMPACT AND/OR ASSESS TECHNOLOGIES
 - DEFINE MEASURES FOR EVALUATION
 - COMPARE EFFECTS OF USING OR NOT USING APPROACHES IN QUESTION (SIMILAR PROJECTS)
 - DETERMINE EFFECTIVENESS OF TECHNOLOGIES IN QUESTION (WHICH ONES HELP AND BY HOW MUCH)

Figure 1.

B58.002

SOFTWARE TECHNOLOGY STUDIES (1977 - 1986)

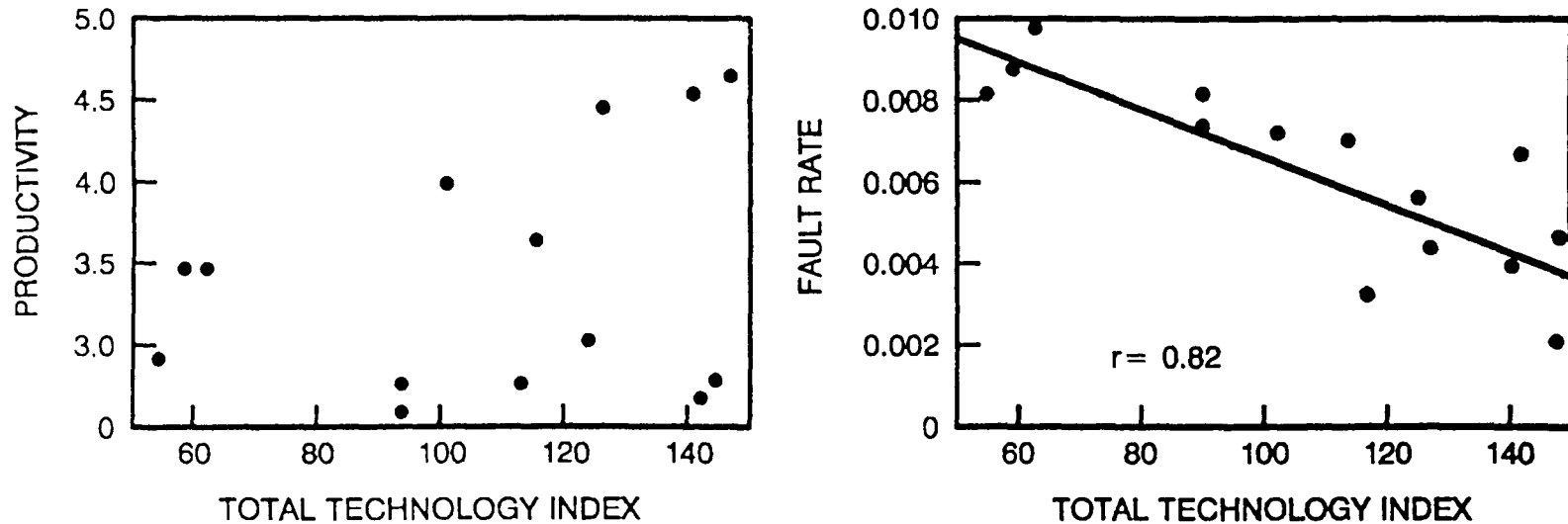
- 56 PROJECTS STUDIED
- RESULTS USED AS BASIS FOR SOFTWARE DEVELOPMENT PRACTICES
- OVER 50 STATE - OF - THE - ART TECHNOLOGIES MEASURED
- OVER 3 MILLION L.O.C. MONITORED (USED IN EXPERIMENTS)
 - TOOLS DEFINED
 - STANDARDS DEVELOPED
 - MANAGEMENT PRACTICES DEVELOPED
 - COST/RESOURCE MODELS

Figure 2.

B58.003

SEL RESULTS REPORTED 12/85

EFFECTS OF DEVELOPMENT "TECHNOLOGY"



1. PRODUCTIVITY COULD NOT BE SHOWN TO INCREASE WITH "TECHNOLOGY INDEX"
2. RELIABILITY IMPROVEMENTS WERE SHOWN TO BE RELATED TO "TECHNOLOGY INDEX"
3. INCREASED TOOL USAGE IMPROVED PRODUCTIVITY

HOW WERE PRODUCTIVITY IMPROVEMENTS ADDRESSED (OVER TIME)?

Figure 3.

APPROACH TO IMPROVED S/W PRODUCTION

1. IDENTIFY POTENTIAL LEVERAGE POINTS

- WHERE DO WE SPEND \$\$?
- WHAT CAUSES PROBLEMS ?
- WHAT COULD BE IMPROVED ?

2. DETERMINE APPROPRIATE METHODS FOR IMPROVING

- TOOLS
- METHODS
- ETC.

3. APPLY - MEASURE - REFINE - ADOPT

Figure 4.

B58.005

IDENTIFIED POTENTIAL LEVERAGE POINTS (1978-1981)

SOME OBSERVATIONS

- 1. INCREASE DEVELOPMENT EFFICIENCY
 - TESTING 30%
 - DOCUMENTATION 35-50%
 - CODE < 15%
 - APPROXIMATELY 40% EFFORT SPENT ON CHANGING/FIXING
 - ERROR RATE IN OPERATIONAL S/W .8 ERRORS/1000 SLOC
- 2. DECREASE REWORK REQUIRED
- 3. CONTINUE DELIVERY OF RELIABLE S/W

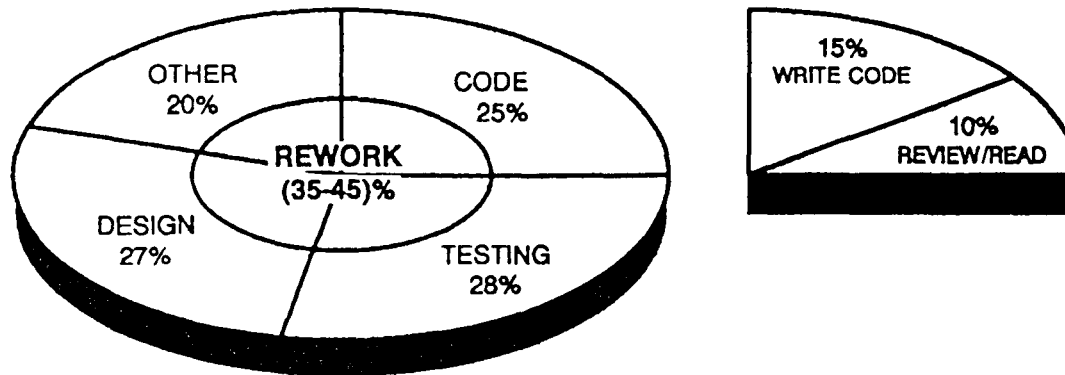


Figure 5.

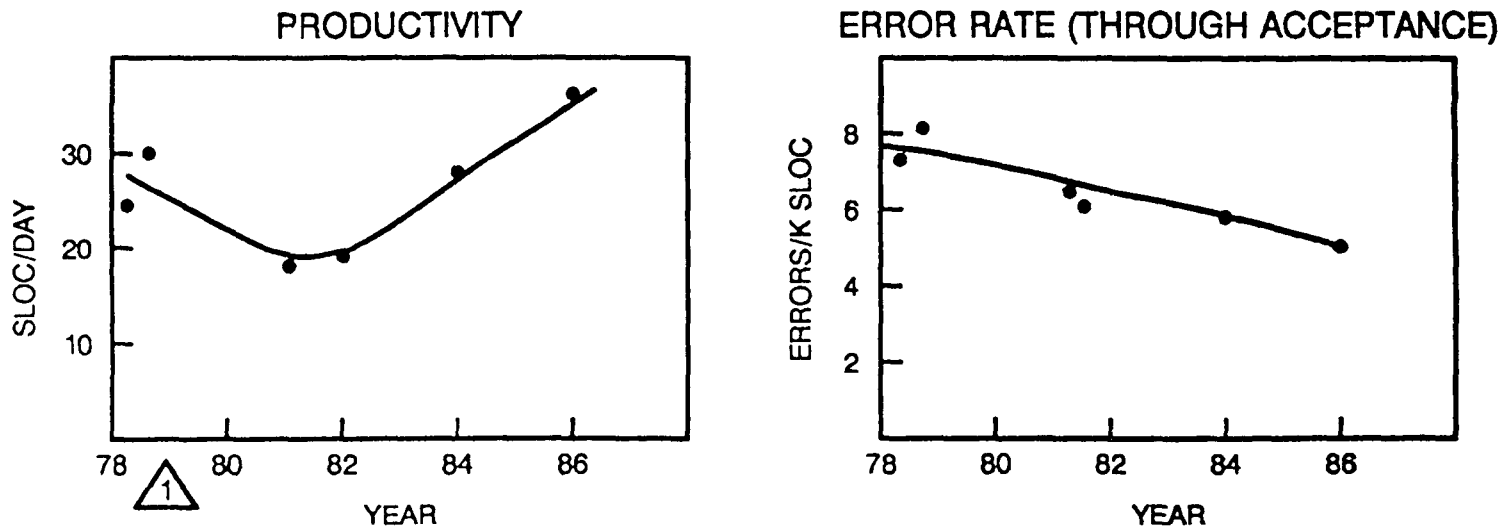
APPROACH TO "QUALITY" IMPROVEMENT

<u>GOAL</u>	<u>SOME TARGETS</u>	<u>SOME MECHANISMS</u>
INCREASE EFFICIENCY	<ul style="list-style-type: none"> ● OPTIMIZE TESTING EFFECTIVENESS ● OPTIMIZE EFFORT DISTRIBUTION ● MANAGEMENT EFFECTIVENESS 	<ul style="list-style-type: none"> ● STRUCTURES TECHNIQUES ● TESTING APPROACHES ● CODE READING ● AUTOMATED TOOLS ● MODELS/MEASURES ●
DECREASE REWORK	<ul style="list-style-type: none"> ● CUT DESIGN ERRORS (75% REPORTED) ● CUT INTERFACE ERRORS (72% REPORTED) <p>(NOT OF CONCERN)</p> <ul style="list-style-type: none"> ● SPECIFICATION ERRORS <5% ● SYNTAX ERRORS <3% 	<ul style="list-style-type: none"> ● PROTOTYPING ● TESTING APPROACHES ● STRUCTURED TECHNIQUES ● IV & V ●
SUSTAIN HIGH QUALITY OF DELIVERED SYSTEMS	<ul style="list-style-type: none"> ● MINIMIZE ERROR RATE ON DELIVERED S/W ● DECREASE SYSTEM COMPLEXITY <p>(AVERAGE "EFFORT TO CHANGE" AVERAGE "EFFORT TO REPAIR")</p>	<ul style="list-style-type: none"> ● MPP ● IV & V ● MEASURES ● MODELS ●

C58.003

Figure 6.

TRENDS* OVER 8 YEARS



EFFICIENCY - PLANNING/MANAGEMENT MORE STABLE
 TESTING EFFORTS MORE EFFECTIVE

REWORK - DESIGN AND INTERFACE ERRORS DECREASED

QUALITY OF DELIVERED S/W - MAINTAINENCE ERROR RATE <.5 ERRORS/K SLOC

△ FORMAL TRAINING IN S/W ENGINEERING TECHNIQUES
 6 MAJOR PROJECTS COMPLETED DURING 2 YEAR PERIOD

* BASED ON 6 PROJECTS OF SIMILAR SIZE AND COMPLEXITY

B58.008

F. McGarry
 NASA/GSFC
 18 of 22

Figure 7.

SOME IMPACTS OF S/W QUALITY IMPROVEMENT EFFORTS

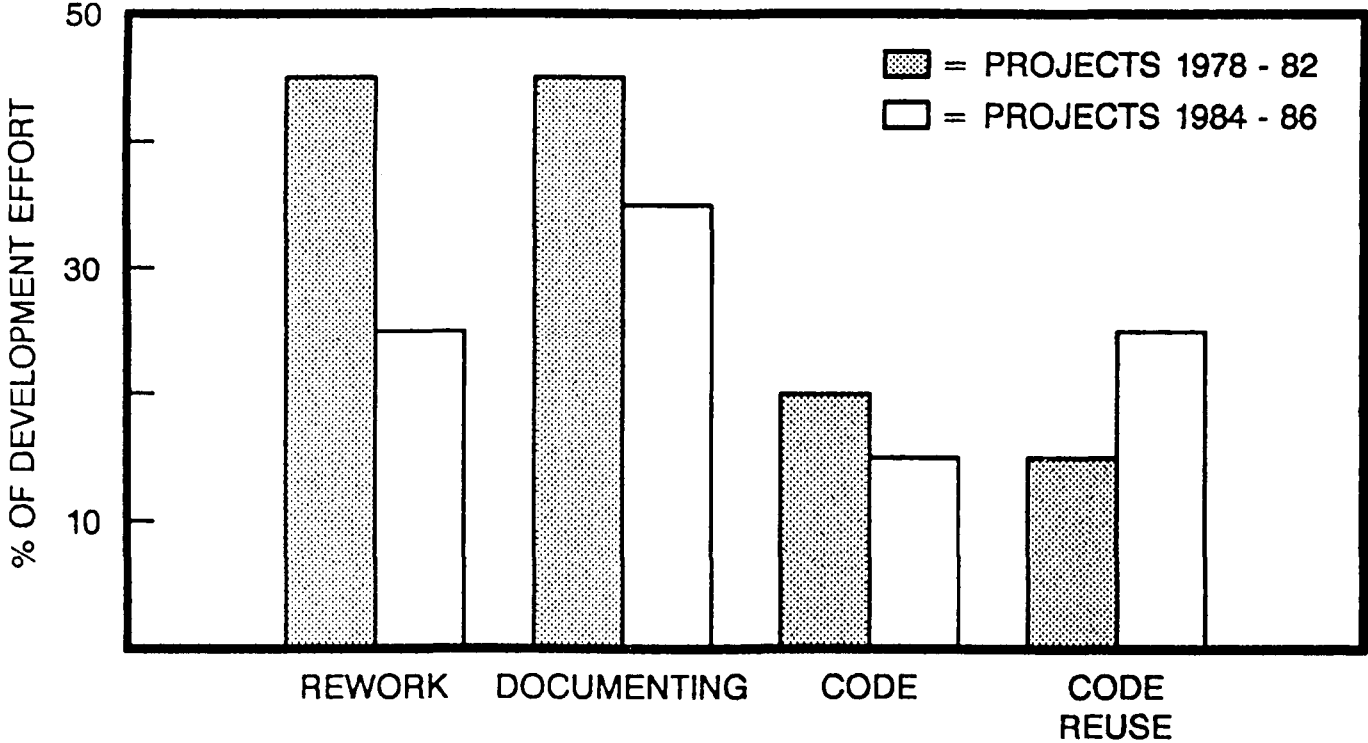


Figure 8.

ADDITIONAL OBSERVATIONS

- AVAILABLE TOOLS NOT USED AS HEAVILY AS ANTICIPATED
- EARLY ('78-'79) HIGH LEVEL OF PRODUCTIVITY IMPRESSIVE
 1. TRAINING
 2. STAFF EXPERIENCE WITH PROBLEM
- STUDIED "METHODOLOGIES" MADE LESS IMPACT THAN HOPED
 2. REINFORCED/TRAINING?
 1. APPLIED CORRECTLY?
 3. MEASURABLE?
- MORE LEVERAGE PROBABLY ATTAINABLE FROM OTHER FACTORS
- MANAGEMENT AIDS WERE KEY BY-PRODUCT OF STUDIES

Figure 9.

B58.009

ENCOURAGING TOOL USAGE IN THE SEL

SOFTWARE DEVELOPMENT ENVIRONMENT (SDE)

- INTEGRATED SET OF DEVELOPMENT TOOLS
 - CONTROL EXECUTIVE - COMMON MMI - IMPLIED METHODOLOGY - INTERFACE TO 12 - 15 TOOLS.
- FIRST EXPERIMENT USING SDE (1 PROJECT USING/ 1 NOT)
 - PRODUCTIVITY AND RELIABILITY SIGNIFICANTLY IMPROVED
 - ALL USERS POLLED FELT SDE HELPED QUALITY OF S/W
 - NEWER DEVELOPERS (<4 YRS. EXPERIENCE) MORE ENTHUSED THAN SEASONED DEVELOPERS

FURTHER WORK IN ENVIRONMENTS HIGHLY PROMISING

Figure 10.

C58.001

REVISED (1986) TARGET LEVERAGE FACTORS

<u>GOAL</u>	<u>WHY</u>	<u>APPROACH</u>
CUT SIZE OF DEVELOPED SYSTEMS	POTENTIAL S/W REUSE NOT BEING ATTAINED	<ul style="list-style-type: none">● TOOLS FOR "REUSE"● ADA● LIBRARY LANGUAGES
ELIMINATE REWORK	STILL SPENDING 25% ON CHANGES AND ERROR FIXING	<ul style="list-style-type: none">● PROTOTYPING● TEST AIDS
INCREASE PROCESS EFFICIENCY	STILL VERY PEOPLE AND 'EXPERIENCE' DEPENDENT NOT 'LEARNING' FROM EXPERIENCE AS FAST AS WE SHOULD	<ul style="list-style-type: none">● ENVIRONMENTS● FORMAL TRAINING● "EXPERT SYSTEMS" FOR MANAGEMENT SUPPORT

B58.013

Figure 11.

TAME - Tailoring A Measurement Environment

Victor R. Basili and H. Dieter Rombach
Dept. of Computer Science
University of Maryland
College Park, MD 20742

More and more project environments suffer from lack of sound knowledge concerning the impact of software process and product characteristics on software quality and productivity. Such knowledge would be important for (1) designing the software process appropriate for achieving particular quality and productivity goals in a given project environment, (2) controlling the fulfillment of given quality and productivity goals throughout the development process based upon quantitative data (as far as possible), (3) providing proper feedback into the ongoing project allowing to take actions where necessary, and (4) allowing post-mortem evaluation of projects for the purpose of learning for the next projects. All these activities depend on sound data collection, validation, and evaluation procedures. This need becomes especially obvious in the current situation of many environments making the transition from traditional languages and related methodologies to Ada and supporting methodologies. In this context, many open questions need to be answered, e.g., whether or not Ada language features and concepts are used appropriately, and how Ada projects should be managed and supported by methods and tools.

For all stated reasons it is necessary to measure and evaluate the quality and productivity of process and product aspects of projects. This can be done by either conducting case studies of ongoing Ada projects or experiments in controlled environments. In both cases concrete measurement and evaluation goals need to be established in a systematic way, measures need to be derived that can help in achieving these goals, and the necessary data need to be collected, validated and interpreted. We have established a methodology that allows us to perform these activities in a systematic way. However, the methodology must be supported by automated tools in order to allow on-line feedback of evaluation results back into ongoing projects. In the long-run, the tools for on-line feedback should be part of each Software Development Environment (SDE), since these environments should provide information to management, development, quality assurance personnel, and others, supporting their decision making processes. Such information would be based on data from the project of interest as well as from previous projects in the same and other environments.

In this paper we present and discuss the TAME (Tailoring A Measurement Environment) project which aims at the development of a prototype measurement and evaluation environment that supports all the previously mentioned activities including the process of setting up measurement and evaluation goals and deriving measures. The prototype currently under development does not interface with an SDE; however, it is designed for being integrated into an SDE in the future. The long-term goal of the TAME project is, however, to come up with guidelines for designing SDEs of the future. We do not believe that evaluation can simply be added to existing SDEs. We believe that SDEs of the future will be driven by TAME-like features allowing for the tailoring of the appropriate software process to project goals and environment characteristics, setting up of measurement and evaluation milestones, and establishing feedback lines. Once the appropriate software process is selected, state-of-the-art tools for performing defined activities such as requirements analysis, design, coding, testing, etc. will be plugged in depending they can provide for the data asked for by the defined process model.

The TAME prototype provides means for collecting, storing, and validating data, computing measures, and interpreting computed values in the context of particular evaluation goals.

A macroscopic view of the TAME architecture shows the system divided into four hierarchically organized layers (from top to bottom): (1) the user interface level, (2) the evaluation level, (3) the measurement level, and (4) the data repository level.

1. The User Interface Level supports the interaction between users and TAME. The TAME prototype will provide a menu driven user interface. In addition, the user interface level contains a tool for setting up the individual measurement and evaluation goals, questions, and measures for each user.
2. The Evaluation Level implements the appropriate context for a particular evaluation session, and contains a processor for running evaluation sessions. The processor performs the evaluation according to the particular needs of the user reflected by a specific set of goals, questions, and measures previously created by the user interface level. In addition, the evaluation processor needs to know the specific authorizations of the user in order to know which functions can be performed by this particular user. The processor also provides analysis functions which, i.e., tell the user whether certain measures can be computed based upon the data currently available in the data repository. This analysis feature of the processor is used during the creation phase of goals, questions, and measures, as well as during the actual evaluation phase according to previously established goals, questions, and measures.
3. The Measurement Level consists of tools for computing measures. The first measurement tools under development are for determining static source code characteristics, data bindings, and structural test coverage. This level is the only level truly dependent on the concepts and languages used for documenting requirements, designs, code, etc. For example in the case of Ada, due to the variety of new language concepts, such as generics, packages, tasks, and exception handling, terms like 'module' or 'data binding' have to be redefined.
4. The Data Repository Level provides the infrastructure for various types of evaluation. This level allows storing and retrieving all kinds of software related data, including evaluation goals, questions, and measures, and authorization data. This level should be as independent as possible of a particular data base management system or a concrete data base structure. It should be implemented as an abstract data type hiding all these implementation details. This data base is designed for all types of information accumulated in Ada projects, not just the information created by measurement and evaluation tools. It also should be capable of interfacing to an existing SDE. From this point of view, this data repository might evolve into a prototype SDE data repository.

The TAME project status is currently as follows: The architectural design of TAME is completed and prototypes of three measurement tools as well as the data repository and the user interface and evaluation level are being implemented. The TAME prototype is to be implemented in Ada (as far as the measurement level is concerned, because we are aiming at Ada projects with this first prototype) and C (for all other, language-independent, levels). TAME will run on a distributed environment, work stations will host instantiations of the user interface level, a main-frame will host all the other levels. Currently we are using SUNs (SUN-3/UNIX) and a MicroVAX/VMS.

The Tame prototype stand alone will provide a useful vehicle for investigating Ada related research questions. Integrated into an SDE, it might allow on-line feedback into ongoing Ada projects.

T A M E

User Interface Level

Evaluation Level

Measurement Level

Data Repository Level

A Comparative Evaluation of Methodologies for Developing
Expert Systems to Aid in Software Engineering Management *

Connie Loggia Ramsey and Victor R. Basili
University of Maryland

ABSTRACT

Although the field of software engineering is relatively new, it can benefit from the use of expert systems. Four prototype expert systems have been developed to aid in software engineering management. Given the values for certain metrics, these systems will provide interpretations which explain any abnormal patterns of these values during the development of a software project. The four expert systems, which solve the same problem, were built using two different approaches to knowledge acquisition, a bottom-up approach and a top-down approach, and two different expert system methods, rule-based deduction and frame-based abduction. A comparison was performed to see which methods better suit the needs of this field. It was found that the bottom-up approach lead to better results than did the top-down approach, and the rule-based deduction systems using simple rules provided more complete and correct solutions than did the frame-based abduction systems.

* Research supported in part by the National Aeronautics and Space Administration Grant NSG-5123 to the University of Maryland. Computer support provided in part by the Computer Science Center of the University of Maryland.

THE VIEWGRAPH MATERIALS
FOR THE
VIC BASILI PRESENTATION FOLLOW

T A M E :
TAILORING A MEASUREMENT ENVIRONMENT

Victor R. Basili
H. Dieter Rombach

Department of Computer Science
University of Maryland
College Park, MD 20740

INTRODUCTION

- **TRANSITION PROBLEMS:**

HOW DO WE

- use Ada concepts properly?
- manage projects properly?
- support Ada projects by methods and tools properly?
- support transition to Ada properly?

- **SOLUTIONS:**

- evaluate various quality/productivity aspects of Ada processes/products
- allow for tailoring processes to specific project needs
- provide tool support
- make measurement & evaluation an integral part of an SDE
(ADDS A NEW DIMENSION TO SDEs)

TAME (TAILORING A MEASUREMENT ENVIRONMENT)

INTRODUCTION

PROJECT PHASES:

- **DEVELOP A PROTOTYPE SUPPORTING ALL KINDS OF MEASUREMENT & EVALUATION ACTIVITIES**
(goal/question/metric paradigm)
- **INTERFACE TAME PROTOTYPE TO AN EXISTING SDE**
(providing on-line feedback into development activities)
- **DEVELOP GUIDELINES FOR THE DESIGN OF FUTURE SDEs**
(support various process models, tailor process models to project goals and environment)

TAME (TAILORING A MEASUREMENT ENVIRONMENT)

MEASUREMENT HISTORY (AT UMD)

- The frequent incorrect usage of methods and tools is not known (NO FEEDBACK) [Basili, Gannon, Yeh, Zelkowitz, ..]
- Single metrics are not sufficient [Basili, Turner]
- We have to design experiments thoroughly [Basili, Reiter]
- We can't just use other people's models (TAILORING) [Basili, Freburger]
- Meta Models and Metrics are suited for transition purposes [Basili, Bailey]
- We have to associate interpretations with metrics [Basili, Doerflinger]
- Goals/Questions/Metrics paradigm for measurement and evaluation [Basili, Weiss]
- Measurement and evaluation is judged based upon "cost/payoff" and "confidence in results" issues (CLASSIFICATION SCHEME FOR EXPERIMENTS) [Basili, Selby]
- Formalize expert knowledge (EXPERT SYSTEM) [Basili, Loggia-Ramsey]
- Formalize evaluation & improvement paradigm [Basili]
- Formalize the tailoring of processes towards project quality goals and environments [Basili, Rombach]

TAME (TAILORING A MEASUREMENT ENVIRONMENT)

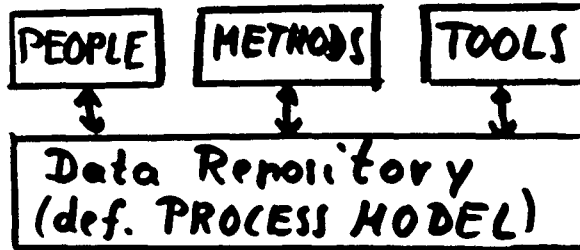
LESSONS LEARNED

- DEVELOP QUALITY A PRIORI
- PROVIDE & SUPPORT FEEDBACK
- DEVELOPMENT METHODS ARE HEURISTIC AND NOT FORMAL
- PREPARE DEVELOPMENT PROCESS FOR MEASUREMENT & EVALUATION
- ALL PROJECT ENVIRONMENTS ARE DIFFERENT
- REUSE EXPERIENCE ONLY AFTER TAILORING IT
- THERE ARE MANY PROCESS MODELS (NEED TO BE TAILORED)
- MANAGEMENT CONTROL IS CRUCIAL AND MUST BE FLEXIBLE
- MEASURES IN ISOLATION ARE USELESS (METRIC VECTOR)
- EXPLORATION MUST BE TOP-DOWN (TO GET THE WHOLE PICTURE)
- DEFINE TERMS (COMPLEXITY, METHODS,)

TAME (TAILORING A MEASUREMENT ENVIRONMENT)

FUTURE SDEs

- SDE MODEL (Environments Workshop, UMD, May 1986):



CONSTRUCTIVE SDEs	
one method or tool	
a set of methods and tools	
a set of methods and tools supporting ONE particular process model	// TAME //
a set of methods and tools supporting a VARIETY of process models	// - short-term - mid-term //
a set of methods and tools supporting a VARIETY of META PROCESS MODELS, which can be TAILORED to specific project needs.	←→ TAME ←→ (long-term) ←→

TAME (TAILORING A MEASUREMENT ENVIRONMENT)

TAME REQUIREMENTS

- **PURPOSE: DEVELOP PROTOTYPE FOR**
 - Establishing evaluation goals
 - Deriving questions / metrics
 - Collecting / validating data
 - Storing / retrieving data
 - Interpreting data wrt. evaluation goals

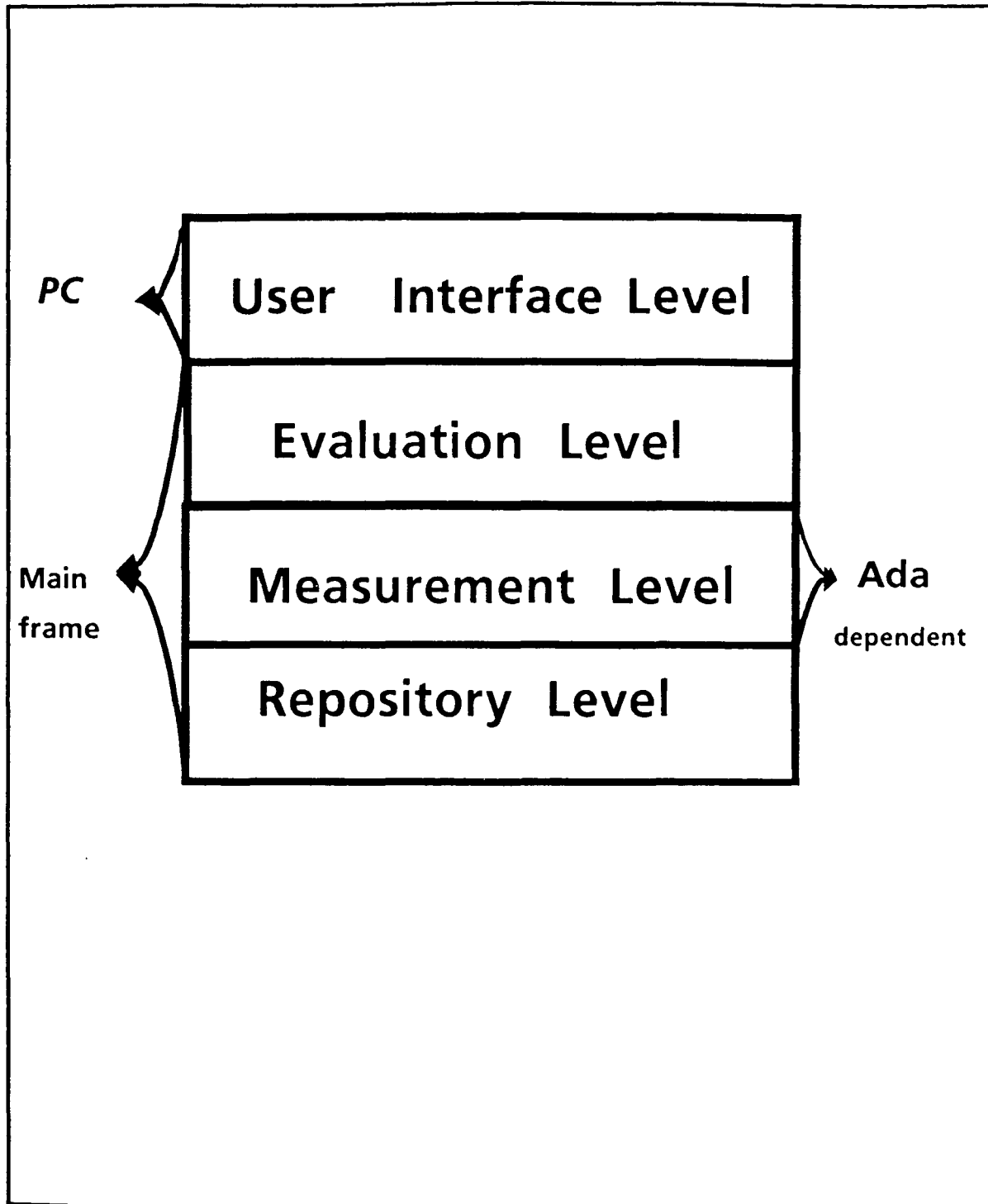
- **POTENTIAL USERS:**
 - Managers
 - Developers
 - QA personnel
 - Researchers

- **USER VIEW OF THE SYSTEM:**
 - A system which is driven by a quantitative / traceable SW quality model (goals/questions/metrics)
 - NOT just a regular SDE augmented with measurement tools
 - Open-ended wrt. metrics & evaluation goals

TAME (TAILORING A MEASUREMENT ENVIRONMENT)

V. Basili
Univ. of Maryland
11 of 24

TAME ARCHITECTURE



TAME (TAILORING A MEASUREMENT ENVIRONMENT)

FIRST PROTOTYPE

- **IMPLEMENT RESTRICTED SUBSET OF THE TAME REQUIREMENTS:**
 - no support of interpretation
 - small subset of measurement tools
 - only minimal SDE interface (access to development documents)
 - restricted data repository
 - Limited emphasis on security / configuration management control strategies
- **TARGET SYSTEM(S):**
 - MicroVAX (VMS) & SUN-3 (UNIX)
 - Implementation should allow for distribution (data repository, measurement tools & evaluation on a main-frame, user-interface level on PC's)
- **IMPLEMENTATION LANGUAGE(S):**
 - ADA (MicroVAX) for Ada-dependent levels (measurement & data repository level)
 - Pascal or C (SUN-3) for the Ada-independent levels (user interface & evaluation level)
- **SCHEDULE:**
 - Complete this prototype by Fall 1987
 - Work on related research issues in parallel

TAME (TAILORING A MEASUREMENT ENVIRONMENT)

CONCLUSIONS

- TAME IS A VERY AMBITIOUS PROJECT
- TAME'S OBJECTIVES GO BEYOND AUTOMATING MEASUREMENT & EVALUATION ACTIVITIES
- TAME WILL HELP DEVELOPING GUIDELINES FOR FUTURE SDEs
- TAME ALLOWS / REQUIRES SOFTWARE ENGINEERING TO INTERFACE WITH OTHER COMPUTER SCIENCE DISCIPLINES
 - data bases
 - artificial intelligence

TAME (TAILORING A MEASUREMENT ENVIRONMENT)

**EXPERIMENTING WITH
EXPERT SYSTEMS
FOR
SOFTWARE MANAGEMENT**

**VICTOR R. BASILI
CONNIE LOGGIA RAMSEY**

MOTIVATION

**Why develop expert systems for
Software engineering ?**

**To capture rules of the SE process and
use them to guide S/W management**

It allows us to:

handle more information

capture corporate knowledge

train new personnel

Methodology

Given a homogeneous environment and data from past software projects,

1. determine useful variables

- easy to collect

- meaningful

- examples

programmer hours

lines of code

Methodology

2. Develop baselines of normalized metrics

- example - average programmer hours per line of code for past projects at specific time intervals
- historical
- environment-specific
- normalize by comparing variables against each other
- average value of baseline is "normal" for past projects
- deviant metric values (more than one standard deviation above or below the average baseline) suggest abnormal project development

Methodology

3. Determine interpretations for metric values which deviate from baseline.

- examples

unstable specifications

good testing

RESEARCH ISSUES

FEASIBILITY OF EXPERT SYSTEMS

- SCIENCE OF SOFTWARE ENGINEERING NOT WELL DEFINED
- KNOWLEDGE BASE EXPLORATORY

METHODS FOR CONSTRUCTION

- TYPE OF INFERENCE MECHANISM
RULE-BASED DEDUCTION VS. FRAME-BASED ABDUCTION
- METHOD OF KNOWLEDGE ACQUISITION
TOP-DOWN VS. BOTTOM-UP

TRANSPORTABILITY

- ARE THE RESULTS GENERALIZABLE
- CAN SYSTEMS BE MOVED TO OTHER ENVIRONMENTS

Methods for Building Expert Systems

- Determine which is best suited for software engineering

- KMS

- Rule-Based Deduction
 - IF <antecedents>
THEN <consequents>

 - used simple rules (one metric clause in antecedent)

 - used certainty factors (indicate certainty of conclusion given the antecedent)

- Frame-Based Abduction
 - one frame for each interpretation

 - hypothesize-and-test cycles

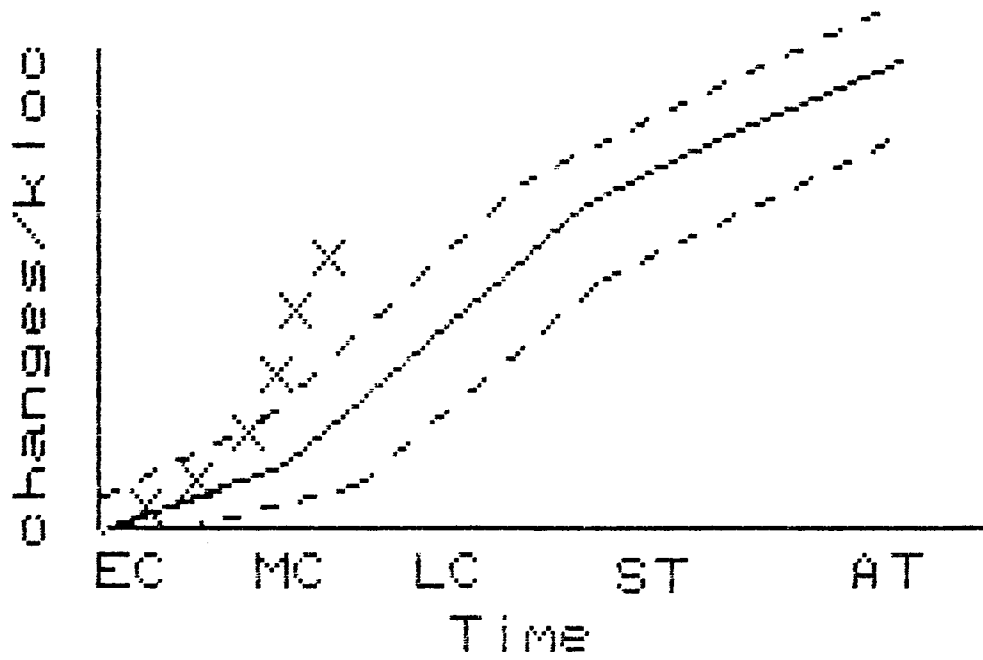
 - generalized set covering model

Example Operation

Given a new project,

- determine values of metrics at particular time phase
- determine whether these values are deviant (using standard deviation test)
- indicate findings to expert system

SCENARIO UTILIZING EXPERT SYST



POSSIBLE INTERPRETATIONS

ERROR PRONE CODE <0.94>

EASY ERRORS OR CHANGES BEING FOUND OR FIXED <0.81>

LOTS OF TESTING <0.75>

LOTS OF TERMINAL JOCKEYS <0.75>

UNSTABLE SPECIFICATIONS <0.50>

NEAR BUILD OR MILESTONE DATE <0.50>

GOOD TESTING OR GOOD TEST PLAN <0.25>

MODIFICATIONS BEING MADE TO RECENTLY TRANSPORTED
CODE <0.25>

AUTOMATING THE DESIGN PROCESS WITH SYNTACTIC-BASED TOOLS

Marvin V. Zelkowitz
Department of Computer Science
University of Maryland
College Park, Maryland 20742

Abstract

This report describes a tool that is being developed to aid in the design process of software production. It is an extension to the SUPPORT syntax directed editor. The idea is to create grammatical descriptions of the design process and embed them into the syntactic structure of code production. The extension of a monolithic environment like SUPPORT which can handle programs of about 5,000 source lines into a distributed environment handling systems of up to 1,000 modules of 100,000 source lines of code is also under study. This report describes some of the experiences to date and gives early indications of how this development will proceed.

1. Introduction

For the past several years, the SUPPORT environment has been under development as a mechanism to aid programmers to build and test reliable Pascal programs [Zelk 84]. From one perspective (e.g., the user community of Freshmen computer science majors at the University of Maryland), SUPPORT can be view as an intergrated environment for the development of Pascal programs that runs on an IBM PC computer. It is based upon a syntax-directed editing paradigm for entry of source text, and it contains an interpreter and debugging tools all managed in a multi-window environment (Figure 1).

Syntax editing differs from the usual character oriented editor in an important way. Instead of the cursor on the screen indicating a position in a program file where text can be entered, in a syntax directed editor the cursor wraps a segment of the program and the user can either type in the appropriate text or can choose an appropriate response from a menu of choices on the screen (Figure 2). In this example, the cursor wraps the statement placeholder, and any text that is a syntactically correct statement may be entered.

The use of windows allows for a programmer to monitor several activites at one time. For example, Figure 3 displays four windows on the screen. The top window is the command window where commands to SUPPORT are echoed. Window 2 is the Variable Trace window where individual variables can be displayed and are automatically undated on the screen when their values change. Window 3 is the Statement Trace window where the programmer can watch the program dynamically execute,

and window 4 is the Execution output window where output from the program appears. The system contains several additional ways to aid the programmer in developing a program including the ability to interrupt and restart execution, to set breakpoints and to display data from the program's run time data display stack.

SUPPORT has been in use at the University of Maryland, and based upon this past year's experience, as well as experience from others [SIGS 84] who have built syntax directed editors, several conclusions can be stated (Figure 4):

- (1) The acceptance of syntax directed editing seems inversely correlated with previous programming experience. That is, more experienced users find the constraints of syntax editing very limiting. Although the syntax editing paradigm has been available for about 8 years, it has not generally caught on. More about this later.
- (2) Syntax editing is a powerful program building technique, but is it perhaps more of a solution looking for a problem to solve. The concept is very simple - programs are stored as parse trees, and commands to manipulate these trees are easy to build. Unfortunately, little work has gone into how to use these commands effectively and how to integrate these commands with the needs of programmers.
- (3) An important issue in all this is the user interface. The syntax editor has to help the user build better programs and not perceived as a roadblock to its acceptance. This user interface has been generally ignored by others.
- (4) One way to aid the user is the concept of an editing hierarchy with syntax editing as only one aspect of this structure.

2. Editing Hierarchy

Syntax editing has generally not be viewed as a powerful editing technique since it is viewed as too constraining to the program. The problem is that the technique is useful in limited situations so an editor that depends upon the technique for total program generation is hard to manage.

For example, program modification is extremely difficult with this paradigm. Usually the entire segment of the program must be deleted and rebuilt since the underlying tree representation must be maintained. In order to avoid such problems, the following classification scheme for editors is being developed (Figure 5) [Zelk 87]. It is assumed that an effective editor contains more than one level of editing feature.

- (1) At the base level is the character oriented editor, much like existing full screen editors. In the SUPPORT case, the user can wrap a segment of source program, pull the text into a separate editing window, modify this text arbitrarily, and have the text reparsed into the program tree on exit from this internal editor.
- (2) A second feature is the file inclusion feature. This works best for passing data declarations like COMMON blocks in FORTRAN among modules.
- (3) A third level is the syntax editing mentioned earlier. Most editors have the first two of these and syntax editing has only this third level. SUPPORT contains all three levels. The include feature reads in full syntax which means that programs

written using other editors or systems can be pulled into the environment. Although SUPPORT is designed as a closed system with its own file system format, it does have commands for importing and exporting source programs so that SUPPORT can be used to interface with other systems.

- (4) The ability to add macros adds to the power and extensibility of the underlying programming language. An experimental version of SUPPORT is looking at macro additions. The inclusion of large macros gives the system a fourth generation language flavor with a fill-in-the-blanks approach to programming while a large number of small macros gives the system the appearance of a language extension and a design language flavor.
- (5) Knowledge representation allows for the ability to use artificial intelligence and expert system technology to aid in the design process. This is currently under study in SUPPORT.
- (6) A complete data flow model allows for compiler optimization technology to build better source programs.

The SUPPORT project is evaluating these alternatives. It is our belief that a system that implements them all will develop as a powerful design language processor and not simply a source code generator.

3. Application to NASA

An important issue to discuss here is what is the relationship of a system like SUPPORT to the programming environment at NASA. Source code entry is not a major problem there, so a syntax directed editor would be of marginal benefit. However, instead of the view of SUPPORT given above, consider the following description (Figure 6):

- (1) The language handled by SUPPORT is defined by a grammar that initializes SUPPORT's internal tables when execution begins. Grammars for subsets of Pascal are currently used by students; however, we have built grammars for C and for Ada, and are experimenting with design grammars useful to NASA.
- (2) SUPPORT contains a window manager for developing any structured text. There is some semantic checking of this text to aid in text generation.
- (3) Much of software production is document preparation: requirements, specification and design documents, source code, test plans, etc. NASA estimates that up to 50% of the cost of a project falls into this area. Thus an effective document preparation system would have a large potential payoff.

An early prototype of SUPPORT for the NASA environment is described by Figure 7. Each NASA FORTRAN module is described by a design prologue and the generation of structure charts is an early indication of overall program structure (Figure 8).

In Figure 9, the grammar read into SUPPORT was designed to look like these design prologues, and in this picture module ProcC is called from both ProcA and ProcB and calls ProcD and ProcE. This can be easily seen by the new window which displays this information automatically (Figure 10). The user can navigate through

the structure chart with the current module always being in the center (ProcC in Figure 10) and the calling modules on line 1 and the called modules on line 3.

The initial prototype seems quite feasible, so the problems that remain are how to integrate the basic window structure of SUPPORT into the operational NASA environment. (This is not as easy as it sounds.) Thus the goals for the current research are summarized by Figure 11:

- (1) SUPPORT is limited to single user systems of up to 5,000 lines of source program. This needs to be extended to systems of several hundred modules and perhaps up to 100,000 lines of code. Internal data structures need to be greatly altered to handle such complexity.
- (2) The design language grammar needs to be studied for effectiveness in this environment.
- (3) NASA needs to be able to test programs on the target computer system which is a large mainframe while SUPPORT was designed to run on DEC VAX and IBM PC computers. Instead of an internal interpreter, the system needs to interact with the running program on another machine in a distributed manner. The major research problem is to develop a protocol for this interaction.

This project has shown the feasibility of using integrated environments within industrial settings. However, in order to make such systems practical, there needs to be further work on both the design language needed by a professional programming group like at NASA and with building distributed integrated environments across diverse hardware.

4. Acknowledgement

This work was supported in part by NASA grant NAG5-368 to the University of Maryland.

5. References

[SIGS 84] *Proceedings of the ACM SIGSOFT Symposium on Practical Software Development Environments*, Pittsburgh PA, April, 1984.

[Zelk 84] Zelkowitz M. V., A small contribution to editing with a syntax directed editor, *ACM SIGSOFT Symposium on Practical Software Development Environments*, Pittsburgh PA, April, 1984, 1-6.

[Zelk 87] Zelkowitz M. V., An editor for program designs, *IEEE Computer Society Compcon*, San Francisco, CA, February, 1987.

THE VIEWGRAPH MATERIALS
FOR THE
M. ZELKOWITZ PRESENTATION FOLLOW

SUPPORT

One View

- Integrated environment for Pascal development
- Based upon syntax directed editor
- Pascal editor, interpreter, debugger, window manager
- Runs on PC-DOS and UNIX systems
- Used at University of Maryland in introductory computer science course for majors

Figure 1.

PROGRAM TEXT:-----

```
program MainProgram (input,output); <.decs. ...>  
begin  
  A := 'X';  
  while A <> '' do  
    begin  
      read(A);  
      <.stmt.>  
    end  
end.
```

CURSOR



MENU



1-block 2-if 3-while 4-proc call 5-:= 6-I/O stmts 7-null

M. Zelkowitz
Univ. of Maryland
6 of 15

Figure 2.

MULTIPLE VIEWS

VARIABLE DISPLAY:-----

STOPPED : StmtCount:24
[CopyToEol]-Ch=
[Add]-Elt=

a

[EmptyQ]-Switch= 1

VARIABLE TRACE

TRACING: [TestRemove] [AddQ]-----

Add(Q1,Q2,{ copy Q1 and Elt to Q2 }Elt);
Switch := '2'
end else begin
Add(Q2,Q1,{ copy Q2 and Elt to Q1 }Elt);
Switch := '1'

PROCEDURE TRACE

STATEMENT TRACE

end
end
end;
procedure Del{ Del }(var QFrom,QTo : text);<.decs. ...>
begin
reset(QFrom);

PROGRAM EXECUTION:-----

***** 2: Continue Execution
The input is :a
***** 25: Breakpoint Reached

EXECUTION OUTPUT

Figure 3.

SUPPORT

Initial results

- Preference for standard editing seems correlated with previous experience
- Other syntax editors generally not used extensively
- Syntax editing is a powerful program building technique, but is it more a solution looking for a problem to solve?
- The user interface has generally been ignored by others
- Development of editing hierarchy

Figure 4.

SUPPORT

Editing hierarchy

- Full screen character editor
- Hierarchical file inclusion
- Syntax editing
- Macro processing
- Knowledge representation
- Data flow model

Figure 5.

SUPPORT

How applied to NASA environment?

- Driven by externally defined grammar
- Intelligent window manager for developing structured text
- Some semantic checking of attributes
- Much of software development is document preparation

Figure 6.

SUPPORT

Design Prototype

- Look at design prologues
- Build structure charts

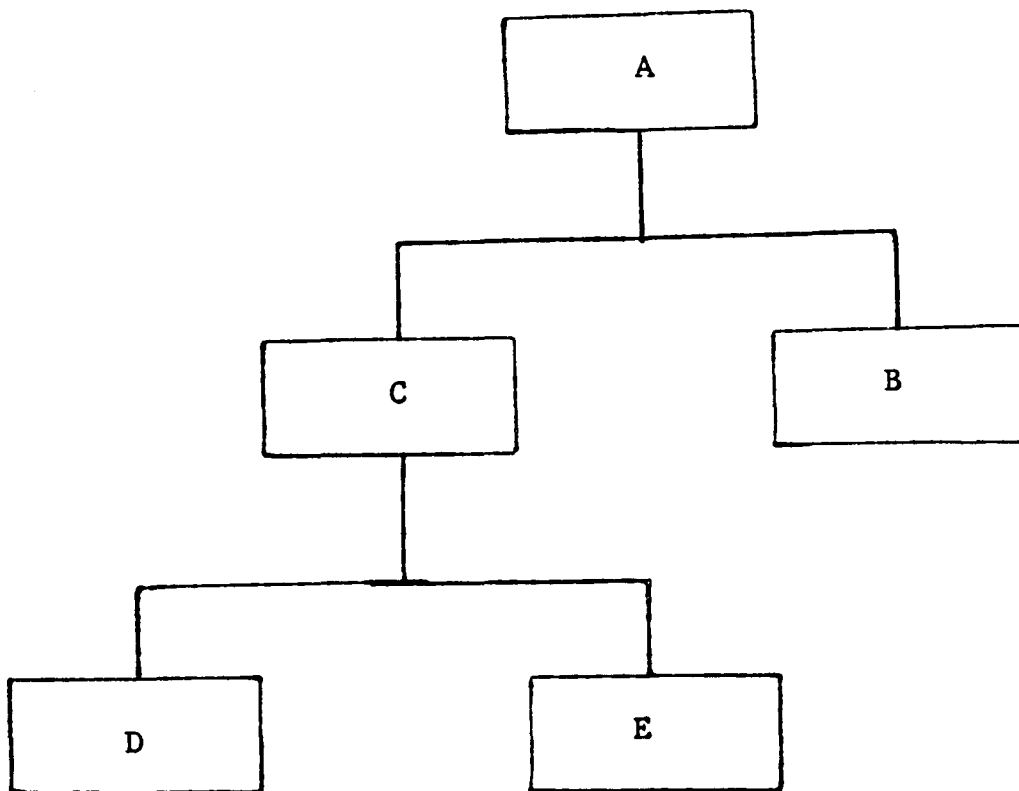


Figure 7.

PROGRAM TEXT:-----

```
PROGRAM <.id.>
C
CCC
C*****
C      PURPOSE : <.texts.>
C
C      ARGUMENT LIST :
C      ARGUMENT TYPE  I/O DIM  DESCRIPTION
C      <.id.> <.typ.> <.iocont.> <.id.> <.texts.>
C
C      CALL BY : <.clist.>
C
C      ROUTINE CALLED : <.rlist.>
C
C      COMMONS REFERENCE: <.comlist.>
C
C      EXTERNAL FILES REFERENCED:
C      FORTRAN UNIT#  FILE NAME OPERATIONS
C      <.id.>
C
C      DESIGNER: <.names.>
C
C      PROGRAMMER: <.name.>
C
C*****
C ENDPROG
C*****
```

Figure 8.

PROGRAM TEXT:-----

```
C
  SUBROUTINE ProcC
CCC
C*****
C  PURPOSE : <.texts.>
C
C  ARGUMENT LIST :
C  ARGUMENT TYPE  I/O DIM  DESCRIPTION<.argcont.>
C
C  CALL BY : ProcA, ProcB
C
C  ROUTINE CALLED : ProcD, ProcE
C
C  COMMONS REFERENCE: <.comlist.>
C
C  EXTERNAL FILES REFERENCED:
C  FORTRAN UNIT#  FILE NAME OPERATIONS
C  <.unitid.>
C
C  DESIGNER: <.names.>
C
C  PROGRAMMER: <.name.>
C*****
C  ENDPROC
C*****
C
  SUBROUTINE ProcD
CCC
C*****
```

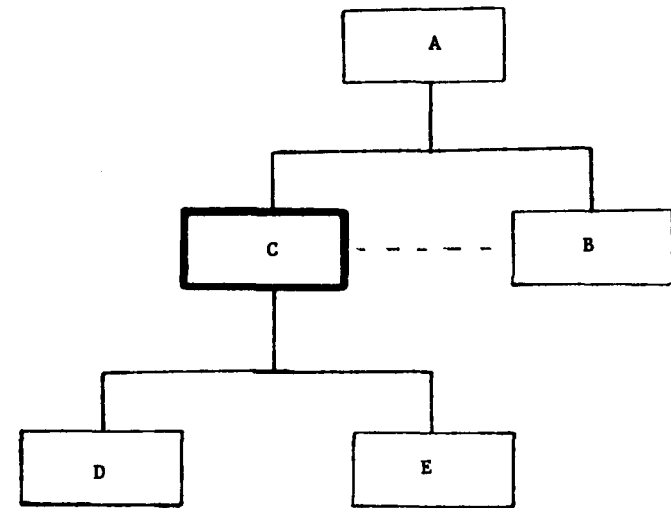
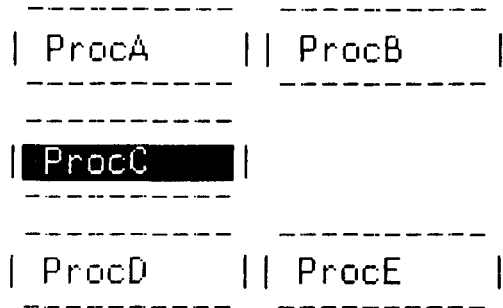


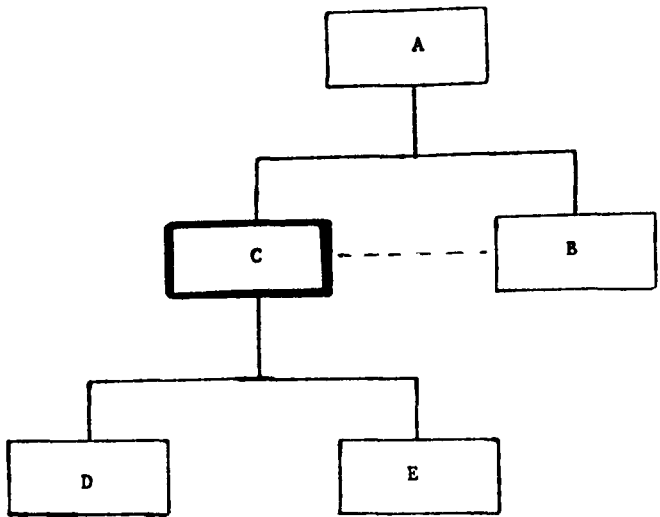
Figure 9.

PROGRAM TREE-----



PROGRAM TEXT:-----

```
C  
C      SUBROUTINE ProcC  
CCC  
C*****  
C      PURPOSE : <.texts.>  
C  
C      ARGUMENT LIST :  
C      ARGUMENT TYPE I/O DIM DESCRIPTION<.argcont.>  
C  
C      CALL BY : ProcA, ProcB  
C  
C      ROUTINE CALLED : ProcD, ProcE  
C  
C      COMMONS REFERENCE: <.comlist.>  
C  
C      EXTERNAL FILES REFERENCED:  
C      FORTRAN UNIT# FILE NAME OPERATIONS  
C      <.unitid.>  
C  
C      DESIGNER: <.names.>
```



M. Zelkowitz
Univ. of Maryland
14 of 15

Figure 10.

SUPPORT

Goals of current project

- SUPPORT currently limited to single user 5,000 line software
- Extend concept to 100,000 lines multi-user developments
- Extend system to create a design processor
- Distributed design - interact with a running program
- Need to consider multi-machine interface

Figure 11.

SEL ADA* EXPERIMENT:
STATUS AND DESIGN EXPERIENCES**

William W. Agresti***
Computer Sciences Corporation

ABSTRACT

The status of the Software Engineering Laboratory (SEL) Ada experiment is reviewed, and the designs produced by the parallel FORTRAN and Ada development teams are compared. The Ada team produced a significantly different design for the spacecraft dynamics simulator. Several lessons learned from the Ada design experience are discussed, including the conditions favoring an Ada-oriented design and the importance of understanding management expectations.

INTRODUCTION

Ada shows promise as a significant contributor to the development of more reliable software. An experiment is in progress at the National Aeronautics and Space Administration's Goddard Space Flight Center (NASA/GSFC) to learn whether Ada will deliver on its promise. The experiment is planned and administered by the Software Engineering Laboratory (SEL) in

*Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).

**Proceedings, Eleventh Annual Software Engineering Workshop, National Aeronautics and Space Administration, Goddard Space Flight Center, December 1986.

***Author's Address: Computer Sciences Corporation, System Sciences Division, 8728 Colesville Road, Silver Spring, Maryland 20910.

the Flight Dynamics Division at NASA/GSFC. Personnel from all three SEL participating organizations--NASA/GSFC, Computer Sciences Corporation (CSC), and the University of Maryland--support the Ada experiment.

The objective of the experiment is to assess the effectiveness of Ada in the flight dynamics software development environment at NASA/GSFC. The experimenters intend to gain an initial understanding of Ada's effect on productivity, reliability, maintainability, reusability, and manageability. This Ada experience is expected to assist in planning the development of software for the NASA Space Station.

The Ada experiment began in January 1985. It involves the parallel development in FORTRAN and Ada of the attitude dynamics simulator for the Gamma Ray Observatory (GRO) spacecraft. Experiment organization and Ada training experiences are discussed in [Agresti 85] and [Murphy, Stark 85]. This paper provides an update on the experiment status, a summary of the FORTRAN and Ada design comparison, and a collection of lessons learned about Ada-oriented design.

EXPERIMENT STATUS

The GRO dynamics simulator is being developed by two teams, one using Ada and the other using FORTRAN, the language typically employed in this environment (Figure 3). Each team was initially staffed at seven programmer/analysts. The development environment is the DEC VAX-11/780 and -8600 computers under VMS, with the Ada team using the DEC Ada Compilation System (ACS).

The simulator is part of the ground support software needed for the GRO mission. Both development teams are building real, operational software, not a "toy" system devised specifically for the experiment. The software will allow analysts to test onboard flight software under conditions that simulate the expected in-flight environment as closely as possible. The FORTRAN simulator consists of 51,000 source lines of code. The Ada simulator, though not finished, will be much larger as measured by source lines.

Figure 4 shows the schedule followed by both teams since the January 1985 start. The FORTRAN team's schedule is typical for this environment, with the exception being the extended acceptance testing period due to lessened schedule pressure after 1985. The Ada team faced the task of first learning Ada. The first 5 months were spent chiefly in Ada training, including the development of a 6000-line Ada pilot project [Agresti 85]. This training period caused the Ada team to lag the FORTRAN team in progressing through the development phases. Another cause of the schedule differences in Figure 4 is the unequal staff effort levels of each team. Figure 5 accounts for this level-of-effort disparity, showing the staff-months of effort attributed to each activity by each development team. For requirements analysis and design, the Ada team expended more effort; for Ada system testing, however, less effort is being estimated. The rationale for this lower estimate is that Ada modules are continually integrated during implementation. The effort typically required in a FORTRAN development for integration during the system testing phase is not expected to be needed by the Ada team.

COMPARING FORTRAN AND ADA DESIGNS

As Figures 4 and 5 indicate, the Ada team is currently in the implementation phase. When the Ada system is finished, the long-awaited comparison to the FORTRAN simulator will be possible. Available now, however, are both designs, so a design comparison was conducted [Agresti et al. 86].

A preliminary comparison of the designs produced by the FORTRAN and Ada teams revealed clear differences in design drivers and design abstractions (Figure 6). The FORTRAN team had more schedule pressure than the Ada team. This condition, when coupled with the presence of FORTRAN-oriented designs for past simulators, led to the adaptation of past designs to serve the new mission's requirements. The Ada team was encouraged to investigate so-called Ada-oriented design approaches, that is, those that exploit Ada's features and capabilities. The team was exposed to both PAMELA [Cherry 85] and Booch's object-oriented approach [Booch 83] during its training. Ultimately, the team developed and applied its own object-oriented methodology [Seidewitz, Stark 86]. The principal design drivers for the Ada team were the encouragement to pursue new design methods and the freedom not to reuse past FORTRAN-oriented designs.

The design abstractions detected in the two designs reflect the differences in design drivers. The FORTRAN design primarily uses procedural abstractions, whereas the Ada design is structured around objects and state machine abstractions. The designs were nearly identical as measured by the number of procedural units defined at the critical design review (CDR): 262 FORTRAN subroutines versus 252 Ada subprograms.

Another focus of the design comparison was the operation of each simulator as prescribed in each design. Figure 7 shows the high-level structure of each design. Worth noting first is the difference in design graphical notation. The FORTRAN design is represented by structure charts showing invocation. The Ada design uses object diagrams, with an arrow from object A to object B showing that object A uses services provided by object B [Seidewitz, Stark 86]. The names used in each design structure of Figure 7 are identical in three of four cases: simulation control, onboard computer (OBC) model, and truth model. The presence of a simulation control structural element in each design is understandable. Because the interaction between the OBC and the rest of the spacecraft and its environment (truth model) is the central element of the attitude dynamics simulation, it is again understandable that each design have such major units.

Although the names of the major units are identical, the operation of the simulators is not. The FORTRAN simulator increments its simulation clock and sequences through the three major units in Figure 7: truth model, OBC model, and simulation output. The Ada design in Figure 7 shows the OBC model object above the truth model object in the diagram. In the Ada team's object-oriented methodology, this arrangement of objects in diagrams means that the OBC model object is "senior" to the truth model object. The timing of the Ada simulator is controlled by the OBC model object, which issues requests to the truth model for needed sensor data. The timing of the OBC model is not under user control as it is in the FORTRAN design.

The differences in design abstractions and simulator operations led to the determination that the Ada design was different in essential ways from the FORTRAN design (Figure 8).

A more detailed comparison of the designs is provided in [Agresti et al. 86].

LESSONS LEARNED ABOUT ADA DESIGN

The Ada team, through its experiences during the design phase, learned a number of lessons about the relationship of Ada and design. The first lesson follows directly from the observation that a different design was produced by the Ada team. Using Ada as the implementation language does not ensure that an Ada-oriented design (i.e., one that exploits Ada's abstractions and features) is produced. For example, in one early Ada development project, the design of the Ada system "looked like a FORTRAN design" [Basili et al. 85]. That an Ada-oriented design was produced in the current experiment was attributed to the following conditions [Agresti et al. 86]:

- The Ada team had the necessary resources to afford not to reuse past designs.
- The Ada team worked directly from system requirements by removing the FORTRAN legacy in requirements documents and recasting system requirements in the multiple views of the Composite Specification Model (CSM) [Agresti 84].
- The Ada team understood alternative design abstractions and was encouraged to explore Ada-oriented design methods (Figure 9).

The first condition--the flexibility not to reuse past designs--is especially important. Reuse is a cost-effective approach in the flight dynamics environment [Card et al. 86]. Not taking advantage of the legacy of past designs is costly but, in this experiment, a key reason why an Ada-oriented design resulted.

The more general issue of legacy must be addressed by software development organizations that have well-established procedures and standards oriented to other languages. Figure 10 shows a range of alternatives for the introduction of Ada, along with some possible consequences. If an organization seeks to exploit Ada's features, the best opportunity exists when Ada is designated at the start of a project. In such a case, the requirements can be specified in a more language-neutral form, and the designers can be free to introduce design abstractions that reflect problem-domain entities and map conveniently to Ada language features.

Several lessons learned during the design phase related to the management of an Ada-oriented design activity (Figure 11). At the CDR of a FORTRAN project, the number of modules is known and used in estimating remaining cost. What, however, should the cost-estimating unit be for an Ada development? At the CDR, the Ada design had 252 subprograms organized into 104 packages; the Ada team used 252 as the size measure for estimating future costs. At project completion, it will be clear whether 356 (252 + 104) would have been a better basis for the cost estimate.

The Ada design products were unfamiliar to higher level managers. The design documents and CDR materials featured object diagrams like the one in Figure 12; managers, however, were accustomed to seeing the structure charts used in FORTRAN designs. The Ada team learned the importance of educating management and review personnel before the CDR so that the design notation would be understood.

The use of Ada-oriented design methods forced a reevaluation of the milestone marking the end of the design phase. In the flight dynamics environment, the criteria for design phase completion are currently FORTRAN oriented. Ada offers

greater opportunity for checking design consistency with the compiler. Compilable package specifications and type definitions seem to be especially important elements to have completed by the CDR.

Project managers monitored the level of effort expended by the Ada team during the design phase (Figure 13). Because all members of the Ada team divide their time between the Ada project and other work, effort levels fluctuated considerably. Figure 13 shows that the seven team members contributed hours to the Ada project at a level lower than two full-time equivalent staff during some months of the design phase. This varying, and generally low, level of effort contributed to the extended schedule shown in Figure 4. The peak effort in Figure 13 coincided with management reviews, demonstrating that the deadline effect is language independent.

Some preliminary lessons learned about staffing were the positive experiences using recent computer science graduates and phasing in new Ada team members (Figure 14). The Ada package specification-body dichotomy facilitates the definition of a "design envelope" to restrict the working span of new staff members.

SUMMARY OBSERVATIONS

The principal observations on the Ada team's design experiences can be summarized as follows (Figure 15):

- The Ada team did produce a different design for the simulator.
- Specific conditions encourage the production of Ada-oriented designs.
- Ada can influence every aspect of design, especially management expectations.

Monitoring of the Ada experiment will continue. The collected experiences in coding and testing--along with the FORTRAN and Ada product comparisons--will certainly provide useful information on the effect and effectiveness of Ada.

ACKNOWLEDGMENTS

The support of the Ada team and the contributions of the Ada experiment managers, F. McGarry and R. Nelson of NASA/GSFC, are appreciated.

REFERENCES

[Agresti 84]

Agresti, W. W., "An Approach for Developing Specification Measures," Proceedings, Ninth Annual Software Engineering Workshop, NASA/GSFC, November 1984

[Agresti 85]

Agresti, W. W., "Measuring Ada as a Software Development Technology in the Software Engineering Laboratory (SEL)," Proceedings, Tenth Annual Software Engineering Workshop, NASA/GSFC, December 1985

[Agresti et al. 86]

Agresti, W. W., V. E. Church, D. N. Card, and P. L. Lo, "Designing With Ada for Satellite Simulation: A Case Study," Proceedings of the First Annual Symposium on Ada Applications for the NASA Space Station, Houston, Texas, June 1986

[Basili et al. 85]

Basili, V. R., E. E. Katz, N. M. Panlilio-Yap, C. L. Ramsey, and S. Chang, "Characterization of a Software Development in Ada," IEEE Computer, September 1985

[Booch 83]

Booch, G., Software Engineering With Ada. Menlo Park, California: Benjamin/Cummings Publishing Co., Inc., 1983

[Card et al. 86]

Card, D. N., V. E. Church, and W. W. Agresti, "An Empirical Study of Software Design Practices," IEEE Transactions on Software Engineering, February 1986

[Cherry 85]

Cherry, G. W., "Advanced Software Engineering With Ada--
Process Abstraction Method for Embedded Large Applica-
tions," Language Automation Associates, Reston,
Virginia, 1985

[Murphy, Stark 85]

Murphy, R., and M. Stark, Ada Training Evaluation and
Recommendation, SEL-85-002, NASA/GSFC, October 1985

[Seidewitz, Stark 86]

Seidewitz, E., and M. Stark, General Object-Oriented
Software Development, SEL-86-002, NASA/GSFC, August 1986

THE VIEWGRAPH MATERIALS
FOR THE
W. AGRETI PRESENTATION FOLLOW

SEL ADA EXPERIMENT: STATUS AND DESIGN EXPERIENCES

**W.W. AGRESTI
COMPUTER SCIENCES CORPORATION**

CSC **COMPUTER SCIENCES CORPORATION**
SYSTEM SCIENCES DIVISION

1038-AGR-(179*)

FIGURE 1

- **EXPERIMENT STATUS**
- **COMPARING FORTRAN AND ADA DESIGNS**
- **LESSONS LEARNED ON ADA DESIGN**

CSC

COMPUTER SCIENCES CORPORATION
SYSTEM SCIENCES DIVISION

1038-AGR-(179*)

FIGURE 2

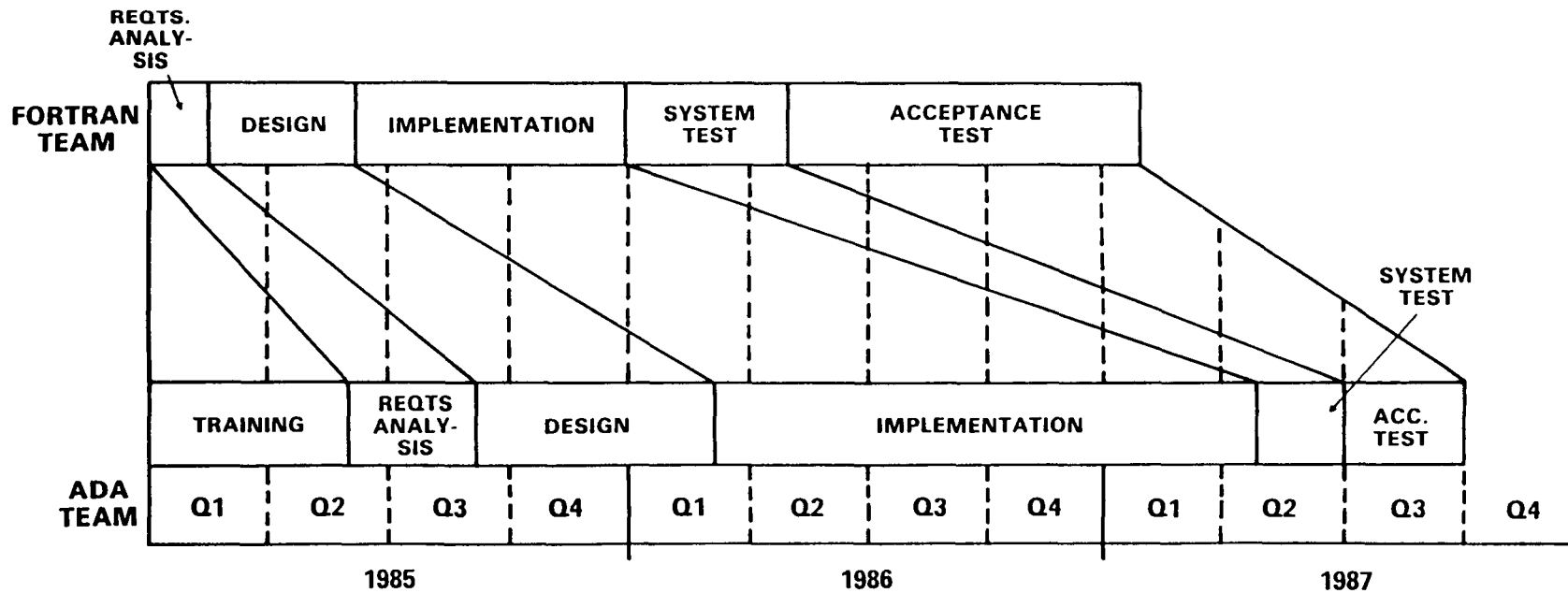
W. Agresti
CSC
12 of 25

SEL ADA EXPERIMENT

- **PARALLEL DEVELOPMENT IN FORTRAN AND ADA**
- **PROJECT: GAMMA RAY OBSERVATORY (GRO) DYNAMICS SIMULATOR**
 - **SIZE: 50,000 (FORTRAN) SOURCE LINES OF CODE**
 - **ENVIRONMENT: VAX-11/780 AND VAX-8600**
 - **STAFFING: 7 PEOPLE ON EACH TEAM**

SCHEDULE*

*EFFORT LEVELS VARY

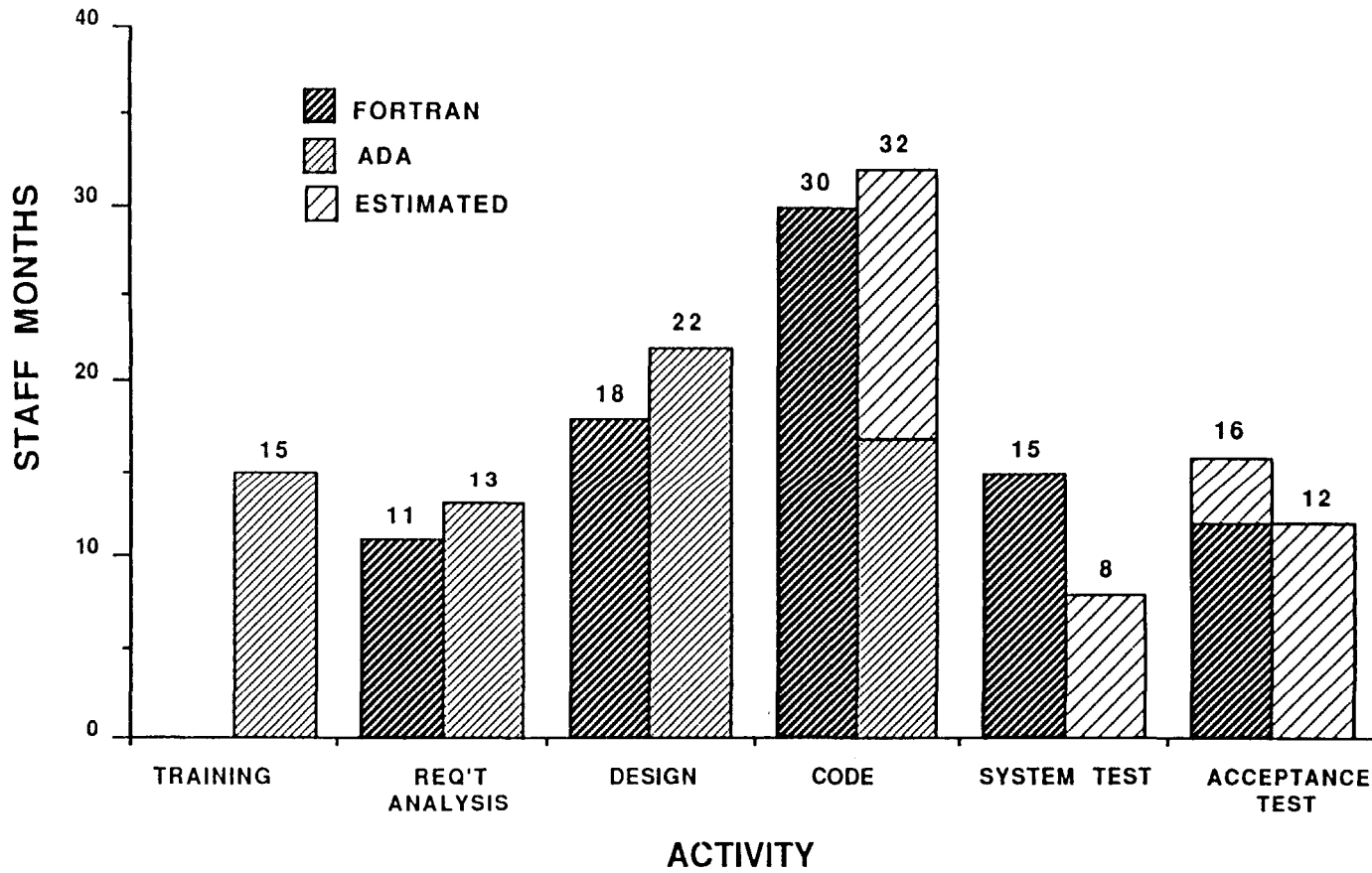


CSC COMPUTER SCIENCES CORPORATION
SYSTEM SCIENCES DIVISION

1038-AGR-(179*)

FIGURE 4

STAFF EFFORT BY ACTIVITY



B2/1/86

FIGURE 5

COMPARING FORTRAN AND ADA DESIGNS

	<u>FORTRAN DESIGN</u>	<u>ADA DESIGN</u>
DESIGN DRIVERS	<ul style="list-style-type: none">● PAST DESIGNS● SCHEDULE CONSTRAINTS	<ul style="list-style-type: none">● NEW DESIGN METHODS● NEW ADA FEATURES
DESIGN ABSTRACTIONS	<ul style="list-style-type: none">● PROCEDURAL	<ul style="list-style-type: none">● OBJECT-ORIENTED● STATE MACHINE
FUNCTIONAL SIZE	<ul style="list-style-type: none">● 262 SUBROUTINES	<ul style="list-style-type: none">● 252 SUBPROGRAMS (104 PACKAGES)

1038-AGR-(179a*)

CSC COMPUTER SCIENCES CORPORATION
SYSTEM SCIENCES DIVISION

FIGURE 6

COMPARING SIMULATOR OPERATIONS

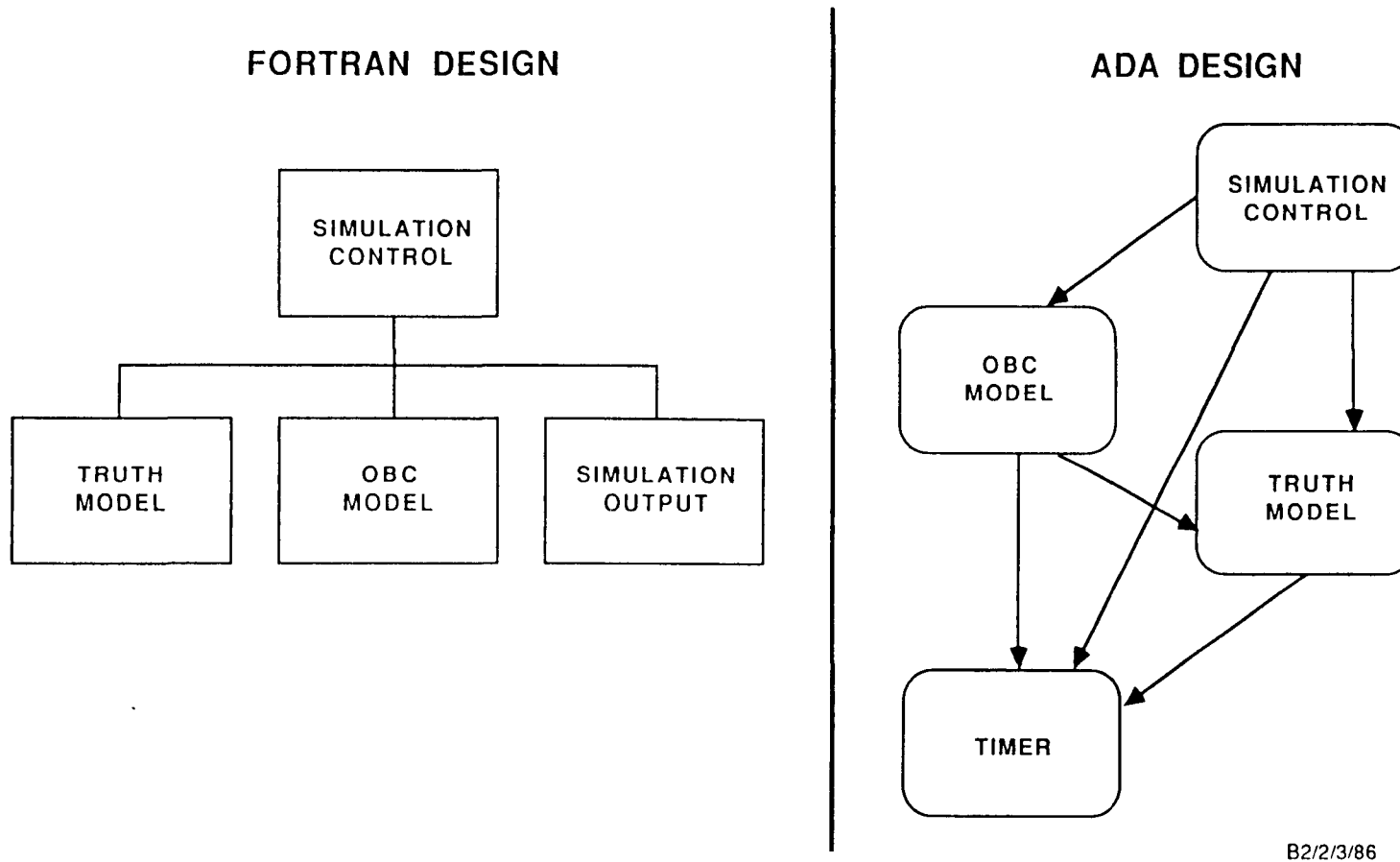


FIGURE 7

COMPARING DESIGN PRODUCTS

- **SIMILARITY OF DESIGNS AT HIGH LEVEL**
 - **REFLECTING ORIGINAL CONTROL PROBLEM**

- **DIFFERENCES NOTED IN ADA DESIGN**
 - **SENIORITY OF OBC OVER TRUTH MODEL**
 - **RESULT OF VIRTUAL MACHINE ABSTRACTION**
 - **SEPARATE TIMING OF OBC AND TRUTH MODEL**
 - **TIMING OF OBC NOT UNDER USER CONTROL**

PREREQUISITES FOR AN ADA-ORIENTED DESIGN

- **WORK DIRECTLY FROM SYSTEM REQUIREMENTS**
 - REMOVE FORTRAN LEGACY
 - ANALYZE REQUIREMENTS FROM MULTIPLE VIEWPOINTS (CSM)
- **UNDERSTAND ALTERNATIVE DESIGN ABSTRACTIONS**
 - USE PILOT PROJECTS
- **OBTAIN NECESSARY RESOURCES TO AFFORD NOT TO REUSE PAST DESIGNS**

FORTRAN-TO-ADA TRANSITION

<u>ALTERNATIVE</u>	<u>CONSEQUENCES</u>
● SOURCE LANGUAGE CONVERSION	● MAINTENANCE PROBLEMS
● ADA AFTER DESIGN	● PROCEDURAL "ADATRAN"
● ADA AFTER REQUIREMENTS ANALYSIS	● EFFECTIVE ONLY IF FORTRAN LEGACY IS REMOVED
● ADA AT PROJECT START	● BEST OPPORTUNITY TO CAST REQUIREMENTS IN A MORE LANGUAGE-NEUTRAL FORM

FIGURE 10

ADA DESIGN: MANAGEMENT ISSUES

- **"EDUCATE" HIGHER LEVEL MANAGEMENT ON DIFFERENCES WITH —**
 - **COST ESTIMATION: WHAT IS A MODULE?**
 - **PRODUCTS: WHERE ARE THE STRUCTURE CHARTS?**
 - **MILESTONES: WHEN IS DESIGN COMPLETED?**
 - **REVIEWS: WHAT IS PRESENTED AT CDR?**

1038-AGR-(179*a)

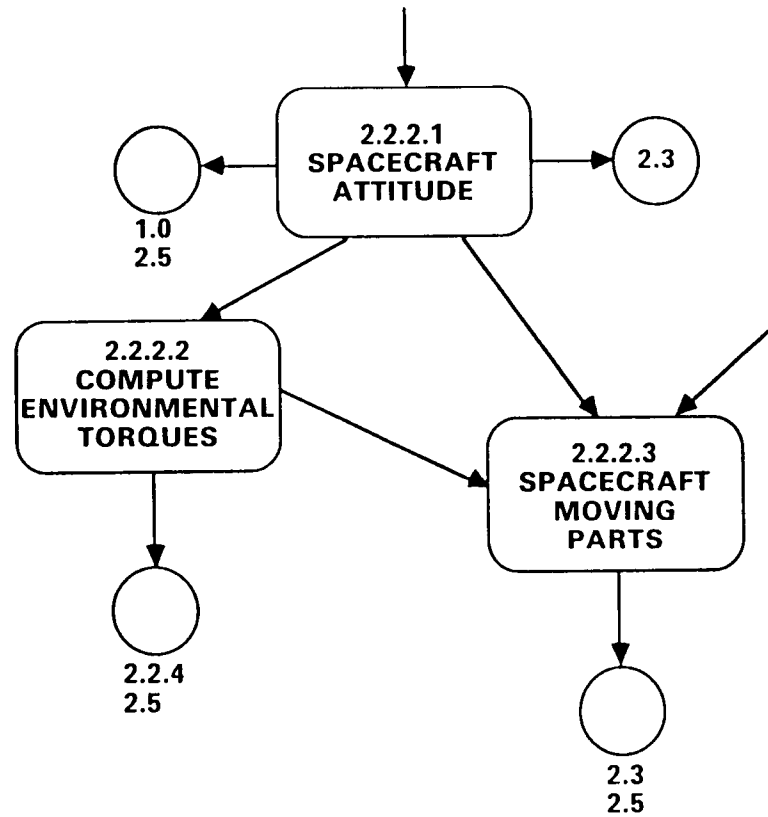
W. Agresti
CSC
21 of 25

CSC

COMPUTER SCIENCES CORPORATION
SYSTEM SCIENCES DIVISION

FIGURE 11

2.2.2 ATTITUDE DYNAMICS

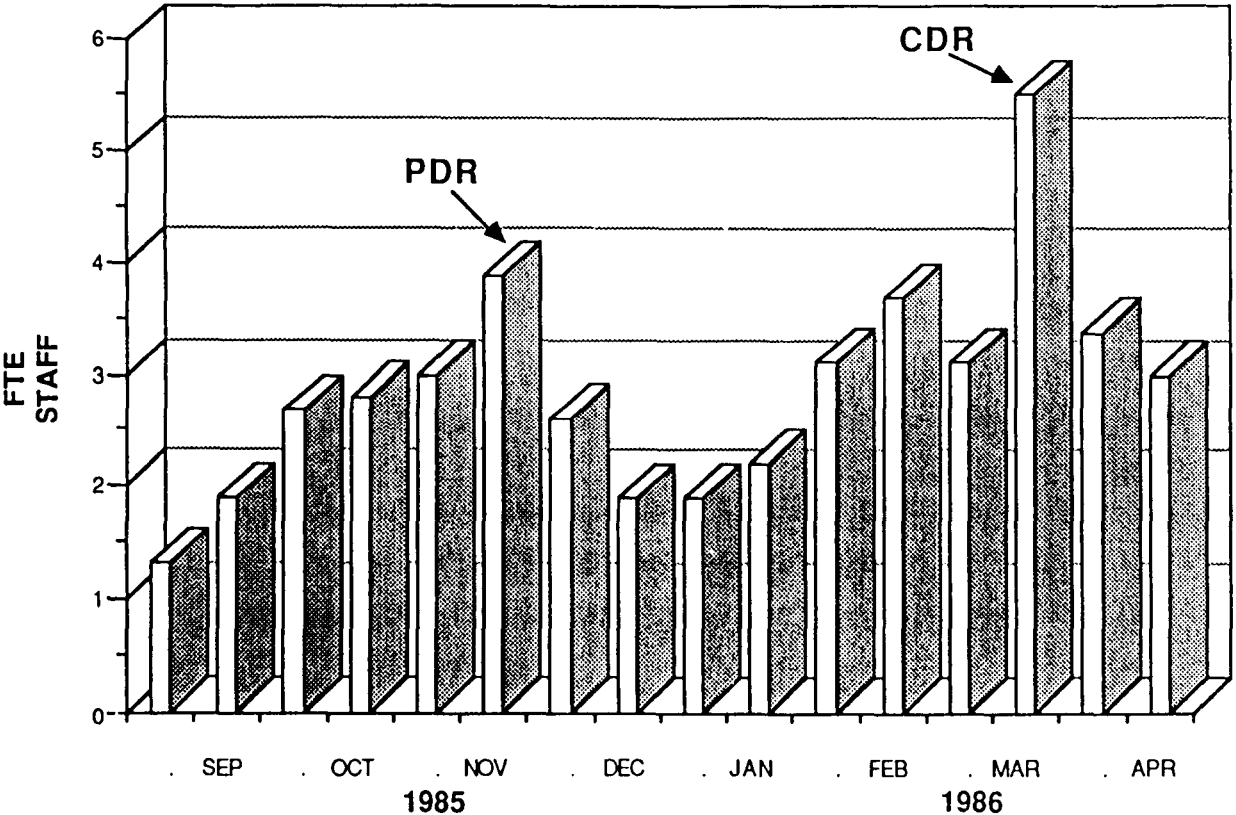


B2/2/4/86

CSC COMPUTER SCIENCES CORPORATION
SYSTEM SCIENCES DIVISION

FIGURE 12

ADA TEAM LEVEL OF EFFORT DURING DESIGN



W. Agresti
 CSC
 23 of 25

FIGURE 13

ADA STAFFING ISSUES

- **POSITIVE EXPERIENCE WITH RECENT COMPUTER SCIENCE GRADUATES DUE TO**
 - PASCAL
 - DESIGN ABSTRACTIONS
 - EXPOSURE TO SEVERAL DIFFERENT LANGUAGES

- **SMOOTH PHASE-IN TO DESIGN AND CODE ACTIVITIES**
 - PACKAGE SPECIFICATION DEFINES "DESIGN ENVELOPE"

1038-AGR-(179a*)

W. Agresti
CSC
24 of 25

CSC

COMPUTER SCIENCES CORPORATION
SYSTEM SCIENCES DIVISION

FIGURE 14

OBSERVATIONS ON ADA DESIGN

- **THE ADA TEAM PRODUCED A DIFFERENT DESIGN FOR THE GRO SIMULATOR**
- **SPECIFIC CONDITIONS ENCOURAGE ADA-ORIENTED DESIGNS**
- **ADA CAN INFLUENCE EVERY ASPECT OF DESIGN, ESPECIALLY, MANAGEMENT EXPECTATIONS**

PANEL #2

EMPIRICAL STUDIES OF SOFTWARE TECHNOLOGY

E. Soloway, Yale University
V. Shen MCC
J. Gaffney, IBM

Studying Software Documentation From A Cognitive Perspective: A Status Report

Elliot Soloway, Jeannine Pinto, Scott Fertig,
Stan Letovsky, Robin Lampert, David Littman, Ken Ewing

Yale University, Department of Computer Science
New Haven, Connecticut 06520

1. Introduction

Software documentation *should* be particularly invaluable for the maintenance programmer: since the maintenance programmer is typically not the original programmer, the maintenance programmer needs to go to some repository for key information --- the kind that that can't really be gleaned from the code itself. In practice, however, software documentation has a bad --- and probably justified --- reputation: programmers don't like to write it (it takes too much time, it isn't as much fun as designing/coding, and guidelines are often vague), or read it (it is never up to date; the only truth is in the code), and managers don't like to pay for it (documentation is the first thing to be cut when a project gets into trouble). Nonetheless, pages of documentation, best measured by the standing foot or the micro-ton, continue to be churned out. Why is there this disparity between intent and realization? What can be done about it? Just perhaps, if we take what looks like an orthogonal perspective, we might be able to identify a few gems. In fact, for 2 years now we have been studying the relationship between software documentation and maintenance from a cognitive perspective; in this brief progress report we attempt to describe the key insights we have had in this effort.

2. Basic Questions Concerning Documentation

There are three basic questions that need to be answered in developing effective documentation:

- *WHAT*: What should the content of documentation be? Should each variable be described? Should control/data flow be described?
- *HOW*: What should the format of documentation be? Should English be used? Should flowcharts? Should PDL?
- *WHEN*: When should the programmer see a particular piece of documentation? By and large, the answer to a programmer's question is probably "in there somewhere," but the cost of searching volumes and volumes and volumes is a demoralizing, time consuming and often fruitless endeavor. The programmer needs to see a particular piece of documentation *when* it is needed --- whatever that might mean.

Clearly, these three issues are intertwined; however, they can profitably be separated out, and studied more or less independently. In particular, our first studies (Section 4) have focused on the *WHAT* question. On the basis of a better understanding of the *WHAT* of documentation, we then initiated a study focusing on the *WHEN* question (Section 5).

* * * * *

The research reported in this paper was supported by the Jet Propulsion Laboratory, California Institute of Technology under contract with the National Aeronautics and Space Administration.

3. Methodology

As we have done with other aspects of the software process, we have taken a cognitive perspective in this research: our approach is to try and understand how programmers *really* use documentation. There are two aphorisms that guide this work:

- "You can observe a lot by just watching." *Yogi Berra*. We need to see what programmers really do in specific situations. Arm chair prognostication hasn't done a lot for software documentation to date. Thus, we video-taped professional programmers doing a maintenance task; we asked them to talk-aloud --- tell us what you are thinking about --- as they went about the task. The talking-aloud methodology is one that is being more widely used in *naturalistic* problem solving situations. Such verbal protocol data provide a window into the cognitive processing that is going on in the head of the programmer. Reaction time studies, the forte of psychologists, are fine for the laboratory when the task being carried out is very focused. However, what would it mean to wire a programmer up to a reaction time meter? What would millisecond differences mean? Also, it is not clear that statistically-oriented studies are useful at this stage of research. That is, there is a standard distinction made in psychological research between:
 - *Theory-testing research*: where one tests hypotheses using statistical methods, and
 - *Theory-building research*: where one tries to simply develop hypotheses.

It is clear to us that we are in the theory-building mode: studying *complex* problem solving behavior is a recent development in cognitive psychology, and there is precious little known. Thus, numbers are not really all that useful here: trends and qualitative remarks are the measurement tools.

- "Little by little grow the bananas." *Judy Soloway*. One typically doesn't learn to ride a bicycle by jumping onto a 10-speed bike; rather, training wheels are a good idea. Similarly, we haven't jumped into studying a 50,000 line program immediately; we need to learn the business first in a somewhat more restricted setting. Once we have some confidence in our methods -- and in ourselves -- then we can jump in over our heads.

In what follows, then, we will outline the two studies we have conducted at JPL, and some of the key observations made in each.

4. Study Area 1: Examining The WHAT Question

There have been two phases in this study area: first, we attempted to understand the baseline -- what do programmers do. Next, we carried out a "manipulation:" can we change the documentation and affect performance; the data from this phase is still under analysis.

4.1. Phase I: What Do Programmers Do With Documentation?

Our initial study was to give professional programmers a program plus some documentation that followed professional guidelines, and ask them to make an enhancement. The program was a 250-line, 14 subroutine, Fortran program that managed a small database of personnel records (containing name, address, etc.), and allowed a user to CREATE, SHOW, UPDATE, DELETE a record. The enhancement was to add a RESTORE command, that would undelete records deleted during the current session with the program.

In studying the video-taped interviews with over 20 professional programmers, two major observations can be made:

- *WHAT programmers needed to know:* The information that was key to making the enhancement was information about the *causal interaction of non-contiguous, non-local, pieces of code*. For example, in order to make a correct patch, programmers needed to realize how the database search routines interacted with the delete/restore commands: the search routines return ONLY active records, since all other previous commands wanted only active records, whereas the restore routine would want the search routines to return a deleted record! This sort of causal information is typically not identified in the code, and it is typically not identified in accompanying paper documentation.
- *Strategies for comprehending the program:* We can coarsely describe two different strategies used by programmers as they went about understanding the program and documentation:
 - *Systematic Strategy:* When using this strategy, programmers started at the beginning of the program and documentation and traced out the flow of the entire program, using various forms of *simulation* (e.g, symbolic, actually plugging in values). Once they understood the program to their own satisfaction, they attempted the patch.
 - *As-needed Strategy:* When using this strategy, programmers used their anticipated patch to guide their study of the program; if additional information appeared to be necessary, then they would attempt to backtrack and find the relevant information.

Predictably, there was an interaction between the strategy employed by the subjects, their acquisition of an understanding of the causal interactions in the program, and a correct enhancement:

- Adopting a systematic strategy invariably led to a correct enhancement.
- Adopting an as-needed strategy lead to mixed results: about half the subjects who adopted this strategy failed to come to understand the causal interactions and thus didn't develop a correct patch.

In otherwords, by not getting a global, more or less complete sense of the program, one is not likely to "back into" global information when one is by and large focused on a local portion of code.

The above observations amount to this: the better one understands a program, the more likely one will be able to change it correctly. Not too surprising, frankly. However, our work has focused on *what it means* to understand a program: what are the key pieces of information that will facilitate a correct view of a program, and in turn, facilitate a correct modification of the program, e.g., an understanding of the causal interactions that obtain between non-local pieces of code. While a systematic strategy provides an essentially fool proof scheme for uncovering such key facts, this strategy becomes impractical on real programs: by and large, it just isn't worth the effort to understand a 100,000 line program in order to create a 15 line patch. (It may not even be possible to truly acquire a global understanding of a very large program!) Thus, almost by definition, programmers will have to adopt an as-needed strategy when they approach a real software maintenance task.

4.2. Phase II: Can We Help Programmers With Better Documentation?

What, then, are the implications for documentation if programmers will be adopting an as-needed strategy? Based on the above research, we can make the following claim: we need to *explicitly* provide programmers information about the non-local, causal interactions in the program. That is, we need take information that is typically gained by simulating the program and allow the programmer access to it, without the programmer actually needing to carry out the simulation. Put still another way: programs are static representations of dynamic entities; we need to abstract out some of the dynamic properties and express them as static descriptions. No mean feat, that.

We have recently carried out another video-taped study with programmers at JPL who were asked to make the same enhancement to the same program described in Section 4.1. However, this time the documentation *did* explicitly contain information about the non-local, causal interactions. For example, in Figure 4-1 we illustrate how we documented this type of information. Notice in particular, that the variable *iptr* is key: it ties the search routines and the delete routine together. We explicitly tell the programmer that there is an interaction taking place. We felt that this explicitness would be a "sign" to the programmer adopting an as-needed strategy: the programmer could use this information in backing into the information needed to make the correct patch.

The results of our new documentation are mixed: in comparison to subjects using the initial documentation, without the explicit information about the non-local, causal interactions, more subjects who use our documentation and who employ an as-needed strategy do make a correct patch; however, that difference is not statistically significant. However, what we are beginning to see is an interaction between a programmer's background and his need/ability to use our new documentation: apparently, subjects who know about database programs, and thus who have some familiarity with the coding tricks used in creating such programs, don't seem to need the "explicit documentation," while those with a less rich background in database programs do seem to benefit most from our new documentation. In otherwords, individual differences are beginning to become more and more apparent. Thus, while analysis of the data is still taking place, we nonetheless feel that our documentation has tapped into something important; now, we are just trying to tease out *exactly* when and why that documentation seems to be useful.

```

SUBROUTINE srch(dbase, ifinal, iptr, name, oldnme, ioldp)
CHARACTER*60 dbase(200,7), name, oldnme
IF (oldnme .EQ. name) THEN
    iptr = ioldp
    RETURN
ELSE
    CALL srch2(dbase, ifinal, iptr, name)
ENDIF
RETURN
END

```

SUBROUTINE NAME: srch(dbase, ifinal, iptr, name, oldnme, ioldp)

PURPOSE: To find the location of the record in the data base; if the previously accessed record is the one being searched for, then that record's index is returned without any search

CALLED-BY: GETNME

CALLS: SRCH2

IMPORTANT INTERACTIONS WITH OTHER ROUTINES:

The name of the most recently accessed record is stored in *oldnme* by the main routine. SRCH uses the value of *oldnme* to optimize the process of locating a record in *dbase*. If *oldnme* = *name* then no search of the database is required; the record pointer, *iptr*, is merely set equal to the pointer to the last record accessed, *ioldp*, and SRCH returns that value. The value of *iptr* is then used by a transaction handler (e.g. DELETE) to perform the user's request on the desired record or to signal an error.

```

SUBROUTINE delete(dbase, iptr, ichnge, name)
CHARACTER*60 dbase(200,7), name
IF (iptr.EQ.0) THEN
    CALL error1(name, 'd')
    RETURN
ENDIF
WRITE(6,*) 'DELETING RECORD FOR:', dbase(iptr, 1)
dbase(iptr, 7) = 'deleted'
ichnge = ichnge + 1
iptr = 0
RETURN
END

```

SUBROUTINE NAME: delete(dbase, iptr, ichnge, name)

PURPOSE: To change the status of a record from 'active' to 'deleted', causing the record to be ignored on a subsequent search.

CALLED-BY: PDB

CALLS: ERROR1

IMPORTANT INTERACTIONS WITH OTHER ROUTINES:

Iptr will be 0 if SRCH and SRCH2 were unable to find an 'active' record with the name specified by the user.

After a record's status field is changed to 'deleted', a user will not be able to SHOW, UPDATE, or DELETE that record, since SRCH2 only returns 'active' records.

All of *dbase* is written to disk just prior to exiting the program if and only if *ichnge* is greater than zero. Since *ichnge* is incremented, the database file, *db*, will be updated at the end of the session by the routine PUTDB.

Figure 4-1: EXAMPLE OF DOCUMENTING NON-LOCAL, CAUSAL INTERACTIONS

5. Study Area 2: Examining The WHEN Question

When a programmer is confronted by the need to make a patch to a *real* program, what information does he need -- and when? To examine this question, we borrowed a real program currently in use at JPL; to protect the names of the innocent, let us call this program the XYZ program. XYZ is approximately 50,000 lines of code and has documentation that stands about 2 feet high. We studied this program for days and developed what we thought was a simple enhancement to the program. We then gave the stack of documentation to two programmers (working independently) at JPL, and asked them to find where in the program the enhancement needed to be made. We video-taped the efforts of these programmers. (We need to point out that XYZ has been used at JPL for some time, and that there were guidelines for developing the documentation. Moreover, the documentation was meant to be useful.)

What happened? Our two subjects each spent close to 2 hours each reading documentation, trying to understand what the system did. That was the first basic question: what are the goals of this program? What are the goals of the various modules in the program? They needed this information *first*; they needed to get a coarse, global sense of what the program was intended to do. The goal information *was* in the documentation; but, it was interwoven with many low-level details that hid the main points and confused the programmer. We repeat, the documentation we used in this study was the real documentation for a real program.

We then rewrote the first manual in the documentation set to reflect the programmer's desire for the overall goals of the programs and the goals of the various main modules. We rewrote the table of contents to this manual to reflect the new content. Our intention was to make this first, overview manual "predicable." That is, our intention was to provide the key goal information upfront, when the subjects apparently needed it. The result? We ran two subjects (again, working independently) on our new documentation: it took each about 20 minutes to identify the module where the patch needed to be made.

We are the first ones to say that this study has many problems with it! Is the documentation for XYZ representative? or is it a worst-case? Were our subjects representative? or were they, too, worst-case? The fact still remains: we were able to reduce time on task from 2 hours to 20 minutes --- a big difference with a small number of subjects. Again, we seem to have tapped into something important; now we need to carry out more crafted studies in order to tease out exactly what is going on.

6. Concluding Remarks

We have been trying to breathe some life into program documentation: our intent is to develop specific prescriptions for improving documentation so as to make it more useful. We have approached the problem from a cognitive perspective: we need to understand how and why documentation is used. From a practical viewpoint, we have identified *what* some key information that should be included in documentation, and we have identified *when* some key information should be made available. From a theoretical viewpoint, we have quite frankly raised more questions than we have answered. But that's good, too. Documentation is like a gold mine; the riches are there; but, not surprisingly, the difficulty of ferreting them out is directly proportional to their value.

Empirical Research on the Design Process: The Field Study

Vincent Shen, Herb Krasner, Neil Iscoe, & Bill Curtis
MCC Software Technology Program
Austin, Texas

Introduction

MCC's Software Technology Program is charged with providing its shareholders with technology that radically improves the productivity and quality of developing large, complex systems. The program has focused on developing technology for aiding the requirements and design process (the "upstream"). As a part of this technology development program, we have a team of 5 research scientists working full-time on empirical studies of the design process. Our presentation will describe one of the studies we have conducted, the initial observations drawn from it, and the implications of these observations for modeling the software design process and guiding the development of design technology.

We are conducting studies of the design process at each of three levels. At the individual level, we are studying design as a cognitive problem-solving process. At the team level, we are studying design as a social interaction process whose goal is to communicate and negotiate mental representations of the behavior of the application system and of the computational structure required to implement it. At the organizational level, we are studying design as the integration of numerous processes (managerial, technical, customer interaction, etc.) that emerge in large organizational endeavors. This presentation will describe initial results from our studies at the organizational level.

Previous Studies of Large Systems Development

Since the mid-1970s, many corporations have been collecting data on the productivity and quality of their large system development projects. These data are often used to determine the major factors affecting productivity and quality in a particular programming environment. These factors are frequently used in driving project cost models. Four of the more important studies were:

1. **IBM Federal Systems Division [WALS77].** Walston and Felix of IBM analyzed 60 reports from project managers. They were interested in identifying the primary factors that affected overall productivity. Of the eight factors identified, two related to the difficulty of interacting with the customer, while

four involved personnel experience and qualifications. The other factor related to the amount of documentation required.

2. **TRW Defence and Space Group [BOEH81].** Boehm collected 63 sets of project information from managers and fellow researchers. The most important factor related to the capability of the personnel assigned to the project. The next three factors involved product complexity, reliability requirements, and timing constraints. Boehm used the data to develop his COCOMO cost estimation models.
3. **ITT Programming Technology Center [VOSB84].** Vosburgh, Curtis, Wolverton, and four others of ITT studied 44 reports from project managers. They found that factors under management control accounted for 1/3 of the variation in productivity. However, factors that were not under management control, those related to the business area, also accounted for 1/3 of the productivity variation. No one factor was sufficient to guarantee improved productivity.
4. **IBM Santa Teresa Laboratory [SHEN85].** Shen, Yu, and Thebaut of Purdue University and Paulsen of IBM studied the historical databases of four projects. They analyzed about 1,400 sets of module-level data and identified several factors that could serve as predictors for error-prone modules.

Although these studies identified important productivity and quality factors, they generally did not elaborate the process through which these factors exerted their influence on the project. The field study reported here was similar to these other studies in the variety of projects involved, but is different in its emphasis on describing how factors exert their influence during the design process. This difference was achieved by collecting a different form of data than had been collected in the studies cited above.

Field Study Design

The field study was designed to gather information from MCC shareholders for input as requirements to our large systems design environment. We have conducted a set of interviews at field sites with members of the design team on 19 large projects. Our purpose was to identify the primary leverage points (i.e., problem areas) in productivity and quality to attack in our research on design environments.

The application areas of the projects studied included avionics, telephony, operating systems, and factory automation; and included projects that contained real-time, distributed, and/or embedded components. These projects were drawn from nine multi-billion dollar corporations in such businesses as defense contracting, computer manufacturing, commercial products manufacturing, and telecommunications. On each project we interviewed the senior systems engineer, the senior software designers, and the

project manager. Often we were also able to talk with someone from testing or quality assurance, a customer, and a division vice president. The interviews were structured with a set of questions designed for each type of position interviewed. Yet the structure was flexible, and was designed to get the interviewee to expound on specific issues or problems faced in designing their particular application. Interviews typically lasted one hour, but interviews with senior systems engineers frequently ran much longer.

During these interviews we focused on: 1) how requirements came to be known and how they were changed, 2) how design decisions were reached and how often and through what mechanism they were changed, 3) what, if any, communication was conducted with customers, users, contracting officers, etc., 4) how design information was communicated among the members of the project, 5) how people were organized at various points during the project, 6) what types of tools were used, and what types should have been available, but weren't, 7) the nature of the hardest design problems faced and how were they tackled, 8) how design was represented at various stages, and 9) how software development was integrated with hardware development.

Observations on Individual Talent

We have identified two types of rare individual talent that frequently develop on projects. The first is that possessed by superdesigners (typically a senior systems engineer). Their talent comes only with a deep understanding of the application domain (avionics, telephony, etc.). It is manifest in their mapping between the behavior exhibited in the application system (how the jet flies and delivers ordinance) and how the computational structure (the software) controls it. Thus, they are mapping between several domains of expertise, only one of which is trained in computer science (the computational domain). The maintenance and communication across project members of a consistent representation scheme for this mapping is the crucial element in managing the development of a system design, and this function is typically performed by the senior systems engineer, not the project manager. This skill will be extremely difficult to automate, since it requires mapping across multiple domains of expertise, a capability not performed well by current expert systems technology.

The second type of rare individual talent is the systems diagnostician. This is a skill that emerges during the development of the system and is possessed by the individual who is sought out for answers on why something doesn't work. This skill is more amenable to automation than that of mapping application behavior into computational structures, since the information required for developing diagnostic expertise is available in the structure of the software.

Because of the crucial role that application knowledge plays in the design of a system, those few who possess it at a level of expertise have tremendous individual leverage in directing the course of a project. We have seen numerous instances where a large design team was taken over by a few individuals who controlled the rest of the design process. That is, if a coalition forms among a few individuals who share a common model of the behavior of the application or of the computational structure to

implement it, they can exercise the power of a majority because other team members are unable to gain consensus for their ideas.

Observations on the Design Process

There are several stages of the design process that are never explicitly represented in software process models, but which are absolutely crucial to the success of the project. These danger of leaving these stages out of a process model is that insufficient time is factored into the project schedule for these activities, and they are therefore frequently cut short to the detriment of productivity an quality later in the project. The first of these stages involves the exploration and selection of a format for abstracting and representing the important features of the application and its behavior. The next stage is the communication and coordination of a common model of the application behavior and computational structure for controlling it among the members of the project. Finally, there is the negotiating and reprioritizing of system features when the underlying tradeoffs in the design are understood.

The stages described above represent a learning process for most of the members of the project. The extent of this learning process will differ by the newness of the technology or the application involved in the project. However, projects are usually planned as if the learning process is a constant, and is small. The coordination of common models and understandings among project members is usually assumed to occur quickly and completely. However, learning is a large consumer of project time in the early phases. When it occurs incompletely it leads to design and interface errors. The representation of these activities as stages in design allows them to be estimated more accurately and planned for purposely.

The amount of time required to learn the structure and behavior expected of a new application is sufficiently great that we often found the best prototype to be a failed project. A prototype which does not exercise the full range of application behavior and function will not provide sufficient learning to design the system correctly. It was through trying to develop the full system and failing that the underlying structure of the application domain was sufficiently teased out to support a successful redesign. This is the "phoenix syndrome" and it is a function of the amount of design experience required to understand the underlying structure of a new application area.

The following presentation contains an elaboration of one set of our results: The Top 5 Problems in Large Development Projects.

References

- [BOEH81] Boehm, B. W. Software Engineering Economics. Englewood Cliffs, NJ: Prentice-Hall, 1981.
- [SHEN85] Shen, V. Y., T. J. Yu, S. M. Thebaut, and L. R. Paulsen. Identifying error-prone software – an empirical study. IEEE Transactions on Software Engineering, 1985, 11 (4), 317–324.
- [VOSB84] Vosburgh, J., B. Curtis, R. Wolverton, B. Albert, H. Malec, S. Hoben, and Y. Liu. Productivity factors and programming environments. Proceedings of the Seventh International Conference on Software Engineering. Washington, DC: IEEE Computer Society, 1984, 143–152.
- [WALS77] Walston, C. E. and C. P. Felix. A method of programming measurement and estimation. IBM Systems Journal, 1977, 16 (1), 54–73.

THE VIEWGRAPH MATERIALS
FOR THE
V. SHEN PRESENTATION FOLLOW



**EMPIRICAL RESEARCH ON THE DESIGN PROCESS:
THE FIELD STUDY**

Vincent Shen, Herb Krasner, Neil Iscoe, & Bill Curtis

MCC Software Technology Program

MCC SOFTWARE TECHNOLOGY PROGRAM

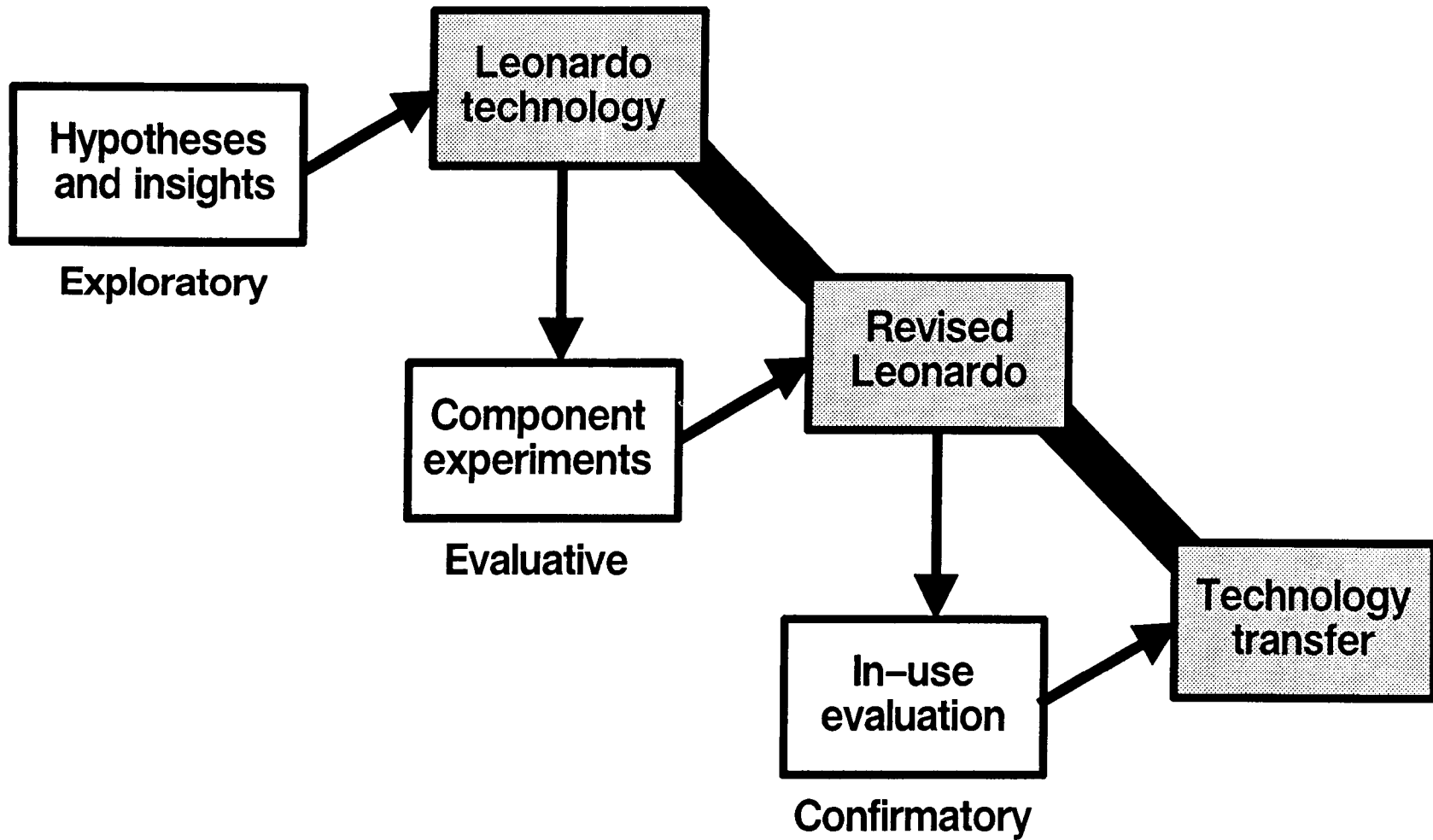
MISSION

Create new software technology that leads to extraordinary increases in the productivity and quality of developing large, complex systems

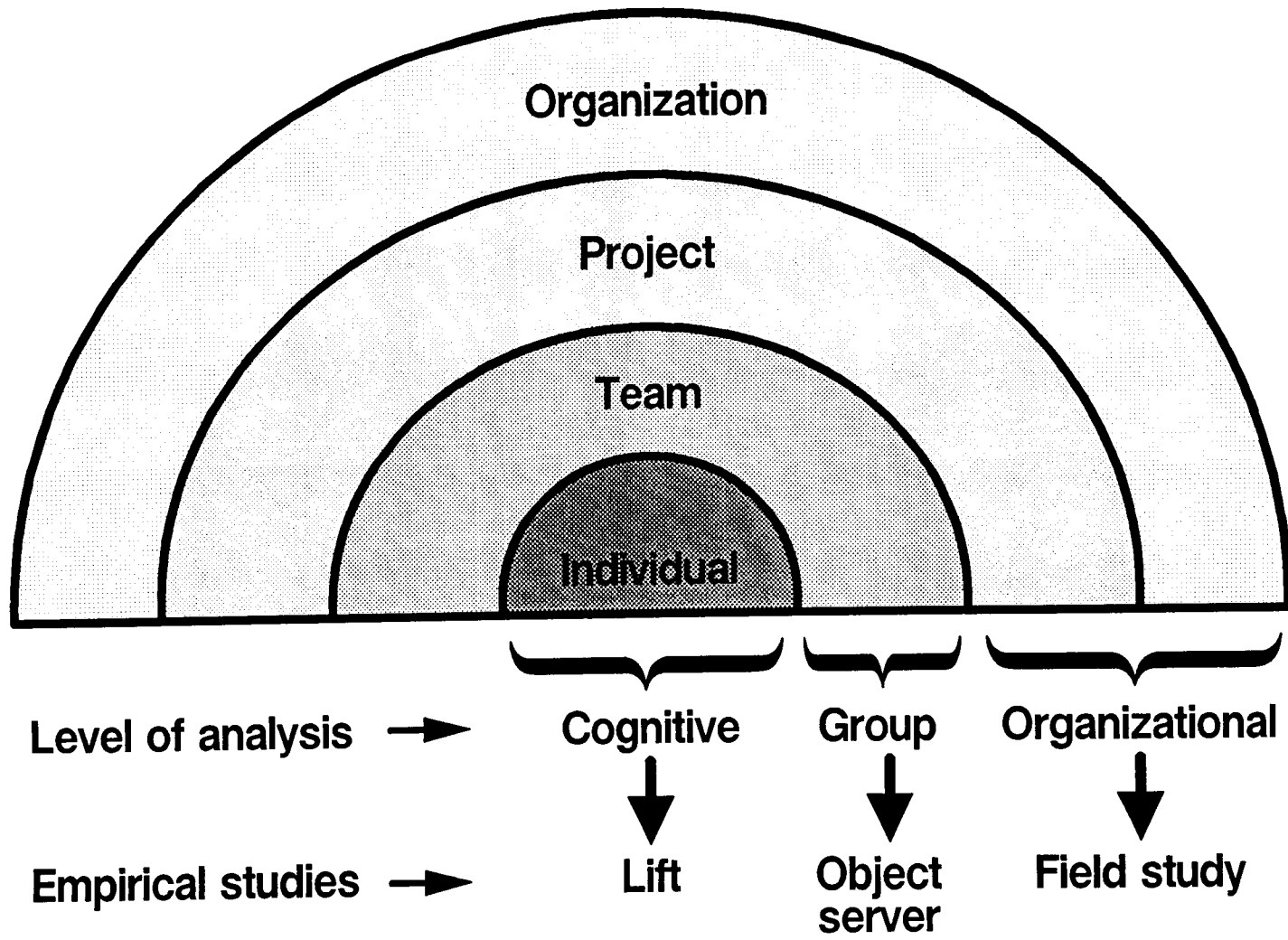
RESEARCH TO PRODUCE A DESIGN ENVIRONMENT: LEONARDO

- The greatest leverage is in requirements and design
- Design technology has lagged implementation technology
- Computers can aid the design of complex software systems
- Existing implementation research is adequate

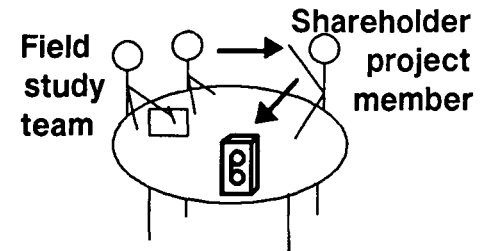
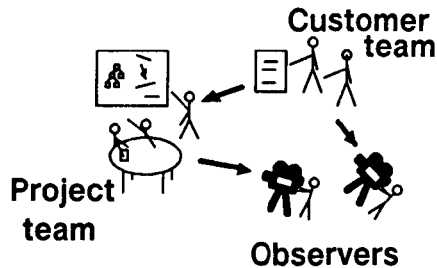
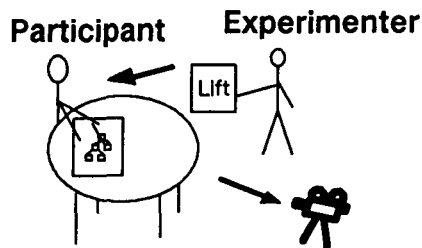
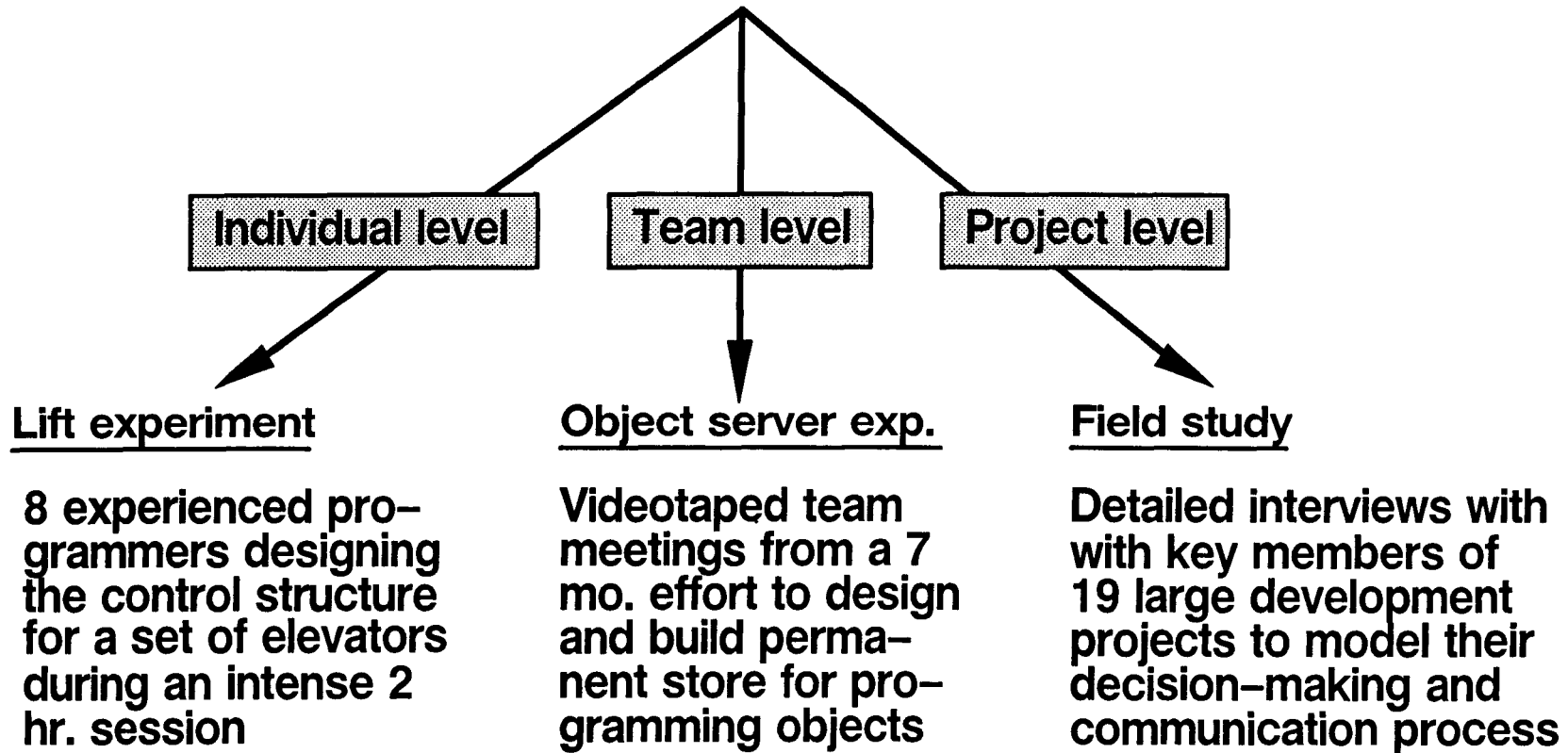
THE ROLE OF EMPIRICAL RESEARCH IN STP



PROCESSES AFFECTING LARGE SYSTEMS DESIGN



EMPIRICAL RESEARCH ON THE DESIGN PROCESS



FIELD STUDY

PURPOSE

To model current projects

- Decision-making processes
- Communication processes

To identify:

- Leonardo requirements
- Key leverage points
- Technology transfer issues
- Design problems

INTERVIEWEES

System engineer

Senior software designers

Project manager

Division general manager

Testing / QA

Customer

LARGE PROJECTS SURVEYED

Proj.	Stage	Size	Nature			Government Sponsored	Primary Application
			Real-Time	Distributed	Embedded		
1	Maint.	400K					Trans. processing
2	Mid	500K	✓	✓			Telephony
3	Mid	300K	✓				Trans. processing
4	Late	55K	✓				Trans. processing
5	Maint.	725K					Operating syst.
6	Late	350K					Compiler
7	Late						Compiler
8	Maint.	250K					Part of operating syst.
9	Term.	0					Software support
10	Mid	70K				✓	CAD (hardware)
11	Maint.		✓			✓	Command & control
12	Late	130K					CAD (software)
13	Late	24K	✓		✓		Radio control
14	Late		✓	✓			Telephony
15	Maint.	50M	✓	✓	✓	✓	Command & control
16	Early	100M			✓	✓	Command & control
17	Mid	50K	✓	✓	✓		Process control
18	Mid	150K	✓		✓	✓	Command & control
19	Late	50K	✓				Part of operating syst.

THE TOP 5 PROBLEMS IN LARGE DEVELOPMENT PROJECTS


- 1. Personnel**
- 2. Communication and coordination**
- 3. Dealing with uncertainty and change**
- 4. Design representation and analysis**
- 5. Technology transfer**

1. Personnel –

To build an effective team

- **Application specialist**
- **Conceptualizer**
- **Boundary spanner**
- **Gate keeper**
- **Implementor**
- **Diagnostician**
- **Feature manager**

Need support for the *informal* network –e.g., role identification, communication break-down detection, etc.



2. Communication and coordination – To know who, what, when, where, why

- **Information overload**
- **Information delay**
- **Information deprivation**

Need intelligent filters and active probes that support quick access to relevant information.

3. Dealing with uncertainty and change –

To reduce wasted effort

- **Uncertainty (floating issues)**
 - **Missing information**
 - **Conflicting information**
- **Change (new information)**
 - **Goals**
 - **Policy and procedures**
 - **People**
 - **Standards**
 - **Technology**
 - **...**

**Need rigorous methods to make the floating issues, changes,
and their impacts explicit.**

4. Design representation and analysis – To record the status and rationale.

- **Representation media**
 - Text
 - Graphics
 - Prototypes
- **Analysis**
 - Analytical models
 - Simulation models

Need mechanisms to compare the behavior of the design with the fuzzy customer's needs.

5. Technology transfer –

To use the best available tools

- **Awareness**
- **Assessment**
- **Culture resistance**

Need to conduct experiments to test learning models of technology transfer.

Conclusions

Research in large complex systems engineering should be focused on meeting the following needs:

1. Support for the *informal* network – e.g., role identification, communication break-down detection, etc.
2. Intelligent filters and active probes that support quick access to relevant information.
3. Rigorous methods to make the specification of floating issues and changes, and their impacts explicit.
4. Mechanisms to compare the behavior of the design with the fuzzy customer's needs.
5. Experiments to test learning models of technology transfer.

A Quantitative Analysis of the Impact
of Modern Software Engineering Techniques
on Software Quality and Development Productivity

J. E. Gaffney, Jr.
T. M. Drabant
W. D. Ceely

IBM, Federal Systems Division
Gaithersburg, Maryland

ABSTRACT

The IBM Federal Systems Division has made a considerable capital investment in its 'software business' through the institutionalization of an integrated set of modern software engineering techniques and technologies. This paper addresses the question, "Has this paid off?" It does this by considering in a quantitative manner, the value of the use of modern software engineering techniques/technologies in terms of their effects on software development productivity and quality. Specifically, the paper presents a quantitative analysis relating the degree of use of several software engineering approaches to these topics. This degree is presented as a single number, the 'software process technology index.'

OVERVIEW

Realizing that technology is the key to attaining consistency in the software development process in terms of higher productivity and increasing levels of quality in the resultant software product, management of the IBM, Federal Systems Division in the late '70's instituted the development and codification of a set of practices and standards and a formal education program aimed at institutionalizing their use in all of FSD's software development organizations. Some of the practices thus presented in an integrated fashion to FSD technical and management personnel (indeed, upper level software management people preceded technical people in the courses) were based in part on extant approaches,¹ such as the design and code inspections conceived by Mike Fagan, of IBM. Other aspects of the body of software technology presented were new, such as the language (PDL or Process Design Language) to be employed to formally record designs.² Indeed, the education program emphasized the use of formal software process structure with defined activities and resultant products produced by each.³ The FSD software education program was the precursor to that of the IBM Software Engineering Institute, a principal component organizational element of the IBM Corporate Technical Institutes.

The 'software process technology index' presented here is a measure of the degree of use of the body of software technologies. By relating this index to software development productivity and quality, we can obtain an assessment of the degree to which this investment has 'paid off.' Our data suggests that higher levels of development productivity and lower levels of error density (upon initial shipment of the software) are associated with higher levels of software technology having been incorporated into the software development process.

The software development process technology index is a number in the range 0-100. It has been evaluated for a variety of large FSD projects totaling more than 3.8M source lines of code. These software efforts took place at two current and one former FSD locations; Gaithersburg, Maryland and Houston, Texas in the first instance and Wayland, Massachusetts, in the second. The 'software process technology index' number signifies the degree of application of the thirteen process and two education attributes presented in Table 1.

Table 1. Software Process Technology Index Attributes

ATTRIBUTES

1. INSPECTIONS
2. STRUCTURED PROGRAMMING
3. STRUCTURED DESIGN LANGUAGE
4. FUNCTION MODEL
5. STATE MACHINE MODEL
6. NETWORK MODEL
7. STRUCTURED SPECIFICATION LANGUAGE
8. UNIT TESTING
9. DEVELOPMENT INTEGRATION TESTING
10. FUNCTION TESTING
11. SYSTEMS TESTING
12. PERFORMANCE AND LIMIT TESTING
13. USER TESTING
14. MANAGEMENT TECHNOLOGY EDUCATION
15. NON-MANAGEMENT TECHNOLOGY EDUCATION

SCORING

1. SCORE EACH ATTRIBUTE 0-16 POINTS
2. INDEX = (SUM OF THE ATTRIBUTE SCORES) x (100/240)

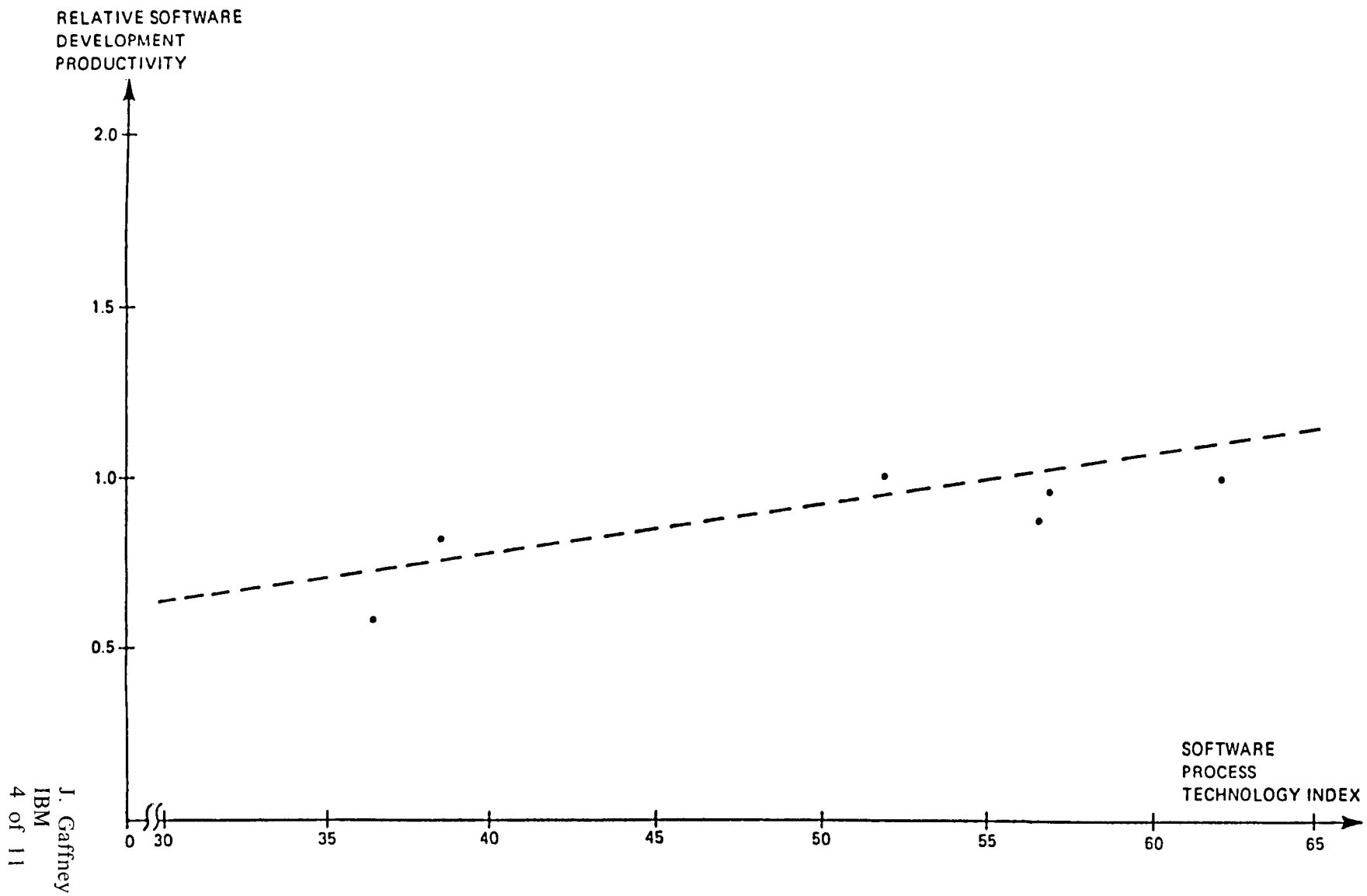


FIGURE 1 RELATIVE SOFTWARE DEVELOPMENT PRODUCTIVITY VS. SOFTWARE PROCESS TECHNOLOGY INDEX

ESTIMATED
RELATIVE NUMBER
OF LATENT ERRORS

2.0

1.5

1.0

0.5

0

30

35

40

45

50

55

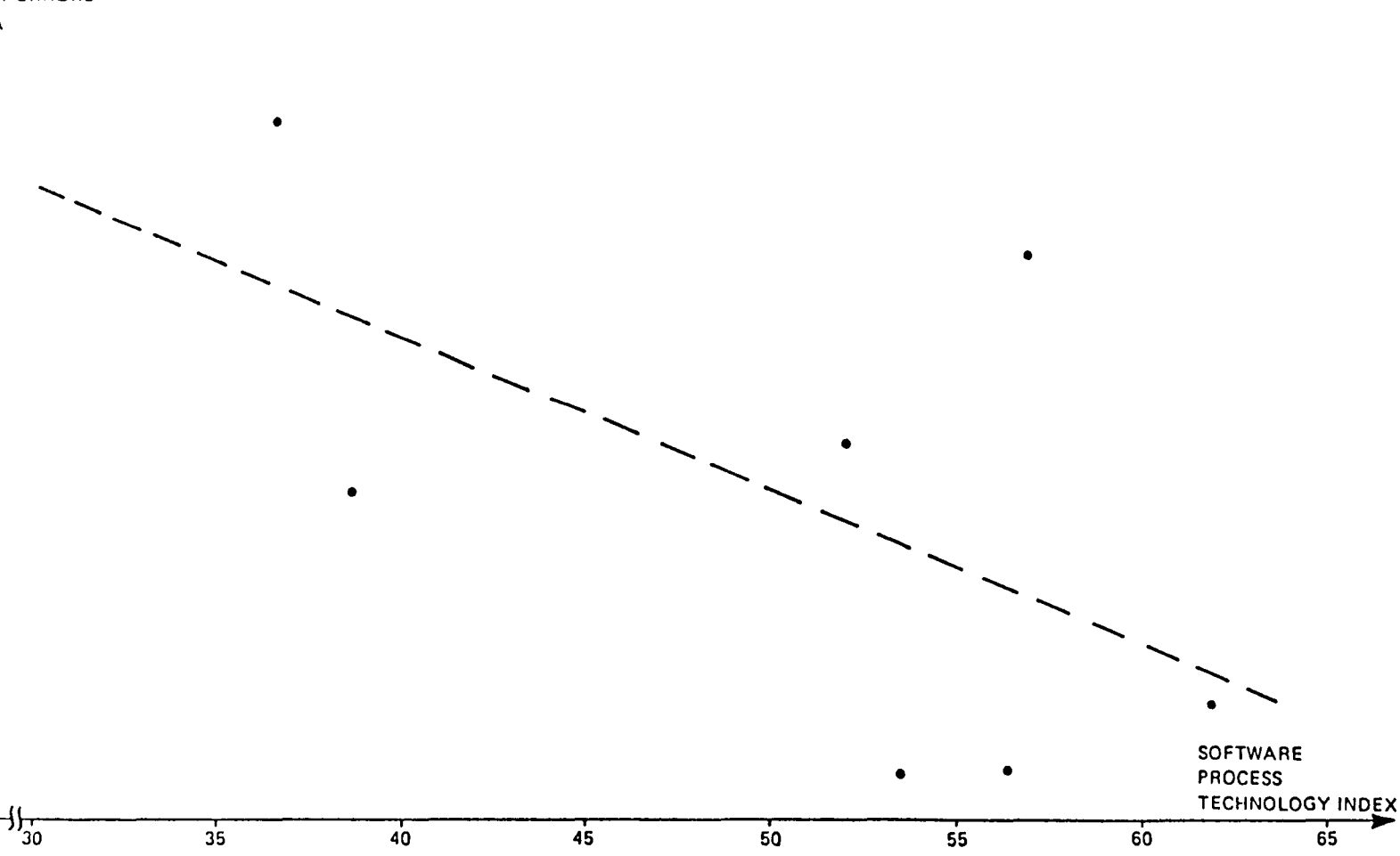
60

65

SOFTWARE
PROCESS
TECHNOLOGY INDEX

J. Gaffney
IBM
5 of 11

FIGURE 2 RELATIVE LATENT ERROR CONTENT VS. SOFTWARE TECHNOLOGY INDEX



The 'software process technology index' is determined by interviewing the software development manager, and/or appropriate members of the software development team, about the degree to which each of the fifteen attributes apply to their project. Each of the attributes, then is given a score in the range 0-16. The total possible score is 240, therefore. The actual score is normalized to be in the range of 0-100 by

multiplying the total actual score by $0.4167 = \frac{100}{240}$.

Thus, the total score may be considered to be a percentage of application of the attributes of modern software technology as dimensioned by the attribute list given in Table 1. Each of the attributes has four sub-attributes which are evaluated in the development of an 'attribute score.' For example, attribute number 5, "State Machine Model," has four components: category of product covered, amount of product covered, degree of technological rigor (with which the model is used), and degree of enforcement of the technology. Each of the other attributes has sub-attributes, correspondingly appropriate to the aspect of the software development process with which it deals. More generally, one may think of each of the attribute scores to be a quantification of the degree of applicability of that item to the development process indicative of the degree of technological rigor and the degree of enforcement used to develop a particular element of software.

Technology Index versus Productivity and Error Content

This section provides some data which relate two key measurables as indicators of the software and the process used to implement it to the 'software process technology index.' These measurables are the software error content, and the development productivity. The index values have been found to have a reasonably strong positive correlation with development productivity and a relatively significant negative correlation with the (estimated) error content of the software.

Figure 1 is a plot of relative values of development productivity and software process technology index; a 'best fit' line to the data is also shown. The 'relative' values are normalized with respect to one of the productivity values. Figure 2 is a plot of relative values of estimated latent (or post ship) errors and corresponding 'software technology index values,' a 'best fit' line to the data is also shown. The relative values are normalized with respect to one of the software product's (estimated) latent error values.

The latent error content of each software product was estimated using a^{5,6} tool, coded in BASIC that runs on the IBM personal computer. This tool estimates the number of errors to be detected in later phases of the development process as well as the latent error content, based on the number of errors found during the inspection processes.¹

Also note that this 'software process technology index' values used in composing these plots and in computing the correlation coefficients were computed applying equal weight to each of the fifteen attributes listed in

Table 1. It is likely that their effects on productivity and quality are, in fact, not uniform.

The sample (linear) correlation coefficient of relative productivity with technology index was found to be 0.697. The corresponding figure for latent error content and technology index was found to be - 0.582. These figures, while not high if evaluated on the same basis on which medical experiments are considered, are reasonable for the computer science field and suggest the 'value added' by software technology. This was a principal purpose of the quantitative analysis reported upon here. The former figure means that 48.6 percent of productivity variation is explainable by the variation in software process technology index. The latter figure suggests that 33.9 percent $(= (0.582)^2 \times 100)$ of the variation is explainable by the software process technology index.

These plots and correlation coefficient values do suggest valuable trends, supporting continued use and expansion of the application of modern technology because of its potential impact on the business in terms of supporting the goals of higher productivity and reduced post-ship errors. The average of about 50 for the software process technology index and about 62 for the peak found so far in the projects we have looked at suggest that there is a lot that can yet be done to further exploit the potential for software process technology in the 'software business.'

Figure 3 provides a plot of relative software development productivity versus relative latent error content (the independent variables shown in Figures 1 and 2, respectively); a line fit through the data points is also shown. The data suggest that higher values of productivity and lower values of latent error content are associated. The sample correlation coefficient value is - 0.6724, which means that 45.2 percent of the variations of one variable is 'explainable' by the other. The negative correlation between development productivity and latent error content suggests that higher degrees of software quality need not be realized at the expense of productivity.

The software process technology index was also evaluated as a function of time, specifically with respect to the date of delivery of the software product whose development process was evaluated. A positive trend was found as shown in Figure 4.

ESTIMATED
RELATIVE NUMBER
OF LATENT ERRORS

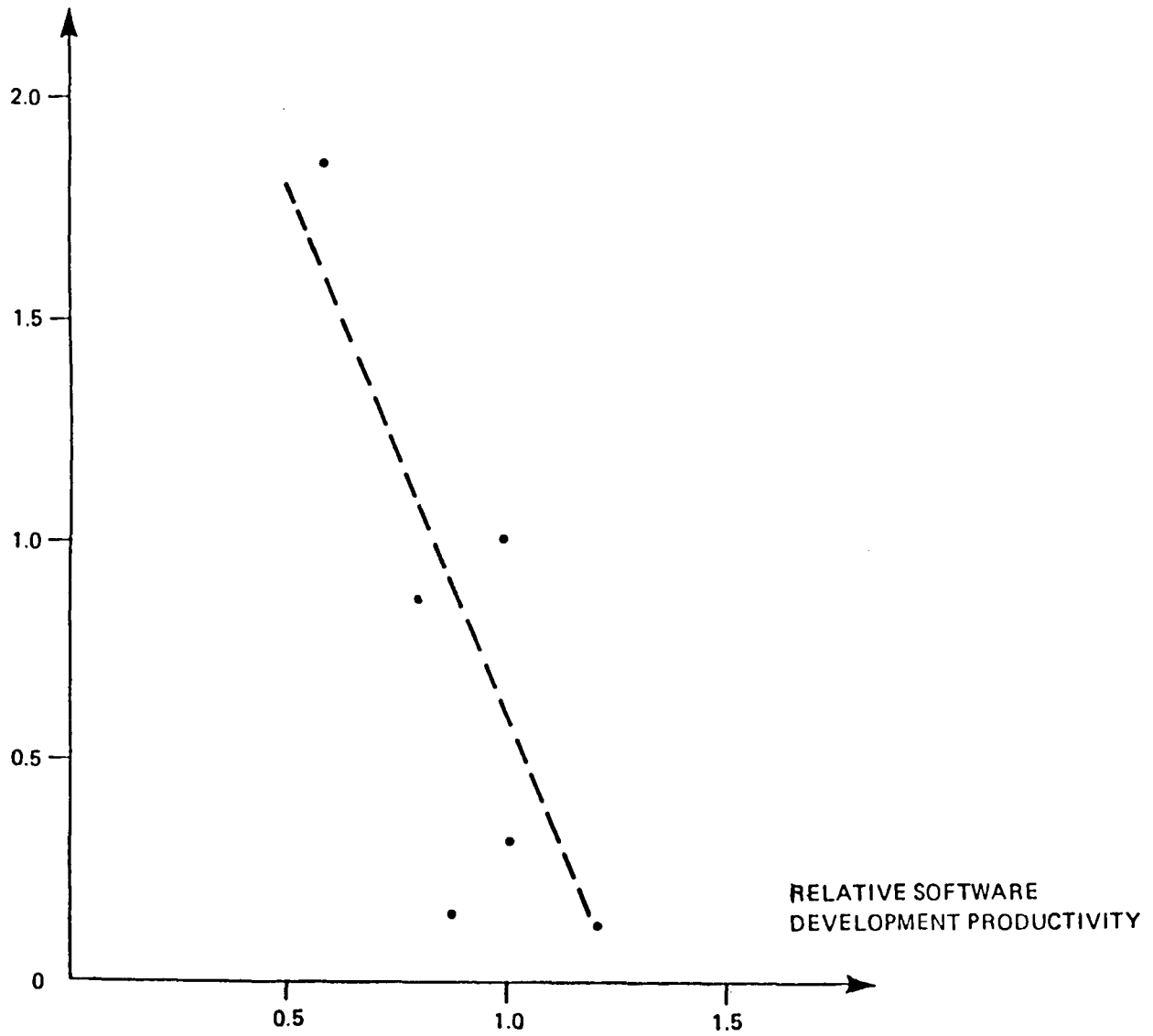


FIGURE 3 RELATIVE LATENT ERROR CONTENT VS. RELATIVE PRODUCTIVITY

SOFTWARE PROCESS
TECHNOLOGY INDEX

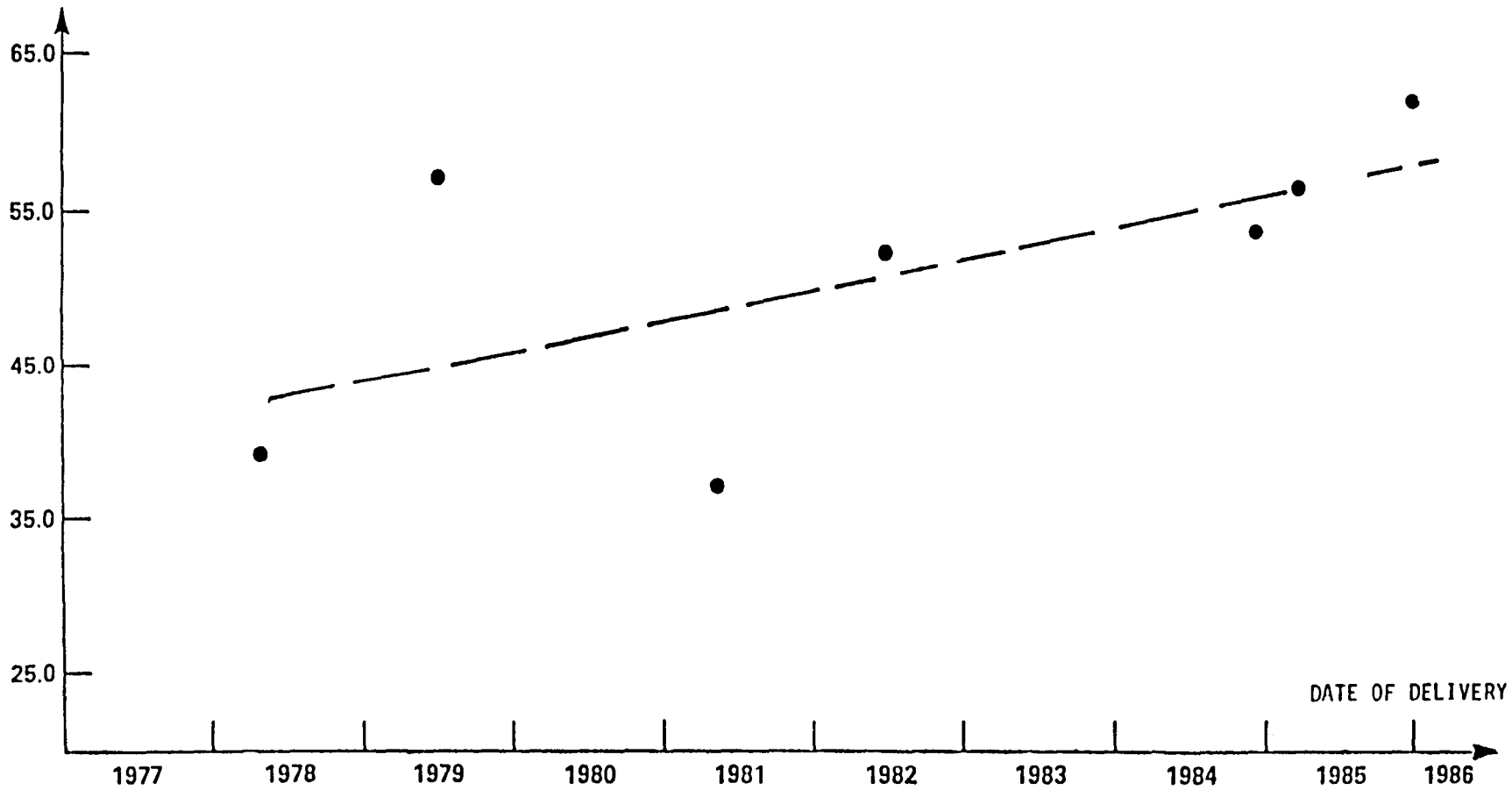


FIGURE 4 SOFTWARE PROCESS TECHNOLOGY INDEX VS. DELIVERY DATE (TIME)

APPLICATION OF THE SOFTWARE PROCESS TECHNOLOGY INDEX

The Software Process Technology Index and its quantitative relationships to productivity and quality indicators, such as latent error content, can be used for various management oriented activities such as:

- o Estimating the likely effects of potential changes in software process technology or productivity and quality ("What's the payoff?")
- o Estimating the effect in uncertainty of the applicability of some element of technology in a certain development situation ("What's the risk?")
- o Validating estimates of development costs ("Are the productivity and technology use levels proposed by the developer compatible?")

Software developers and management personnel can employ empirically derived quantitative relationships between the technology index and productivity or latent (post-ship) error content to validate an estimate of new software productivity or quality obtained using other techniques.^{4,5} Also, such relationships between the technology of the software process and the resultant software product might be used for planning purposes, such as for predicting the possible impact on software development productivity or the quality of the delivered software product of proposed improvements in the technology being used in software development.

The authors believe both intuitively and from specific empirical observation that the higher the Technology Index is for a software development project and its associated process, the higher will be the productivity of the software engineers and the quality will be better of the software they develop, in terms of residual latent defects of the delivered product. Data presented supports this belief.

Specific benefits of higher technology index scores that have been detected include the following:

- o Fewer errors inserted into the developing product.
- o Earlier detection/removal of inserted errors.
- o Better predictability of schedule and cost of development; the authors deduce that these benefits accrue from using a well defined and automated software development process. It would include mechanisms for preventing error insertion (syntax directed editors, and the like), early error detection, (formal inspections, static and dynamic design/code analyzer) confidence raising (automated test generators, coverages measures) and clearly specified criteria and enforcement mechanisms for entry into and exit from each phase of development.

REFERENCES

1. Fagan, M. E., "Design and Code Inspections to Reduce Errors in Program Development," "IBM Systems Journal," Volume 15, Number 3, 1976, Page 182.
2. Linger, R. C., et al, "Structured Programming: Theory and Practice," Addison-Wesley, 1979.
3. Quinnan, R. E., "The Management of Software Engineering, Part V," "IBM Systems Journal," Volume 19, Number H, 1980, Page 466.
4. Gaffney, J. E., Jr., "On Predicting Software Related Performance of Large-Scale Systems"; Proceedings of the 15th Conference of the "Computer Measurement Group," CMG XV, December, 1984, San Francisco.
5. Gaffney, J. E., Jr. and Martello, S. J., "A Model For Prediction of Latent Errors Using Data Obtained During the Development Process," 9th Software Engineering Workshop, NASA, GSFC, November, 1984 (Page 196, "Proceedings").
6. Gaffney, J. E., Jr., "On Predicting Software Related Performance of Large-Scale Systems," CMG XV, December, 1984, San Francisco; also printed in the "Proceedings of the 15th Annual Conference of the Computer Measurement Group."

PANEL #3

SOFTWARE ENVIRONMENTS

E. Spafford, Georgia Tech.
B. Boehm, TRW
T. Wasserman, IDE, Inc.

The *MOTHR*A Software Testing Environment*

Richard A. DeMillo[†]
Eugene H. Spafford

Software Engineering Research Center
Georgia Institute of Technology
Atlanta, Georgia 30332-0280
+1 404 894-3180

ABSTRACT

The value of software testing in the development of large software systems is well-documented. Unfortunately, the development and employment of an integrated test plan is often avoided due to the costs associated with testing. These costs include more than just capital expenses associated with obtaining test systems and software. They also include the time and effort involved in educating personnel in the use of the testing system, the time taken to run the tests, and the costs of rerunning the tests after errors are found and corrected. Furthermore, some forms of testing are difficult or impossible to run incrementally, and they produce results which may be difficult to use in correcting or enhancing the tested software.

The *MOTHR*A Environment is an integrated set of tools and interfaces that support the planning, definition, preparation, execution, analysis and evaluation of tests of software systems. The support provided by *MOTHR*A is applicable from the earliest stages of software design and development through the progressively later stages of system integration, acceptance testing, operation and maintenance. *MOTHR*A has been designed to address some of the cost concerns mentioned above. Two primary design criteria, in particular, are significant in this regard. First, the *MOTHR*A interfaces—particularly user interfaces—are high-bandwidth. This allows us to present more information during testing and retesting. Coupled with proper design and integration with familiar displays, it should obviate the need for extensive training to use *MOTHR*A.

Secondly, the overall *MOTHR*A architecture imposes no *a priori* constraints on the size of the software systems that can be tested in the environment. The practical meaning of this criterion is that the same architecture is able to service programs varying in size from individual modules of less than 10³ source lines to fully integrated systems of more than 10⁶ lines. The human user—the *tester*—is able to apply comparable functions across a familiar interface as the software being tested evolves in size and complexity by several orders of magnitude. In fact, the only indicators of size or complexity that have ties to the *MOTHR*A architecture are the operating system cost penalties and performance delays inherent in manipulating massive objects. All other costs and resource demands are under the direct control of the tester. In most cases, the tester will choose to allow critical resources such as *time* or *memory* to grow linearly with program size and complexity. The tester may, however, choose to conserve these resources by sacrificing other resources (*e.g.*, dollars) or even by reducing the fidelity of the test. These are ulti-

mately economic decisions determined by the relative costs of tests and failures—*MOTHRA* does not *legislate* or even *favor* one kind of decision in preference to another.

An important mechanism for meeting these criteria is that *MOTHRA* is reconfigurable, allowing the integration of user and system tools with which the tester may already be familiar, and allowing the system to make use of different underlying hardware architectures of differing capabilities. We address this in *MOTHRA* by the use of *thematic tools* for software testing. It has been our experience that software testing is most effective when the test procedures can be reduced to a set of well-understood and natural activities. Since *MOTHRA* supports tests of both very small and very large programs, the details of the tools that are actually invoked vary in power and scope. However, even very different tools can implement basic *themes* that are carried along throughout the several phases of testing. For example, programmers in modern development environments interact increasingly with an array of very powerful source language debuggers. Even though formal testing methodologies and debugging are very different activities, the *debugging theme* can be used as a metaphor to carry the tester from tool to tool as the software being tested evolves.

One *MOTHRA* system has been constructed using the AT&T Bell Labs *Blit* interactive bitmap display terminal running under the control of a UNIX window manager called *Layers*. The host environment is a modestly configured VAX 11/780 running UNIX 4.3 BSD. Another version has been implemented on VAXstations[®] running Ultrix 1.2 and the X Window System. However, the architecture of *MOTHRA* encourages re-hosting. Furthermore, explicit operations allow *MOTHRA* processes to spawn parallel and vectorized processes for execution by a Cyber 205 (or any other powerful parallel machine).

January 23, 1987

* The work presented in this paper was funded, in part, by RADC contract F30602-85-C-0255.

† The authors may be reached by e-mail addressed to:

Internet: rad@gatech.EDU spaf@gatech.EDU
uucp: ...!{akgua,decvax,hplabs,seismo}!gatech!{rad,spaf}

• UNIX is a registered trademark of AT&T Technologies.

The *MOTHR*A Software Testing Environment*

Richard A. DeMillo[†]
Eugene H. Spafford

Software Engineering Research Center
Georgia Institute of Technology
Atlanta, Georgia 30332-0280
+ 1 404 894-3180

1. Introduction

The *MOTHR*A Environment is an integrated set of tools and interfaces that support the planning, definition, preparation, execution, analysis and evaluation of tests of software systems. *MOTHR*A is designed to be used starting at the earliest stages of software development and continuing through the progressively later stages of system integration, acceptance testing, operation and maintenance.

The *MOTHR*A system satisfies three primary criteria. First, its interfaces—particularly user interfaces—are high-bandwidth. Second, the overall architecture imposes no *a priori* constraints on the size of the software systems that can be tested in the environment. While these seem to be unrelated criteria that address issues at differing levels of detail, they are, in fact, closely linked.

Since the ability to process very large integrated software is an explicit design goal, increasing the effective *feedback bit rate*¹ along key interfaces is an obvious way to design for acceptable functional performance. The bandwidth of the interface is simply the feedback bit rate that it supports. Bitmap displays and windowing are the usual means of increasing the bandwidth of user displays, for instance. Less obvious are techniques which increase the effective bit rate by graphical compression, statistical sampling, and analog representations. In *MOTHR*A information is highly compressed for presentation to the tester. This provides a high-bandwidth user interface in which structural and dynamic information is summarized graphically and exact representations of algorithm and program behavior are replaced by inexact *animations* of behavior, higher-order descriptions of process execution, and non-procedural specifications of program function.

The practical meaning of the second requirement is that the same architecture should be able to service programs varying in size from individual modules of less than 10³ source lines to fully integrated systems of more than 10⁶ lines. That is, the human user—the *tester* should be able to apply comparable functions across a familiar interface as the software being tested evolves in size and complexity by several orders of magnitude.

In fact, virtually the only indicators of size or complexity that have ties to the *MOTHR*A architecture are the operating system cost penalties and performance delays inherent in manipulating massive objects. All other costs and resource demands are under the direct control of the tester. In most cases, the tester will choose to allow critical resources such as *time* or *memory* to grow linearly with program size and complexity. The tester may, however, choose to conserve these resources by sacrificing other resources (*e.g.*, dollars) or even by reducing the fidelity of the test. These are ultimately economic decisions determined by the

¹ This use of the term *feedback bit rate* is apparently due to S. C. Johnson and refers to the natural measures of work and efficiency in software development environments. Roughly speaking, the feedback bit rate is the number of bits transferred across an interface (from host to user) per atomic user interface operation.

relative costs of tests and failures. *MOTHR*A does not *legislate* or even *favor* one kind of decision in preference to another.

The key to this approach is to design an environment in which most primitive operations are implemented as *local transformations* of data objects. Global operations, on the other hand, are never applied to these objects but rather are defined in terms of primitive transformations of more complex *atomic* objects.²

*MOTHR*A satisfies these requirements by first organizing the user interface around a high-resolution bit map display with adequate graphics and windowing capabilities and, second, by using the display as a tester's *view* into a larger (virtual) test context. A view is defined by a consistent set of object instances that comprise a meaningful state for the *MOTHR*A system. Such a state contains sufficient information for applying a set of primitive operations and generating test-related data and results in the form of new object instances. The tester need have only a dim idea about the representation or physical location of aspects of the test which are not in view. As a matter of fact, the total context of a sufficiently complex test may not be meaningful to a software tester at all; in this instance, a large team of testers will each have differing views of the test, the total context of which is really only understood by systems engineers.

One of our major concerns has been to make *MOTHR*A reconfigurable. For the most part, *MOTHR*A does not attempt to re-create capabilities provided by the environment in which it is hosted. The guiding principle has been to structure *MOTHR*A as a *subenvironment*^{DeMi86} of an overall software development or support environment. This implies both a certain closure and a robust interface. The *MOTHR*A architecture supports as a function any meaningful composition of basic functions. This is accomplished through an object-oriented architecture and user interface. There are several motivations for not viewing *MOTHR*A as highly integrated into a more global host environment. Foremost among these are the need for isolation and protection of test-related processes.

This same goal is also addressed in *MOTHR*A by the use of *thematic tools* for software testing. It has been our experience that software testing is most effective when the test procedures can be reduced to a set of well-understood and natural activities. Since *MOTHR*A supports tests of both very small and very large programs, the details of the tools that are actually invoked vary in power and scope. However, even very different tools can implement basic *themes* that are carried along throughout the several phases of testing. For example, programmers in modern development environments interact increasingly with an array of very powerful source language debuggers. Even though formal testing methodologies and debugging are distinct activities, the *debugging theme* can be used as a metaphor to carry the tester from tool to tool as the software being tested evolves. For example, program mutation^{DeMi78, Budd81, Howd82} requires testers to construct sets of tests to demonstrate that certain basic design and programming errors are not present.³ A fundamental activity in program mutation is *revealing bugs* in the mutant programs. Powerful debuggers are therefore useful tools during the tests and can be carried along as thematic tools. Many other test methodologies can, in turn, be reduced to mutation testing.^{Acre79, Budd81} Thus, these methodologies can also be supported by the thematic tools.

² We use the term *object* to mean a collection of data and operations on that data. An *atomic object* is one which allows only atomic operations, in the sense of *view atomicity*.^{Alc83} We do not address concepts like reliability or fault tolerance with the design of *MOTHR*A. Further, the exact structure of these objects (*active* or *passive*, etc.) does not matter. The object paradigm is intended as simply a design approach to the construction of *MOTHR*A.

³ In this sense, program mutation is a kind of fault detection experiment, as might be carried out to detect faults in digital circuits. Here, the experiments are applied to software and the fault model is the space of likely errors that programmers make. The "local transformations" mentioned previously are simply the fault insertion operations. This technique is general enough to simulate common coverage-based tests such as statement, branch, and path coverage as well as many other systematic software tests.

There are subsidiary issues that are addressed in the design of *MOTHR*A. Foremost among these is our belief in *capitalizing* the software development effort at an appropriate level. The notion of capital-intensive software engineering and production is not a new one. For the *MOTHR*A development group, this point of view has led us to a fairly cavalier attitude toward trading machine cycles for human effort in conducting a test. Provided only that it can be justified economically, *MOTHR*A will spawn machine-intensive tasks and organize them for execution by a computer resource of appropriate power. This function is called *resource-shifting* and, although it is under the control of the tester, *MOTHR*A organizes and partitions all test views to accommodate such remote processing.

2. User Views

Testers interact with *MOTHR*A through a *view* of the test. The tester's view presents images representing global test status as well as local objects, attributes and processes. There may be several views to which the tester has access at any one time, but these views must be accessed serially and the user cannot have two simultaneous and distinct views of tests.

Some of the objects in view are entirely local and *private* to the user. For example, the user may create a temporary file as an aid in deriving appropriate test cases. These objects are under the complete and total control of the current view, and the user who "owns" the view can create copy, share, and destroy these objects at will. At the other extreme are those objects that are *shared* by all views. These objects are typically under the control of agents or processes external to *MOTHR*A. An example of such a shared object is the source listing of the software being tested. Such objects might be the property of configuration management and library tools residing in a host environment. These tools enforce a specified set of rights to access or modify the shared objects. *MOTHR*A operations on any shared objects in view respect the rights inherited from the external owners or managers of these objects. Intermediate to these private and shared objects are the *public* objects. Objects that are public represent the visible activity of the test. These objects are generated by testers and by *MOTHR*A tools. Public objects may include test cases and results, traceability mappings between test events and specifications, and error/fault statistics. Some of these public objects are transient while others are persistent. Occasionally, a transient object (e.g., test case number 6) affects a persistent object (e.g., the error count for path number 26) and is incorporated into the *MOTHR*A object base according to predefined dependencies, relationships, and operations in much the same fashion as source code files dependencies are treated by the UNIX *make* utility.^{Feld79} The exact nature of these dependencies define a *policy* that is unique to the test and its organization. *MOTHR*A does not define these policies—it only enforces them.

In physical appearance, a view is bounded by the edges of a high-resolution bitmap display. Each window in the view gives the tester access to certain objects and operations that are currently meaningful. The tester *selects* windows, objects, and operations with a mouse that can be used to point to windows and their contents and to pull down menu selections that are displayed under user control.

*MOTHR*A interfaces have been implemented for the Bell Labs *Blit* interactive bitmap display terminal⁴ running under the control of a UNIX window management executive called *Layers*, and on Digital Equipment's VAXstation II color and black-and-white display terminals running under the X Window System.^{Sche86} These particular instances of the user interface are, however, not the only ones possible. The underlying architecture effectively dissociates the physical properties of the display from the tools which the display accesses. In essence, the display is treated as just another tool in the environment. Other display tools can be substituted provided that the environment's interface conventions are satisfied.

⁴ The AT&T 5620 Dot-Mapped Display. See [Pike84], for example.

2.1. Functions and Operations

We will begin by briefly describing a typical set of functions that the tester invokes. These functions are generally invoked in a sequence of views, called a *run*. Runs may be suspended (saving the complete view at the time of suspension) and resumed at any time. However, atomic operations are non-interruptible. Therefore, the view that is actually associated with a suspended run may contain objects resulting from values returned at a later time by on-going atomic operations. These are managed by a data- and event-driven harness. The same mechanism is used to manage multiple views of a test. A single display, for instance, may be used to invoke a series of functions applied to two different source modules. Since only one view at a time can be available, the tester can invoke a set of atomic actions and suspend the run to begin a run for the second module.

2.1.1. Run Initiation

The key shared objects are the source files.⁵ A run is initiated by identifying a set of source files and associating the name of the run with those files. *MOTHR*A handles the parsing of the source files to a convenient internal form and also manages the naming conventions for modules and other syntactic units contained in those files.

2.1.2. Test Level Selection

A test plan may specify any of several *levels* of testing to be performed.^{Budd81} Examples of these levels are *statement analysis*, *predicate and domain analysis*,^{Whit78} and *coincidental correctness analysis*. Statement analysis is used for determining that every statement in the program has been executed and has some effect on the functional behavior of the program. Predicate and domain analysis are used to determine that all branches and specified paths are properly selected and that domains associated with these predicates are properly defined. Coincidental correctness analysis is used to test for the presence of a wide variety of computational errors, including various arithmetic, data flow, and interface errors.^{Good79}

Within each level, the user may also choose a *strength* of test, represented by a percentage. The exact meaning of a strength value depends on the specific level of testing and certain subsets of the levels that may be selected. For example, if the user selects the statement analysis level at 100%, the test can only be passed by constructing tests that fully exercise every statement in the program. Within the predicate and domain analysis level at 90% strength, the tester will be required to construct tests that with 90% certainty determine the boundaries of predicate domains.

The levels of test are defined in terms of certain *mutant* operators.^{Budd78} That is, source code transformations that implement the desired level of testing. For example, in the statement analysis level, mutant operators called *san* and *sdl* are used to determine whether each statement has been executed and to what effect. The *san* operator replaces each source statement by a special statement called *trap* that raises an exception. Unless test cases are provided that raise all possible exceptions, all statements cannot have been exercised. On the other hand, the operator *sdl* replaces each statement by a *no-op*. Unless the transformed programs behave differently than the program being tested, the test data does not demonstrate that the given statements have any functional effect on program behavior.

Within the levels, classes of these mutant operators may be selected by the tester. In these cases, the tester will use the selected operators to implement specialized testing strategies.^{Acre79} These selections may be made on the basis of known or suspected weaknesses, or perhaps upon economic considerations (*e.g.*, the tester may only have the resources available to test 25% of the mutants in a specified time span).

⁵ *MOTHR*A is a multi-lingual environment. In the current version, *MOTHR*A is limited to processing Fortran 77 (the complete language) and Ada (a large subset). Later versions are planned for C, Modula 2, Lisp, and possibly others.

Selection of levels, mutant types, and strengths may also be associated with source code components. For example, during a unit test, the user may select only a certain subroutine for a particular level and strength of testing. During software integration testing, the tester may choose an incremental (*i.e.*, bottom-up) strategy in which a given level and strength are successively applied to units, then to integrating software that calls these subroutines, and so on.

2.1.3. Test Data Selection and Execution

An important test function is the construction of tests and the execution of the program on the test data. The creation of a set of test cases is essentially an *editing* function. The editing may be under the control of the human tester, who is trying to meet some specified level of testing (*e.g.*, testing for the presence of all coincidental correctness errors of a given type), an automated test data *generator*, a simulator, or even some data capture device that records digitalized inputs from sensors, operators and communications channels. Creation of appropriate tests is a key function. We will return to it again after some other supporting functions have been described.

The actual *testing* is carried out by executing programs on the test data. The results are observed by an *oracle* that decides whether or not the program has behaved properly. The notion of *proper behavior* can be quite complex. In unit and module testing, the concept is usually identified with functional correctness—that is, consistency with a written formal or informal specification. In later views of a more highly integrated software system or subsystem, correctness is less important than meeting functional or user requirements. The oracle mediates all of these authorities. If a formal specification is available, the oracle consults it. If a human user is the authority, the oracle takes advice from this source. If the behavior cannot be assessed without additional instrumentation, the oracle receives instrumented output and reacts accordingly.

If unacceptable behavior is observed, the policies in force for the test determine the next course of action. In some cases, the test proceeds after the nature and location of the error is recorded in a public record. In other cases, the cause of the failure is located and fixed immediately, resulting in a new view of the test.

2.1.4. Test Status Evaluation

During the testing process, the tester eventually wants to know whether or not testing has been completed. This determination may be subjectively made or it may be specified quite precisely and unambiguously. The latter case is obviously the more interesting one in *MOTHR*A.

Test status is instrumented and reported as dynamic progress toward meeting test goals specified during run initiation. The user may be interested in overall progress toward completing a test specified for a given level and strength. By the same token, the user may be interested in whether or not a test has been carried out to reveal a specific error or type of error. In all of these cases, test status can be defined in terms of a single primitive function: execution of a *mutant program* on the test data. If the test data—in the judgement of an oracle—does not distinguish the program being tested from the mutant program, then the mutant is said to be *live* and is reported as such. If, on the other hand, the oracle determines that the mutant behavior varies significantly from the behavior of the original program, then the mutant is marked *dead*.

Dynamic information on test progress can be displayed in graphical and tabular format and is archived in public and shared objects according to test policies enforced by *MOTHR*A.

2.1.5. Test Data Creation Revisited

Test status evaluation is used to guide the test creation process. The tester may elect to stop testing at this point or to strengthen the test data by attempting to kill some live mutants.

If all currently enabled mutants have been killed, the tester may wish to create new mutant types or begin testing a different subroutine.

In this process, the user is aided by the evaluation displays as well as by tools that may be imported. Suppose, for example, that the tester is attempting to kill all mutants that replace integer constants n with $n+1$ and $n-1$ (as might be required for domain analysis). In addition to reporting that these mutants remain alive, *MOTHR*A allows the user to examine the effects of these mutants in the context of the original program or even to browse through related source lines or live mutants. More powerful *test case* editing capabilities are available to create new tests, modify previous tests or to capture the results of other test data generators. If the user has an especially difficult time in constructing a test that kills these mutants, he may import a debugger to attempt to exhibit that the mutants are in fact "buggy" versions of the program.

2.2. The Display

The technology used in the display and the material presented in that display are critical to the design of *MOTHR*A. The *MOTHR*A window layout presents the user with a view of all the objects that were described above. Based on our classification of objects we have defined the following subwindows (displays) within the *MOTHR*A display:

- **Mutant Status Manipulation:** The icons that define and reference specified mutant types, aggregations of these types, and the levels and strengths of tests that can be defined from them.
- **View Status:** The graphic symbols or textual displays that represent the progress of the current view toward test objectives, or other measures of completion.
- **Test Cases:** Any object—whether constructed by the tester or captured from an external source such as a simulator—that is used to stimulate the software being tested.
- **Source Language Representations:** Each view of the test defines a fragment of the software being tested, and a source language representation of such a fragment is a high-level description of the fragment. By definition, the most primitive constructs in any source language representation are the source lines of code; all other representations associate text or graphical information with sets of source lines.
- **Command Line:** Terse communications, prompts and system status reports are directed to a degenerate (one line) window called the command line.

Testers may query and modify attributes displayed in any of these subwindows. Transient information and data are displayed by whatever means is most appropriate for the display tool. In our implementation, such transient data are displayed in windows that overlay (and may sometimes obscure) the fixed windows just described. An example of a transient object might be one of the thematic tools mentioned in Section 1. The tester must make any explicit interfaces and functional dependencies between transient objects and *MOTHR*A objects since none are implicit in our design of *MOTHR*A.

The *MOTHR*A Display handles "global" information in two distinct ways. First, it gives the tester access to objects not in the current view. For example, to initiate the testing session, the tester provided file names that were meaningful to the host's file system, even though *MOTHR*A does not contain file management capabilities. Second, simply touching and changing the attributes of objects in the Display can have affects on the other windows in the view—thus the Display encapsulates a set of "global" relationships for the rest of the view. For example, selecting a random sampling of substitution mutants results in a propagation of mutant status information to the other subwindows, such as the View Status subwindow.

Attributes of objects displayed in each window can be modified dynamically, so that, for instance, the display format of the source language text can be changed to bring the live mutants into view. More complex interactions between view and source windows are

possible. For example, the tester can point to a histogram "bar" in the view window and cause the corresponding live mutants to appear in the source window.

3. Subenvironment Architecture

Supporting the user display is a collection of tools bound together by an *information interface* and hosted on another environment. Specified access pathways and ports allow information, commands, and signals to flow between *MOTHRA* and the host environment. While most of these have operating system dependencies, they have been hidden in higher level constructs that appear to be primitives to *MOTHRA*. Although the overall design is robust, implementing these primitives is easier in some environments than in others.

For example, one of the reasons for conceiving *MOTHRA* as a subenvironment of a host is the need to control and manipulate faulty processes. Unlike most programming environments, the *intent* of *MOTHRA* is to execute faulty processes. While most software developers would like to consider failure to be an abnormal condition, the *MOTHRA* user deliberately seeks it out through the process of killing mutant programs. Many of the failures induced in this way are benign (the mutant program runs to completion but delivers incorrect results). Approximately one fourth of the mutants generated,⁶ however, are not benign. They generate processes that run seriously amok and must be tightly controlled. The modes of failure in these processes run from simple errors such as division by zero to storage allocation and concurrency errors that could harm unrelated processes if allowed to proceed unconstrained.

An important aspect of these definitions is that the system defines a *process* at each time n , rather than just a state. This is a key idea for several reasons. First, the atomicity of actions may result in several intermediate states before any other *MOTHRA* function can be applied. Second, the display architecture and logical driver together constitute a data and event driven network of autonomous processes and unique definitions of sequences of states may not be possible in certain circumstances, whereas definitions of sequences of processes can be defined in terms of the external actions needed to invoke them. Third, error recovery and roll-back procedures as well as look-ahead optimization are easier to define and implement. Fourth, we anticipate the use of *MOTHRA* in conjunction with nondeterministic system testing procedures; recording and replaying test scenarios and associating internal test events with software inputs is relatively easy to implement if each major time step of the environment corresponds to a history of states.

The information interface is the *MOTHRA backplane*. In many respects, *MOTHRA* combines the features of both open and closed programming systems. *MOTHRA* is closed in that the fixed windows of a view and the objects, attributes and operations associated with them define an Entity-Relationship (E-R) model^{Chen80} that cannot be modified. Thus the process monitors, test data generators, instrumentation and other tools associated with the fixed windows can always count on certain dependencies and relationships among essential objects in view—ensuring, for instance, reproducible behaviors.

On the other hand, *MOTHRA* is open to the extent that any E-R model-respecting tool whatsoever can be attached to the backplane. Editing is a simple example of a transient activity that can be imported in this way. Any file can be edited by any editor provided only:

- the file is editable by the editor in question;
- the point in time at which the editor is invoked does not preempt or interrupt an action defined to be atomic in the E-R model;
- no attributes or properties are introduced by the editor's actions or side effects that contradict attributes or properties of the E-R model.

In other words, any tool can be imported to the user's view, provided that the user is able to plug (or wire) that tool into the backplane. This is a particularly valuable design for a

⁶ In our testing so far.

testing environment, since many testing tools share common tool fragments. It also permits some novel interactions between the host and *MOTHR*A environments. A software developer, for example, can attach a mutant generation and execution capability as a background activity during coding and debugging. This is a generalization of Weinberger's dynamic instruction counting tool.^{Wein84} The underlying E-R model allows the processes of mutant generation and execution to be decoupled from the integrating framework provided by the display architecture (recall that the display technology is simply another tool that plugs into the backplane). One application of this capability is the inexpensive maintenance of test status throughout the development process by keeping killed mutant status information for object code.

4. Resource Shifting

The process of creating and executing mutant programs on the test cases $\tau_1, \tau_2, \dots, \tau_k$ can be done serially in one of two logical orderings. The first ordering would be to apply the test cases, one at a time, to each live mutant and observe the results. The second ordering is where all test cases are applied to each live mutant and the results observed. All such serial processes consist of on the order of $\mu \times k$ independent transactions, where μ is the number of enabled mutants and k is the number of tests to be executed.⁷ In either case, we are presented with a series of independent tasks.

Simply spawning these independent tasks to m independent parallel processors reduces the elapsed time for processing the test cases against the mutants to:

$$\frac{\mu \times k}{m} + OVERHEAD.$$

Since the *OVERHEAD* can be compressed to one of the serial protocols mentioned above, this amounts to a linear speed up on independent parallel processors. However, large blocks of these tasks have an internal structure that can be exploited to achieve more impressive speed gains.

For example, the substitution mutants of a simple assignment (using C-like notation) can be written in one of the following forms:

$$*lhs = operand_1 \times operand_2 \Rightarrow *lhs' = operand_1 \times operand_2 \quad (1)$$

$$*lhs = operand_1 \times operand_2 \Rightarrow *lhs = operand'_1 \times operand_2 \quad (2)$$

$$*lhs = operand_1 \times operand_2 \Rightarrow *lhs = operand_1 \times operand'_2 \quad (3)$$

Furthermore, the order in which these mutants appear is fixed once the program is known. At the time *mutgen* returns a value, the mutant statements (1)-(3) are equivalent to a *vector operation*

$$LHS = OPERAND_1 \otimes OPERAND_2,$$

where \otimes is the vectorized binary operation and the vectors $LHS[i]$, $OPERAND_1[i]$, and $OPERAND_2[i]$ are defined respectively to be $*lhs$, $operand_1$, and $operand_2$ if $i = 0$. For $i \geq 1$, the vector positions are defined by the mutant definitions (1)-(3). Thus, the substitution mutant executions are equivalent to a series of vector operations (followed by inner product operations to determine which mutants have been killed).

Interleaving the generation of vectorized expressions with parallel tasks can result in a multiplicative speed-up. This is especially attractive for the case of substitution mutants since for a typical n line program, the (worst-case) number of substitution mutants grows

⁷ Some simplification is possible by "short-circuiting" an iteration once a mutant has been killed (there is no need to apply further test cases to a dead mutant), but we will ignore that and other optimizations in the following presentation so as to make it more accessible.

proportional to

$$\binom{n}{2}$$

which is the dominant term in the expression denoting the worst-case complexity of mutant generation and execution. For moderately sized software systems (*e.g.*, systems for which $10^4 \leq n \leq 10^6$) complete tests have required several days of dedicated computer time. With interleaved parallel tasking and vectorization on processors with MIPS rates in the 50-100 range, a thousand-fold speed-up is possible, bringing these tasks to within the reach of real-time responses.

This has led us to consider seriously the possibility of *shifting resources* to accommodate such processor intensive tasks. *MOTHR*A is designed to be hosted on hardware configured with multiple machines of varying capabilities.

For example, one host might consist of the bitmap displays, object definitions, and file services required for tester interaction. We assume also that whatever programming environment serves as the *host environment* for *MOTHR*A can be accessed through this host. In particular, editing and other transient functions do not make any demands on subsequent layers.

A second host consists of large-to-medium granularity parallel processors. Each of these processors operates on a common memory with appropriate programmer control of parallelism. The tester may—when local resource thresholds are exceeded—*shift gears*. The result is the spawning of blocks of independent parallel tasks for each of the processors. Coordination of destination processors and the collection and collation of the results of process execution is the responsibility of a process that resides on the first host. It is intended that the tester have complete control over the allocation of parallel resources. At present, however, this control is restricted to partitioning the serial tasks mentioned above in some appropriate manner.

In the same manner, vectorization is carried out as described above and the vectorized code and test cases are sent to a third host. Since the result of the vector operation is itself a vector, only this result is returned from this host. The precise format of vector operations is a machine-dependency that cannot be easily removed, although we anticipate that UNIX systems capable of 100-500 scalar MIPS with powerful vector extensions to C will become widely available. For the current version of *MOTHR*A, however, we are adopting a conservative approach. For example, long chains of data dependencies within loops are being partitioned to avoid vectorization difficulties.

The experimental performance studies of resource-shifting will be reported in detail elsewhere.

5. Conclusion

The *MOTHR*A environment described in this paper is currently implemented and running in a multi-host environment consisting of Digital Equipment VAX 11/780 and 11/750 mini-computers, VAXstation II workstations, AT&T Blit bitmap display terminals and a Control Data Cyber 205 supercomputer. Version 1.0 of *MOTHR*A contains at least primitive implementations for the functions described above, although many of the most desirable integrating features (*e.g.*, automating the transmission of vectorized processes from the VAX host to the Cyber 205) are not fully functional. Thus far, *MOTHR*A has been used to test Fortran 77 programs in the 20-500 line range. With current memory and other constraints (there are no *MOTHR*A design constraints) complete testing of 1,000-10,000 line Fortran programs seems well within the capabilities of Version 1.0.

A second version that exploits optimization opportunities and will be tailored to extremely large-scale applications is under design.

Although user experience with *MOTHR*A is currently confined to our development group, we expect Version 1.0 to be available on a limited scale to a community of 30-50 software

testers. In spite of the care we have taken to ensure that fundamental design concepts really match the needs of realistic software testing, we anticipate that many hitherto unidentified issues will surface. These experiences will be analyzed and reported at a later date. We are optimistic, however, that a software testing environment architected as described above will deliver acceptable levels of computing resources to the important problem of how to test and evaluate the quality and reliability of large software systems. Furthermore, we anticipate that the system will be easily learned and easily used, thus leading to improvements in testing and software production.

References

Acre79.

Acree, A. T., R. A. DeMillo, T. A. Budd, R. J. Lipton, and F. G. Sayward, "Mutation Analysis," TECHNICAL REPORT GIT-ICS-79/08, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1979.

Allc83.

Allchin, J. E., "An Architecture for Reliable Decentralized Systems," PH.D. DISS., School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1983. Also released as technical report GIT-ICS-83/23

Budd78.

Budd, T. A., R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "The Design of a Prototype Mutation System For Program Testing," *PROCEEDINGS NCC, AFIPS CONFERENCE RECORD*, pp. 623-627, 1978.

Budd81.

Budd, T. A., "Mutation Analysis: Ideas, Examples, Problems, and Prospects," in *Computer Program Testing*, ed. B. Chandrasekaran and S. Radicchi, pp. 129-148, North-Holland, 1981.

Chen80.

Chen, P. P., *Entity-Relationship Approach to Systems Analysis and Design*, North-Holland, 1980.

DeMi78.

DeMillo, R. A., R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *COMPUTER*, vol. 11, no. 4, pp. 34-43, IEEE, April 1978.

DeMi86.

DeMillo, R. A., "Functional Capabilities of a Test and Evaluation Subenvironment in an Advanced Software Engineering Environment," TECHNICAL REPORT GIT-SERC-86/07, Software Engineering Research Center, Georgia Institute of Technology, Atlanta, GA, 1986.

Feld79.

Feldman, S. I., "Make—A Program for Maintaining Computer Programs," *SOFTWARE PRACTICE AND EXPERIENCE*, vol. 9, pp. 255-265, 1979.

Good79.

Goodenough, J. B., and S. L. Gerhart, "Towards A Theory of Test Data Selection," *TRANSACTIONS ON SOFTWARE ENGINEERING*, vol. SE-1, no. 2, pp. 156-173, IEEE, June 1979.

Howd82.

Howden, W. E., "Weak Mutation Testing," *TRANSACTIONS ON SOFTWARE ENGINEERING*, vol. SE-8, no. 4, pp. 371-379, IEEE, July 1982.

Pike84.

Pike, R., "The Blit: A Multiplexed Graphics Terminal," *BELL LABORATORIES TECHNICAL JOURNAL*, vol. 63, no. 8, pp. 1607-1630, AT&T, October 1984.

Sche86.

Scheifler, R. W., and J. Gettys, "The X Window System," *TRANSACTIONS ON GRAPHICS*, no. 63, ACM, 1986.

Wein84.

Weinberger, P. J., "Cheap Dynamic Instruction Counting," *BELL LABORATORIES TECHNICAL JOURNAL*, vol. 63, no. 8, pp. 1815-1826, AT&T, October 1984.

Whit78.

White, L. J., E. I. Cohen, and B. Chandrasekaran, "A Domain Strategy for Computer Program Testing," *TECHINCAL REPORT OSU-CISRC-TR-78-4*, Ohio State University, 1978.

THE VIEWGRAPH MATERIALS
FOR THE
E. SPAFFORD PRESENTATION FOLLOW



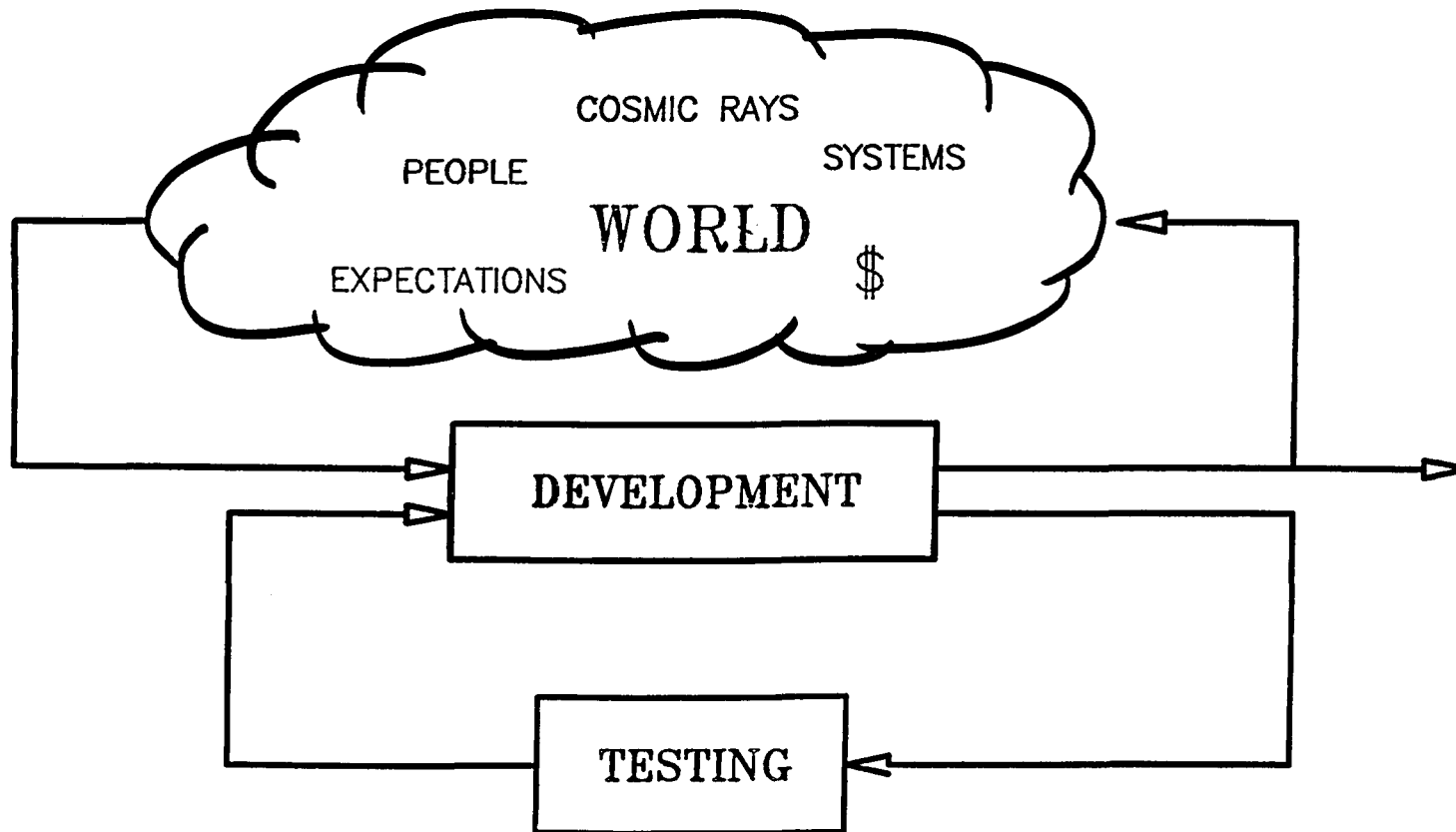
THE **MOTHRA** SOFTWARE TESTING ENVIRONMENT

Software Engineering Research Center
Georgia Institute of Technology
Atlanta, Georgia 30332

TESTING IS A RISK-REDUCING ACTIVITY

- * INCREASE CONFIDENCE THAT SYSTEM BEHAVES AS INTENDED
- * DEMONSTRATE MODES OF FAILURE
- * FIND AND REMOVE INTRINSIC CAUSES OF FAILURE

**STRICTLY SPEAKING,
TESTING IS NOT PART OF THE DEVELOPMENT PROCESS**



TESTING IS A CONTROL ON THE DEVELOPMENT PROCESS

COSTS

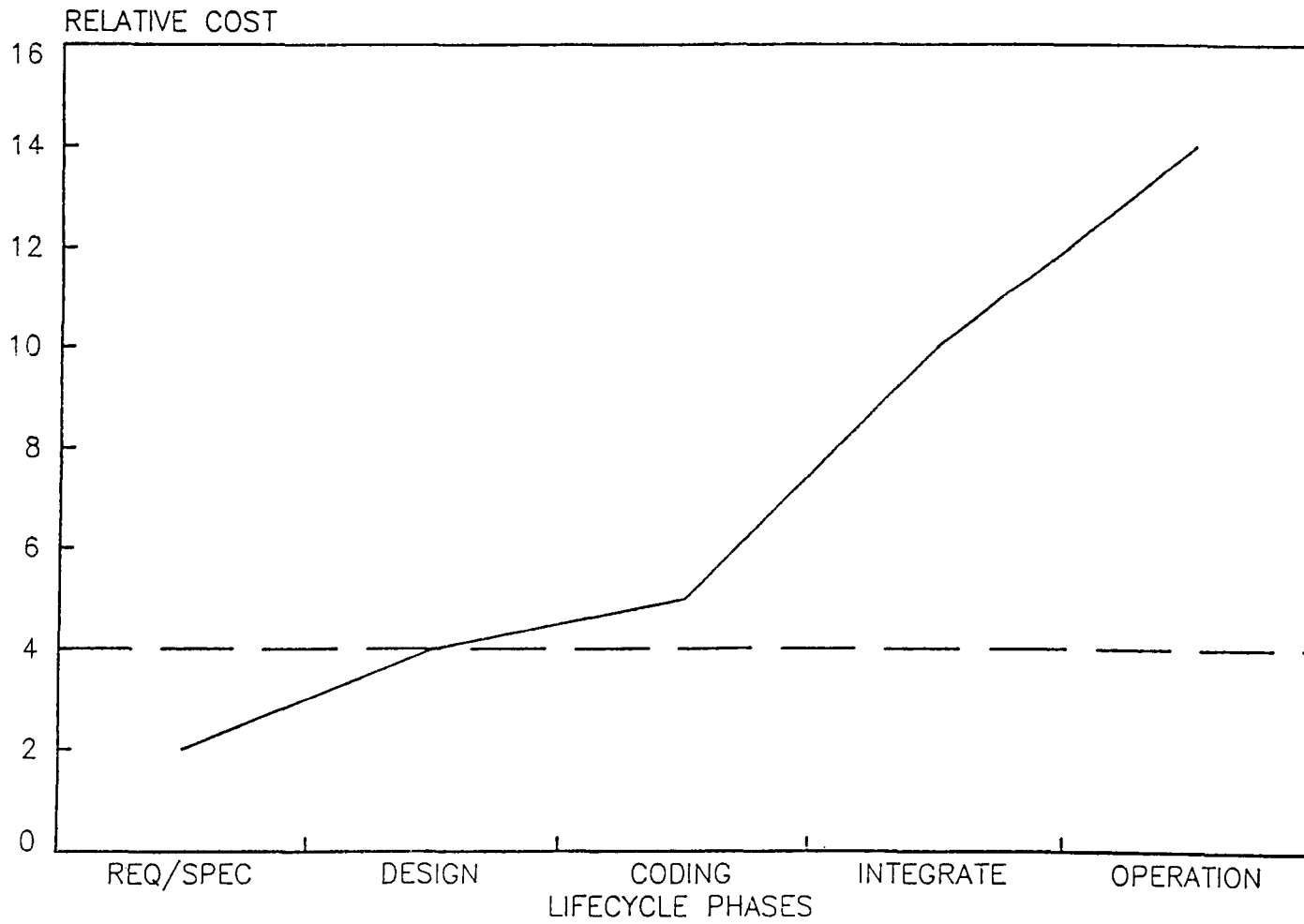
COSTS DUE TO TESTING

COST OF ERRORS

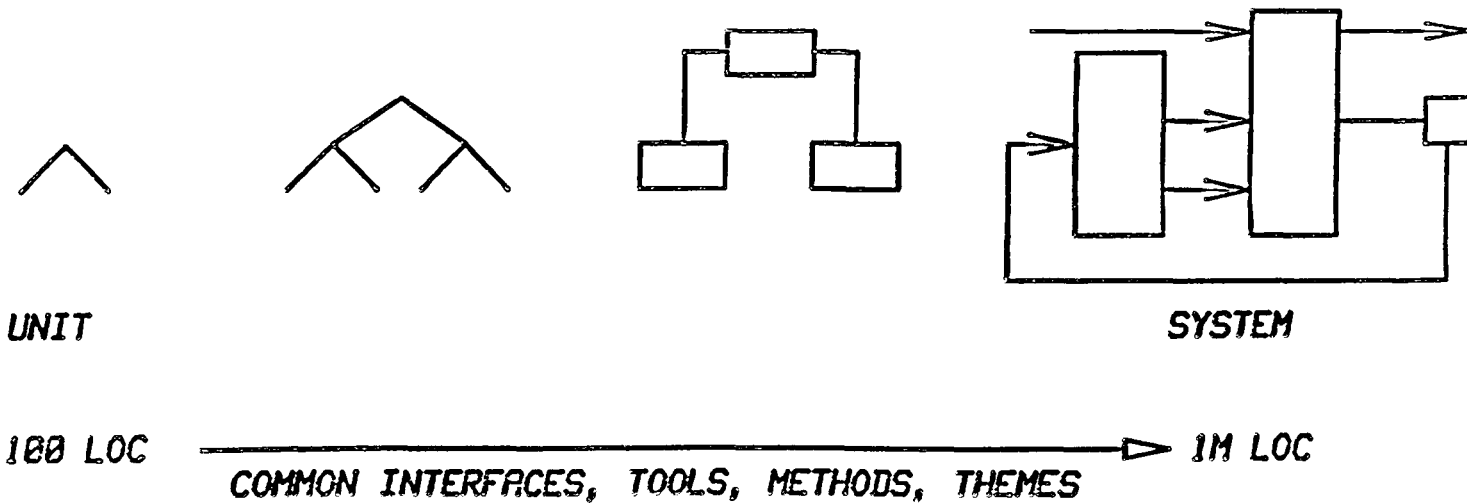
RELATIVE COST OF ERROR CORRECTION

REL. COST

DESIGN COST



THE MOTHRA SOFTWARE TESTING ENVIRONMENT



HIGH BANDWIDTH INTERFACES
GRAPHICS, WINDOWING, ANIMATION, DATA COMPRESSION, OBJECT-ORIENTED

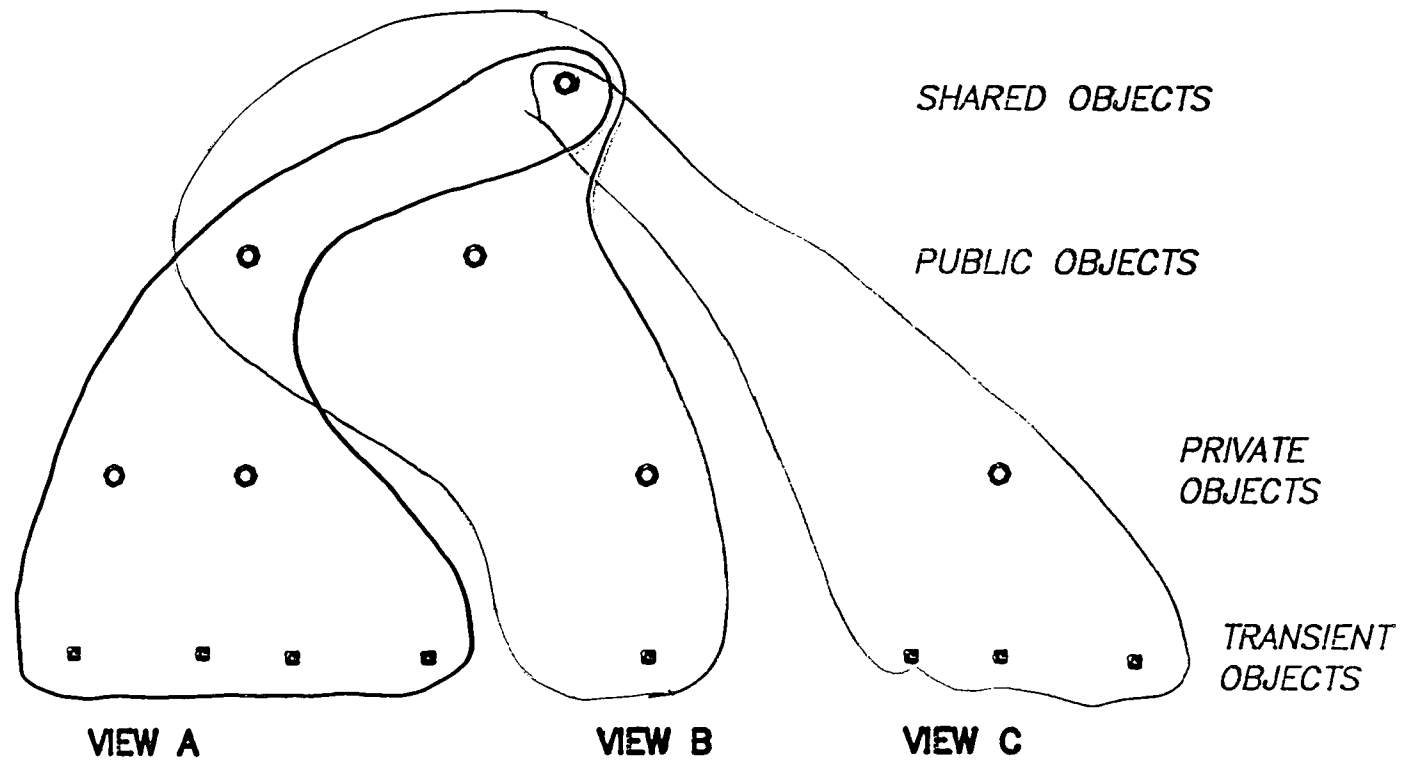
NO A PRIORI SIZE CONSTRAINTS
ATOMIC OPERATIONS ARE LOCAL TRANSFORMATIONS OF EXECUTABLE OBJECTS

THREE ASPECTS OF MOTHRA

- * VIEWS -- A WAY OF MANAGING
LARGE TESTS
- * THEMATIC TOOLS -- A METHODOLOGY FOR
USER INTERACTIONS
- * SHIFTING GEARS -- CAPITALIZING
THE TEST AT AN APPROPRIATE LEVEL

VIEWS OF LARGE TESTS

BOUNDARIES OF BITMAP DISPLAY DEFINE A WINDOW INTO
A MORE GLOBAL TEST CONTEXT



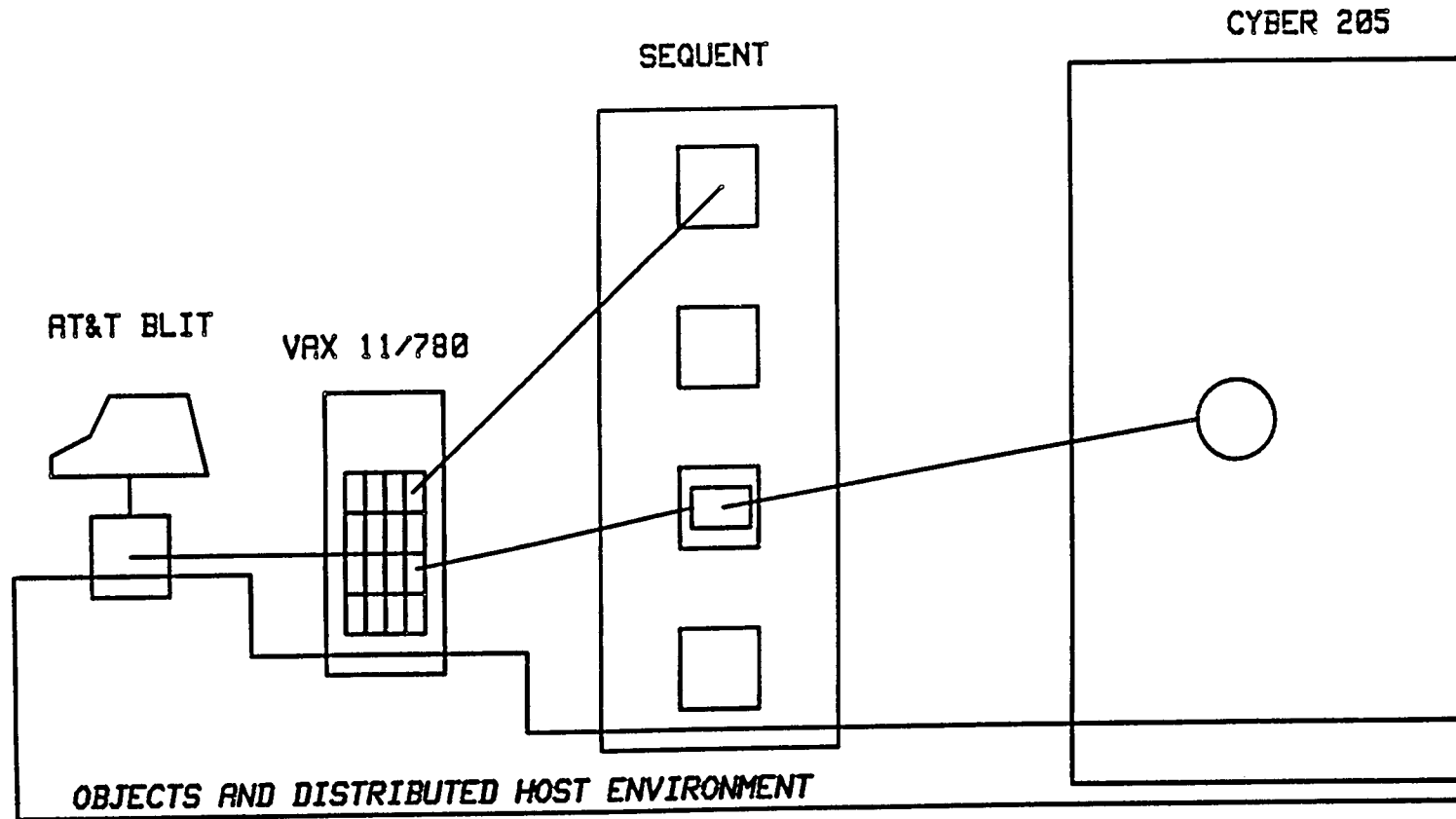
VIEW DEFINE CONSISTENT, MEANINGFUL STATES FOR MOTHRA

COMMON TOOL THEMES

- * *INSTRUMENTATION*
 - * *PI-LIKE SYMBOLIC DEBUGGERS*
 - * *ZERO-IMPEDANCE PROBES*

- * *EDITING*
 - * *TEST ADVISOR*
 - * *THE ORACLE*

SHIFTING GEARS IS EASY
ONCE THE ENVIRONMENT IS CAPITALIZED



1 MIP ▶ 200 MIPS

INCREASED COST
INCREASED FIDELITY
INCREASED SIZE

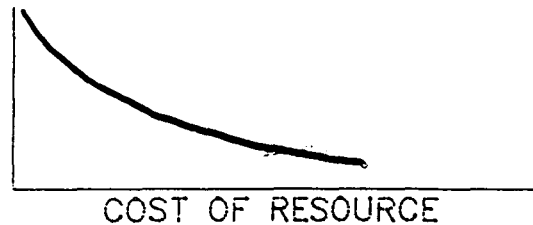
INCREMENTAL COST OF TEST

$$C(R,P,L,S) = \frac{\text{COST OF TESTING P AT LEVEL L AND STRENGTH S ON R}}{\text{COST OF RUNNING P ON R}}$$

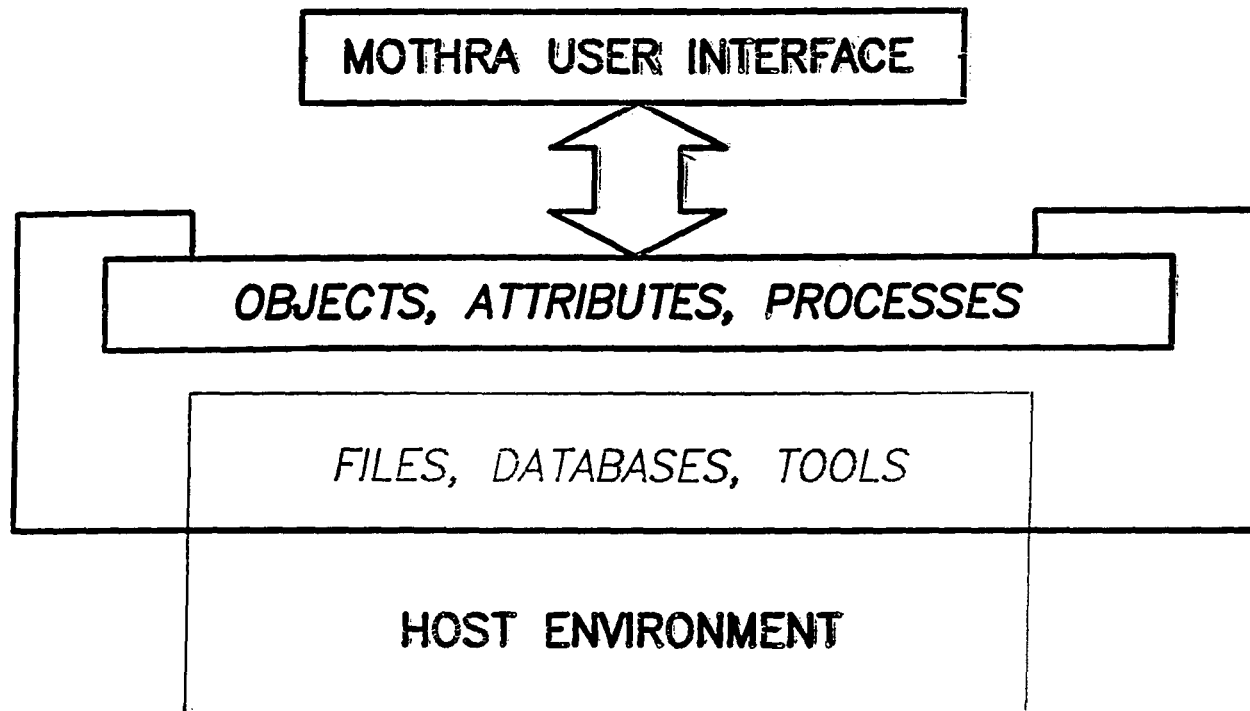
*FIRST SET OF EXPERIMENTAL RESULTS
(NOT TO BE GENERALIZED)*

$$\begin{aligned} C(\{\text{VAX 780, CYBER 205}\}, P, \{\text{sal, pdl, ccl}\}, 1.00) &= 1.02 \\ C(\{\text{VAX 780}\}, P, \{\text{sal, pdl, ccl}\}, 1.00) &> 500 \end{aligned}$$

INCREMENTAL
COST OF TEST



MOTHRAS IS DESIGNED TO BE A SUBENVIRONMENT



- * *4.2 BSD ON VAX 11/780 (Layers on Blit)*
- * *ULTRIX ON VAXStation II (X-Windows)*

TECHNIQUE

*Given: program P
test data T*

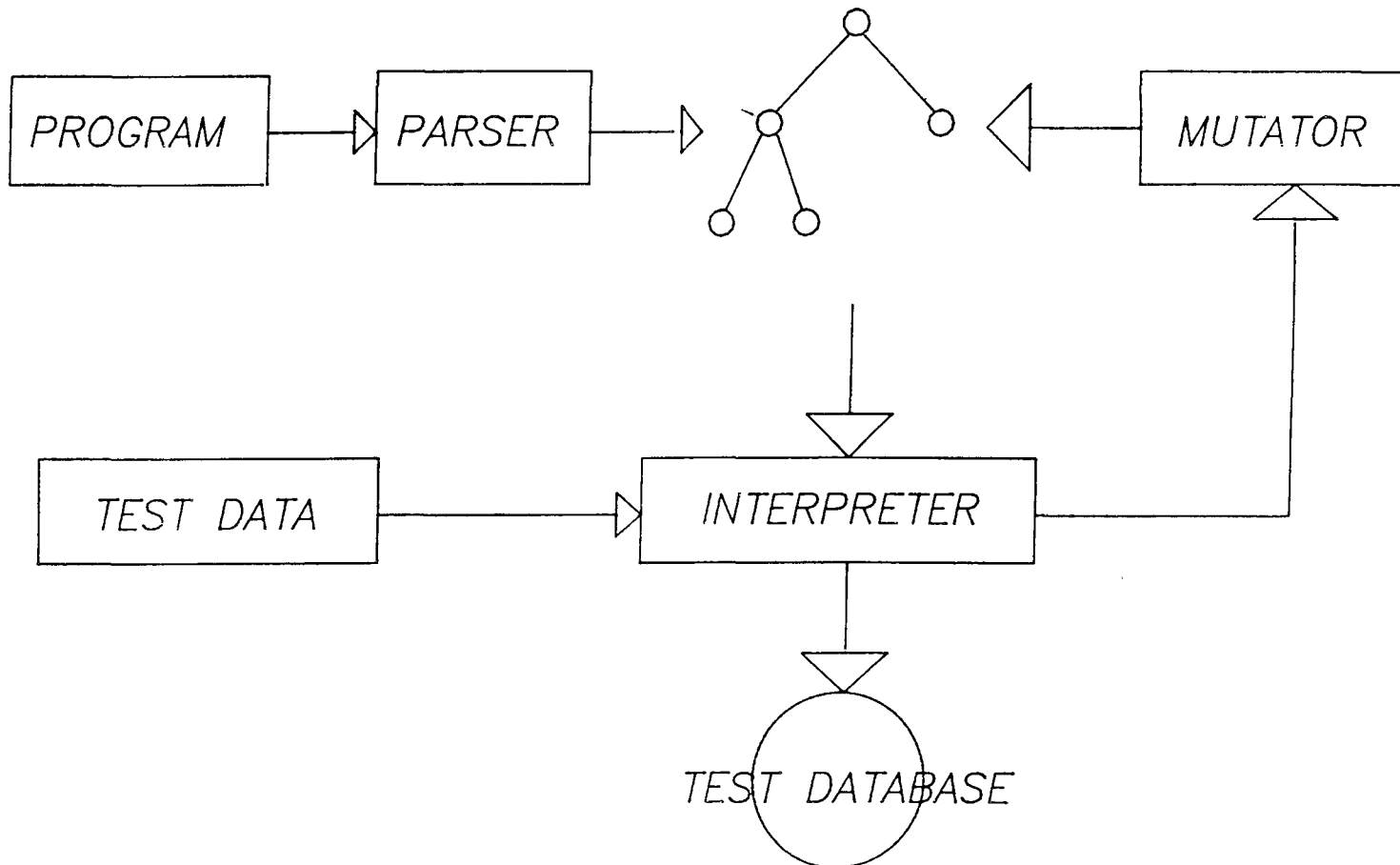
*Construct: a set of MUTANT
programs $M(P)$*

$M(P, T) = \% \text{ of programs in } M(P)$

(1) not equivalent to P

(2) give different results on T

AUTOMATIC MUTATION SYSTEM



Enter Experiment Name:									
Level:	sal	pdl	cdl	Test Strength: 100%					
Class:	ary	con	ctl	dmn	opm	prd	scl	stm	ell
MOTHR Fortran-77 Mutation System									
Software Engineering Research Center Georgia Institute of Technology Atlanta, GA 30332									

ALL Array Mutant Types

Levels: pdl ccl Test Strength: 100%

Class: con dnn omm prd scl sta all

Types:

```

INTEGER A(10),N
DO 200 J=1,N-1
  DO 100 J=1,N-J
    IF (A(I).LE.A(J)) GOTO 100
    TEMP=A(I)
    A(I)=A(J)
    A(J)=TEMP
100  CONTINUE
200  CONTINUE
RETURN
END

SUBROUTINE SELECT(L,N)
INTEGER L(N),N
INTEGER I,J,MAX,TEMP

J=N
IF (J.LT.2) GOTO 99
MAX=J
I=J-1
5  IF (I.LT.1) GOTO 20
  IF (L(I).LE.L(MAX)) GOTO 10
  MAX=I
  I=I-1
  GOTO 5
20  TEMP=L(MAX)
  L(MAX)=L(J)
  L(J)=TEMP
  J=J-1
  GOTO 1
99  RETURN
END

SUBROUTINE INSERT (L,N)
INTEGER L(N),N
INTEGER KEY,I,J

J=2
IF (J.GT.N) GOTO 99
KEY=L(J)
I=J-1

```

Directory
 Edit Source
 Display Symbol Table
 Display CodeFile

absc 56%

acrc 82%

acrs 92%

acrc 82%

Mutant Profiles
 Specify Mutants
 Experiment State
 Test Histograms

for L : 2
 Enter 2 values for the
 array L: 0 -1
 Enter value for N: 2
 Test Case #5
 Running Original Program
 Input Values for test case 5.
 L 0 -1
 N 2
 Execution stopped because:
 Normal Termination.
 There were 22 statements
 executed.
 Output Val: Directory
 Select Test Case #5
 Query
 Edit Test Case
 L -1 0
 N 2
 Is this the correct output.

Level: sol pdl ccl **Test Strength: 1003**

Class: ary con cti dmn opa prd scl stl otl

```

INTEGER A(10),N
DO 200 J=1,N-1
DO 100 J=1,N-J
IF (A(I).LE.A(J)) COTO 100
TEMP=A(I)
100 CONTIN
200 RETURN
END

```

running

Initializing...

argv[0] = pd.m

()ints: ClrWindow=<function?>

()ints: MutleyEnabled=<function?>

getshort()

getstr()

putbuf()

putchar()

putlong()

putnchar()

putshort()

putstr()

printf()

putc()

putchar()

puts()

sprintf()

strlen()

sync()

clr()

clrEol

ClrWindow

ConMsg

DrawBarGraph

Draw_HIL

Drect

Help

InitBarGraph

InitMutTypes

InitMuts

InitSysMon

Initialize

Jdisplay

Jrect

MousePos

ConMsg

DrawBarGraph

Draw_HIL

Drect

Help

InitBarGraph

InitMutTypes

InitMuts

InitSysMon

Initialize

Jdisplay

Jrect

MousePos

MutleyEnabled

the

Z

ogram

st. case 5.

because

on.

ments

st case #5.

L-1 0

N-2.

Is this the correct output.

TECHNICAL APPROACH

- * *BUILD PROTOTYPE BASED ON PROGRAM MUTATION APPROACH TO TESTING*
 - o *SYSTEMATIC AND QUANTITATIVE*
 - o *WELL-DEVELOPED THEORETICAL BASIS*
 - o *EXTENSIVE EXPERIMENTAL VALIDATIONS*
 - o *SCALES UP*
 - o *IDEAL FOR SUPERCOMPUTER IMPLEMENTATION*

- * *ADAPT PROTOTYPE TO ADA*

- * *CONDUCT FEASIBILITY DEMONSTRATIONS*

PROJECT STATUS

- * *TWO IMPLEMENTATIONS OF VERSION 1
INSTALLED AND BEING TESTED*
 - PROCESSES COMPLETE FORTRAN 77 LANGUAGE
 - LAYERS (VAX 11/780) AND X-WINDOWS (VAXStations)

- * *ADA CAPABILITY DESIGNED*
 - ERROR OPERATORS DEFINED
 - ARCHITECTURE SPECIFIED

- * *PERFORMANCE STUDIES INITIATED*
 - VERSION 1 OPTIMIZATION
 - SUPER COMPUTER IMPLEMENTATIONS

A Value-Chain Analysis of Software Productivity Components

Barry W. Boehm, TRW Inc.
Phillip N. Papaccio, TRW Inc.

Summary

This paper summarizes a recent value-chain analysis of software productivity components at TRW. It explains the various value chain components and percentages, assesses their implications for improving software productivity, and elaborates on some further data analysis performed to address one of the major value chain components: rework costs.

1. The Software Product Value Chain

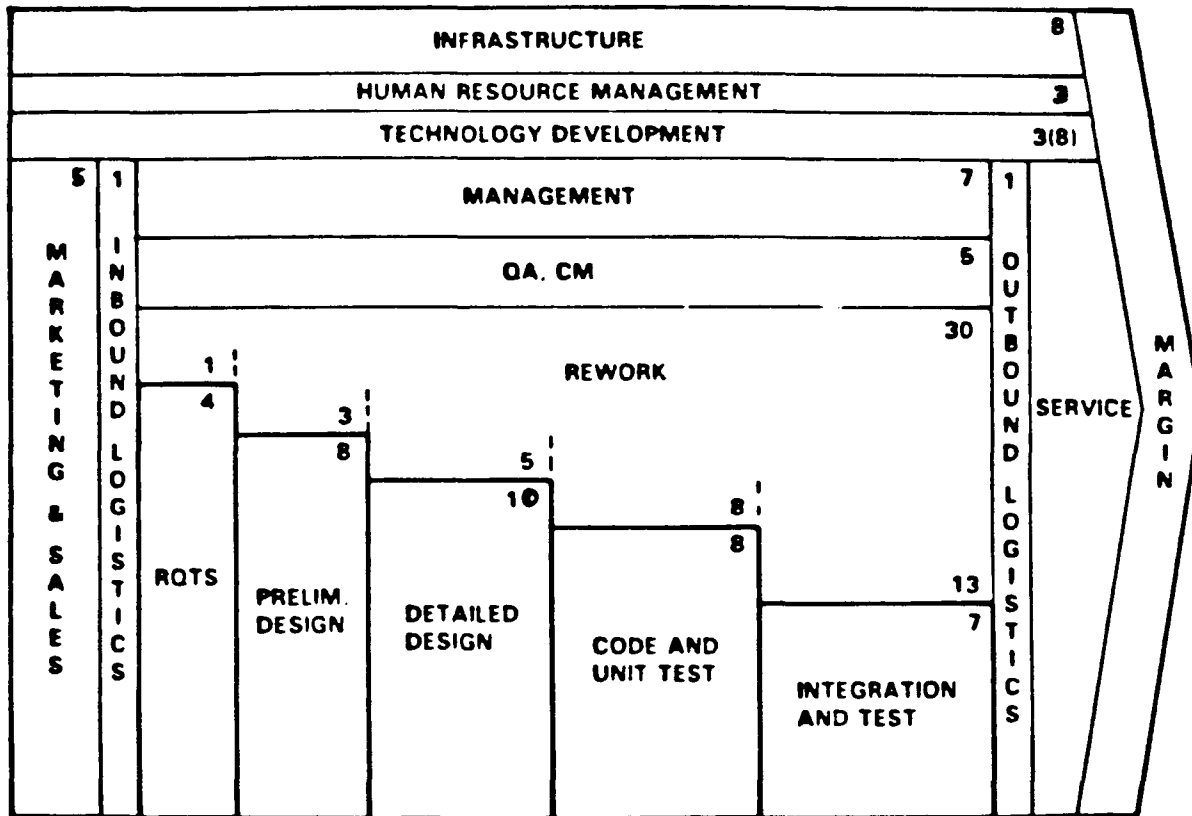
The value chain, developed by Porter and his associates at the Harvard Business School [Porter, 1980; Porter, 1985], is a useful method of understanding and controlling the costs involved in a wide variety of organizational enterprises. It identifies a canonical set of cost sources or value activities, representing the basic activities an organization can choose from to create added value for its products. Figure 1 shows a value chain for software development representative of experience at TRW. Definitions and explanations of the component value activities are given below. These are divided into what [Porter, 1985] calls primary activities (inbound logistics, outbound logistics, marketing and sales, service, and operations) and support activities (infrastructure, human resource management, technology development, and procurement).

Primary Activities

Inbound logistics covers activities associated with receiving, storing, and disseminating inputs to the products. This can be quite large for a manufacturer of, say, automobiles; for software it consumes less than 1% of the development outlay. (For software, the related support activity of *procurement* is also included here).

Outbound logistics covers activities concerned with collecting, storing, and physically distributing the product to buyers. Again, for software, this consumes less than 1% of the total.

Figure 1. Software Development Value Chain



Marketing and sales covers activities associated with providing a means by which buyers can purchase the product and inducing them to do so. A 5% figure is typical of government contract software organizations. Software product houses would typically have a higher figure; internal applications-programming shops would typically have a lower figure.

Service covers activities associated with providing service to enhance or maintain the value of the product. For software, this comprises the activities generally called software maintenance or evolution. For simplicity, Figure 1 avoids including a service cost component in the development value chain; a life-cycle value chain is presented and discussed as Figure 2 below.

Operations covers activities associated with transforming inputs into the final product form. For software, operations typically involves roughly four-fifths of the total development outlay.

In such a case, the value-chain analysis involves breaking up a large component into constituent activities. Figure 1 shows such a breakup into management (7%), quality assurance and configuration management (5%), and the distribution of technical effort among the various development phases. This phase breakdown also covers the cost sources due to rework. Thus, for, example, of the 20% overall cost of the technical effort during the integration and test phase, 13% is devoted to activities required to rework deficiencies in or reorientations of the requirements, design, code, or documentation; the other 7% represents the amount of effort required to run tests, perform integration functions, and complete documentation even if no problems were detected in the process.

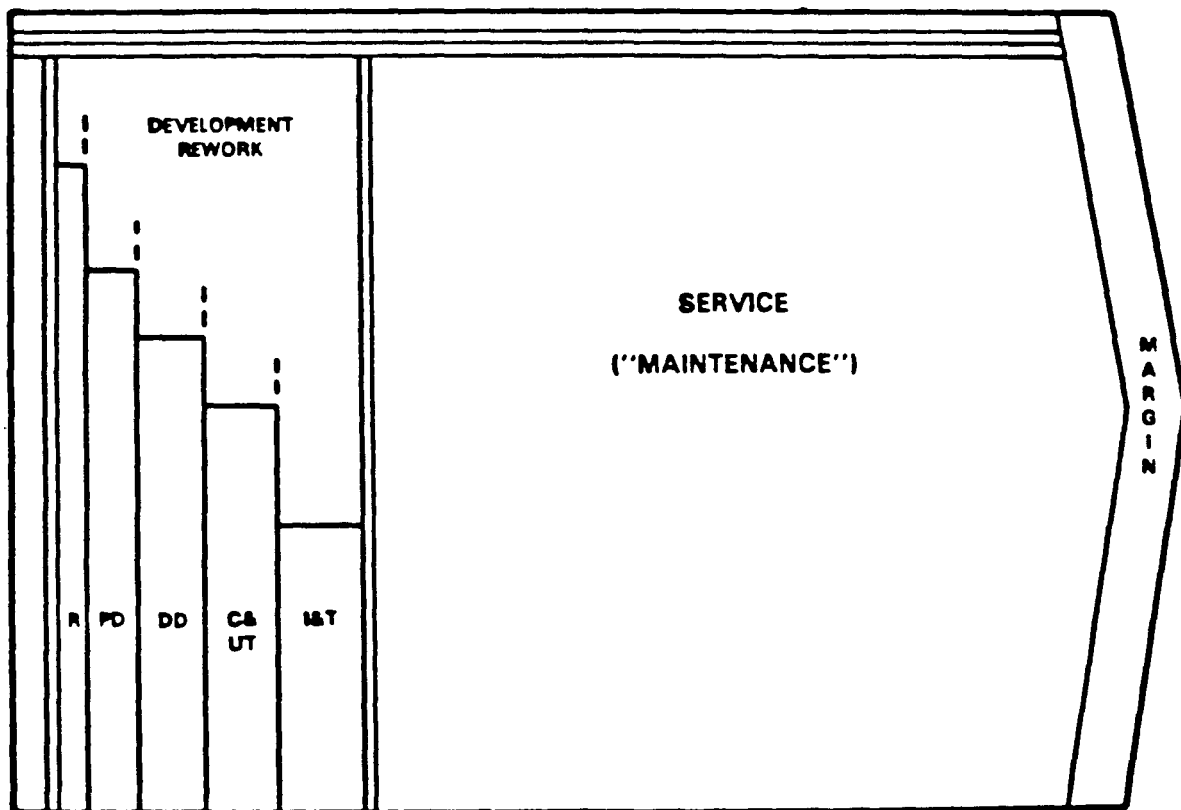
Support Activities

Infrastructure covers such activities as the organization's general management planning, finance, accounting, legal, and government affairs. The 8% figure is typical of most organizations.

Human resource management covers activities involved in recruiting, hiring, training, development, and compensation of all types of personnel. Given the labor-intensive and technology-intensive nature of software development, the 3% figure indicated here is a less-than-optimal investment.

Technology development covers activities devoted to creating or tailoring new technology to improve the organizations products or processes. The 3% investment figure here is higher than many software organizations, but still less than

Figure 2. Software Life-Cycle Value Chain



optimal as an investment to improve software productivity and quality.

Margin and Service

Margin in the value chain is the difference between the value of the resulting product and the collective cost of performing the value activities. As this difference varies widely among software products, it is not quantitatively defined in Figure 1. As discussed above, *service* is best quantified as a software life-cycle value chain as shown as Figure 2, with roughly 70% of the value activity devoted to service or evolution-related activity. However, since the component activities involved during evolution do not differ markedly from those which go on during software development, we will continue to focus on Figure 1 as a source of insights into understanding and controlling software costs.

Software Development Value Chain Implications

The primary implication of the software development value chain is that the "Operations" component is the key to significant improvements. Not only is it the major source of software costs, but also most of the remaining components such as "Human Resources" will scale down in a manner proportional to the scaling down of Operations cost.

Another major characteristic of the value chain is that virtually all of the components are still highly labor-intensive. Thus, there are significant opportunities in providing automated aids to make these activities more efficient and capital-intensive. Further, it implies that human-resource and management activities have much higher leverage than their 3% and 7% investment levels indicate.

The breakdown of the Operations component indicates that the leading strategies for cost savings in software development involve:

- *Making individual steps more efficient*, via such capabilities as automated aids to software requirements analysis or testing.
- *Eliminating steps*, via such capabilities as automatic programming or automatic quality assurance.
- *Eliminating rework*, via early error detection, or via such capabilities as rapid prototyping to avoid later requirements rework.

In addition, further major cost savings can be achieved by reducing the total number of elementary Operations steps, by developing products requiring the creation of fewer lines of code. This has the effect of reducing the overall size of the Value Chain itself. This source of savings breaks down into two primary options:

- *Building simpler products*, via more insightful front-end activities such as prototyping or risk management.
- *Reusing software components*, via such capabilities as fourth-generation languages or component libraries.

2. The Software Productivity Improvement Opportunity Tree

This breakdown of the major sources of software cost savings leads to the *Software Productivity Improvement Opportunity Tree* shown in Figure 3. This hierarchical breakdown helps us to understand how to fit the various attractive productivity options into an overall integrated software productivity improvement strategy.

Further discussions of the various productivity options are provided in [Boehm, 1986a]. As one example involving further data analysis, we studied the distribution of rework costs on a sample of 1378 problem reports on two large TRW software projects. These studies indicated that rework instances tend to follow a Pareto distribution: 80% of the rework costs typically result from 20% of the problems. Figure 4 shows some typical distributions of this nature from recent TRW software projects; similar trends have been indicated in [Rubey et al, 1975],[Formica, 1978], and [Basili-Weiss, 1981]. The major implication of this distribution is that software verification and validation activities should focus on identifying and eliminating the specific *high-risk* problems to be encountered by a software project, rather than spreading their available early-problem-elimination effort uniformly across trivial and severe problems. Even more strongly, this implies that a *risk-driven* approach to the software life-cycle such as the spiral model [Boehm, 1986] is preferable to a more *document-driven* model such as the traditional waterfall model.

Figure 3. Productivity Improvement Opportunity Tree

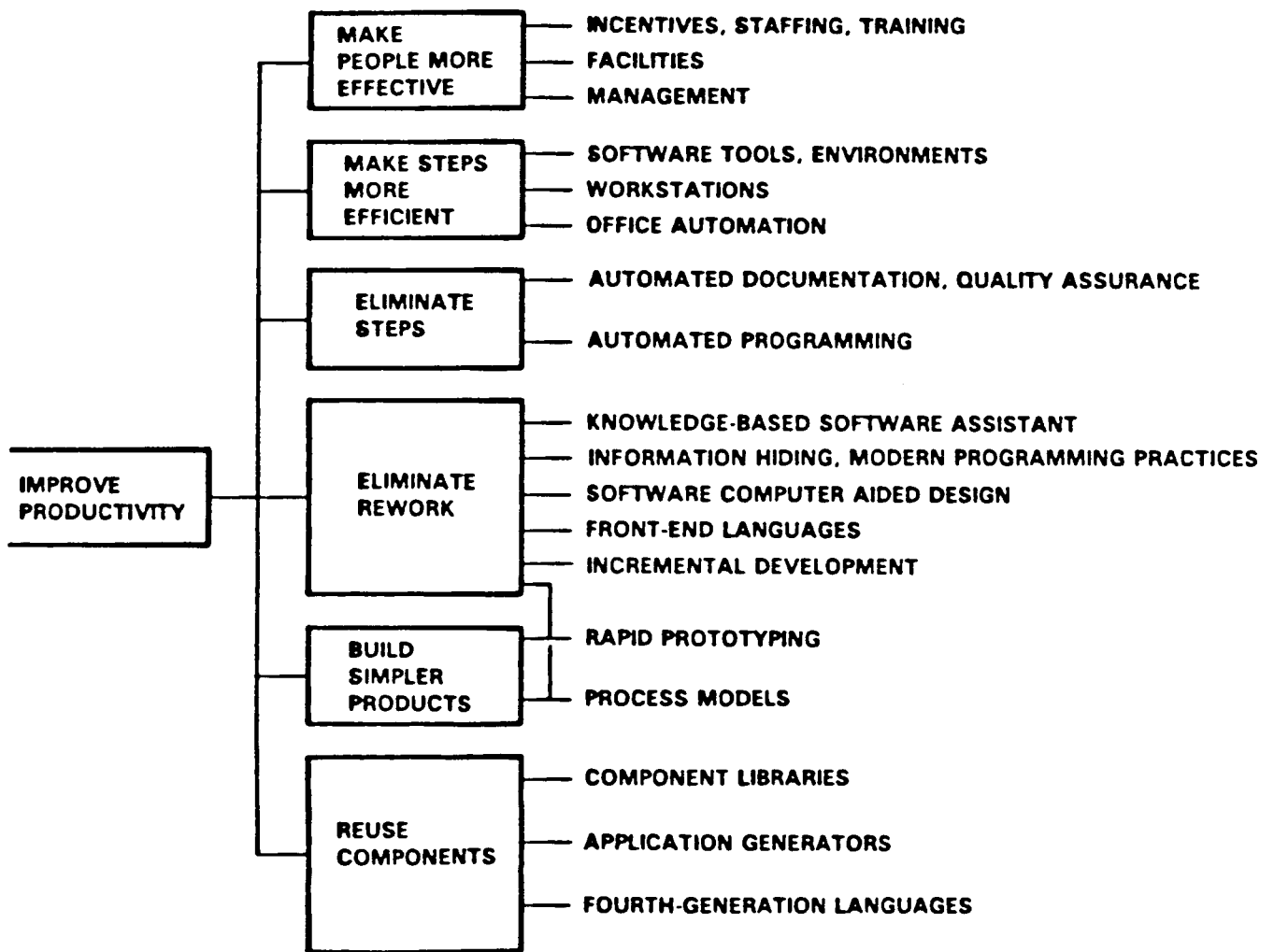
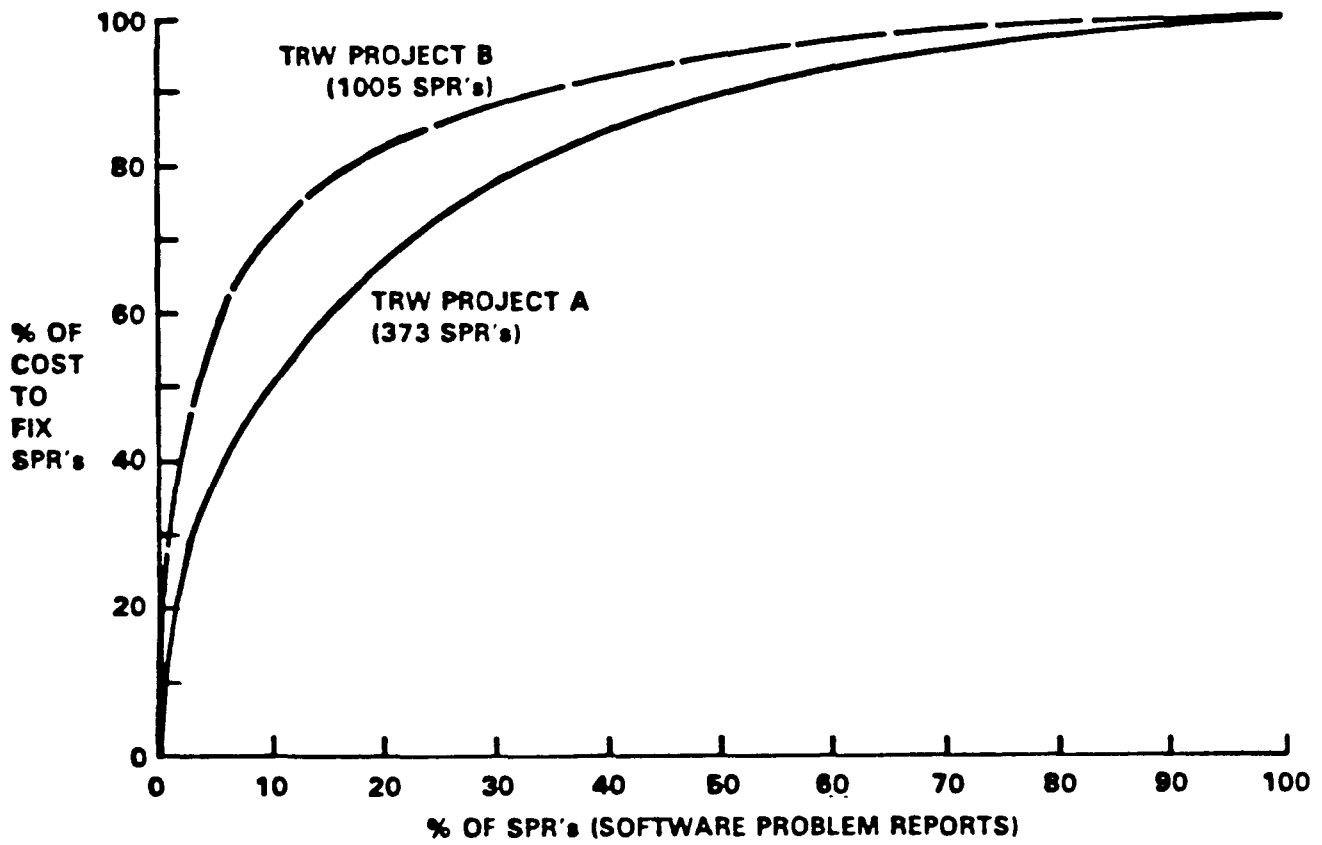


Figure 4. Rework Costs are Concentrated in a Few High-Risk Items



[Basili-Weiss, 1981]. V. R. BASILI and D. M. WEISS, "Evaluation of a Software Requirements Document by Means of Change Data", *Proceedings, Fifth International Conference on Software Engineering*, IEEE, March 1981, pp. 314-323.

[Boehm, 1986]. B. W. BOEHM, "A Spiral Model of Software Development and Enhancement", *Proceedings, IEEE Second Software Process Workshop, ACM Software Engineering Notes*, March 1986.

[Boehm, 1986a]. B. W. BOEHM, "Understanding and Controlling Software Costs", *Proceedings, IFIP 86*, North Holland, 1986.

[Formica, 1978]. G. FORMICA, "Software Management by the European Space Agency: Lessons Learned and Future Plans", *Proceedings, Third International Software Management Conference, AIAA/RAeS*, London, October 1978, pp. 15-35.

[Porter, 1980]. M. E. PORTER, *Competitive Strategy: Techniques for Analyzing Industries and Competitors*, New York: The Free Press, 1980.

[Porter, 1985]. M. E. PORTER, *Competitive Advantage*, New York: The Free Press, 1985.

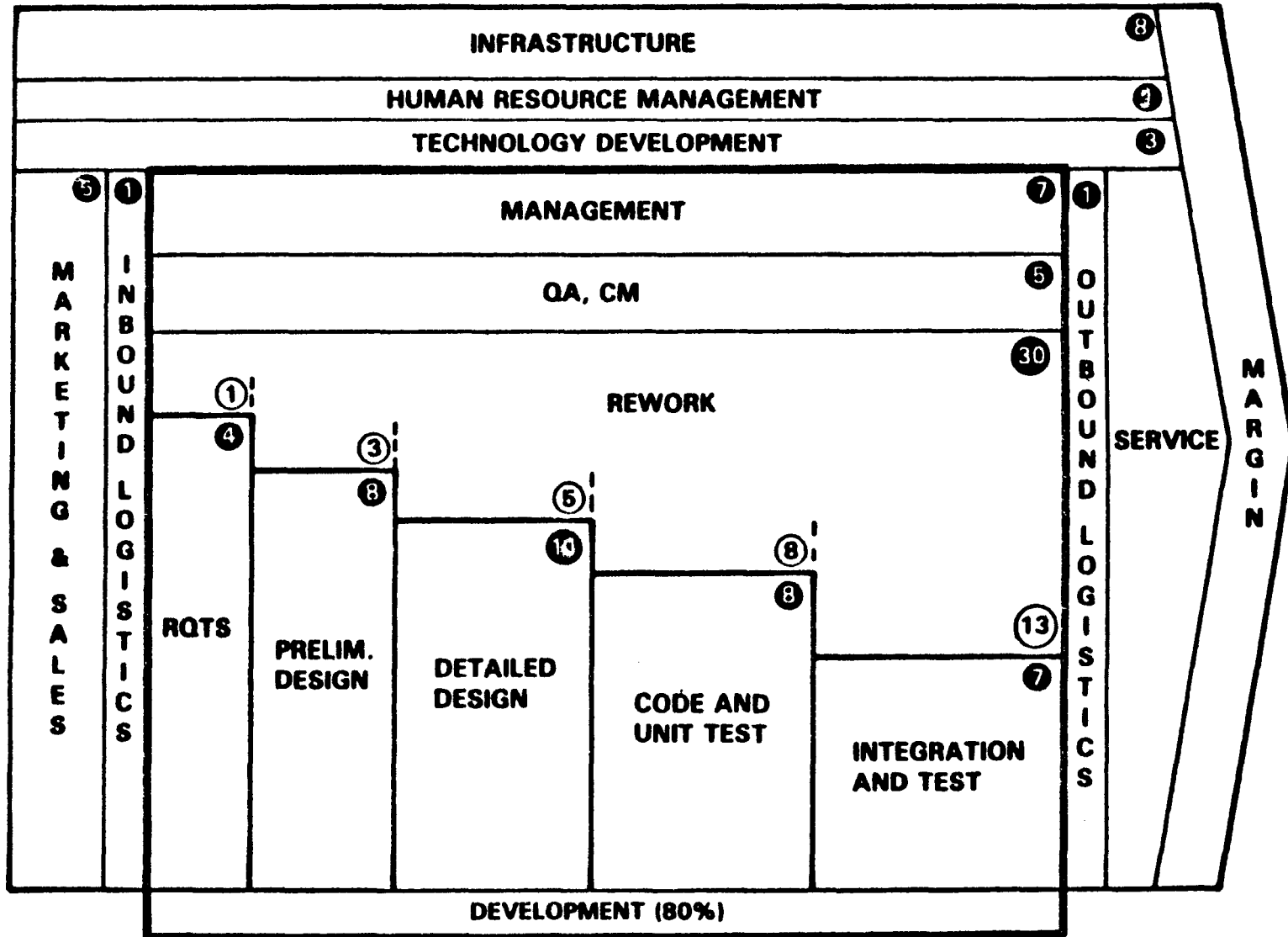
[Rubey et al, 1975]. R. J. RUBEY, J. A. DANA, and P. W. BICHE, "Quantitative Aspects of Software Validation", *IEEE Trans. Software Engineering*, June 1975, pp. 150-155.

A VALUE CHAIN ANALYSIS OF SOFTWARE PRODUCTIVITY COMPONENTS

**BARRY BOEHM
PHIL PAPACCIO, TRW**

- **SOFTWARE VALUE CHAIN ANALYSIS**
- **SOFTWARE PRODUCTIVITY OPPORTUNITY TREE**
- **TRW QUANTUM LEAP PROGRAM IMPLICATIONS**
- **RISK MANAGEMENT IMPLICATIONS**
- **THE SPIRAL MODEL OF THE SOFTWARE PROCESS**

SOFTWARE DEVELOPMENT VALUE CHAIN

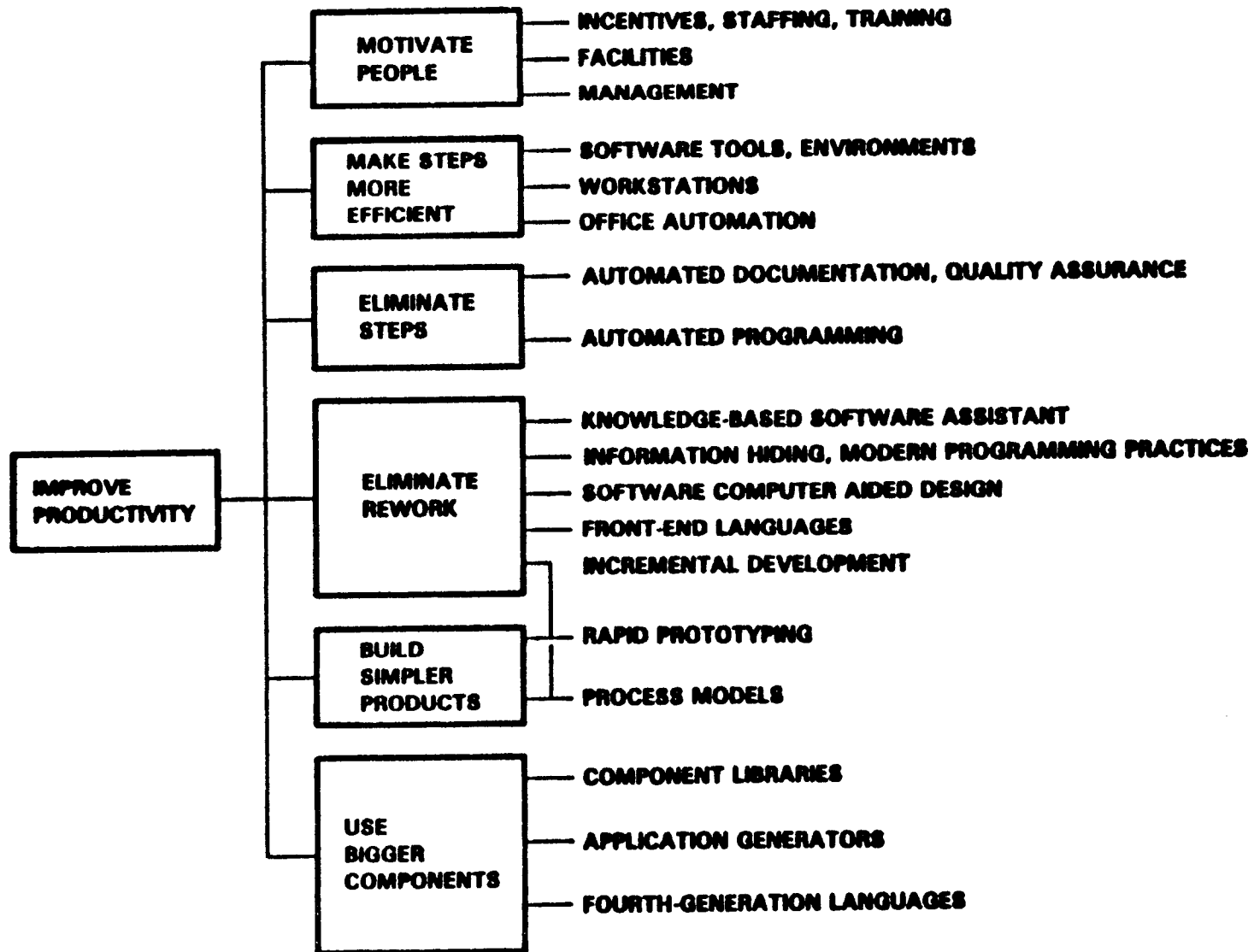


VALUE CHAIN CHARACTERISTICS

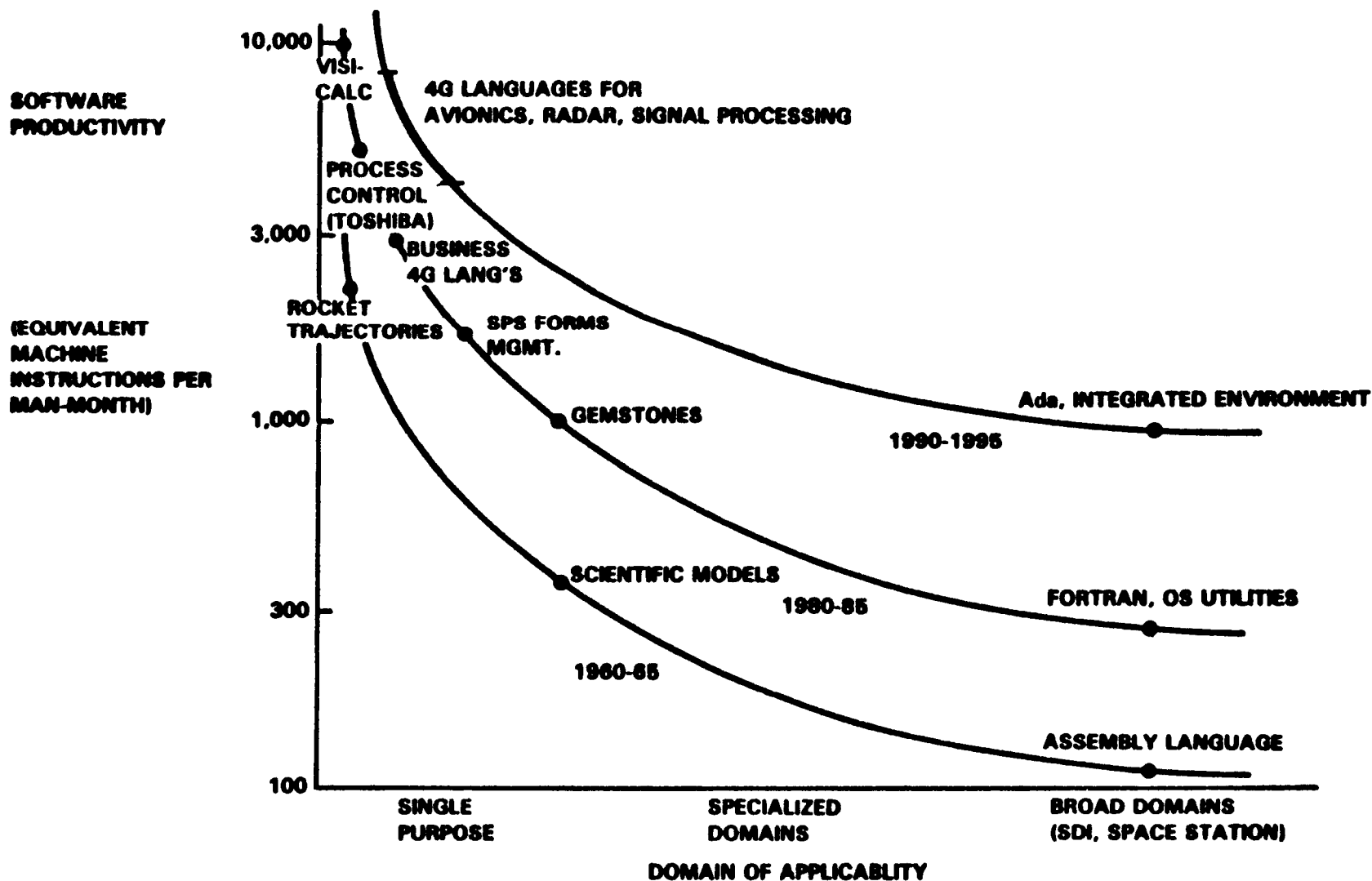
- **DEVELOPMENT AREA IS KEY**
 - **LARGEST COST SOURCE**
 - **LABOR-INTENSIVE**
 - **MOST OTHER PARTS SCALE TO DEVELOPMENT COSTS**

- **BIGGEST OPPORTUNITIES**
 - **ELIMINATE REWORK**
 - **ELIMINATE STEPS**
 - **MAKE STEPS MORE EFFICIENT**
 - **MOTIVATE PEOPLE**
 - **DEVELOP LESS CODE**
 - USE BIGGER COMPONENTS**
 - BUILD SIMPLER PRODUCTS**

PRODUCTIVITY IMPROVEMENT OPPORTUNITY TREE



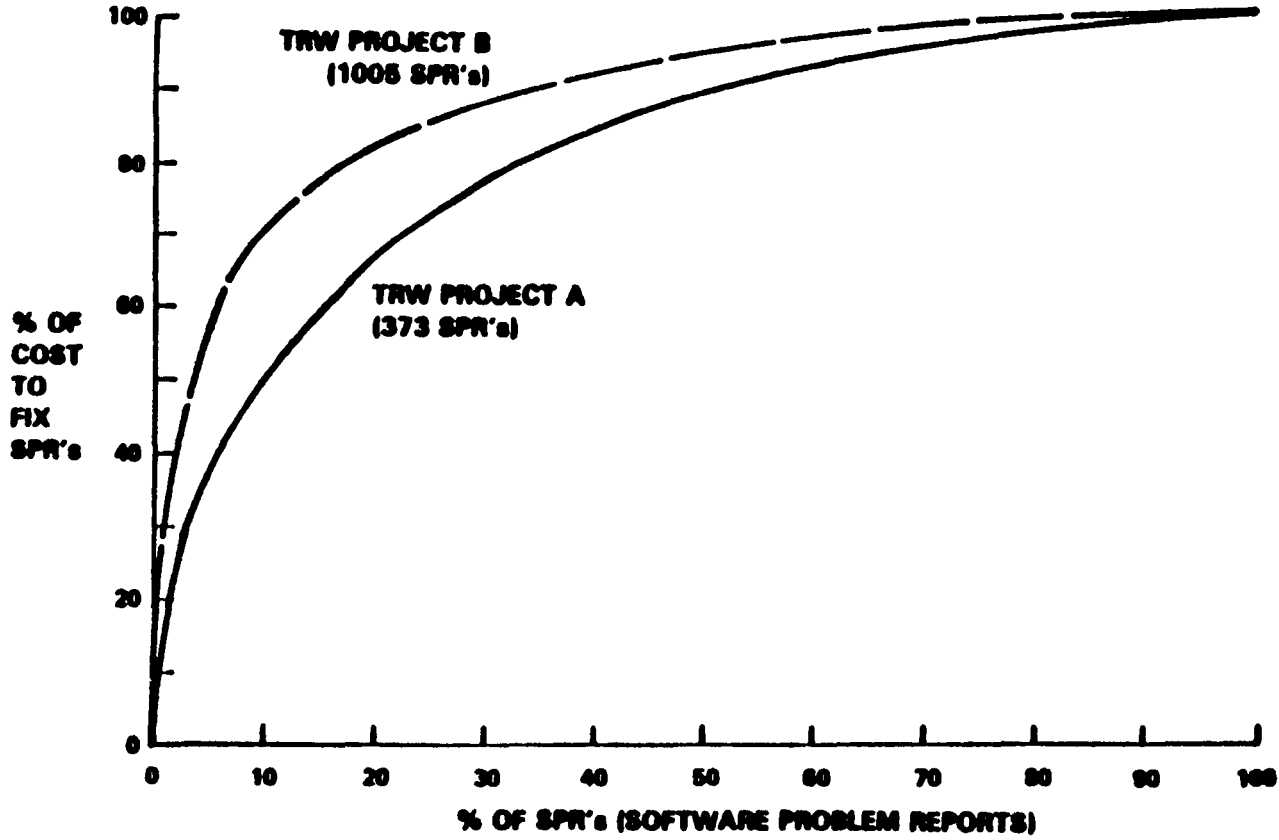
SOFTWARE TECHNOLOGY AND PRODUCTIVITY TRENDS



Keys to Systems and Software Quantum Leap

- Reusable software components
 - Component library
 - 4th generation languages for some DSG applications (factor of 10 improvements possible)
- Eliminating rework and scrap
 - Rapid prototyping, CASE, SCAD
- Fully integrated support environment
 - Knowledge-based software assistant
 - Transition to ADA programming language
- Complementary rethinking of development process
 - Methods and metrics
 - Management approach

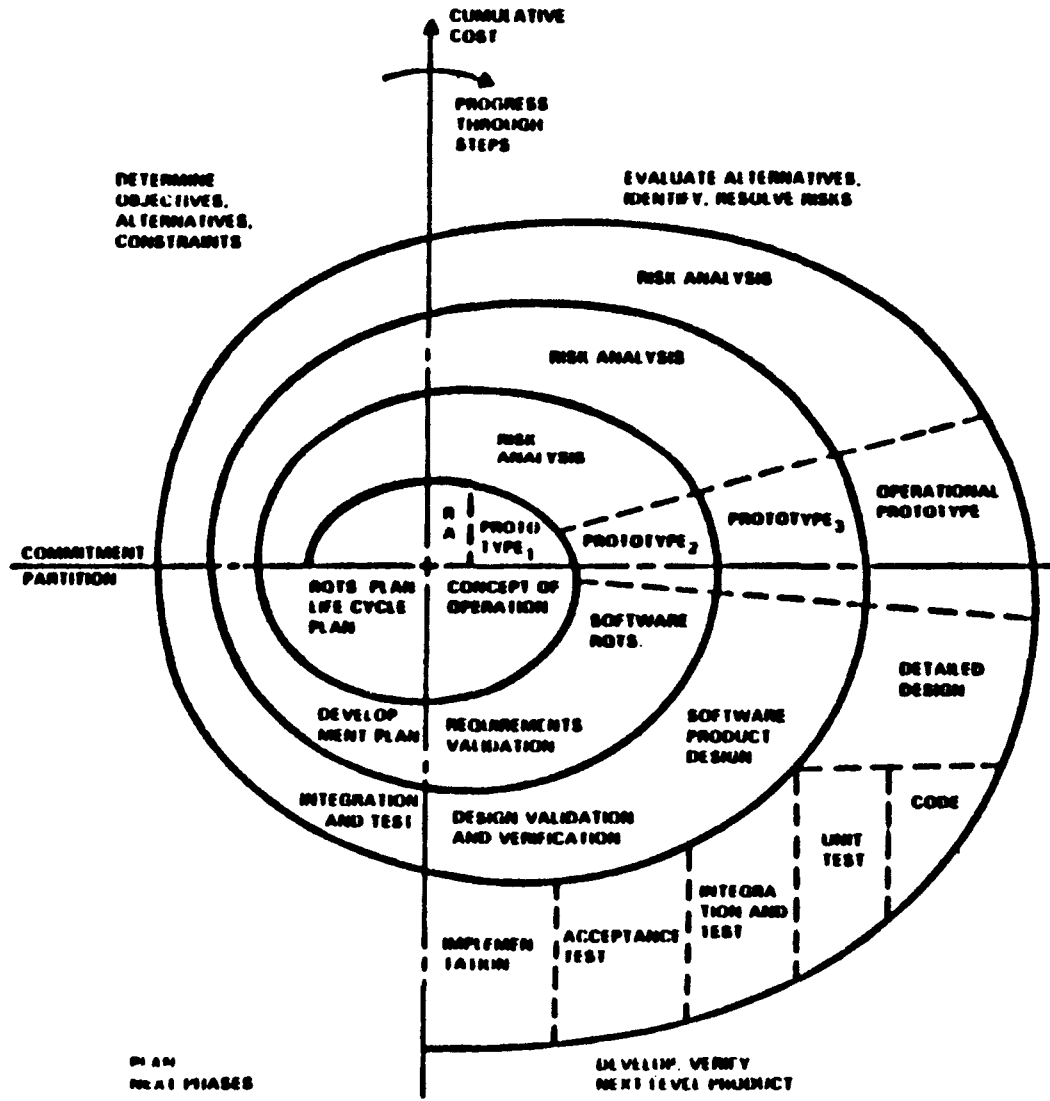
REWORK COSTS ARE CONCENTRATED IN A FEW HIGH-RISK ITEMS



SPIRAL MODEL OF SOFTWARE PROCESS

- **CYCLICAL VS. LINEAR PROCESS**
- **INDUCTIVE, SYNTHETIC VS. DEDUCTIVE, ANALYTIC**
 - **ACCOMMODATES REUSE, PROTOTYPING**
- **RISK-DRIVEN VS. DOCUMENT-DRIVEN**
- **RISK CONSIDERATIONS DETERMINE MIX OF ACTIVITIES**
 - **WATERFALL, EVOLUTIONARY DEVELOPMENT ARE SPECIAL CASES**

SPIRAL MODEL OF SOFTWARE PROCESS



Visible Connections: The Open Architecture of the Software through Pictures Environment

Anthony I. Wasserman

Interactive Development Environments, Inc.
150 Fourth Street, Suite 210
San Francisco, CA 94103

Summary

Interactive Development Environments has created Software through Pictures™, an integrated ensemble of tools, on the principles of "open architecture". A set of graphical editors support methods for software analysis and design, including Structured Systems Analysis, Structured Design, Entity-Relationship modeling, and User Software Engineering (USE). The tools store information in a common data repository, called the IDE Data Dictionary, and allow users to structure their work by systems within projects.

The editors are supported by a variety of tools for such tasks as completeness and consistency checking of diagrams, generation of code and code skeletons, and generation of picture descriptions in pic or PostScript™. There are approximately 90 such support tools for the six editors, including the IDEtool that provides a mouse-and-menu interface to the entire environment.

A fundamental decision in the design of the Software through Pictures environment was to build upon an "open architecture" for software. Open architecture is a well-understood concept in computer hardware, but we wanted to apply it as completely as possible to software.

We established the following criteria for our open architecture, which we named Visible Connections™:

- (1) it should be possible to invoke *every component* independently; users should have information about all of the arguments and options that can be provided to each tool;
- (2) all interfaces to the tools and to their inputs and outputs should be published so that other tools can build upon those interfaces;
- (3) the database schema used by the IDE data dictionary should be visible and extensible.
- (4) all files used to produce user-visible messages, such as error messages, should be visible and modifiable;

The collection of tools and files that are present in an open architecture should be organized in such a way that the user of the tool set is presented with a coherent view of the environment. A particularly effective way to achieve this goal is to organize the tools and files by logical level. At least four such levels should be present in an open architecture for a software development environment:

- (1) Integrated Environment Level
This level is a starting point for the user of the environment and starts up any global mechanisms that are needed by tools working in that environment.
- (2) Tool Level(s)
One or more levels is then devoted to the various tools that exist in the environment. If tools invoke one another, then they can often be separated with the calling tools on the "higher" level and the called tools on the "lower" level. Various libraries and utilities used by the tools can often be associated with a separate level.
- (3) Data repository level
Tools in an integrated environment should be built upon a common data repository. This level

contains the programs that manage the data repository, possibly a database management system.

(4) File interface level

This level contains text files used and/or produced by the tools in the environment in a form suitable for use by other tools. For the architecture to be fully open, all of these files should be user-readable, and should either be free text or a file with a well-defined syntax.

We perceived several advantages in the open architecture approach:

- (1) users already have tools, either locally developed or separately acquired, that they would like to integrate with our tools; furthermore, they would be adding to those tools in the future;
- (2) the visibility of flags and arguments for each component provides the user with the ability to customize the tools;
- (3) access to files, messages, and schemas provides similar possibilities for tool customization, extensibility, and integration.

There are also some disadvantages associated with an open architecture, including the following:

- (1) it is easy to add new tools that use existing interfaces, but more difficult to modify existing tools when the external interface must be changed; users may have built tools that depend on the existing interface and even conversion routines may be insufficient;
- (2) interfaces, such as file formats, message files, and databases, must be saved in a form that allows them to be accurately described and easily processed by other tools; some performance penalties may result from this requirement;
- (3) there are many more aspects of the environment that can be affected by the user and the developer is unable to predict and to test all of the changes that might be made by various users, some of which might lead to system errors;
- (4) because there are so many aspects that can be affected by the user, it is more difficult for the user to obtain comprehensive knowledge about the tools and environment; additional training and experience is needed for a "software environment administrator."

The Visible Connections architecture has served us and our users well, and we regard it as a significant contribution to the understanding of integrated software development environments. It yields four advantages that are valuable both to developers and users of software development environments:

- (1) it allows user customization of the environment to support local preferences;
- (2) it allows extensibility of the environment by both developers and users with minimal impact on the existing base;
- (3) it encourages the development of modest-sized software components rather than large monolithic tool systems;
- (4) it can provide multiple interfaces to the same functions, thereby providing appropriate support for different classes of users.

Software tool environments are changing very rapidly, with many different tool builders, and it would be helpful if various tool building efforts could be effectively combined with one another. The open architecture approach can effectively contribute to this goal.

TMUnix is a trademark of AT&T Bell Laboratories. Software through Pictures and Visible Connections are trademarks of Interactive Development Environments, Inc. PostScript is a trademark of Adobe Systems, Inc. Ada is a registered trademark of the U.S. Department of Defense (Ada Joint Program Office).

THE VIEWGRAPH MATERIALS
FOR THE
A. WASSERMAN PRESENTATION FOLLOW

Key issues in software environments

- Software tool architecture
 - Open vs. closed
 - Tool interconnections
- Project database (“repository”)
 - What is an object?
 - What should be saved?
- Computing support for software development
 - Workstations -- dedicated processing
 - High resolution displays -- graphics
 - Multiple windows
- Life cycle coverage
- Physical workspace
 - Office environments
 - Staff support
 - Ergonomics

Issues in Architectures for Software Development Environments

- Open vs. Closed Approach
- User View
- Internal communication structure
- Underlying database support

What is an open software architecture?

- Multiple level access
 - separate invocations for different levels
 - suitable for distributed environment
- All interfaces published
 - file formats
 - database schemas
- Uses common standards
 - ASCII text
 - pic, PostScriptTM languages

Advantages of open software architecture

- **User customization of environment**
 - Local options and preferences
 - Translation of text messages
- **User extensibility of environment**
 - Add local tools
 - Add other vendors' tools
- **Small components**
 - No monolithic closed system
- **Multiple interfaces to same functions**
 - Supports different types of users

Multiple levels in a software tool environment

- Integrated user view
- Software tools
- Data repository and management functions
- File interface

Goals of the Software through Pictures Environment

- Customizability
- Extensibility
- Usability
- Sharing
- Project database
- Checking
- User control
- Hardware technology

Software through PicturesTM

- Framework for integrated software development environment
 - Extensive opportunities for customization
 - Powerful mechanisms for extensibility
 - Integrated front end for environment
 - Tool communication through database
- Graphical editors for analysis, design, and prototyping
- Support for different views of software models
 - Data Structure Editor
 - Entity Relationship Editor
 - Dataflow Diagram Editor
 - Structure Chart Editor
 - Transition Diagram Editor
 - State Transition Editor
- Completeness and consistency checking of models
- Multiuser support
 - Shared database
 - Diagram locking
 - Integrated version control
 - Works in heterogeneous network environment
- Built upon Visible ConnectionsTM open architecture
 - Published file formats
 - Published DBMS schema and scripts
 - User-modifiable message files
 - User-modifiable project directory structure
 - Visible linkage to user-selected version control system
 - Use of standards

IDE Tool Architecture

- Unified user view
 - IDEtool
- Location independence
 - Tool Information File
- Open architecture
 - Published file formats for extensions
 - Published database schemas
 - Published individual tool invocations
 - User-modifiable IDEtool specification
 - Visible message files
- Project database
 - IDE Data Dictionary
 - History information
 - Linked to Software through Pictures editors
 - Extensible schema

IDEtool

- Startup window
 - Provides uniform view of environment
 - Takes advantage of mouse/menu/icons/windows
- Created dynamically at runtime
 - Uses text file
- Provides access to all IDE tools and options
 - User customizable
- Easily extended for new tools
- Integrated help and error messages
- Hides command structure and underlying OS
 - Reduces learning time for tools
 - Reduces error rate for new users

IDEtool window regions

- Tool stripe
- Message area
- Control panel
 - Project and system information
 - Version information (optional)
 - Control buttons
 - Help Mode
- Tool group area
- Command area
- Options and arguments area
- Teletype subwindow

IDE TOOL (3.0)

Project Directory: /usr/ide/project
System: example Level:

Execute Reset Quit [DB Open] [Help Mode]

				[Diagram Utilities]
				[Project Database]
				[Version Control]
				[IDE Data Dictionary]
				[More Utilities]

[Edit Diagram] [Print Diagram] [Check Decomposition] [Generate IDE/DD]
[Check IDE/DD] [Pspec for diagram] [Pspec for a single Process] [Pic]
[PostScript]

Draw: Data Flow Diagram Control Flow Diagram
Show: Data Flows Control Flows All Flows
Symbols: Gane/Sarson Marco/Yourdon
Process Index Position: Top Bottom
Show full Process Index

```

shellto
/dev/sd
home (1
/usr/ide
home (12
home (13
screendump | lpr -v -h
home (14) ll
screendump | lpr -v -h

```



IDE TOOL (3.0)

Project Directory: /usr/ide/project/
System: example Level: 0

Execute Reset Quit [DB Open] [Help Mode]

[Diagram Utilities]
[Project Database]
[Version Control]
[IDE Data Dictionary]
[More Utilities]

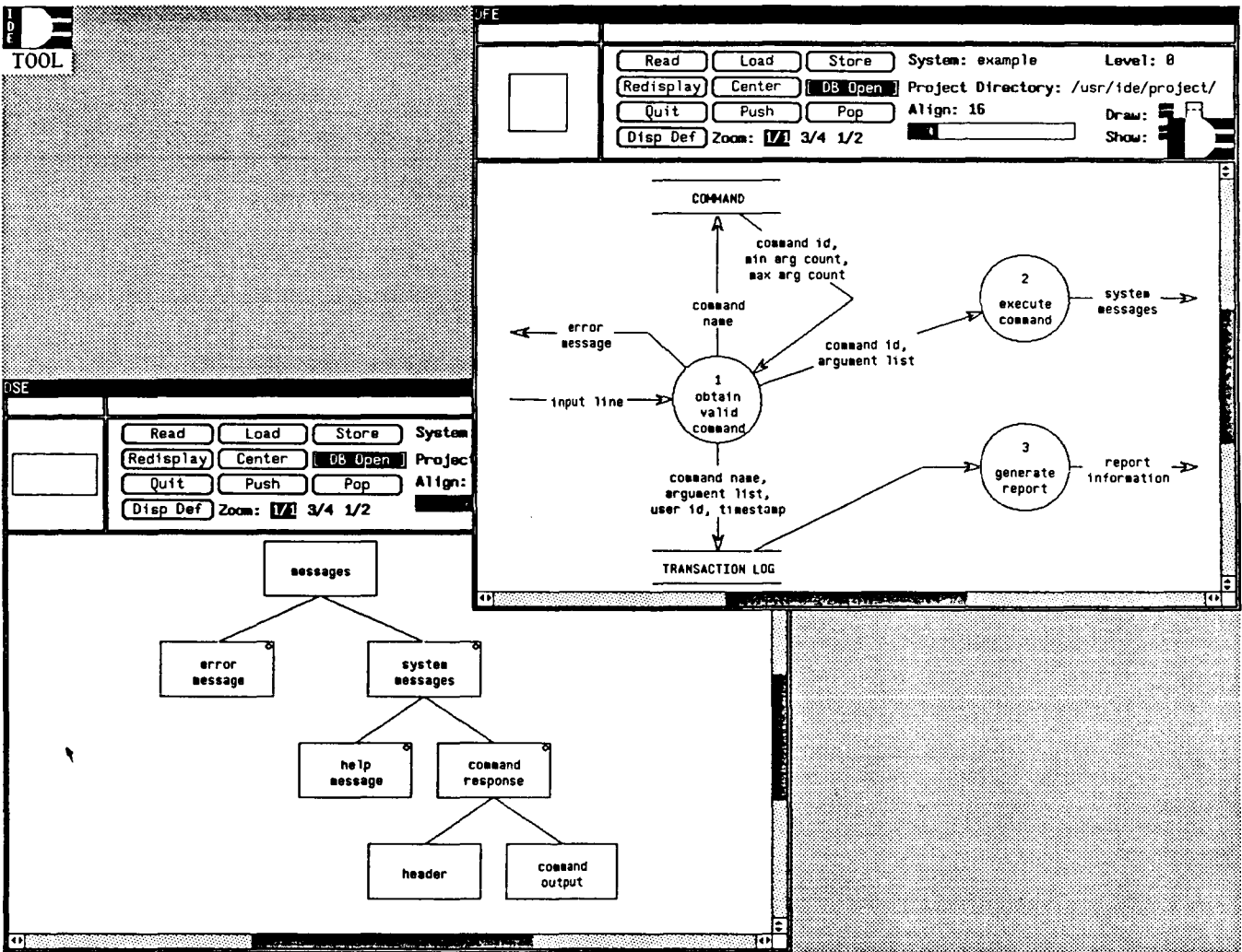
[Edit Diagram] [Print Diagram] [Check Decomposition] [Generate IDE/DD]
[Check IDE/DD] [Pspec for diagram] [Pspec for a single Process] [Pic]
[PostScript]

Draw: Data Flow Diagram Control Flow Diagram
Show: Data Flows Control Flows All Flows
Symbols: Gane/Sarson deMarco/Yourdon
Process Index Position: Top Bottom
Show full Process Index

Read Load Store System: example Level: 0
Redisplay Center DB Open Project Directory: /usr/ide/project/
Quit Push Pop Align: 32 Draw: [] []
Disp Def Zoom: 1/1 3/4 1/2 Show: [] []

```

graph TD
    InputLine[input line] --> Node1((1 obtain valid command))
    Node1 -- error message --> ErrorMessage[error message]
    Node1 -- command name --> Command[COMMAND]
    Node1 -- "command id, min arg count, max arg count" --> Command
    Node1 -- "command id, argument list" --> Node2((2 execute command))
    Node2 -- system messages --> SystemMessages[system messages]
    Node1 -- "command name, argument list, user id, timestamp" --> TransactionLog[TRANSACTION LOG]
    TransactionLog --> Node3((3 generate report))
    Node3 -- report information --> ReportInfo[report information]
  
```



Developers can modify the environment

- Processes activated by pop-up menu selections
- Processes specific to diagram node types
- Processes to be activated with one command
- Startup menu and user interface (IDEtool)
- Text editors for symbol editing PDL and “mini-spec” templates
- Pop-up menu labels
- Project database schema
- Help and error messages
- Panel button labels
- Printing interface
- Print status messages
- Tool information file
- Locations of tools, databases, message files

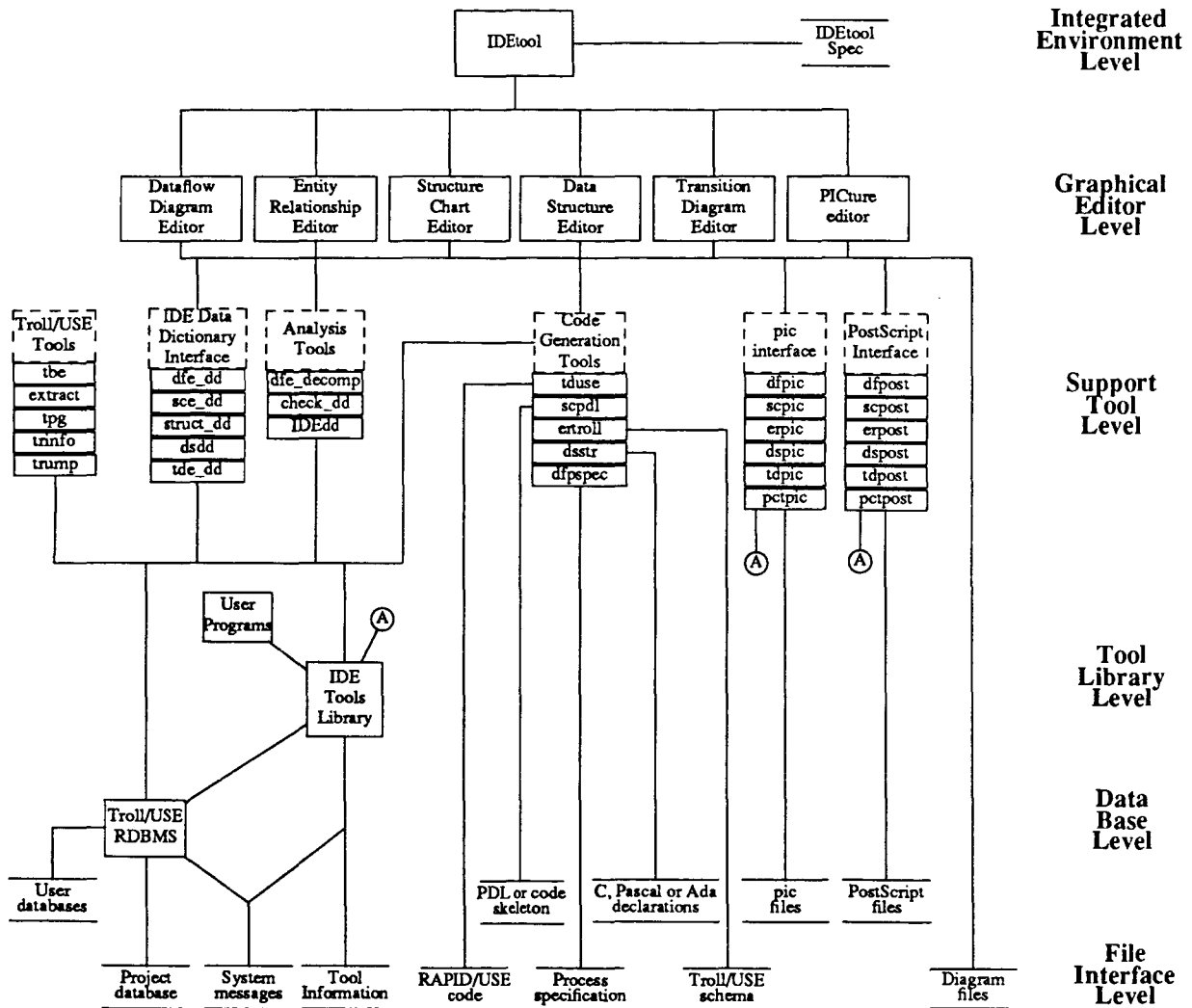
Visible and Modifiable Messages

- Error messages
- Help messages
- Menu items
- Panel items
 - Buttons
 - Sliders
 - Icons
- DBMS scripts for IDE Data Dictionary
- “Minispec” and “PDL” templates
- Project directory creation script
- IDEtool specification file
- Tool Information file
- Formats for output reports
- Easy modification and/or translation

Tool Information File

- Location independence
- Parameterized environment with initial defaults
- Global/local/personal versions
- Attribute-value pairs in text file
- Tool Information functions in IDE tools library

IDE Multilevel Open Architecture



Published file formats

Published database schema for IDE Data Dictionary

User-modifiable messages and report formats

Diagrams stored in ASCII

Pic and/or PostScript™ output of diagrams

Tool Information File for location independence and customization

IDEtool built from textual IDEtool specification file

IDE Tools are Extensible

Feature	Benefit
<ul style="list-style-type: none">• All files are stored in ASCII or are easily convertible to ASCII	<ul style="list-style-type: none">— Supports open architecture for tools
<ul style="list-style-type: none">• All IDE file formats are documented	<ul style="list-style-type: none">— Allows users to build additional tools that interface easily with IDE tools
<ul style="list-style-type: none">• All IDE graphics are translated into device-independent graphics languages: pic and PostScript	<ul style="list-style-type: none">— Supports most popular high resolution printers
<ul style="list-style-type: none">• All Software through Pictures tools are supported by a compact and efficient relational DBMS (Troll/USE)	<ul style="list-style-type: none">— Supports important software engineering concept of project database— Can be used both for project and application database requirements— Allows users to add new relations and attributes
<ul style="list-style-type: none">• IDE tools communicate through a user-customizable tool information file	<ul style="list-style-type: none">— IDE software is <i>location-independent</i>— Users can modify IDE standard environment to suit local needs
<ul style="list-style-type: none">• The tool information file is built upon the IDE tools library	<ul style="list-style-type: none">— Both IDE and users can add new attributes and values to the tool information file as the tool set is extended— User programs can be easily linked to Troll/USE and RAPID/USE

Integrated development environment

- IDEtool startup window
- Linkage to project directory/system
- Mouse/menu/icon interface
- Customization of default options
- Extensibility of tools, commands, arguments
- Integrated project database
- Integrated locking and version control
- Integrated help
- Integrated error reporting

PANEL #4

SOFTWARE TESTING

J. Knight, University of Virginia
N. Leveson, University of California, Irvine
A. Goel, Syracuse University

DATA DIVERSITY
AN APPROACH TO FAULT-TOLERANT SOFTWARE

Paul E. Ammann John C. Knight

Department of Computer Science
University of Virginia
Charlottesville, Virginia.

A Summary

Submitted To The Eleventh Annual Software Engineering Workshop
Goddard Space Flight Center
Greenbelt, Maryland.

In an effort to provide substantial improvements in reliability of the software for crucial applications, various methods of building fault-tolerant software have been proposed. Such software is constructed with the assumption that faults may survive the development process, and the structure is designed to cope with the effects of those faults at execution time.

The two best-known existing methods of building fault tolerant software are N -version programming [1] and recovery blocks [2]. N -version programming requires the separate, independent preparation of multiple (i.e. " N ") versions of a piece of software for some application. These versions are executed in parallel in the application environment; each receives identical inputs and each produces its version of the required outputs. The outputs are collected by a voter and, in principle, they should all be the same. In practice there may be some disagreement. If this occurs, the results of the majority (assuming there is one) are assumed to be the correct output, and this is the output used by the system.

A recovery block is a structure in which the results of an algorithm are subjected to an acceptance test. If the results are deemed unsatisfactory, the state of the machine that existed just prior to execution of the algorithm is restored and an alternate algorithm is executed. This process may be repeated until satisfactory outputs are produced or the set of alternates is exhausted.

Both of these techniques rely on the availability of multiple implementations of an algorithm to allow faults to be tolerated. This is termed *design diversity* because the different implementations are assumed to contain different designs and thereby, it is hoped, different faults. Clearly, either method incurs the capital investment required to construct the multiple implementations. N -version programming also incurs the operational cost of executing the algorithms.

It has been observed frequently that software often fails at boundary points in the input space. A program may work well for many input cases, survive extensive testing, and then fail on an input case associated with a boundary condition. This condition may take the form of what seem to be an obscure set of conditions in the input data. The boundary need not be an obvious boundary in the set of values taken by any specific input. In fact, the boundary is usually associated with a transition in the required processing algorithm.

This phenomenon is well known and is, in fact, the basis of a theoretical, but not very practical, testing technique [3]. However if that method is not used, faults associated with boundary conditions frequently fail to show up in testing precisely because they require an exact and unexpected set of circumstances that are not generated during testing. It is usually not sufficient to generate a test data set whose values are merely close to the values which cause the program to fail. If, during testing, the special case is not generated exactly, the software usually works correctly on the generated test cases.

There is a strong implication from these observations that if software fails under certain execution conditions, it is very likely that a minor perturbation of those execution conditions would allow the program to work. This is the observation underlying a new approach to fault tolerance that we term *data diversity*.

In the diverse-data approach to fault tolerance, we make use of the noted phenomenon by reexecuting the same software that failed but with slightly different inputs. As with any form of fault tolerance, an approach using data diversity has to be able to do detect the error. The application of data diversity therefore involves combining the notion of executing identical software with slightly different inputs and some scheme for error detection.

Clearly, this general approach is not suitable for all application areas. It is however well suited to control systems in which sensors are read and actuators set. Sensors are noisy and inaccurate, and small systematic perturbations of sensor values in order to cope with a fault would still allow the software to generate acceptable outputs.

We have defined a program structure that takes advantage of data diversity by executing multiple *copies* of a *single* program. The copies are executed in parallel and each receives a slightly modified form of the input data. The results are voted in much the same way as occurs in an *N*-version system. We refer to the program structure that results as an *N-copy program*.

We have also defined a modification of the recovery block structure that uses data diversity. We term this approach the *retry block*. The concept of the retry block is quite simple. Rather than preparing multiple alternate algorithms as is done with the recovery block, there is only one version of the algorithm.

It is supplemented by an acceptance test that has the same form and purpose as the acceptance test in a recovery block. The semantics of the retry block are to execute the algorithm normally and follow it with evaluation of the acceptance test. If the acceptance test is passed, the retry block is complete. If the acceptance test fails, the algorithm is executed a second time but the data is modified prior to execution. Clearly, this approach can be repeated and the retry block executed many times if desired.

We have obtained empirical evidence of the expected performance of data diversity by examining the effect of minor changes to the input data on the known faults in the programs produced for the Knight and Leveson experiment [4]. The results indicate that, although the performance of data diversity is quite varied, it can produce a substantial reduction in failure probability at virtually no cost.

This paper will describe data diversity as an approach to fault-tolerant software in detail and present the results of empirical studies.

REFERENCES

- (1) L. Chen and A. Avizienis, "N-version programming: A fault-tolerance approach to reliability of software operation," *Digest of Papers FTCS-8: Eighth Annual International Conference on Fault Tolerant Computing*, Toulouse, France, pp. 3-9, June 1978.
- (2) T. Anderson and P.A. Lee, *Fault Tolerance: Principles and Practice*, Prentice Hall International, 1981.
- (3) L. White and E. Cohen, "A Domain Strategy For Computer Program Testing", *IEEE Trans. on Software Engineering*, Vol. SE-6, No. 3, May 1980.
- (4) J.C. Knight and N.G. Leveson, "An Experimental Evaluation Of The Assumption Of Independence In Multi-Version Software", *IEEE Trans. on Software Engineering*, Vol. SE-12, No. 1, January 1986.

THE VIEWGRAPH MATERIALS
FOR THE
J. KNIGHT PRESENTATION FOLLOW



UVA

DEPARTMENT OF COMPUTER SCIENCE

DATA DIVERSITY
AN APPROACH TO SOFTWARE
FAULT TOLERANCE

Paul E. Ammann John C. Knight

Department Of Computer Science
University Of Virginia
Charlottesville, Virginia 22903

Sponsored By NASA Grant Number NAG1-605



FAULT TOLERANCE THROUGH *DESIGN* DIVERSITY

- *N*-Version Programming:

- Multiple Implementations Of A Specification, Developed Independently
- Executed In *Parallel*
- Outputs Voted To Select System Output
- Faults Tolerated By Presumed Differences In *Design* Of The Versions

- Recovery Block:

- Multiple Implementations Of A Specification, Developed Independently
- Executed In *Series*
- Output Checked By An Acceptance Test
- Faults Tolerated By Presumed Differences In *Design* Of The Versions



FAILURE REGIONS

- Input Space For Most Programs Is Hyperspace With Many Dimensions
- For Example:
 - Twenty Floating Point Inputs
 - Twenty-Dimensional Space
- Sometimes Varies From Execution To Execution
- Certain Region(s) In Input Space Contain Data Cases Causing Failure
 - Termed *Failure Regions*
- What Are Their Characteristics?
 - Shape
 - Size
 - Etc
- Seeing Some Might Provide Insight



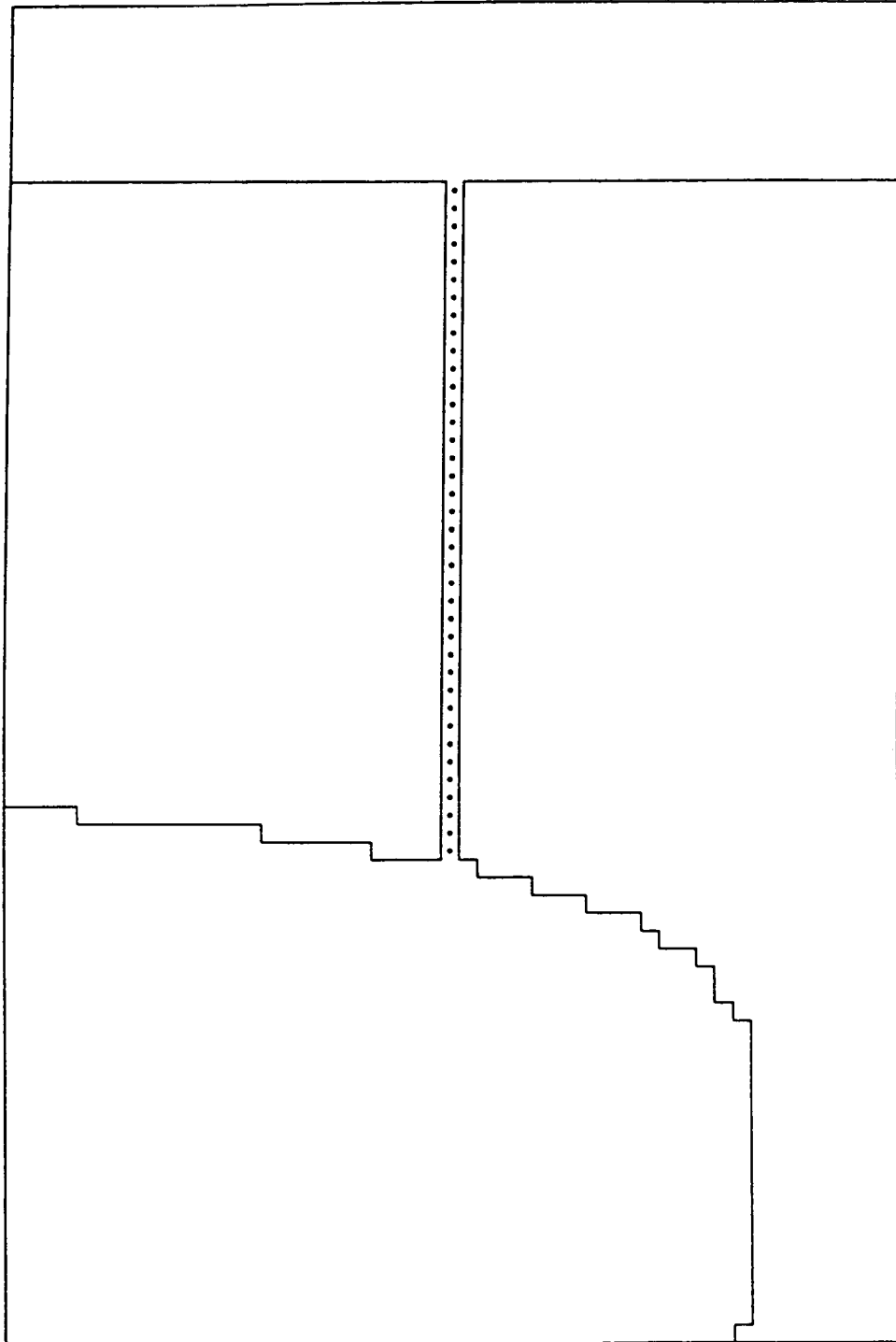
CROSS SECTIONS

- Cannot Display Multi-Dimensional Spaces
- Can Display Two-Dimensional Cross Sections
- Approach:
 - Systematically Vary Two Inputs Across Range
 - Keep All Other Inputs Fixed
 - Determine Correctness Of Output
 - Plot Correct vs Incorrect
 - Plot Transitions In Output Values
- Previous Knight/Leveson Experiment:
 - Twenty-Seven Programs
 - One Million Test Cases
 - Identified Faults
- Many Sections Obtained From These Faults



UVA

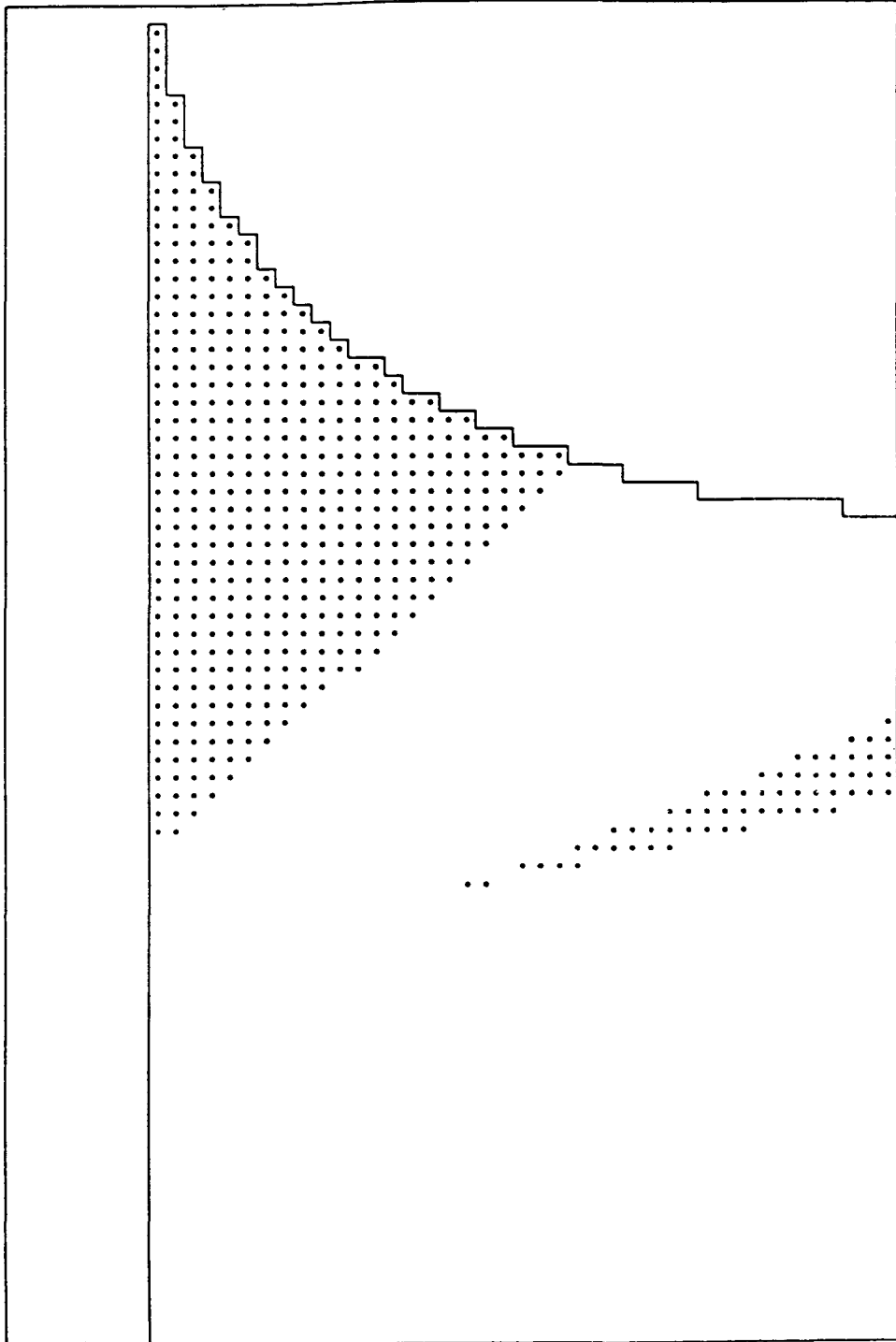
DEPARTMENT OF COMPUTER SCIENCE





UVA

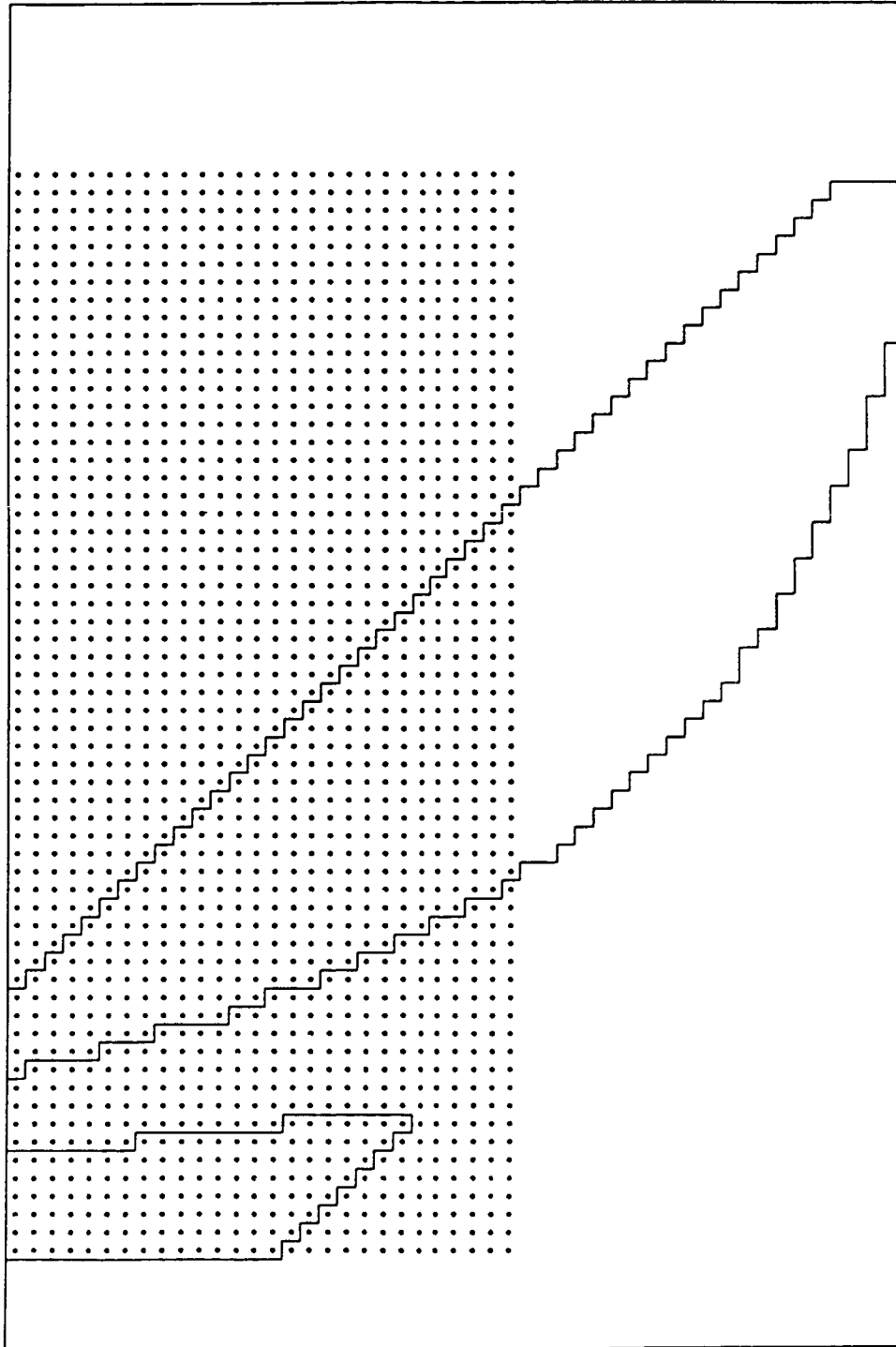
DEPARTMENT OF COMPUTER SCIENCE

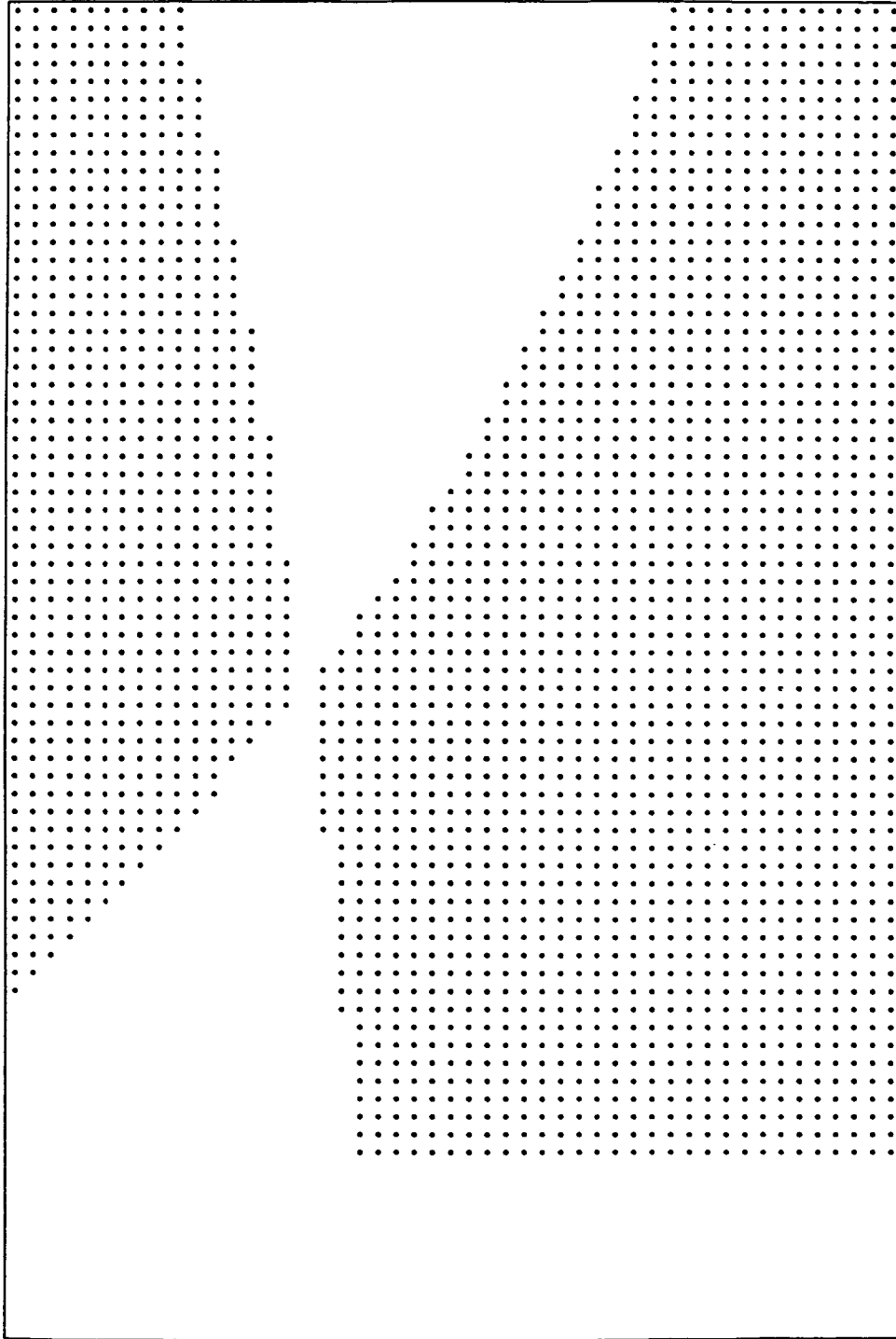


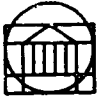


UVA

DEPARTMENT OF COMPUTER SCIENCE

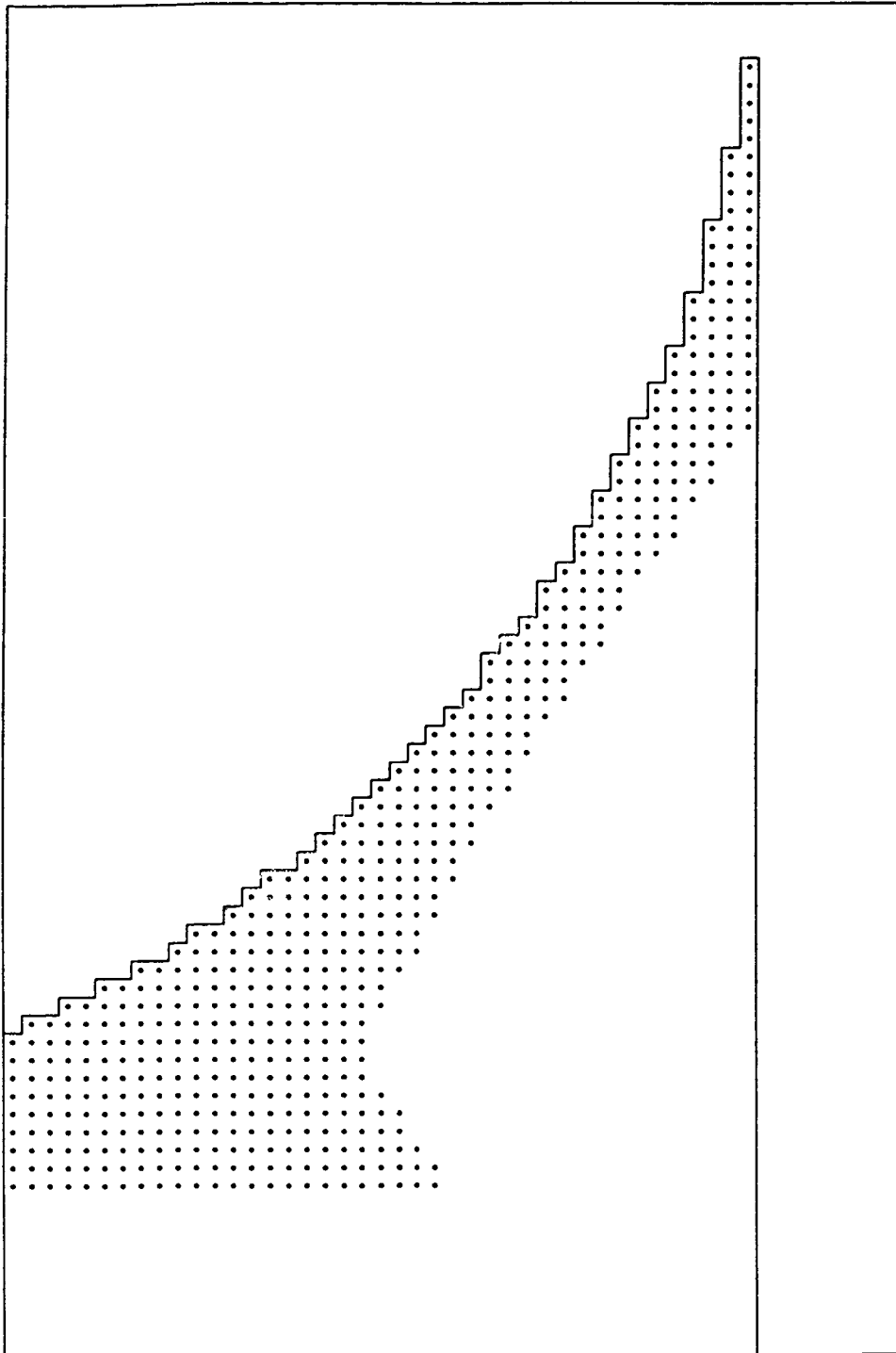


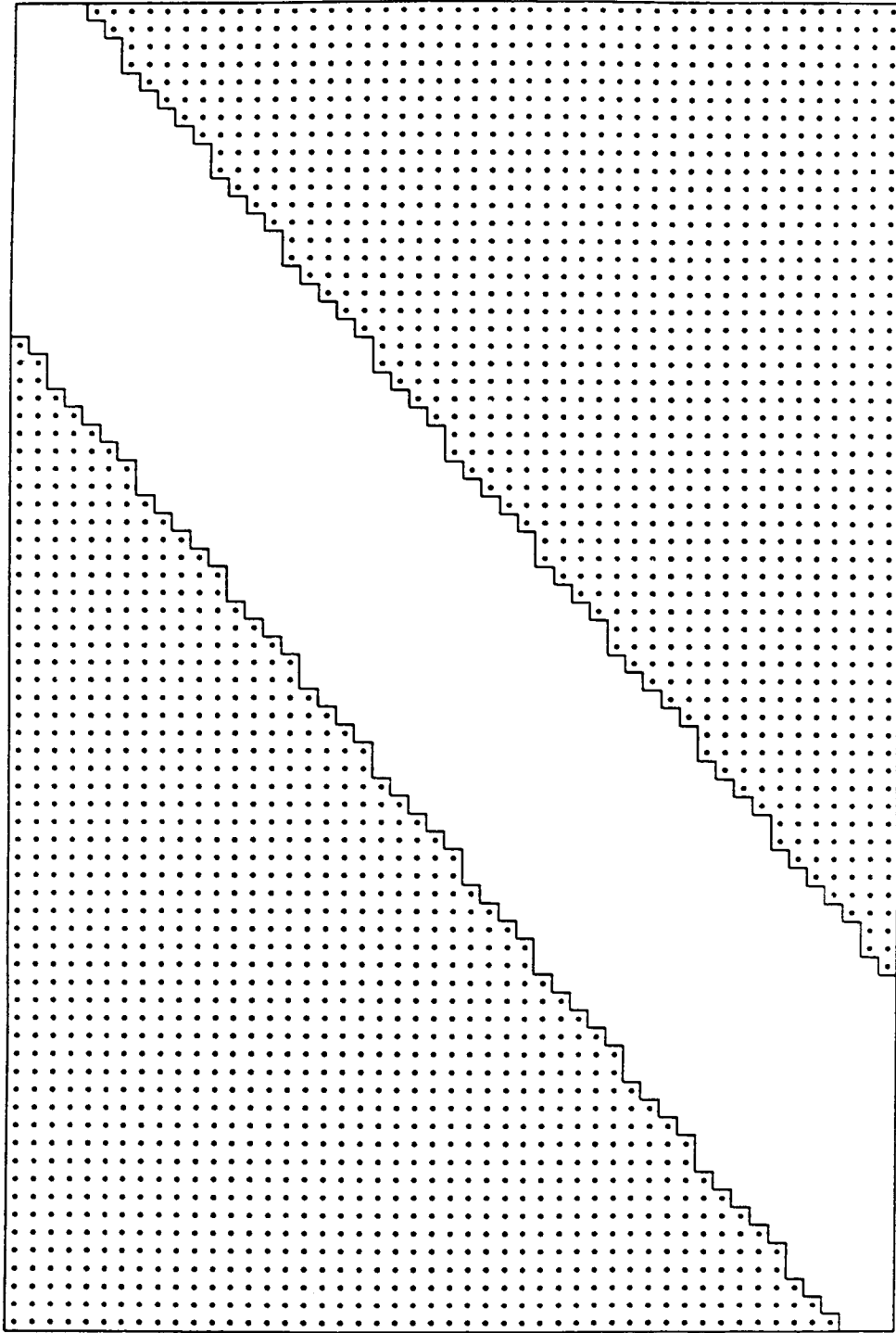


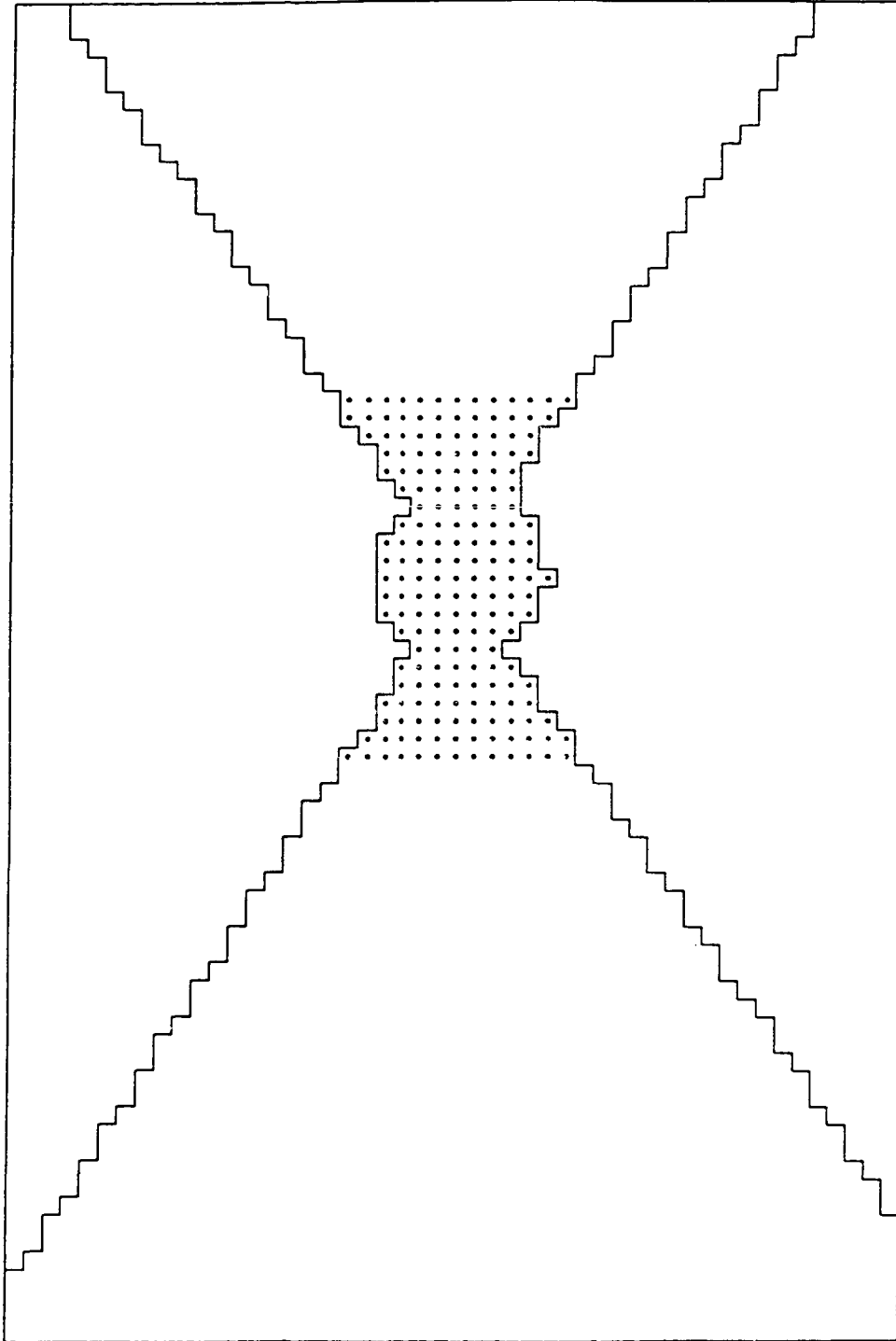


UVA

DEPARTMENT OF COMPUTER SCIENCE









DATA DIVERSITY

- Programs Tend To Fail On “Special” Cases
 - Why?
 - Because Failure Regions Tend To Be:
 - Small
 - Irregularly Shaped
 - Close To Output Transitions
- How Could This Observation Be Exploited?
 - If Program Fails, Rerun With Slightly Different Data.
 - If New Data Outside Failure Region, Program Will Succeed.
 - If Output Still Acceptable, Fault Has Been Tolerated.
- We Term This, Fault Tolerance By *Data Diversity*.

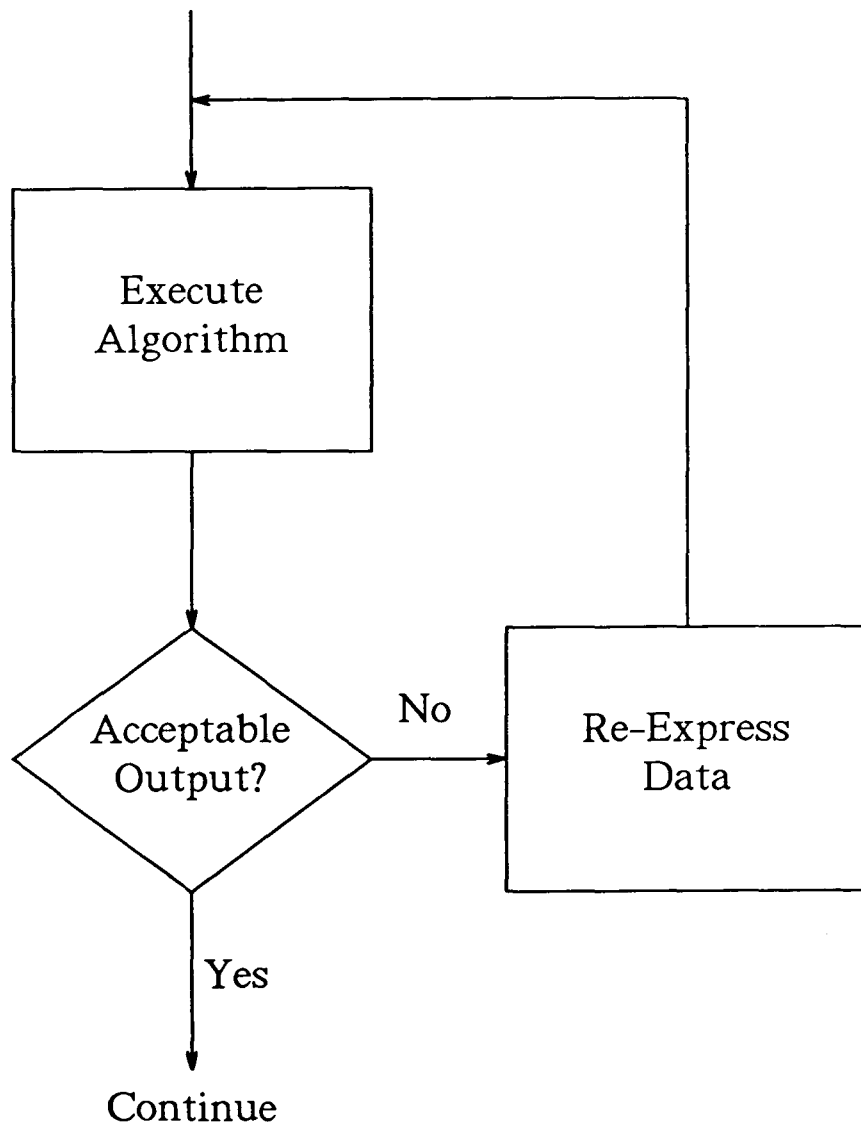


PREVIOUS WORK

- Idea Suggested Previously As Special Cases
- Gray - Tandem Corp
 - Rerun Asynchronous System
 - Different Order Of Events Often Avoids Fault
- Shepherd et al - Cranfield Inst. Tech.
 - Run Multiple Versions With Inputs Skewed In Time
- No Need To Rely On Chance Reordering Nor On Data Changing With Time
- We Propose General Diversity In Data
 - *Re-Express* Data Algorithmically

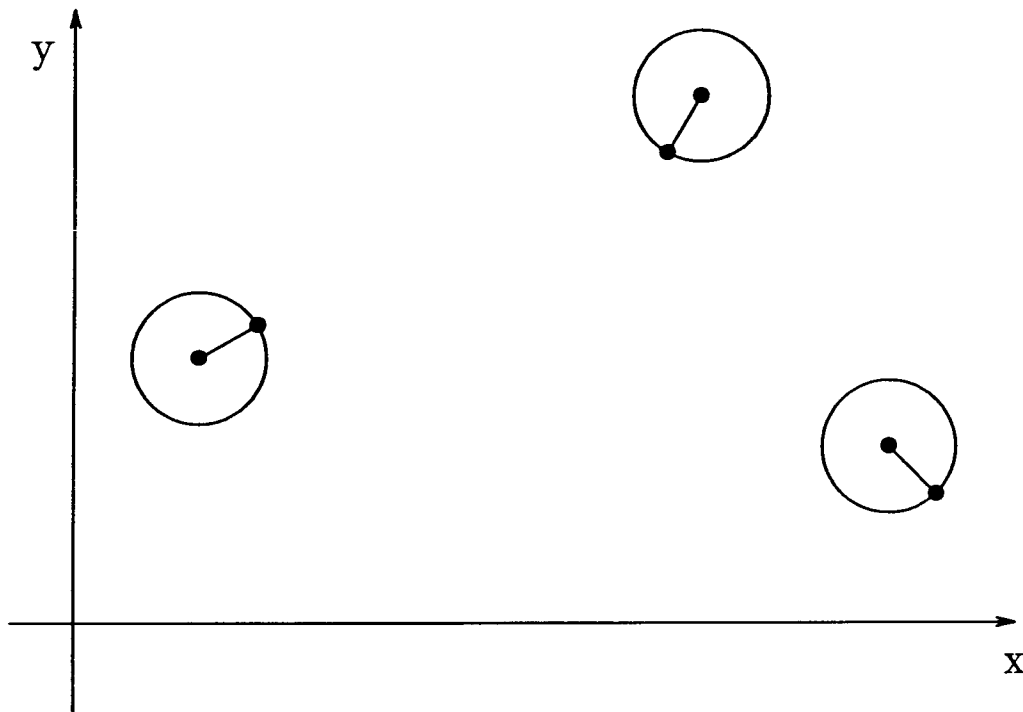


DATA DIVERSITY USING THE RETRY BLOCK





DATA RE-EXPRESSION





RETRY BLOCK

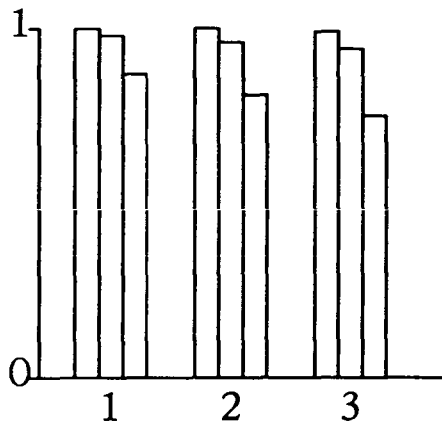
RELATIVE PERFORMANCE

Retries	1			2			3		
Displ	0.001	0.01	0.1	0.001	0.01	0.1	0.001	0.01	0.1
Fault									
6.1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
6.2	1.00	0.98	0.87	1.00	0.96	0.81	0.99	0.94	0.75
6.3	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
7.1	0.92	0.59	0.26	0.87	0.43	0.11	0.80	0.29	0.03
8.1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
8.2	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
9.1	0.99	0.90	0.39	0.97	0.83	0.19	0.97	0.74	0.07

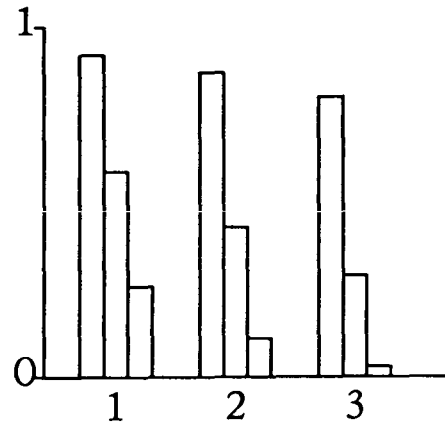


RETRY BLOCK

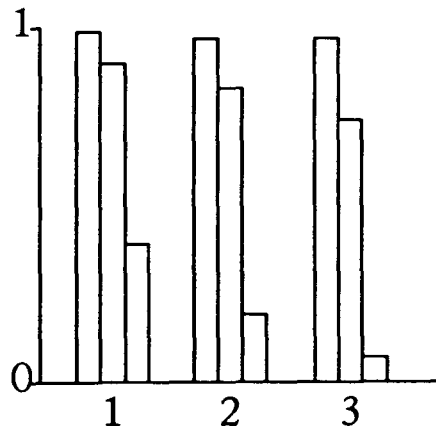
RELATIVE PERFORMANCE



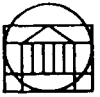
Fault 6.2



Fault 7.1



Fault 9.1



CONCLUSIONS

- Data Diversity Is Inexpensive
 - Single Implementation
 - Minor Costs Associated With Re-Expression And Error Detection
- Data Diversity Works
 - Empirical Study Showed:
 - Some Faults Tolerated Very Well
 - Some Faults Tolerated Not So Well
 - Performance Widely Varied
- Data Diversity Not Universally Applicable
 - Some Data Cannot Be Re-Expressed
 - Many Control Systems Read Noisy And Inaccurate Sensors, Data Diversity Should Work Well

Self-Checking Software*

*Sung D. Cha
Nancy G. Leveson
Timothy J. Shimeall*

Dept. of Information & Computer Science
University of California, Irvine
Irvine, CA 92717

John C. Knight

Dept. of Computer Science
University of Virginia
Charlottesville, VA 22903

Abstract

This paper presents the results of an empirical study of error detection using self-checks. The goal of this study was not just to obtain quantitative results but to learn more about such checks and how they might best be implemented. This information may result in better methods for formulating checks, making them easier to write and more effective. The analysis of the checks revealed that there are great differences in the ability of individual programmers to design effective checks. We found that some checks that might have been effective failed to detect a fault because they were badly placed, and there were numerous instances of checks signaling non-existent errors. In general, specification-based checks alone were not as effective as combining them with code-based checks.

Goals of the Study

The ability to produce ultra-reliable computer systems in such industries as aerospace and defense is becoming increasingly important. Although research in hardware design has yielded computer architectures of potentially very high reliability, the state of the art in software development is not as advanced. Current software engineering methods cannot guarantee ultra-high software reliability, and formal verification and synthesis are not able to deal with software of the required size and complexity.

It has been proposed that fault tolerance techniques be used to make the software function correctly despite the presence of faults in the code. It is hoped that this will provide the

*This work was supported in part by NASA under grant numbers NAG-1-511, and NAG-1-668, in part by NSF grant DCR 8406532, and in part by MICRO grants cofunded by the state of California, Hughes Aircraft Co., and TRW.

required reliability, although empirical evidence is meager. The few real projects that have attempted to apply software fault tolerance have used ad hoc methods, and little or no data was collected on the effectiveness of the techniques.

We are engaged in a long-term effort to evaluate and improve software fault tolerance techniques and to determine when and how they should be applied. Our first experiment, which was reported at a previous Goddard Software Engineering Workshop, tested the fundamental assumption that software versions that are developed independently will fail in a statistically independent manner [Knight and Leveson (1986a)]. A conclusion of this experiment is that models of reliability improvement must include the possibility of correlated failure patterns among the N versions. Using the programs generated for this experiment, we have been able to demonstrate some reliability improvement using 3-version voting although it was not as great as might be necessary to achieve ultra-high reliability in practical systems. We have also examined in detail the faults that caused correlated failures to determine their common characteristics and to determine if it appeared that changes in the way N -version software is developed might help to minimize them [Brilliant, Knight, and Leveson (1986b)].

Because of the limited success of the voting technique, we have attempted to examine alternatives. The primary alternative to voting is acceptance tests or other types of self-tests embedded in the software. A recent experiment by Anderson *et. al.* (1985) used recovery blocks, but few conclusions were reached about acceptance tests outside of the fact that they are hard to write. Information about self-checking software is important not only for fault-tolerance, but also for more general software engineering techniques since acceptance tests are a subset of the more general run-time assertion used in exception-handling and testing. Although acceptance tests, assertions, and exception handling mechanisms have been included in programming languages and systems, little information is available about the difficulty of

writing effective self-checks in software. More information about the use of self-checks to detect software errors might result in better methods for formulating checks, making them easier to write and more effective. Our goal in this study was not merely to provide numerical data, but to learn more about such checks and how they might best be implemented. The next section describes the design of the study. Following this, the results are described and conclusions drawn.

Experimental Design

This study uses the programs developed for a previous experiment by [Knight and Leveson (1986a)]. Twenty-seven versions of a program to read radar data and determine whether an interceptor should be launched to shoot down the object (hereafter referred to as the Launch Interceptor Program, or LIP) were prepared from a common specification by graduate students and seniors at the University of Virginia and the University of California, Irvine. Extensive efforts were made to ensure that individual students did not cooperate or exchange information about their program designs during the development phase. The twenty-seven LIP programs have been analyzed by running one million randomly generated test cases on each program and locating the individual faults that were detected during the testing procedure.

In the present study, 8 students from UCI and 16 students from UVA were employed for a week's time to instrument the programs with self-checking code in an attempt to detect errors in the programs. Eight programs were selected from the 27 and each was randomly assigned to three students (one from UCI and two from UVA). The students were all graduate students in computer science with an average of 2.35 years of graduate study. Professional experience ranged from 0 to 9 years with an average of 1.7 years. None of the participants had

prior knowledge of the LIP program nor were they familiar with the results of the previous experiment. There was no significant correlation found between a participant's graduate or industrial experience and their success at writing self-checks.

Participants were provided with a brief explanation of the study along with an introduction to writing self-checks. All also read Chapter 5 on Error Detection from a textbook on fault tolerance [Anderson and Lee (1981)]. The participants were first asked to study the LIP specification and to write checks using only the specification, the training materials, and any additional references the participants desired. When they had submitted their initial checks, they were randomly assigned a program to instrument. The participants were asked to write checks with and without looking at the code in order to determine if there was a difference in effectiveness between self-checks designed by a person working from the requirements alone and those for which the person has access to and information about the program code. On the one hand, the person working only from the requirements might provide more independence by not being influenced by the written code. However, it could also be argued that looking at the code will suggest different and perhaps better self-checks. Because we anticipated that the process of examining the code might result in the participants detecting faults through code-reading alone, participants were asked to report any such detected faults but to still attempt to write a self-check to detect the fault.

The instrumented versions were subjected to an acceptability test (200 randomly generated test cases) as in the previous experiment. The original versions were known to run correctly on those data, and we wanted to attempt to remove obvious faults introduced by the self-checks. If any false alarms were raised (faults reported that did not actually exist) or if new faults were detected which had been introduced into the program by the instrumentation, the programs were returned to the participants for correction. Along with the instrumented

version, participants submitted time sheets, background profile questionnaires, and descriptions of all program faults identified by code reading.

After the instrumented programs had passed the acceptability test, they were executed using the test cases on which they had failed in the previous experiment along with 20,000 new randomly-generated test cases to see if new faults might have been detected. Finally, the self-checks were carefully examined and catalogued as to type of check and effectiveness.

Results

The first task of the experiment participants was to read through the program requirements specification and to design self-checks based solely on that specification. These self-checks were found to fall into four groups based on the general strategy of check used:

- [1] *Duplication Checks*: self-checks that duplicate the functionality of the code and compare results. Most, but not all, of the self-checks in this group use algorithms different from the original source code.
- [2] *Structural Checks*: self-checks that verify the proper use of data structures or the proper semantics of code. Examples include a check which verifies that the exit condition of a loop is true immediately following the loop and a check that verifies that data values have not been improperly overwritten.
- [3] *Reversal Checks*: self-checks that reverse the operation performed by the code and then see if the results are consistent with the input data.
- [4] *Consistency Checks*: self-checks that determine if the results have certain properties. Examples of consistency checks include range checking, arithmetic exception checking, and type checking.

Table 1 shows the classification of the self-checks designed from the specification. Note that the largest number of checks written were consistency checks followed by duplication checks. Performance is discussed later, but Tables 3 and 5 show that a total of 33 self-checks were completely or partially effective in detecting errors. Of these 33 effective checks, 4 (or 12%) were formulated by the participants after looking at the requirements specification only. The remaining 88% of the effective checks were designed after the the participants had looked at the code. It has been suggested that acceptance tests in the recovery block structure must be based on the specification alone [Anderson and Lee (1981)]. Our results indicate that effectiveness of the self-checking can be improved when the specification-based (acceptance test) checks are refined and expanded by source code reading and a thorough and systematic instrumentation of the program. It appears that it is very useful for the instrumentor to actually see the code when writing self-checks.

#	Type of checks used					Total
	Duplication	Structural	Reversal	Consistency	Other*	
Total	149	23	76	218	11	477

Table 1: Specification-Based Self-Checks

The second task of the participants was to instrument a particular program with self-checks. No limitations were placed on the participants as to how much time could be spent (although they were paid only for a 40 hour week which effectively set an upper bound[†]) or how much code could be added. The amount of time reported spent ranged from 19.5 hours to 52 hours. There was no statistically significant relationship between the number of hours claimed to have been spent (as reported on the timesheets) by the participants and whether or not they detected any program faults.

*These self-checks were too vague to be classified

[†]Several reported spending more than 40 hours on the project.

Table 2 describes the change in length in each program during instrumentation[†]. Note that there is a great variation in the amount of code added, ranging from 48 lines to 835 lines. Participants added an average of 37 self-checks, varying from 11 to 97. Despite this variation, there was no correlation between the total number of checks inserted by a participant and the number of those checks that were effective at finding faults. That is, more checks did not necessarily mean better fault detection.

Version #	Number of Lines			Increase			
	original	a	b	c	a	b	c
3	757	909	1152	805	152	395	48
6	643	859	887	700	216	244	57
8	600	1046	1356	824	446	756	224
12	573	1121	696	806	548	123	233
14	605	905	1342	712	300	737	107
20	533	611	1368	596	78	835	63
23	349	1065	417	544	716	68	195
25	906	1644	1016	1022	738	110	116

Table 2: Lines of Code Added During Instrumentation

Table 3 classifies the program-based self-checks in terms of strategy used and effectiveness. Checks are classified as effective if they correctly report the presence of an error during execution. Two partially effective checks by participant 23a that detect an error most (but not all) of the time are counted as effective. Ineffective checks are those that do not signal an error when one occurs during run-time in the module being checked. False alarms signal an error when no error is present. Finally, the effectiveness is classified as unknown if the check does not signal an error and the module being tested is correct.

[†]In order to aid the reader in referring to previously published descriptions of the faults found in the original LIP programs, the programs are referred to in this paper by the numbers previously assigned in the original experiment. A single letter suffix is used (a, b, or c) to distinguish between the three independent instrumentations of the programs.

#	Effectives				Ineffectives				False Alarms				Unknowns				Total
	D	S	R	C	D	S	R	C	D	S	R	C	D	S	R	C	
Total	19	0	0	14	10	28	0	50	5	2	0	3	73	168	31	462	865

Table 3: Self-Check Classification

It can be seen from Table 3 that duplication and consistency checks were about equally effective in detecting faults although more consistency checks were used. For these programs, structural and reversal checks were not effective, but this may have been influenced by the types of faults that were actually in the programs. We examined the ineffective self-checks (checks on code that contained faults but did not detect the faults) in detail. They appear to fail due to one or more of the following reasons:

- Wrong self-check strategy – the participant used a type of self-check inappropriate to detect the fault present in the code. For example, use of a structural check when the fault was an inadvertent substitution of one variable for another in an expression.
- Wrong check placement – the participant placed the self-check in a location where not all results were checked, and the fault was on a different path.
- Use of the original faulty code in the self-check – the participant falsely assumed a portion of the code was correct and called that code as part of the self-check.

It should be noted that the placement of the checks may be as crucial as the content. This has important implications for future research in this area and for the use of self-checking in real applications.

It should not be assumed that a false alarm involved a fault in the self-checks. In fact, there were cases where an error message was printed even though both the self-check and the original code were correct. This occurred when the self-check made a calculation using a different algorithm than the original code. Because of the inaccuracies introduced by finite

precision arithmetic compounded by the difference in order of operations, the self-check algorithm sometimes produced a result which differed from the original by more than the allowed tolerance. Increasing the tolerance does not necessarily solve this problem in a desirable way. This same problem occurred in our previous experiment and is discussed in detail elsewhere [Brilliant, Knight, and Leveson (1986a)].

Some faults were detected while the participants were reading the code. The numbers in Table 4 refer to the numbering used to identify the individual faults in [Brilliant, Knight, and Leveson (1986b)]. Three faults were reported that actually were not faults; the participant misunderstood the code.

Version	Fault
3a	3.3
6a	6.1
	6.2
12c	12.1
20b	20.2
20c	20.2
25a	25.1
	25.3

Table 4: Faults Detected Through Code-Reading

Table 5 summarizes the detected faults by how they were found. 20% of the detected faults were detected by specification-based checks, 40% by code-reading, and 40% by code-based checks. Note that often more than one check detected the same fault in the code-based case, which was not true of the specification-based or code-reading faults.

Object	Due To			Total
	Spec-based Design	Code Reading	Code-based Design	
Faults Detected	4	8	8	20
Effective Checks	4	8	21	33

Table 5: Fault Detection Classified by Instrumentation Technique

One final way of looking at the results of this study is to consider the number of faults detected and introduced by the participants. Table 7 shows this information.

#	Already Known Faults		Other Faults Detected	Newly Added Faults	
	Present	Detected		Incorrect	NoAnswer
3a	4	1	0	0	0
3b		0	0	0	0
3c		0	0	0	0
6a	3	2	1	0	1
6b		0	0	1	0
6c		0	0	0	0
8a	2	2	0	0	0
8b		0	0	1	0
8c		0	1	3	0
12a	2	1	1	0	0
12b		0	0	0	2
12c		1	1	1	1
14a	2	0	0	0	0
14b		0	0	0	4
14c		0	0	0	0
20a	2	0	0	1	0
20b		1	1	2	0
20c		1	0	0	0
23a	2	2	0	3	1
23b		0	0	0	0
23c		0	0	0	0
25a	3	2	1	0	0
25b		1	0	0	1
25c		0	0	0	0
total	20	14	6	12	10

Table 6: Summary of Fault Detection

This data makes very clear the difficulty of writing effective self-checks. Of 20 previously known faults in the programs, only 11 were detected (the 14 detected known faults in Table 6 include some multiple detections of the same fault) and only 3 of the 11 detected faults were found by more than one of the three participants instrumenting the same program. It should be noted, however, that the versions used in the experiment are highly reliable (an average 99.9165% success rate on the previous one million case testing), and many of the faults are

quite subtle. We could find no particular types of faults that were easier to detect than others. Individual differences in ability appear to be important here.

One rather unusual case occurred. One of the new faults detected by participant 8c was detected quite by accident. There *is* a previously unknown fault in the program. However, the checking code contains the same fault. An error message is printed because the self-check code uses a different algorithm than the original, and finite precision problems cause the self-check to differ from the original by more than the allowed real-number tolerance. We discovered the new fault while evaluating the error messages printed, but it was entirely by chance. This same thing occurred in modules which did not contain a fault, and in that case the error message was classified as a false alarm (as discussed above). Our decision was to classify the self-check as effective because it does signal a fault when a fault does exist, but this is a subjective choice.

It is very interesting that the self-checks detected 6 faults not previously detected by comparison of twenty-seven versions of the program with a gold version over a million test cases. After closer examination of the newly discovered faults, we found that one of the reasons they were not uncovered previously is that the strategy of test case selection did not include those test cases that would have revealed the faults. This points out the well-known difficulty in selecting appropriate test cases. The fact that the self-checks uncovered new faults implies that they may have some advantages over voting alone. The faults were not detected during the previous testing because the voting procedure could compare only the final result of computations (since different algorithms were used), whereas the self-check verified the validity of intermediate results as well. For the few cases in which it arose, the faults did not affect the correctness of the final output. However, under different circumstances the final output would have been incorrect.

Although new faults were introduced through the self-checks, this is not very surprising. It is known that changing someone else's program is difficult and whenever new code is added to a program there is a possibility of introducing faults. All software fault tolerance methods involve adding additional code of one kind or another to the basic application program. The major causes of the new faults were an algorithmic error in a redundant computation, use of an uninitialized variable during instrumentation, logic error, use of Heron's formula, infinite loops added in instrumentation, out of bounds array reference, etc. The use of uninitialized variables occurred due to incomplete program instrumentation. A participant would declare a temporary variable to hold an intermediate value during the computation, but fail to assign a value on some path through the computation. A rigorous acceptability test may have detected these faults earlier, especially those that cause an abnormal termination of the program.

Conclusions

This study was not designed to provide definitive answers to any particular questions, but instead to attempt to determine what the important questions are. This should guide us and others in the design of further experiments, in the evaluation of current proposals, and in the design of new methodologies. Some important questions arise as a result of this study that need to be answered such as:

- [1] There appear to be great differences in individual ability to design effective self-checks. This suggests that more training or experience might be helpful. Our participants had little of either although all were familiar with the use of pre- and post-conditions and assertions to formally verify programs. The data suggests that it might also be interesting to investigate the use of teams to instrument code.
- [2] The programs were instrumented with self-checks in our study by participants who did

not write the original code. It would be interesting to compare this with instrumentation by the original programmer. A reasonable argument could be made both ways. The original programmer, who presumably understands the code better, might introduce fewer new faults and might be better able to place the checks. On the other hand, separate instrumentors might be more likely to detect faults since they provide a new view of the problem. More comparative data is needed here.

- [3] Placement of self-checks appeared to cause problems. Some checks that might have been effective failed to detect a fault because they were badly placed. This implies either a need for better decision-making and rules for placing checks or perhaps different software design techniques to make placement easier.
- [4] Specification-based checks alone were not as effective as using them together with code-based checks. This implies that fault tolerance may be enhanced if the alternate blocks in a recovery block scheme, for example, are also augmented with self-checks along with the usual acceptance test. This may also apply to pure voting schemes such as *N*-version programming. A combination of fault-tolerance techniques may be more effective than any one alone. More information is needed on how best to integrate these different proposals.
- [5] The process of writing self-checks is obviously difficult. However, there may be ways to provide help with this process. For example, Leveson and Shimeall (1983) suggest that safety analysis using software fault trees [Leveson and Harvey (1983)] can be used to determine the content and the placement of the most important self-checks. Other types of application or program analysis may also be of assistance. Finally, empirical data about common fault types may be important in learning how to instrument code with self-checks.

Many promising research topics, empirical studies, and experiments are suggested by the results of this study that may lead to better procedures for software error detection.

Acknowledgements

The authors are pleased to acknowledge the efforts of the experiment participants: David W. Aha, Tom Bair, Jack Beusmans, Bryan Catron, Harry S. Delugach, Siamak Emadi, Lori Fitch, W. Andrew Frye, Joe Gresh, Randy Jones, James R. Kipps, Faith Leifman, Costa Livadadas, Jerry Marco, David A. Montuori, John Palesis, Nancy Pomicter, Mary Theresa Roberson, Karen Ruhleder, Brenda Gates Spielman, Yellamraju Venkata Srinivas, Tim Strayer, Gerald Reed Taylor III, and Raymond R. Wagner, Jr.

REFERENCES

- [1] T. Anderson, P.A. Barrett, D.N. Halliwell, and M.R. Moulding, "An Evaluation of Software Fault Tolerance in a Practical System", *Digest of Papers FTCS-15: Fifteenth Annual Symposium on Fault-Tolerant Computing*, pp. 140-145, June 1985.
- [2] T. Anderson, and P.A. Lee, *Fault Tolerance: Principles and Practice* Englewood Cliffs, NJ, Prentice-Hall Intl., 1981.
- [6] S.S. Brilliant, J.C. Knight, and N.G. Leveson, "The Consistent Comparison Problem in N-Version Software", submitted for publication, 1986a.
- [7] S.S. Brilliant, J.C. Knight, and N.G. Leveson, "Analysis of Faults in an N-Version Software Experiment", submitted for publication, 1986b.
- [13] J.C. Knight and N.G. Leveson, "An Experimental Evaluation of the Assumption of Independence in Multi-Version Programming", *IEEE Transaction on Software Engineering*, pp. 96-109, January 1986a.
- [15] N.G. Leveson, and P.R. Harvey, "Analyzing Software Safety", *IEEE Transactions on Software Engineering*, Vol. SE-9, No. 5, pp 569-579, 1983.
- [16] N.G. Leveson, and T.J. Shimeall, "Safety Assertions for Process-Control Systems", *Digest of Papers FTCS-13: Thirteenth Annual Symposium on Fault-Tolerant Computing*, pp 236-240, June 1983.

AN EXPERIMENTAL COMPARISON OF FORTRAN AND
ADA PROGRAM RELIABILITY

Amrit L. Goel¹

Joseph Cavano²

F. Youwakim Farhat¹

Tom Little¹

Presented at the Eleventh NASA/SEL Annual
Software Engineering Workshop NASA/GSFC
Greenbelt, Maryland
December 3, 1986

¹Syracuse University, Syracuse, NY 13244

²RADC, GAFB, Rome, NY 13441

This work was supported in part by SCEE and
the Rome Air Development Center, GAFB

This report documents an experiment investigating the effect of Fortran and Ada languages on program reliability. The experimental design employed here was a 2^3 full factorial design, i.e., a design in three variables, each at two levels. The variables and their levels were language (Fortran and Ada), programmer experience (intermediate and advanced) and application type (scientific and text processing). Due to experimental and resource constraints, this study concentrated on a 2^2 factorial design in two variables, language and programmer experience. Some of the experimental points in this design were replicated to determine precision of the effect of language and experience on program reliability. The scientific problem used here was the Launch Interceptor Program (LIP), a simple but realistic, anti-missile system which has been used elsewhere in connection with software testing and fault-tolerance research. The second problem used was the well-known text-formatting program. The programmers were graduate students in computer engineering with varying degrees of experience in programming languages.

The programmers developed their own designs from given specifications and did independent compilations and unit testing. Data on these activities were collected for comparison purposes. The function testing of each LIP version was done after removing errors detected during unit testing and compilation. Fifty-four test cases for this purpose were developed manually using a hybrid of structure-dependent and structure-independent testing techniques.

In this paper we present the results of an experiment investigating the effect of Fortran and Ada languages on program reliability. The experimental design employed here was a 2^2 full factorial design, i.e., a design in two variables, each at two levels. The variables and their levels were language (Fortran and Ada) and programmer experience (intermediate and advanced). Some of the experimental points in this design were replicated to determine precision of the effects on program reliability. The scientific problem used here was the Launch Interceptor Program (LIP), a simple but realistic, anti-missile system which has been used elsewhere in connection with software testing and fault-tolerance research. The second problem used was the well-known text-formatting program. The programmers were graduate students in computer engineering with varying degrees of experience in programming languages.

The programmers developed their own designs from given specifications and did independent compilations and unit testing. Data on these activities were collected for comparison purposes. The function testing of each LIP version was done after removing errors detected during unit testing and compilation. Fifty-four test cases for this purpose were developed manually using a hybrid of structure-dependent and structure-independent testing techniques.

Since it was not feasible to determine the oracle for the LIP programs, a variation of majority voting technique was used to determine correct values. Specifically, elements of the conditions met vector, the CMVE's, were compared from each version and the majority of five was taken to be the correct values. This technique proved to be very useful and efficient in testing and debugging. Also, this method was successful in providing a very high structural and functional coverage of the programs.

Errors revealed by the above fifty-four test cases were removed one-at-a-time and the remaining test cases were run on these incrementally corrected versions. The plots of cumulative error symptoms versus test case number seem to follow homogenous Poisson processes in each case and provide some useful insights into the error symptom occurrence behavior.

Operational testing of the LIP programs was done after removing errors found during function testing using three sets of test cases. The first set of 120 tests was developed manually using the same hybrid techniques as for function testing. The other two sets of 100 and 1000 tests were based on random testing.

Data on errors were collected during development and unit testing, function testing and operational testing. This included numbers, causes and types of error and times taken to isolate and fix them. Reliability comparison between Ada and Fortran programs were based on the total number of errors, as well as on errors found during development and functional testing, functional and

operational testing and operational testing. Some comparisons were also based on error density, the number of errors per 100 non-comment lines of code. Further analyses were based on error causes (design, coding and programmer) as well as on error types (control, data and interface). Following are the main results of these analyses.

The number of errors in the Ada programs were about 70% less than those in Fortran when comparisons were based on errors found during all phases. If errors during development and unit testing were excluded, i.e., if only functional and operational testing data were considered, the Ada programs had about 78% less errors. Similar differences were found for data based on error causes and error types.

Using error density during development and functional testing as a measure, the average difference between Ada and Fortran programs was 5.7 errors per 100 non-comment lines with a standard deviation of 1.35 units. The effect of programmers' experience was to reduce error density by 2.1 with a standard deviation of 1.35 units. If data during functional and operational testing alone is considered, then the Ada programs had 3.20 less errors per 100 non-comment lines. The effect of experience during these phases was not statistically significant.

Summarized above are the most important results of this study. Even though they are based primarily on implementations of only one problem, they do indicate that there is a statistically significant evidence in support of higher reliability of Ada programs. The extent of this difference, however, is likely to vary from one application to another as well as across different development environments.

THE VIEWGRAPH MATERIALS
FOR THE
A. GOEL PRESENTATION FOLLOW

STUDY OBJECTIVE

Assess the effect of Fortran and Ada languages on Program Reliability in a controlled Experimental Environment

Specifically, determine the number of errors detected during development and unit testing, function testing, and operational testing

Also, analyze detected errors for causes, symptoms, types, etc.

The observed numbers are used to assess the relative reliability of the programs developed in these languages

EXPERIMENTAL DESIGN

A 2^3 FULL FACTORIAL DESIGN

FACTORS

LEVELS

Language

Fortran ADA

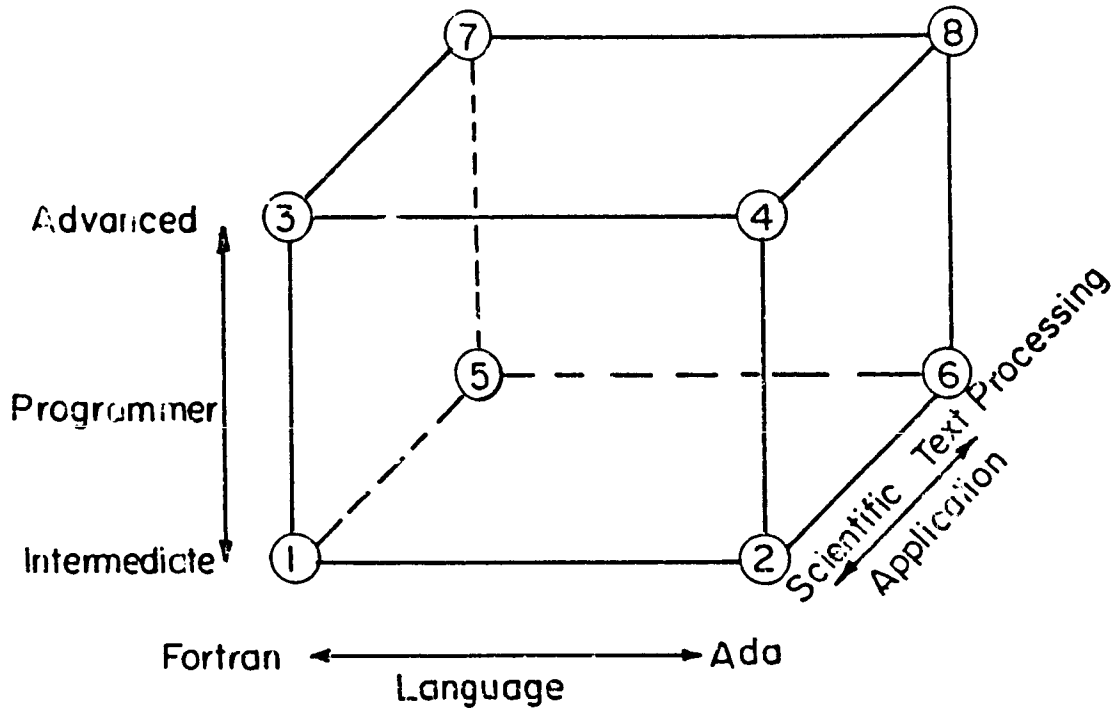
Programmer
Experience

Intermediate Advanced

Application
Type

Scientific Text Processing

A 2^3 FULL FACTORIAL DESIGN

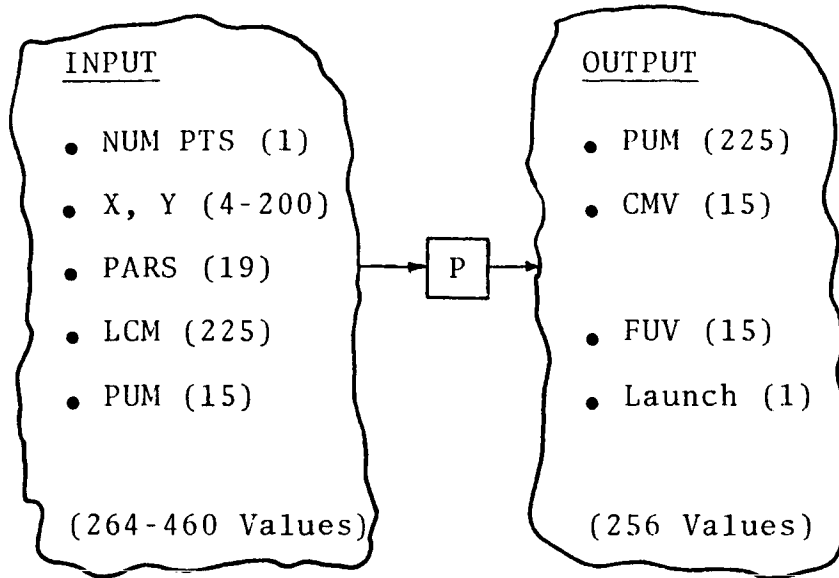


LAUNCH INTERCEPTOR PROGRAM

- . Simple, but realistic anti-missile system.
- . Studied elsewhere^{*} in connection with fault-tolerant software research.
- . Program reads inputs which represent radar reflections, checks whether some prespecified conditions are met and determines if the reflections come from an object that is a threat and if yes, signals a launch decision.

^{*} Knight and Leveson, IEEE-TSE, January 1986.

SCHEMATIC OF LIP

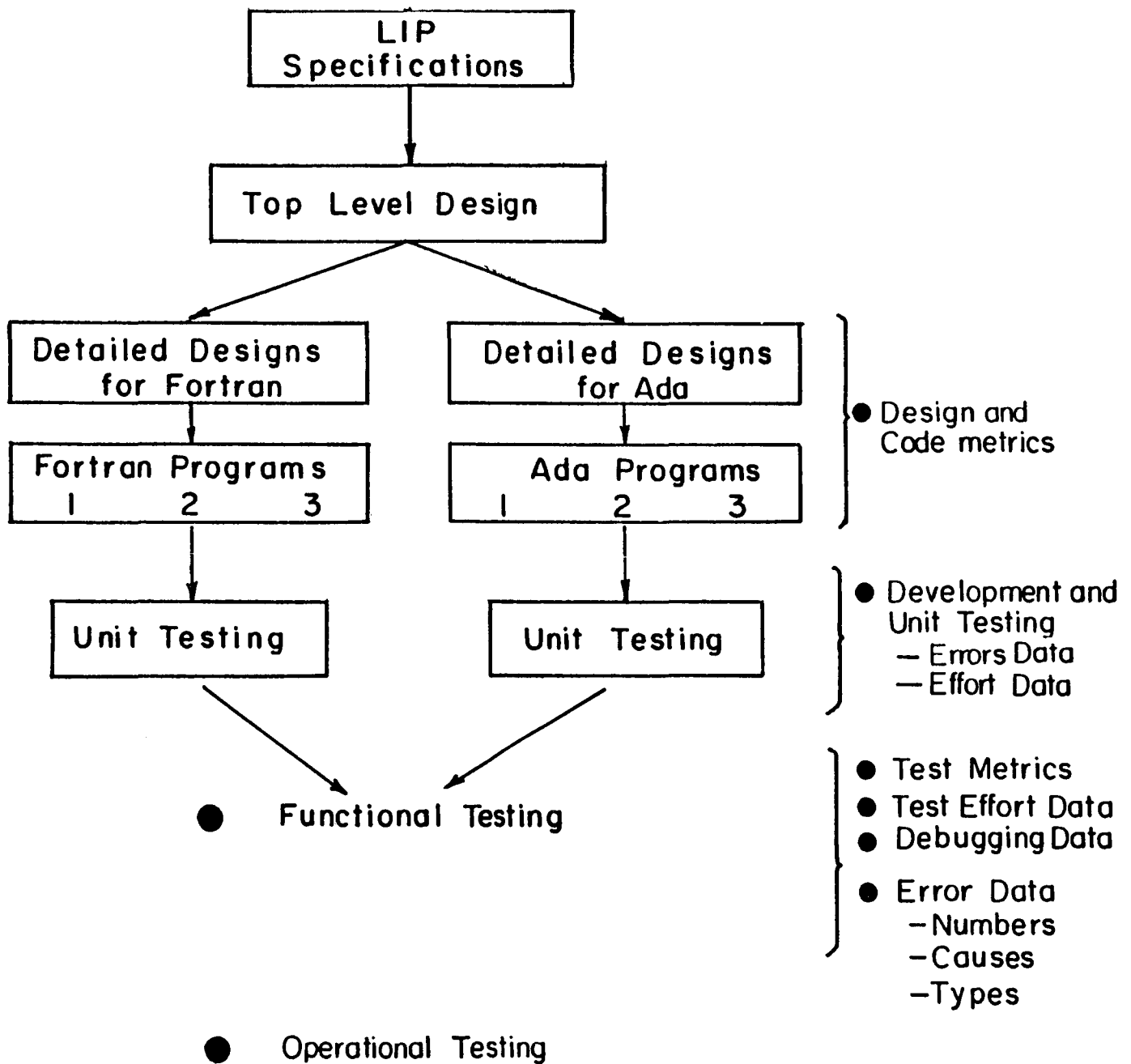


EXAMPLE

Launch Interceptor Conditions

LIC 1: There exists at least one set of two consecutive data points that are a distance greater than LENGTH1 apart

LIC 11: There exists at least one set of three data points separated by exactly E PTS and F PTS...that are the vertices of a triangle with area greater than AREAL



EXPERIMENTAL APPROACH: LIP

- . Developed six versions in Fortran and ADA
 - Two each by intermediate programmers
 - One each by advanced programmers
- . Collected data on errors and effort during development.
- . Unit testing - some structure based, some function based.
- . LIP testing - 54 test cases derived from specifications and ADA Code.
- . Further testing on 'corrected' versions (120 test cases).
- . Random testing on 'corrected' versions (100 + 1000 test cases).
- . Analysis of errors: numbers, causes, type, etc. to assess reliability.

Some Metrics for the LIP Programs

	F1	F2	A1	A2	F3	A3
Source Lines	696	446	691	624	526	851
Non-Comment Lines	550	442	632	498	439	600
Executable Statements	212	246	214	184	174	137
Packages	*	*	6	4	*	3
Subprograms	21	19	30	23	28	29
Procedures	*	*	4	4	*	1
Subroutines	13	19	*	*	28	*
Tasks	*	*	15	15	*	0
Functions	8	0	11	4	0	28
Exceptions	*	*	2	0	*	2
Raise	*	*	0	0	*	3

RELIABILITY ASSESSMENT

- . Based on an analysis of errors found during development and testing.
- . Errors analyzed by
 - Numbers
 - Causes
 - Symptoms
 - Types

SUMMARY OF NUMBER OF ERRORS
FOUND DURING VARIOUS PHASES

	<u>F1</u>	<u>F2</u>	<u>A1</u>	<u>-A2</u>	<u>F3</u>	<u>A3</u>
Dev. & Unit Testing	24	28	8	7	10	4
LIP Testing	16	18	5	4	15	4
<hr/>						
Subtotal	40	46	13	11	25	8
<hr/>						
<u>OT</u>						
120 TC	1*	1	0	0	1	1
100 Random	2*	0	1	0	1	0
1000 Random	3*	0	0	0	0	0
<hr/>						
Total OT	4	1	1	0	2	0
<hr/>						
Total (All Phases)	44	47	14	11	27	8

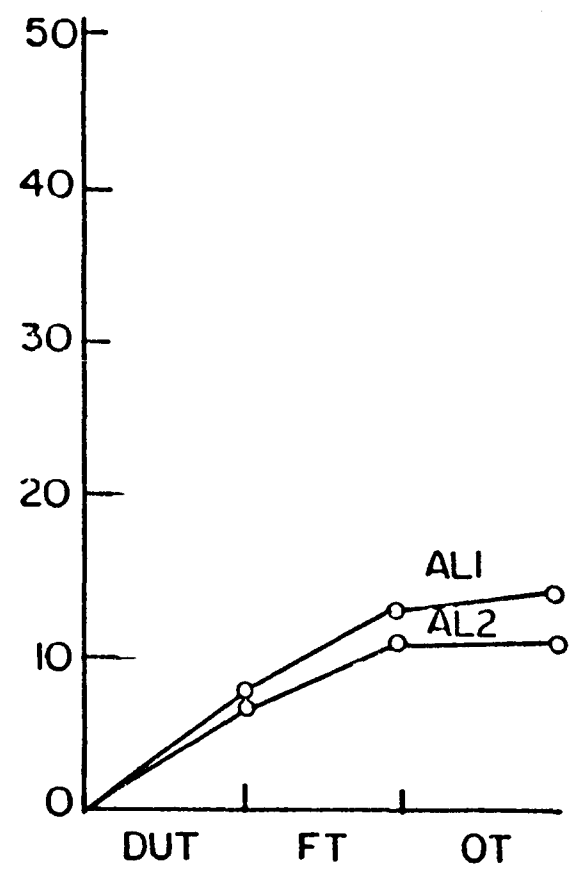
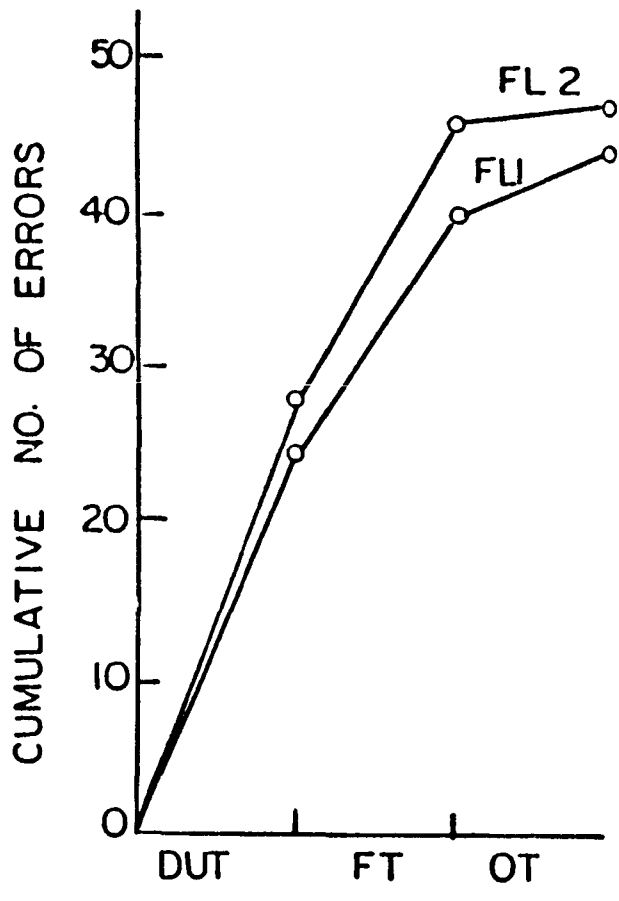
(* one common error)

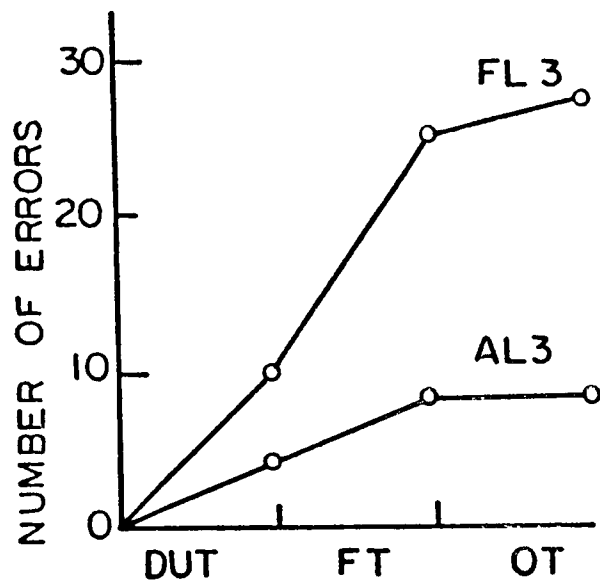
TOTAL ERRORS AND ERROR DENSITIES

<u>INTERMEDIATE</u>	<u>F</u>	<u>A</u>	<u>% DIFFERENCE</u>
Average Number	45.5	12.5	72.5
Average Density	9.1	2.1	76.9

ADVANCED

Number	27.0	8.0	70.4
Density	5.7	1.3	77.2





LIP AND OPERATIONAL TESTING

<u>INTERMEDIATE</u>	<u>F</u>	<u>A</u>	<u>% DIFFERENCE</u>
Average Number	19.5	5.0	74.4
Average Density	4.07	0.88	78.4

ADVANCED

Number	17.0	4.0	76.5
Density	3.88	0.67	82.7

ERROR CLASSIFICATION^{*}

Phase : Design, Coding

Cause : Previous fix,
Programmer error

Symptoms: Infinte loop

Type : Computation, Interface

* Basili & Perricone, Selby

DESIGN AND CODING ERRORS

<u>INTERMEDIATE</u>	<u>F</u>	<u>A</u>	<u>% DIFFERENCE</u>
Design	13.5	5.0	62.9
Coding	29.5	7.0	76.3
 <u>ADVANCED</u>			
Design	10.0	3.0	70.0
Coding	15.0	5.0	66.7

ERROR CAUSES: LIP TESTING

	<u>F1</u>	<u>F2</u>	<u>A1</u>	<u>A2</u>	<u>F3</u>	<u>A3</u>
Pro- grammer Error	4	10	0	1	9	0
Previous Fix	2	0	0	0	2	0
Incorrect Imple- mentation of Specs.	6	4	2	3	2	3
Clerical	3	4	2	0	2	1
Program. Language Misunder- standing	1	0	0	0	0	0
Total	16	18	4	4	15	4

ERROR SYMPTOMS: LIP TESTING

	<u>F1</u>	<u>F2</u>	<u>A1</u>	<u>A2</u>	<u>F3</u>	<u>A3</u>
Overflow/ Underflow	1	4	0	1	0	0
Infinite Loop	0	0	0	1	0	0
Wrong Result	15	14	5	2	15	4
<hr/>						
Total	16	18	15	4	15	4
<hr/>						

ERROR TYPES: LIP TESTING

	<u>F1</u>	<u>F2</u>	<u>A1</u>	<u>A2</u>	<u>F3</u>	<u>A3</u>
Computation	4	6	2	1	5	2
Control	6	4	0	2	4	0
Interface	5	1	0	0	1	2
Data	1	7	3	1	5	0
Total	16	18	5	4	15	4

APPENDIX A

ATTENDEES OF THE 1986 SOFTWARE ENGINEERING WORKSHOP

AGRESTI, BILL W.	COMPUTER SCIENCES CORPORATION
ALDRIDGE, JACK	McDONNELL-DOUGLAS ASTRONAUTICS CO.
ANDERSON, JOYCE	COMARCO, INC.
ANDREW, DON A.	FINANCIAL MANAGEMENT SERVICE
ANTALEK, RICHARD W.	SOCIAL SECURITY ADMINISTRATION
ARMSTRONG, MARY	IIT RESEARCH INSTITUTE
ARNOLD, ROBERT	MITRE CORP.
ARTHUR, JAMES	VIRGINIA TECH UNIV.
ASKEW, MICHAEL B.	GTE CORPORATION
AYERS, EVERETT	ARINC RESEARCH CORP.
BAILEY, JOHN	
BAILIN, SID	CTA
BARNES, BILL P.	NASA/GSFC
BARRETT, CURTISS C.	NASA/GSFC
BASILI, VIC	UNIVERSITY OF MARYLAND
BIGWOOD, DOUGLAS W.	U.S. DEPT. OF AGRICULTURE
BILLINGS, DIANE	FEDERAL COMMUNICATIONS COMMISSION
BLIZZARD, MICHAEL	LOCKHEED EMSCO (PB-9)
BLUM, BRUCE	APPLIED PHYSICS LAB
BOEHM, BARRY	TRW
BOND, JACK	U.S. DEPT. OF DEFENSE
BOND, LISA T.	NAVAL SURFACE WEAPONS CENTER
BOON, DAVE	COMPUTER SCIENCES CORP.
BOOTH, ERIC	COMPUTER SCIENCES CORP.
BRANDT, KIM	NASA/GSFC
BREDESON, MIMI	SPACE TELESCOPE SCIENCE INST.
BROPHY, CAROLYN	UNIVERSITY OF MARYLAND
BROWN, KEITH L.	
BROWN, ROBERT G.	IBM CORP.
BUCHANAN, GEORGE A.	IITRI
BUELL, JOHN	COMPUTER SCIENCES CORP.
BURLEY, RICK	NASA/GSFC
CAIN, BETTY	DEPT. OF TREASURY
CALLENDER, DAVID	NASA/JSC
CAPRIOTTI, DAVID	BURROUGHS CORP.
CARD, DAVE	COMPUTER SCIENCES CORP.
CARLSON, JOHN P.	OAD CORP.
CARLSON, ROBERT	NASA/ARC
CARRIO, MIGUEL	TELEDYNE BROWN ENGINEERING
CARSON, JOHN H.	GEORGE WASHINGTON UNIVERSITY
CARTER, HAROLD	MARTIN MARIETTA CORP.
CARY, JOHN	GEORGE WASHINGTON UNIVERSITY
CHASSON, MARGARET C.	IBM CORP.
CHEN, CHI-FANG	
CHENOWETH, HALSEY B.	WESTINGHOUSE ELECTRIC COMPANY
CHEUVRONT, STEVE	COMPUTER SCIENCES CORP.
CHMURA, LOUIS J.	NAVAL RESEARCH LAB.
CHU, RICHARD	MARTIN MARIETTA AEROSPACE
CHUNG, ANDREW	FAA TECHNICAL CENTER
CHURCH, VIC	COMPUTER SCIENCES CORPORATION
CISNEY, LEE	NASA/GSFC
CLAPP, JUDITH A.	THE MITRE CORPORATION

CLARK, DAVID
CLEMENTS, PAUL
COHEN, VIC
COLLINS, MICHAEL D.
COOK, JOHN F.
COOK, LAURA
CORNEL, ROGER
COTNOIR, DONNA
COUCHOUD, CARL
CROTEAU, LEE
CYPRYCH, GENE
DAILY, JACK
DAVIS, ANN
DECKER, WILLIAM
DELONG, SUZANNE
DEMEESTER, RICHARD H.
DEVANE, ARYNE
DICKSON, CHARLES H.
DISKIN, DAVID
DONNELLY, LAURIE
DOUGLAS, FRANK J.
DUVALL, LORRAINE
DYMOND, KEN
EBERHART, H. O.
ELLIOTT, DEAN F.
ELLIS, WALTER
ELOVITZ, HONEY S.
ESKER, LINDA
EVANGELIST, MICHAEL
FABISZAK, CATHY
FANG, AI
FATH, RICHARD
FINDLEE, JOHN
FISHTAHLER, LARRY
FORMANAK, KATHLEEN
FORSYTHE, RON
FRAHM, MARY J.
FRANKEL, SHEILA
FRANKLIN, JUDE
FREUND, AL
FRIEDMANN, DAN
GAFFNEY, JOHN
GARRICK, JOE
GERSTNER, DIETWALD
GETTIER, CHARLES
GIFFIN, GEOFF
GODFREY, SALLY
GOEDDEKE, VINCENT
GOEL, AMRIT
GOGIA, B. K.
GOLDBERG, ALEXANDER
GOLDEN, JOHN R.
GORDON, LLAYDON
GORDON, MARC D.

SYSTEMS DEVELOPMENT CORP.
NAVAL RESEARCH LAB
EPA
MARTIN MARIETTA CORP.
NASA/GSFC
GSC
U.S. EPA
COMPUTER SCIENCES CORP.
SOCIAL SECURITY ADMINISTRATION
NYMA
COMPUTER SCIENCES CORP.
NAVAL RESEARCH LAB
COMPUTER SCIENCES CORP.
COMPUTER SCIENCES CORP.
COMPUTER SCIENCES CORP.
GENERAL ELECTRIC
FCC
USDA-ARS-ASRI-HYDROLOGY
U.S. CENSUS BUREAU
LOCKHEED
PROFESSIONAL SOFTWARE SERVICES
DUVALL COMPUTER TECHNOLOGIES, INC.
NATIONAL BUREAU OF STANDARDS

NASA/GSFC
IBM/FEDERAL SYSTEM DIVISION
SOFTWARE A & E
COMPUTER SCIENCES CORP.
MCC
NASA/GSFC
NASA/HQ
FCC AMD-IM
DEPT. OF TREASURY
COMPUTER SCIENCES CORP.
MARTIN MARIETTA-ATC DIVISION
NASA/WALLOPS FLIGHT FACILITY

NATIONAL BUREAU OF STANDARDS
PRC/GIS
ALLEN BRADELY CO.

IBM CORPORATION--FSD
NASA/GSFC
NASA/HQ
IIT RESEARCH INSTITUTE
JET PROPULSION LAB
NASA/GSFC
NASA/MARSHALL SPACE FLIGHT CENTER
SYRACUSE UNIVERSITY
MARTIN MARIETTA AEROSPACE
MARTIN MARIETTA CORP.
EASTMAN KODAK CO.
CSC
MCI CORP.

GREEN, DANIEL
GREEN, SCOTT
GREEN, STAN
GREENBERG, DIANA
GREGOR, ROBERT M.
HALTERMAN, KAREN
HARDY, ROBERT
HEASTY, RICHARD
HECK, JOANN L.
HELLER, GERRY
HENRY, SALLIE
HENRY, STEPHANIE
HILL, DONNA E.
HILLMER, DOUG
HOLMES, BARBARA
HORMBY, TOM
HOUSER, WALTER
HULL, LARRY
HUTTMAN, GREG
HYBERTSON, DUANE
IDELSON, NORMAN L.
ITKIN, DAVID
JACKSON, GEORGE
JAWORSKI, ALLAN
JELETIC, JIM
JOESTING, DAVID
JONES, CHRISTOPHER
JOO, BOK G.
JORDAN, LEON
JUN, LINDA
KARDATZKE, OWEN
KATZ, BETH
KAUSCH, CHUCK
KELLY, KIM R.
KESSINGER, RICHARD
KESTER, RUSH
KLENK, JOHN
KLITSCH, GERALD N.
KNIGHT, JOHN
KOWALCHACK, BONNIE
KOZIUK, FRANK
KRAMER, NANCY
KRUSZEWSKI, GEORGE
KUHN, RICK
KURIHARA, TOM
LABAW, BRUCE
LAMAS, NIKI
LANGDON, WOODY
LAVALLEE, DAVID
LAZARO, JOE
LEADER, KAREN
LEBAIR, BILL
LEDBETTER, JOAN M.
LEVESON, NANCY G.

IITSG/GAIO
NASA/GSFC
IIT RESEARCH INSTITUTE
U.S. EPA
SOFTWARE A & E
OAO CORPORATION
FCC
COMPUTER SCIENCES CORP.
RMS TECHNOLOGIES, INC.
COMPUTER SCIENCES CORP.
DEPARTMENT OF COMPUTER SCIENCE
NASA/GSFC
NSWC
U.S. CENSUS BUREAU
GSC NASA/GSFC
JOHN HOPKINS UNIVERSITY
GSA/IRMSKMP
NASA/GSFC
CENSUS BUREAU
LOCKHEED
APPLIED PHYSICS LAB
UNIVERSITY OF MARYLAND
ACTION
FORD AEROSPACE
NASA/GSFC
BENDIX FIELD ENGINEERING CORP.
IIT RESEARCH INSTITUTE
UNIVERSITY OF MARYLAND
COMPUTER SCIENCES CORP.
NASA/GSFC
NASA/GSFC
UNIVERSITY OF MARYLAND
NASA/GSFC
IBM CORP.
SOFTECH, INC.
GTE GOVERNMENT SYSTEMS
THE MITRE CORPORATION
COMPUTER SCIENCES CORP.
UNIVERSITY OF VIRGINIA
UNIVERSITY OF MARYLAND
IIT RESEARCH INSTITUTE

KETRON, INC.
NATIONAL BUREAU OF STANDARDS
U.S. DEPT. OF TRANSPORTATION
NAVAL RESEARCH LAB
CENSUS BUREAU
COMPUTER SCIENCES CORP.
FORD AEROSPACE CO.

IIT RESEARCH INSTITUTE
NASA/GSFC
USDA-OIRM-ATSD
UNIVERSITY OF CALIFORNIA

LEVINE, DAVID R.
LEWIS, BLAIR
LEWIS, JAMES M.
LIN, CHI Y.
LIPHAN, BILL
LIU, JEAN C.
LIVELY, MARY
LLOYD, MICHAEL R.
LOESH, BOB
LORD, YVONNE
LUCAS, JANICE
LUCAS, TODD R.
LUCZAK, RAY W.
LUPTON, GLENN
LURIE, BRIAN
LYTTON, VICTOR H.
MAJORS, SHEILA
MALAY, ROBERT M.
MARCINIAK, JOHN
MARTIN, RONALD
McCOMAS, DAVE
McENEARNEY, MARK
McGARRY, FRANK E.
McGARRY, MARY ANN
McGOVERN, DAN
McKENZIE, MERLE
McMAHON, SUE
MEDUIN, JENNIFER B.
MEYER, MARGARET
MOHRMAN, CARL C.
MONTGOMERY, AL
MORGAN, CAROL
MUCKEL, JERRY
MULCAHY, KEVIN
MULLEN, PETER
MUNDY, CORNELIA C.
MURPHY, CYNTHIA
MURPHY, ROBERT
MYERS, PHILIP I.
NANCE, RICHARD E.
NAPKORI, JOE
NELSON, ROBERT
NGUYEN, BAO T.
NICHOLAS, DAVID A.
NOONAN, ROBERT
NORCIO, TONY F.
NORMAN, KENT
OBREBSKI, KRZYSZTOF
OEI, CHARLES
OHLMACHER, JANE
ONG, JOHN
OSBORNE, WILMA
OVERDECK, BETSEY L.
OWENS, ANTHONY

INTERMETRICS, INC.
JET PROPULSION LAB
U.S. CENSUS BUREAU
JPL
DEPT. OF TREASURY
COMPUTER SCIENCES CORP.
U.S. EPA
GENERAL DYNAMICS
SYSTEM TECHNOLOGY INSTITUTE
WESTINGHOUSE DEFENSE & ELEC. CENTER
DEPT. OF TREASURY

COMPUTER SCIENCES CORP.
DIGITAL EQUIPMENT CORP.
BURROUGHS CORP.
U.S. DEPT. OF AGRICULTURE
CSC

MARCINIAK & ASSOCIATES
FCC
NASA/GSFC
BUREAU OF LABOR STATISTICS
NASA/GSFC
IITRI-ECAC/U.S. NAVAL STATION
FAA TECHNICAL CENTER
JET PROPULSION LAB
JET PROPULSION LAB
NASA/GSFC
U.S. ARMY OFFICE OF TEST & EVAL. AGENCY
MARTIN MARIETTA ATC
NYMA
IIT RESEARCH INSTITUTE
COMPUTER SCIENCES CORP.
U.S. EPA
MARTIN MARIETTA AEROSPACE
NAVAL SURFACE WEAPONS CENTER
THE MITRE CORP.
NASA/GSFC
COMPUTER SCIENCES CORP.
VIRGINIA TECH UNIV.
SYSTEM DEVELOPMENT CORP.
NASA/GSFC
HQ USAF/SCTT
JET PROPULSION LAB
COLLEGE OF WILLIAM AND MARY
NAVAL RESEARCH LAB
UNIVERSITY OF MARYLAND
PROGRAM RESOURCES, INC.
SDC/BURROUGHS
SOCIAL SECURITY ADM.
NASA/GSFC
NATIONAL BUREAU OF STANDARDS
COMPUTER SCIENCES CORP.
FCC

PAGE, JERRY
PALMER, JAMES G.
PALMQUIST, SALLY
PAPPAS, EUGENE
PARKER, DON
PASSALACQUA, TERESA
PAVNICA, PAUL
PERKINS, DOROTHY
PERKINS, TOBY
PERRY, SANDRA
PETER, MICHAEL
PETERSEN, JANE B.
PIETRAS, JOHN
PIXTON, JERRY
PLETT, MICHAEL E.
PRABHAKAR, N. D.
PRESTON, DAVID
PRINCE, ANDY
PUTNEY, BARBARA
QUANN, EILEEN S.
QUIMBY, KELVIN
QUINN, JEAN T.
RAY, SUSAN
REDWINE, SAM
REED, KARL
RHOADS, TOM
RICHARDSON, EDDIE
ROBBINS, DON
ROBERTS, REBEKAH
ROBINSON, RICHARD
ROHR, JOHN A.
ROMBACH, DIETER H.
ROSENFELD, ROCHELLE
ROWE, DENNIS
ROY, DANIEL M.
RUFFNER, ALAN
SAMII, MINA V.
SANDBORGH, RAYMOND E.
SAWYER, DON M.
SCALISE, GARY
SCHUBERT, KATHY
SCHULTHEISZ, ROBERT
SCHULTZ, DAVID
SCHWARTZ, DAVID
SCIULLO, ED
SCOTT, EUGENE H.
SCOTT, LEIGHTON
SEIDEWITZ, ED
SEMMELE, RALPH
SERAFIN, PAUL
SHEN, VINCENT Y.
SHI, LEON
SHI, XIAOHONG
SHIEK, ELVIRA

COMPUTER SCIENCES CORP.
JOHNS HOPKINS APPLIED PHYSICS LAB
IIT RESEARCH INSTITUTE
IIT RESEARCH INSTITUTE
NASA/GSFC
CENSUS BUREAU

NASA/GSFC
SPERRY CORP.
CSC
GENERAL SERVICES ADMINISTRATION
AUTOMETRIC, INC.
MITRE CORP.
SYSTEMS DEVELOPMENT CORP.
COMPUTER SCIENCES CORP.
AT & T BELL LABORATORIES
IIT RESEARCH INSTITUTE
PRC SYSTEMS SERVICES
NASA/GSFC
COMPUTER SCIENCES CORP.
COMPUTER SCIENCES CORP.
NAVAL RESEARCH LABORATORY
NASA/GSFC
INSTITUTE FOR DEFENSE ANALYSIS
UNIVERSITY OF MARYLAND
COMPUTER SCIENCES CORP.
FCC
NSA/CSS
PRC
THE MITRE CORPORATION
JET PROPULSION LAB
UNIVERSITY OF MARYLAND
GTE GOVERNMENT SYSTEMS
MITRE CORPORATION
CENTURY COMPUTING
MCI CORP.
COMPUTER SCIENCES CORP.
UNISYS CORP.
NASA/GSFC
DEPARTMENT OF TRANSPORTATION
NASA/LERC
WEDLARS III
COMPUTER SCIENCES CORP.
BENDIX
NATIONAL LIBRARY OF MEDICINE
COMPUTER SCIENCES CORPORATION
NSA
NASA/GSFC
APL
EG & G
MCC
COMPUTER SCIENCES CORP.
UNIVERSITY OF MARYLAND
NASA/GSFC

SHOAN, WENDY
SMITH, DAN
SMITH, NANCY
SMITH, PAT
SMITH, PEG M.
SNYDER, GLENN
SO, MARIA
SOLOMON, DAVID
SOLOWAY, ELLIOT
SPAFFORD, EUGENE H.
SPEIZER, HOWARD M.
SPENCE, BAILEY
SPIEGEL, DOUG
SPIEGEL, MITCHELL
STARK, MICHAEL
STAUFFER, MIKE
STEINBACHER, JODY
STOKES, ED
STONE, DAWN F.
STOTTS, DAVID
STUART, WILLIAM
SUDDITH, STEVE
SWEENEY, JOHNNY
SWEET, BILL
SZULEWSKI, PAUL
TALLEY, RONALD D.
TARDIF, MICHELLE
TASAKI, KEIJI K.
TASKY, DEBORAH L.
TAUSWORTHE, ROBERT C.
TENG, BRENDA
THOMAS, JOE H.
THOMPSON, JOHN T.
TREASURE, DAVE
TRIOUFIS, KOSTOS
TSAGOS, DINO
TURNROSE, BARRY
URI, CAROL
USHER, GEORGE
VALETT, JON
VERNACCHIO, AL
VOIGT, SUSAN
VOLTZ, SUSAN
WALDO, KAREN E.
WALLACE, DOLORES
WANG, SHOUL
WASSERMAN, ANTHONY I.
WATSON, STAN
WEISS, DAVID
WENDE, CHARLES
WERLING, RICHARD
WILLIAMS, MIKE
WONG, YEE
WOOD, RICHARD

NASA/GSFC
FORD AEROSPACE & COMM. CORP.
NASA/GSFC
NSWC
NASA HEADQUARTERS
COMPUTER SCIENCES CORP.
NASA/GSFC
COMPUTER SCIENCES CORP.
YALE UNIVERSITY
SOFTWARE ENGINEERING RESEARCH CENTER
U.S. CENSUS BUREAU
COMPUTER SCIENCES CORP.
NASA/GSFC
GTE SYSTEMS
NASA/GSFC
GENERAL ELECTRIC CO.
JET PROPULSION LAB
COMPUTER SCIENCES CORP.
COMPUTER SCIENCES CORP.
UNIVERSITY OF MARYLAND
BUREAU OF CENSUS
GSC
DEPT. OF TREASURY
SOFTWARE ENGINEERING INSTITUTE
C.S. DRAPER LABS

NASA/GSFC
NASA/GSFC
U.S. CENSUS BUREAU
JPL
COMPUTER SCIENCES CORP.
NASA/MSFC
FORD AEROSPACE
CENSUS BUREAU

COMPUTER SCIENCES CORP.
FCC AMD-IM
TREASURY-FMS-IS
NASA/GSFC
NASA/GSFC
NASA/LARC
NASA/GSFC
IIT RESEARCH INSTITUTE
NATIONAL BUREAU OF STANDARDS

INTERACTIVE DEVELOPMENT ENVIRONMENTS, INC.
NASA/GSFC
U.S. CONGRESS
NASA/GSFC

NYMA
COMPUTER SCIENCES CORP.
COMPUTER SCIENCES CORP.

WU, LIQUN
WU, SABINA L.
WU, YEN
YANG, CHAO
YEE, MARY
YOUUMAN, CHARLES
ZAMANI, DORY
ZAVELER, SAUL
ZELKOWITZ, MARV
ZYGIELBAUM, ART

UNIVERSITY OF MARYLAND
IITRI
IITRI
NASA/GSFC
LOGICON, INC.
THE MITRE CORPORATION
ACTION
U.S. AIR FORCE
DEPT. OF COMPUTER SCIENCES
JET PROPULSION LAB

APPENDIX B

STANDARD BIBLIOGRAPHY OF SEL LITERATURE

The technical papers, memorandums, and documents listed in this bibliography are organized into two groups. The first group is composed of documents issued by the Software Engineering Laboratory (SEL) during its research and development activities. The second group includes materials that were published elsewhere but pertain to SEL activities.

SEL-ORIGINATED DOCUMENTS

SEL-76-001, Proceedings From the First Summer Software Engineering Workshop, August 1976

SEL-77-002, Proceedings From the Second Summer Software Engineering Workshop, September 1977

SEL-77-004, GSFC NAVPAK Design Specifications Languages Study, P. A. Scheffer and C. E. Velez, October 1977

SEL-78-005, Proceedings From the Third Summer Software Engineering Workshop, September 1978

SEL-78-006, GSFC Software Engineering Research Requirements Analysis Study, P. A. Scheffer and C. E. Velez, November 1978

SEL-78-007, Applicability of the Rayleigh Curve to the SEL Environment, T. E. Mapp, December 1978

SEL-78-302, FORTRAN Static Source Code Analyzer Program (SAP) User's Guide (Revision 3), W. J. Decker and W. A. Taylor, July 1986

SEL-79-002, The Software Engineering Laboratory: Relationship Equations, K. Freburger and V. R. Basili, May 1979

SEL-79-003, Common Software Module Repository (CSMR) System Description and User's Guide, C. E. Goorevich, A. L. Green, and S. R. Waligora, August 1979

SEL-79-004, Evaluation of the Caine, Farber, and Gordon Program Design Language (PDL) in the Goddard Space Flight Center (GSFC) Code 580 Software Design Environment, C. E. Goorevich, A. L. Green, and W. J. Decker, September 1979

SEL-79-005, Proceedings From the Fourth Summer Software Engineering Workshop, November 1979

SEL-80-002, Multi-Level Expression Design Language-Requirement Level (MEDL-R) System Evaluation, W. J. Decker and C. E. Goorevich, May 1980

SEL-80-003, Multimission Modular Spacecraft Ground Support Software System (MMS/GSSS) State-of-the-Art Computer Systems/Compatibility Study, T. Welden, M. McClellan, and P. Liebertz, May 1980

SEL-80-005, A Study of the Musa Reliability Model, A. M. Miller, November 1980

SEL-80-006, Proceedings From the Fifth Annual Software Engineering Workshop, November 1980

SEL-80-007, An Appraisal of Selected Cost/Resource Estimation Models for Software Systems, J. F. Cook and F. E. McGarry, December 1980

SEL-81-008, Cost and Reliability Estimation Models (CAREM) User's Guide, J. F. Cook and E. Edwards, February 1981

SEL-81-009, Software Engineering Laboratory Programmer Workbench Phase 1 Evaluation, W. J. Decker and F. E. McGarry, March 1981

SEL-81-011, Evaluating Software Development by Analysis of Change Data, D. M. Weiss, November 1981

SEL-81-012, The Rayleigh Curve As a Model for Effort Distribution Over the Life of Medium Scale Software Systems, G. O. Picasso, December 1981

SEL-81-013, Proceedings From the Sixth Annual Software Engineering Workshop, December 1981

SEL-81-014, Automated Collection of Software Engineering Data in the Software Engineering Laboratory (SEL), A. L. Green, W. J. Decker, and F. E. McGarry, September 1981

SEL-81-101, Guide to Data Collection, V. E. Church, D. N. Card, F. E. McGarry, et al., August 1982

SEL-81-102, Software Engineering Laboratory (SEL) Data Base Organization and User's Guide Revision 1, P. Lo and D. Wyckoff, July 1983

SEL-81-104, The Software Engineering Laboratory, D. N. Card, F. E. McGarry, G. Page, et al., February 1982

SEL-81-106, Software Engineering Laboratory (SEL) Document Library (DOCLIB) System Description and User's Guide, W. Taylor and W. J. Decker, May 1985

SEL-81-107, Software Engineering Laboratory (SEL) Compendium of Tools, W. J. Decker, W. A. Taylor, and E. J. Smith, February 1982

SEL-81-110, Evaluation of an Independent Verification and Validation (IV&V) Methodology for Flight Dynamics, G. Page, F. E. McGarry, and D. N. Card, June 1985

SEL-81-203, Software Engineering Laboratory (SEL) Data Base Maintenance System (DBAM) User's Guide and System Description, P. Lo, June 1984

SEL-81-205, Recommended Approach to Software Development, F. E. McGarry, G. Page, S. Eslinger, et al., April 1983

SEL-82-001, Evaluation of Management Measures of Software Development, G. Page, D. N. Card, and F. E. McGarry, September 1982, vols. 1 and 2

SEL-82-003, Software Engineering Laboratory (SEL) Data Base Reporting Software User's Guide and System Description, P. Lo, September 1982

SEL-82-004, Collected Software Engineering Papers: Volume 1, July 1982

SEL-82-007, Proceedings From the Seventh Annual Software Engineering Workshop, December 1982

SEL-82-008, Evaluating Software Development by Analysis of Changes: The Data From the Software Engineering Laboratory, V. R. Basili and D. M. Weiss, December 1982

SEL-82-102, FORTRAN Static Source Code Analyzer Program (SAP) System Description (Revision 1), W. A. Taylor and W. J. Decker, April 1985

SEL-82-105, Glossary of Software Engineering Laboratory Terms, T. A. Babst, F. E. McGarry, and M. G. Rohleder, October 1983

SEL-82-406, Annotated Bibliography of Software Engineering Laboratory Literature, D. N. Card, Q. L. Jordan, and F. E. McGarry, November 1986

SEL-83-001, An Approach to Software Cost Estimation, F. E. McGarry, G. Page, D. N. Card, et al., February 1984

SEL-83-002, Measures and Metrics for Software Development, D. N. Card, F. E. McGarry, G. Page, et al., March 1984

SEL-83-003, Collected Software Engineering Papers: Volume II, November 1983

SEL-83-006, Monitoring Software Development Through Dynamic Variables, C. W. Doerflinger, November 1983

SEL-83-007, Proceedings From the Eighth Annual Software Engineering Workshop, November 1983

SEL-84-001, Manager's Handbook for Software Development, W. W. Agresti, F. E. McGarry, D. N. Card, et al., April 1984

SEL-84-002, Configuration Management and Control: Policies and Procedures, Q. L. Jordan and E. Edwards, December 1984

SEL-84-003, Investigation of Specification Measures for the Software Engineering Laboratory (SEL), W. W. Agresti, V. E. Church, and F. E. McGarry, December 1984

SEL-84-004, Proceedings From the Ninth Annual Software Engineering Workshop, November 1984

SEL-85-001, A Comparison of Software Verification Techniques, D. N. Card, R. W. Selby, Jr., F. E. McGarry, et al., April 1985

SEL-85-002, Ada Training Evaluation and Recommendations From the Gamma Ray Observatory Ada Development Team, R. Murphy and M. Stark, October 1985

SEL-85-003, Collected Software Engineering Papers: Volume III, November 1985

SEL-85-004, Evaluations of Software Technologies: Testing, CLEANROOM, and Metrics, R. W. Selby, Jr., May 1985

SEL-85-005, Software Verification and Testing, D. N. Card, C. Antle, and E. Edwards, December 1985

SEL-85-006, Proceedings From the Tenth Annual Software Engineering Workshop, December 1985

SEL-86-001, Programmer's Handbook for Flight Dynamics Software Development, R. Wood and E. Edwards, March 1986

SEL-86-002, General Object-Oriented Software Development, E. Seidewitz and M. Stark, August 1986

SEL-86-003, Flight Dynamics System Software Development Environment Tutorial, J. Buell and P. Myers, July 1986

SEL-86-004, Collected Software Engineering Papers: Volume IV, November 1986

SEL-86-005, Measuring Software Design, D. N. Card, October 1986

SEL-RELATED LITERATURE

Agresti, W. W., Definition of Specification Measures for the Software Engineering Laboratory, Computer Sciences Corporation, CSC/TM-84/6085, June 1984

⁴Agresti, W. W., V. E. Church, D. N. Card, and P. L. Lo, "Designing With Ada for Satellite Simulation: A Case Study," Proceedings of the First International Symposium on Ada for the NASA Space Station, June 1986

²Agresti, W. W., F. E. McGarry, D. N. Card, et al., "Measuring Software Technology," Program Transformation and Programming Environments. New York: Springer-Verlag, 1984

¹Bailey, J. W., and V. R. Basili, "A Meta-Model for Software Development Resource Expenditures," Proceedings of the Fifth International Conference on Software Engineering. New York: IEEE Computer Society Press, 1981

¹Basili, V. R., "Models and Metrics for Software Management and Engineering," ASME Advances in Computer Technology, January 1980, vol. 1

Basili, V. R., Tutorial on Models and Metrics for Software Management and Engineering. New York: IEEE Computer Society Press, 1980 (also designated SEL-80-008)

³Basili, V. R., "Quantitative Evaluation of Software Methodology," Proceedings of the First Pan-Pacific Computer Conference, September 1985

¹Basili, V. R., and J. Beane, "Can the Parr Curve Help With Manpower Distribution and Resource Estimation Problems?", Journal of Systems and Software, February 1981, vol. 2, no. 1

¹Basili, V. R., and K. Freburger, "Programming Measurement and Estimation in the Software Engineering Laboratory," Journal of Systems and Software, February 1981, vol. 2, no. 1

³Basili, V. R., and N. M. Panlilio-Yap, "Finding Relationships Between Effort and Other Variables in the SEL," Proceedings of the International Computer Software and Applications Conference, October 1985

⁴Basili, V. R., and D. Patnaik, A Study on Fault Prediction and Reliability Assessment in the SEL Environment, University of Maryland, Technical Report TR-1699, August 1986

²Basili, V. R., and B. T. Perricone, "Software Errors and Complexity: An Empirical Investigation," Communications of the ACM, January 1984, vol. 27, no. 1

¹Basili, V. R., and T. Phillips, "Evaluating and Comparing Software Metrics in the Software Engineering Laboratory," Proceedings of the ACM SIGMETRICS Symposium/Workshop: Quality Metrics, March 1981

³Basili, V. R., and C. L. Ramsey, "ARROWSMITH-P--A Prototype Expert System for Software Engineering Management," Proceedings of the IEEE/MITRE Expert Systems in Government Symposium, October 1985

Basili, V. R., and R. Reiter, "Evaluating Automatable Measures for Software Development," Proceedings of the Workshop on Quantitative Software Models for Reliability, Complexity, and Cost. New York: IEEE Computer Society Press, 1979

²Basili, V. R., R. W. Selby, and T. Phillips, "Metric Analysis and Data Validation Across FORTRAN Projects," IEEE Transactions on Software Engineering, November 1983

³Basili, V. R., and R. W. Selby, Jr., "Calculation and Use of an Environments's Characteristic Software Metric Set," Proceedings of the Eighth International Conference on Software Engineering. New York: IEEE Computer Society Press, 1985

Basili, V. R., and R. W. Selby, Jr., Comparing the Effectiveness of Software Testing Strategies, University of Maryland, Technical Report TR-1501, May 1985

⁴Basili, V. R., R. W. Selby, Jr., and D. H. Hutchens, "Experimentation in Software Engineering," IEEE Transactions on Software Engineering, July 1986

²Basili, V.R., and D. M. Weiss, A Methodology for Collecting Valid Software Engineering Data, University of Maryland, Technical Report TR-1235, December 1982

³Basili, V. R., and D. M. Weiss, "A Methodology for Collecting Valid Software Engineering Data," IEEE Transactions on Software Engineering, November 1984

¹Basili, V. R., and M. V. Zelkowitz, "The Software Engineering Laboratory: Objectives," Proceedings of the Fifteenth Annual Conference on Computer Personnel Research, August 1977

Basili, V. R., and M. V. Zelkowitz, "Designing a Software Measurement Experiment," Proceedings of the Software Life Cycle Management Workshop, September 1977

¹Basili, V. R., and M. V. Zelkowitz, "Operation of the Software Engineering Laboratory," Proceedings of the Second Software Life Cycle Management Workshop, August 1978

¹Basili, V. R., and M. V. Zelkowitz, "Measuring Software Development Characteristics in the Local Environment," Computers and Structures, August 1978, vol. 10

Basili, V. R., and M. V. Zelkowitz, "Analyzing Medium Scale Software Development," Proceedings of the Third International Conference on Software Engineering. New York: IEEE Computer Society Press, 1978

³Card, D. N., "A Software Technology Evaluation Program," Annais do XVIII Congresso Nacional de Informatica, October 1985

⁴Card, D. N., V. E. Church, and W. W. Agresti, "An Empirical Study of Software Design Practices," IEEE Transactions on Software Engineering, February 1986

³Card, D. N., G. T. Page, and F. E. McGarry, "Criteria for Software Modularization," Proceedings of the Eighth International Conference on Software Engineering. New York: IEEE Computer Society Press, 1985

¹Chen, E., and M. V. Zelkowitz, "Use of Cluster Analysis To Evaluate Software Engineering Methodologies," Proceedings of the Fifth International Conference on Software Engineering. New York: IEEE Computer Society Press, 1981

⁴Church, V. E., D. N. Card, W. W. Agresti, and Q. L. Jordan, "An Approach for Assessing Software Prototypes," ACM Software Engineering Notes, July 1986

²Doerflinger, C. W., and V. R. Basili, "Monitoring Software Development Through Dynamic Variables," Proceedings of the Seventh International Computer Software and Applications Conference. New York: IEEE Computer Society Press, 1983

Higher Order Software, Inc., TR-9, A Demonstration of AXES for NAVPAK, M. Hamilton and S. Zeldin, September 1977 (also designated SEL-77-005)

³McGarry, F. E., J. Valett, and D. Hall, "Measuring the Impact of Computer Resource Quality on the Software Development Process and Product," Proceedings of the Hawaiian International Conference on System Sciences, January 1985

³Page, G., F. E. McGarry, and D. N. Card, "A Practical Experience With Independent Verification and Validation," Proceedings of the Eighth International Computer Software and Applications Conference, November 1984

³Ramsey, J., and V. R. Basili, "Analyzing the Test Process Using Structural Coverage," Proceedings of the Eighth International Conference on Software Engineering. New York: IEEE Computer Society Press, 1985

⁴Seidewitz, E., and M. Stark, "Towards a General Object-Oriented Software Development Methodology," Proceedings of the First International Symposium on Ada for the NASA Space Station, June 1986

Turner, C., and G. Caron, A Comparison of RADC and NASA/SEL Software Development Data, Data and Analysis Center for Software, Special Publication, May 1981

Turner, C., G. Caron, and G. Brement, NASA/SEL Data Compendium, Data and Analysis Center for Software, Special Publication, April 1981

³Weiss, D. M., and V. R. Basili, "Evaluating Software Development by Analysis of Changes: Some Data From the Software Engineering Laboratory," IEEE Transactions on Software Engineering, February 1985

¹Zelkowitz, M. V., "Resource Estimation for Medium Scale Software Projects," Proceedings of the Twelfth Conference on the Interface of Statistics and Computer Science. New York: IEEE Computer Society Press, 1979

²Zelkowitz, M. V., "Data Collection and Evaluation for Experimental Computer Science Research," Empirical Foundations for Computer and Information Science (proceedings), November 1982

Zelkowitz, M. V., and V. R. Basili, "Operational Aspects of a Software Measurement Facility," Proceedings of the Software Life Cycle Management Workshop, September 1977

NOTES:

¹This article also appears in SEL-82-004, Collected Software Engineering Papers: Volume I, July 1982.

²This article also appears in SEL-83-003, Collected Software Engineering Papers: Volume II, November 1983.

³This article also appears in SEL-85-003, Collected Software Engineering Papers: Volume III, November 1985.

⁴This article also appears in SEL-86-004, Collected Software Engineering Papers: Volume IV, November 1986.